LABORATORY FOR
COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-87

ANCILLARY REPORTS: KERNEL DESIGN PROJECT

DAVID D. CLARK, EDITOR

JUNE 1977

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TM -87

ANCILLARY REPORTS: KERNEL DESIGN PROJECT

David D. Clark, editor

June 30, 1977

The research reported here was sponsored in part by Honeywell Information Systems Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order No. 2641 which was monitored by ISTAO under contract No. F19628-74-C-0193.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE

MASSACHUSETTS 02139

FOREWORD

For the past three years, the Computer Systems Research Division of the Laboratory for Computer Science has performed a series of engineering studies on the Multics operating system. The goal was to demonstrate the feasibility of producing a version of a full function general purpose operating system with a "security kernel" simple enough that its correct operating can be certified by some form of auditing. During this project, several results of an interim nature were published as internal group memos, and were never subsequently published in any publicly available form. This memo contains seven such reports that contain interesting results not otherwise reported. These seven reports deal with four areas:

- Analysis of bugs discovered in the Multics system.
- Survey of the initial size of the Multics kernel.
- Detailed design specification of two level process manager.
- Performance evaluation of the multi-process page manager.

D. D. Clark

TABLE OF CONTENTS

REPAIRED SECURITY BUGS IN MULTICS

by J. H. Saltzer

A short time ago I began to compile a list of all known ways in which a user may break down or circumvent the protection mechanisms of Multics. The list is quite interesting, and available for individual study, but until the problems are repaired, it does not seem wise to distribute it widely. On the other hand, it would be wise to promote discussion of the topic, so as problems are fixed, I will publish their descriptions.

Examining post-mortems of fixed bugs may initially strike one as unrewarding, but there are some potential payoffs. Since one of our objectives is to discover how to construct a simple, auditable supervisor which has a very low probability of such errors, the following questions seem worthy of discussion about each bug:

1. How did it get in to the system? What design decisions helped create an environment in which the error was made?
2. Why was it not detected immediately, at checkout time or during system installation? What better auditing tools might have resulted in earlier detection?
3. Was a design principle violated, thereby leading to the error?
4. Is this bug a member of a class of errors, of which there may be more examples in Multics? What design principle or auditing technique might be useful in eliminating all such related errors?

By way of definition, let us use the following arbitrary definition of security-related problems: those which permit

1) unauthorized disclosure of information.
2) unauthorized changing of information.
3) denial of accessibility to authorized users.

A bug which may be exploited to force a system crash is considered to be in the third category. To constrain our area of concern, only security-related problems which are part of operating system design or implementation are

of interest. For example, the practice of leaving the door to the machine room unlocked is not of interest to us. (Unless the cause is a bad design feature of the operating system which prevents convenient system operation inside locked doors.)

Recently Repaired Security Bugs

Several problems were fixed in the installation of system 18.0 which simplified the access control strategy of the system:

1. The CACL ring brackets trap. Before system 18.0, every ACL and CACL entry contained its own separate ring bracket specification, leading to great ease in slipping up, especially if one creates a segment in a strange directory without first checking its CACL. This trap was fallen into by the linker in the following way: if a user in ring 4 called a ring 1 entry for the first time, the linker tried to create a new combined linkage section for ring 1 in the process directory. If the user had previously planted a link with the name "combined_linkage_1.01" in his process directory, the combined linkage segment would actually be created wherever he wished -- in some other directory, for example. Although the linker carefully set the ACL of the new segment to permit ring-one access only, the CACL of the target directory could give access in higher rings to other users.

   Since 18.0 fixed this problem by making the ring bracket specification a property of the segment, as specified by the creator, rather than a property of the individual ACL or CACL entry.

   It should be noted that a contribution to this trap was made by the automatic system feature of allowing segments to be created through links. It would perhaps make sense to allow protected subsystems to specify that they do not want this feature, so that when they create a segment by name, it is created exactly where they expect.

   Security Principle: If the protection status of a segment depends on its position in the naming hierarchy, the creator of a segment nust be given complete control of that position; no one else may be allowed to influence its position.

This principle is currently at odds with two system deficiencies, both of which lead to desire to put links in the process directory:

a) an inflexible process directory record quota scheme, which leads to the need to place some system segments in other directories.

b) the automatic discarding of a process directory contents upon accidental process termination, which leads to a need to place some system segments elsewhere so that they may be examined to discover the reason for the process termination.

It seems quite clear that solutions to these two practical problems must be found before the basic security principle can be followed.

2. AST overflow bug. Before system 18.0 was installed, there was a requirement that whenever a segment is active, all directories superior to the segment must also be active. If a user created a directory tree deeper than the AST size, he could overflow the AST with unremovable entries. This would cause a system crash.

   Although this method of systematically crashing the system has now been fixed by 18.0, which does not require that superior directories be active, it illustrates another unfollowed security principle: table overflows and other unexpected (impossible) events must be handled gracefully without crashing the system, since the assumption that the overflow (or whatever) cannot be systematically produced by an attacker is hard to verify; worse, a system change elsewhere later may render the assumption incorrect.

3. Blank names bug. If a directory contained an entry for a segment with an all-blank name, deletion of that directory would cause a system crash. System 18.0 fixed this bug, which again was based on assumption that the user could not force an impossible condition to occur, so no recovery for the impossible condition was provided.

4. fs_get bug. Entry fs_get$ref name failed to initialize its error handler, so when it got an error return from kst_man (e.g., KST has overflowed) it attempted to reset a lock it never set, crashing the system. This one seems to be a simple programming error, since setting up the error handler fixed the trouble. Some technique of auditing which detects this class of bug is needed.

One other bug has been recently fixed, in system 17.11:

5. Argument validation bug. The software validation of arguments on cross-ring calls permitted pointers with indirect modifiers to be used, but it did not follow the indirect chain to see where it led. A user could supply an indirect argument pointer in a call to a supervisor entry which writes into an argument, and thrreby redirect the writing back into a supervisor database. This bug was fixed by changing the software validation to forbid indirect modifiers in argument pointers.

   This bug has some aspects similar to those of bug number 1, above, in that unexpected indirection can easily be overlooked.

   This bug would have been automatically fixed by the 6180 argument validation hardware, which will also automatically take care of about 30 other argument address validation troubles which have been uncovered by systematically auditing the supervisor entries.

A CENSUS OF RING 0

by Victor L. Voydock

Introduction

A major research area of the Computer Systems Research Group is to investigate the problem of producing a certifiable computer operating system. The first approach to this problem could have been to attempt to audit the Multics ring 0 supervisor as it then existed. That is, to read all of the programs which comprised the ring 0 supervisor and determine whether or not they did what they were supposed to do. It was clear that this was not a practical approach due to the size and complexity of ring 0 and the lack of a precise (or even imprecise) specification of its functions.

An approach which immediately suggested itself was to simplify ring 0 so that it could be audited. Before this could be done in any organized way it was necessary to have a clearer idea of what was in ring 0, so it was decided to take a census of ring 0. This document reports the results of that census.

Approaches

The census analyzes ring 0 from various points of view:

1. A notebook of ring 0 interfaces.
2. A functional breakdown of hcs_ entries.
3. A functional breakdown of all ring 0 segments.
4. A breakdown of all ring 0 segments by source language.

The notebook of interfaces describes every way that ring 0 can be entered by means of a call. It is a first (albeit crude) attempt to provide a functional specification of ring 0. It is available for study to anyone who is interested. The functional breakdown of hcs_ entries will be des-

cribed in a later RFC. The rest of this document deals with approaches 3 and 4.

Method of Census Taking

The information in Tables I-VI was gathered from the two directories which contain copies of all ring 0 object segments: >ldd>hard>bc and >ldd>hard>o. The information describes system 20.10a, a 6180 system installed on 8/15/73. The text section sizes were obtained from the object maps. The segment count indicates the number of separately translated pl1 and ALM segments. The entry point count includes segdefs, as well as standard entry points. Thus this count is slightly inaccurate since a few procedure segments (such as the FIM) have data segdefs imbedded in them. (There is no way to distinguish a data segdef from a procedure entry point segdef.) The translator names were obtained from the object segments using object_info_.

The functional categories (a complete list appears in Table II) are somewhat arbitrary. Any attempt to put labels on things is bound to distort reality somewhat. Comments on major classification flaws are welcome.

Most of the categories are self-explanatory. (Table VI has a list of all segments in each category.) Physical Storage Management consists of everything which is used to manage the physical storage of segments (core control, page control, bulk store control, etc.). Error Handling and Tracing contains all error handlers not local to one major category (e.g. syserr, verify_lock). Major categories are listed in Table I. Utility (Internal) contains utility segments which are not local to one major category (e.g. privileged_mode_ut). Utility (Shared with other rings) contains utility programs which are also used by rings other than zero (e.g. clock_, signal_, pl1_operators_). Obsolete contains segments which exist only for compatibility (either with other parts of the system or with user programs), and transfer vectors which can be thrown away when the appropriate procedures are converted to version 2 pl1. All obsolete segments can (eventually) be removed from ring 0 without affecting users.

General Observations

Finally, some general observations should be made.

First, ring 0 is much smaller than expected - about 157,000 words of text section (executable code and read only data). A large but not monstrous amount of code. For example, the bare bones of the pℓ1 compiler (parse, semantic translator and code generator) take up 118,000 words of text and this figure more than doubles if pℓ1 IO, the file manager and the pℓ1 runtime library are included. Why then is ring 0 so complex and hard to understand? Another measure of complexity is the number of distinct functional units - procedure entry points in pℓ1 terminology. Ring 0 contains 1201 entry points. (The bare bones pℓ1 compiler in contrast, contains 325 entry points.) A large number of entry points can be a symptom rather than a cause of complexity (when it is either) - reducing the number of entry points will not necessarily result in a simpler system. But, nevertheless, an investigation should be made to determine why there are so many entry points and to what extent they contribute to the complexity of Ring 0. This investigation might provide insight into how the system might be more simply organized.

The second observation is that the amount of assembly language generated code in ring 0 is larger than expected. 12.4% of non-obsolete ring 0 procedure text is ALM generated. If one views pℓ1_operators_ as an extension of every object segment and excludes it from the total, the figure drops to about 10%. This is still quite high. If, as a very rough estimate, one assumes an average of 5 words of text section per pℓ1 source statement, our results indicate (see Table IV) that ring 0 consists of about 29,000 lines of pℓ1 source and about 15,000 lines of ALM source.

Fortunately, the amount of ALM can probably be reduced significantly. All 64 non-obsolete ALM procedure segments in ring 0 (see Table V), have less than 2000 words of text section each and all but 9 have less than 400 words of text section each. A cursory study has uncovered 13 segments which can be immediately converted to pℓ1 with no loss of system efficiency and additional study will undoubtably uncover others. Dave Reed is currently investigating this area.

Finally, Tables I, II and VI suggest a number of areas in which simplification might yield a significant reduction in the size of ring 0:

initialization - One of the oldest parts of the system. Can probably be reorganized and simplified

salvager - Its size indicates that either it is a collection of ad hoc methods or that the system data bases are not well organized with respect to salvagability.

tty dim and ARPA network - Duplicate functions should be merged. An investigation should also be made into why the ttydim is so large.

interrupt handling - Rich Feiertag's work on simplifying the way interrupts are handled should greatly reduce the complexity, if not the size of the IO system and of Physical Storage Management.

linker, search rules - Phil Jansen's work on removing the linker from ring 0 will remove a complicated function from ring 0 but will not greatly reduce the size of ring 0 (about 3%).

A Final Comment

Through the use of binding, the actual number of free standing procedure segments in ring 0 is 50 (instead of 305), and the number of accessible entry points is 909 (instead of 1201). A more judicious choice of binding might further reduce the number of accessible entires. Some accessible entries implement primitives used by outer rings and some functional areas span more than one segment. Nevertheless, the number of accessible entires is a rough measure of the connectivity of the various functional areas of ring 0. A study of the interrelations of the 50 free standing procedure segments may lead to insights into the overall structure of ring 0.

Table I: Breakdown by Major Categories
(System 20.10a)

Category	% of total	Words of text section	Number of segments	Number of entries
File System/Virtual Memory	36.7	57727	93	476
Initialization/Reconfiguation/ Shutdown	15.4	24312	56	102
IO System	15.	23602	33	117
ARPA Network	12.1	19143	34	158
Utility	9.	14269	38	122
Obsolete	5.3	8400	16	71
Process Management	5.	7809	26	95
Interrupt/Fault Dispatching	1.2	1966	8	59
Other (Put in ring 0 for no good reason.)	.2	353	1	1
Total		157581	305	1201
Total (minus obsolete)		149181	289	1130

Table II: More Detailed Breakdown
(System 20.10a)

Category	Words of text section	Number of segments	Number of entries
I. File System/Virtual Memory	57727	93	476
A. File System	18111	24	125
B. Salvager	11840	15	41
C. Linker/Search Rules/ Working Directory	4572	11	30
D. Segment Control	7069	13	29
E. Physical Storage Management	11719	21	209
F. Other (Things which overlap categories)	4416	9	42
II. Initialization/Reconfiguration/Shutdown	24312	56	102
A. Initialization/Shutdown	19501	46	81
B. Reconfiguration	3207	4	7
C. Other (Things which overlap categories)	1604	6	14
III. IO System	23602	33	117
A. IOM/355	4533	13	38
B. Typewriter Control	11558	7	25
C. IOAM	2963	6	31
D. Printer Control	2247	4	9
E. Tape Control	2301	3	14
IV. ARPA Network	19143	38	158
V. Utility	14269	37	122
A. Error Handling and Tracing	3431	11	28
B. Utility (Internal)	1923	7	41
C. Utility (Shared with other rings)	8915	20	53
VI. Obsolete	8400	17	71
VII. Process Management	7809	26	95
A. Process Creation/Status/ Destruction	4655	19	32
B. Inter-Process Communication	1838	2	11
C. Traffic Control	1943	2	40
D. Timers/ips masking	375	3	12
VIII. Interrupt/Fault Dispatching	1966	8	59

Table III: Breakdown by Bound Segment

(System 20.10a)

Bound Segment Name	Words of text section	Words of linkage section	Number of entries
bound_sss_wired	1040	48	13
bound_active_1	1136	96	12
bound_error_active	1232	104	6
bound_error_wired	1104	96	14
bound_file_system	22984	696	110
bound_gim_active	2208	112	14
bound_init_1	2272	304	14
bound_init_2	3264	200	7
bound_io_init	2248	144	5
bound_iom_active	6566	216	40
bound_iom_imp_dim_	7364	376	58
bound_iom_imp_status	4320	320	25
bound_iom_wired	6920	440	29
bound_mseg_prim	1024	88	7
bound_network0_	8192	720	26
bound_page_control	9552	320	73
bound_process_creation	7320	416	27
bound_salvager	11432	344	35
bound_sss_active_	4632	88	43
bound_sss_wired_	3336	64	18
bound_system_faults	19128	616	111
bound_tc_wired	1432	168	16
bound_temp_1	6928	272	17
bound_temp_2	648	104	5
bound_tty_active	7600	136	21

Table IV: Breakdown by Language
(System 20.10a)

Category	% of ALM	Words of Text ALM	Words of Text PL/I	Number of Segments ALM	Number of Segments PL/I
Interrupt/Fault Dispatching	70.2	1381	585	7	1
Utility	41.4	5907	8362	15	23
Obsolete	35.5	2989	5411	9	7
Process Management	23.6	1842	5967	4	22
Initialization/Configuration/Shutdown	14.	3406	20906	10	46
File System/Virtual Memory	7.4	4273	53454	19	74
IO System	6.9	1628	21974	8	25
ARPA Network	.5	92	19051	1	33
Other	0.	0	353	0	1
Total	13.6	21488	136093	73	232
Total (minus obsolete)	12.4	18529	130652	64	225
Total (minus obsolete and p&l_operators)	10.1	14711	130652	63	225

Table V: List of ALM Procedure Segments by Category

Category	Language	Words of text	Words of linkage	Number of entry points	Segment Name
1-SI	alm	116	56	0	bootstrap2
1-SI	alm	1712	8	0	bootstrap1
1-SI	alm	242	8	6	slt_manager
1-SI	alm	262	8	1	pre_link_2
1-SI	alm	272	8	1	pre_link_1
1-SI	alm	30	22	1	build_template_pds
1-SI	alm	38	10	1	shutdown_switch
1-SI	alm	382	36	4	tape_reader
1-SI	alm	64	14	3	privileged_mode_init
1-SI,RC	alm	288	76	5	init_processor
2-ID	alm	220	32	4	signaller
2-ID	alm	240	90	21	wired_fim
2-ID	alm	272	18	1	fault_error
2-ID	alm	28	8	3	parity_check
2-ID	alm	297	102	15	ii
2-ID	alm	320	74	9	fim
2-ID	alm	4	8	2	return_to_ring_0_
3-FS,SC,S	alm	58	8	2	hash_index
3-L	alm	172	14	2	get_defptr
3-L	alm	62	8	1	datmk_util_
3-L	alm	96	8	3	lot_maintainer
3-S	alm	154	20	6	salv_free_store
3-SC	alm	46	10	2	kst_man
3-SC,SSM	alm	80	12	5	get_ptrs_
3-SSM	alm	104	60	26	page
3-SSM	alm	1300	142	21	page_fault
3-SSM	alm	136	72	6	device_control
3-SSM	alm	142	36	7	free_store
3-SSM	alm	218	52	5	bulk_store_control
3-SSM	alm	220	36	14	pc_trace
3-SSM	alm	220	56	20	master_pxss_page
3-SSM	alm	234	42	2	pre_page
3-SSM	alm	336	36	15	pd_util
3-SSM	alm	52	16	7	meter_disk
3-SSM	alm	563	12	19	page_error
3-SSM	alm	80	24	5	page_util
4-PC	alm	34	16	2	level
4-PC	alm	6	8	1	gate_init
4-T	alm	28	18	3	vclock
4-TC	alm	1774	196	39	pxss
5-I	alm	12	12	1	ioam_check
5-I	alm	38	8	1	call_detacher
5-IOC	alm	22	8	4	dn355_util
5-IOC	alm	511	24	9	iom_manager
5-IOC	alm	8	10	1	dstint
5-P	alm	430	8	1	prt_300_conv
5-P	alm	587	10	1	prt_ccnv
5-TP	alm	20	8	1	tape_checksum_

Table V - page 2

Category	Language	Words of text	Words of linkage	Number of entry points	Segment Name
6-E	alm	106	56	3	emergency_shutdown
6-E	alm	18	10	1	check_trailer
6-E	alm	24	16	1	syserr
6-UI	alm	138	34	3	wire_stack
6-UI	alm	22	8	1	fm_checksum_
6-UI	alm	26	16	3	get_proc_id
6-UI	alm	501	74	18	privileged_mode_ut
6-UI	alm	61	16	1	absadr
6-US	alm	10	12	1	clock_
6-US	alm	14	8	2	unwinder_util_
6-US	alm	18	8	3	all_rings_util_
6-US	alm	206	8	6	condition_
6-US	alm	28	10	2	wired_utility_
6-US	alm	3818	42	5	pl1_operators_
6-US	alm	917	8	2	formline_
7-N	alm	92	8	1	imp_status_driver
8-O	alm	113	8	1	old_freen_
8-O	alm	12	16	4	fast_hc_ipc_tv
8-O	alm	143	10	1	old_alloc_
8-O	alm	2574	14	13	pl1_operators
8-O	alm	30	8	1	move_
8-O	alm	50	54	23	sss_active_tv_
8-O	alm	53	8	2	old_area_
8-O	alm	6	10	1	tty_read_tv
8-O	alm	8	12	2	tty_write_tv
8-O	pl1	220	32	2	accept_alm_obj

Note: see Table VI for an explanation of category abbreviations.

Table VI: List of Ring 0 Segments by Category
(System 20.10a)

The following category abbreviations are used:

1. Initialization/Reconfiguration/Shutdown
   - RC - Reconfiguration
   - SI - Shutdown
2. IH - Interrupt/Fault Dispatching
3. File System/Virtual Memory
   - FS - File System
   - L - Linker/Search Rules/Working Directory
   - S - Salvager
   - SC - Segment Control
   - SSM - Physical Storage Management
4. Process Management
   - PC - Process Creation/Status/Destruction
   - IPC - Inter-Process Communication
   - T - Timers/ips masking
   - TC - Traffic Control
5. IO System
   - I - IOAM
   - IOC - IOM/355
   - P - Printer Control
   - TP - Tape Control
   - TT - Typewriter Control
6. Utility
   - E - Error Handling and Tracing
   - UI - Utility (Internal)
   - US - Utility (Shared with other rings)
7. N - ARPA Network
8. O - Obsolete

Multiple tags indicate segments which fall in multiple categories, e.g. a tag of FS,S indicates a segment used both by the File System and the Salvager.

Category	Language	Text Size (words)	Linkage Size (words)	Number of entries	Segment Name
1-RC	pl1	266	32	2	dsu270_reconfig
1-RC	v2pl1	2032	86	1	reconfig
1-RC	v2pl1	291	34	2	add_memory
1-RC	v2pl1	518	38	2	delete_pd_records
1-SI	alm	116	56	1	bootstrap2
1-SI	alm	1712	8	1	bootstrap1
1-SI	alm	242	8	3	sit_manager
1-SI	alm	262	8	1	pre_link_2
1-SI	alm	272	8	1	pre_link_1
1-SI	alm	30	22	1	build_template_pds
1-SI	alm	38	10	1	shutdown_switch
1-SI	alm	382	36	4	tape_reader
1-SI	alm	64	14	3	privileged_mode_init
1-SI	pl1	1199	48	1	initialize_dims
1-SI	pl1	189	38	1	syserr_init
1-SI	pl1	192	44	4	delete_segs
1-SI	pl1	354	62	2	shutdown
1-SI	pl1	397	36	1	load_system
1-SI	pl1	469	112	1	tc_init
1-SI	pl1	470	42	1	segment_loader
1-SI	pl1	53	26	1	clock_init
1-SI	pl1	68	26	1	build_template_dsegs
1-SI	pl1	73	36	1	initializer
1-SI	pl1	744	68	2	update_sst_pl1
1-SI	pl1	383	206	4	initialize_faults
1-SI	pl1	92	22	1	find_peripheral
1-SI	pl1	98	34	1	tc_shutdown
1-SI	v2pl1	1057	72	1	scs_init
1-SI	v2pl1	137	32	1	init_hardcore_gates
1-SI	v2pl1	1526	34	1	tty_init
1-SI	v2pl1	163	38	1	init_sys_var
1-SI	v2pl1	164	26	1	initialize_gim
1-SI	v2pl1	1701	76	2	init_branches
1-SI	v2pl1	187	66	2	init_collections
1-SI	v2pl1	223	36	1	init_root_dir
1-SI	v2pl1	252	26	1	dn355_init
1-SI	v2pl1	27	18	1	io_init
1-SI	v2pl1	300	44	2	wired_shutdown
1-SI	v2pl1	325	22	2	make_sdw
1-SI	v2pl1	364	38	2	trace_init
1-SI	v2pl1	382	36	11	tape_io
1-SI	v2pl1	436	102	1	tape_init
1-SI	v2pl1	49	14	1	init_str_seg
1-SI	v2pl1	527	24	1	iom_data_init
1-SI	v2pl1	658	28	2	make_branches
1-SI	v2pl1	71	12	1	bulk_store_init
1-SI	v2pl1	751	30	1	init_sst
1-SI	v2pl1	757	40	1	scas_init
1-SI	v2pl1	89	20	1	printer_init
1-SI	v2pl1	956	40	4	dsu190_init
1-SI,RC	alm	288	76	5	init_processor
1-SI,RC	pl1	456	82	3	stop_cpu
1-SI,RC	pl1	54	22	1	find
1-SI,RC	v2pl1	120	26	1	prds_init

Table VI - page 3

Category	Language	Text Size (words)	Linkage Size (words)	Number of entries	Segment Name
1-SI,RC	v2pl1	153	22	2	freecore
1-SI,RC	v2pl1	533	84	2	start_cpu
2-ID	alm	220	32	*	signaller
2-ID	alm	240	90	21	wired_fim
2-ID	alm	272	18	1	fault_error
2-ID	alm	28	8	3	parity_check
2-ID	alm	297	102	15	ii
2-ID	alm	320	74	9	fim
2-ID	alm	*	8	2	return_to_ring_0_
2-ID	v2pl1	585	34	4	parity_fault
3-FS	pl1	1056	56	5	acl_
3-FS	pl1	163	28	1	check_gate_acl_
3-FS	pl1	1764	150	5	append
3-FS	pl1	222	38	3	ring0_init
3-FS	pl1	265	30	2	acc_list_
3-FS	pl1	275	22	1	match_star_
3-FS	pl1	282	52	1	force_access
3-FS	pl1	337	40	*	quotaw
3-FS	pl1	35	26	2	quota_util
3-FS	pl1	355	40	5	fs_alloc
3-FS	pl1	549	72	5	ringbr_
3-FS	pl1	650	78	2	del_dir_tree
3-FS	pl1	862	128	7	find_
3-FS	v2pl1	1056	56	3	star_
3-FS	v2pl1	1160	94	3	delentry
3-FS	v2pl1	1232	78	15	set
3-FS	v2pl1	1484	80	11	quota
3-FS	v2pl1	1662	70	13	status_
3-FS	v2pl1	212	20	2	make_seg
3-FS	v2pl1	2437	104	17	asd_
3-FS	v2pl1	437	34	3	level_0_
3-FS	v2pl1	491	34	3	fs_move
3-FS	v2pl1	559	64	3	chname
3-FS	v2pl1	566	62	3	truncate
3-FS,S	pl1	1087	52	5	acc_name_
3-FS,SC	pl1	197	30	1	move_file_map
3-FS,SC	pl1	304	56	*	dir_control_error
3-FS,SC	pl1	337	52	2	access_mode
3-FS,SC	pl1	485	78	5	sum
3-FS,SC,S	alm	58	8	2	hash_index
3-FS,SC,S	pl1	572	56	4	hash
3-L	alm	172	14	2	get_defptr
3-L	alm	62	8	1	datmk_util_
3-L	alm	96	8	3	lot_maintainer
3-L	pl1	134	24	1	get_defname
3-L	v2pl1	1036	68	*	link_snap
3-L	v2pl1	125	20	2	unsnap_service
3-L	v2pl1	234	28	1	rest_of_datmk_
3-L	v2pl1	313	30	1	get_defname_
3-L	v2pl1	532	36	1	initiate_search_rules
3-L	v2pl1	770	54	7	fs_search
3-L	v2pl1	998	66	7	link_man
3-S	alm	154	20	5	salv_free_store
3-S	pl1	1087	40	2	salv_check_thread
3-S	pl1	1087	80	2	salv_check_map
3-S	pl1	1207	82	1	salv_rebuild_directory
3-S	pl1	1408	60	1	salvage_entry
3-S	pl1	194	32	1	salv_clean_ast
3-S	pl1	1979	100	3	salvage_directory

Table VI - page 4

Category	Language	Text Size (words)	Linkage Size (words)	Number of entries	Segment Name
3-S	pl1	247	48	3	salv_truncate
3-S	pl1	369	58	5	salv_name
3-S	pl1	372	54	1	salv_delete_dir
3-S	pl1	421	48	3	salv_print
3-S	pl1	516	54	5	salv_check_ptr
3-S	pl1	597	38	1	salv_rebuild_names
3-S	pl1	761	48	4	salv_rebuild_acl
3-S	v2pl1	1441	88	2	on_line_salvager
3-SC	alm	46	10	2	kst_man
3-SC	pl1	115	32	1	kst_entry_check
3-SC	pl1	373	30	1	activate
3-SC	pl1	436	46	2	setfaults
3-SC	pl1	440	34	2	kstsrch
3-SC	pl1	549	40	2	updatep
3-SC	v2pl1	1044	76	6	makeunknown
3-SC	v2pl1	586	64	2	pagefault
3-SC	v2pl1	652	60	3	deactivate
3-SC	v2pl1	687	62	1	seg_fault
3-SC	v2pl1	689	56	4	initiate
3-SC	v2pl1	720	54	1	get_aste
3-SC	v2pl1	732	40	2	makeknown
3-SC,L	v2pl1	1296	66	14	fs_get
3-SC,SSM	alm	80	12	3	get_ptrs_
3-SSM	alm	104	60	26	page
3-SSM	alm	1300	142	21	page_fault
3-SSM	alm	136	72	5	device_control
3-SSM	alm	142	36	7	free_store
3-SSM	alm	218	52	5	bulk_store_control
3-SSM	alm	220	36	14	pc_trace
3-SSM	alm	220	56	20	master_pxss_page
3-SSM	alm	234	42	2	pre_page
3-SSM	alm	336	36	15	pd_util
3-SSM	alm	52	16	7	meter_disk
3-SSM	alm	563	12	19	page_error
3-SSM	alm	80	24	5	page_util
3-SSM	pl1	123	32	1	assign_device
3-SSM	pl1	290	60	3	get_disk_meters
3-SSM	pl1	388	58	1	move_device
3-SSM	pl1	420	58	7	pc_wired
3-SSM	pl1	497	52	4	wire_proc
3-SSM	pl1	742	78	15	pc_trace_pl1
3-SSM	v2pl1	1548	50	4	pc_abs
3-SSM	v2pl1	1847	52	16	dsu190_control
3-SSM	v2pl1	2259	82	11	pc
4-IPC	pl1	368	60	5	fast_hc_ipc
4-IPC	v2pl1	468	54	6	hc_ipc
4-PC	alm	34	16	2	level
4-PC	alm	6	8	1	gate_init
4-PC	pl1	132	42	3	pim
4-PC	pl1	161	48	2	init_proc
4-PC	pl1	24	24	1	stop_process
4-PC	pl1	241	48	1	activate_segs
4-PC	pl1	261	58	3	deact_proc
4-PC	pl1	283	48	1	deactivate_segs
4-PC	pl1	371	80	1	terminate_proc
4-PC	pl1	485	38	1	makestack
4-PC	pl1	70	40	3	proc_info
4-PC	pl1	90	26	2	access_viol
4-PC	v2pl1	1250	88	3	act_proc

Table VI - page 5

Category	Language	Text Size (words)	Linkage Size (words)	Number of entries	Segment Name
4-PC	v2pl1	134	26	1	proc_int_handler
4-PC	v2pl1	175	26	1	outward_call_handler
4-PC	v2pl1	175	38	3	ring_alarm
4-PC	v2pl1	21	12	1	get_page_trace
4-PC	v2pl1	067	48	1	initialize_kst
4-PC	v2pl1	75	20	1	get_process_usage
4-T	alm	28	18	3	vclock
4-T	v2pl1	258	30	3	set_alarm_timer
4-T	v2pl1	89	20	3	ips_
4-TC	alm	1774	198	39	pxss
4-TC	pl1	169	30	1	wired_olm
5-I	alm	12	12	1	ioam_check
5-I	alm	38	8	1	call_detacher
5-I	pl1	161	34	3	ioam_ut1
5-I	pl1	198	56	7	dstm_
5-I	pl1	098	52	6	ioam_ut
5-I	v2pl1	1856	76	13	ioam_
5-IOC	alm	22	8	4	dn355_util
5-IOC	alm	511	24	3	iom_manager
5-IOC	alm	8	10	1	dstint
5-IOC	v2pl1	144	18	1	gim4
5-IOC	v2pl1	1616	56	3	dn355
5-IOC	v2pl1	173	18	1	gloc_stat
5-IOC	v2pl1	210	30	2	gim_alloc
5-IOC	v2pl1	32	14	1	channel
5-IOC	v2pl1	388	32	1	gim3
5-IOC	v2pl1	393	32	1	gim1
5-IOC	v2pl1	64	18	1	check
5-IOC	v2pl1	878	72	6	gim_assign
5-IOC	v2pl1	34	18	1	gim2
5-P	alm	430	8	1	prt_300_conv
5-P	alm	587	10	1	prt_conv
5-P	v2pl1	242	20	1	printer_status
5-P	v2pl1	988	86	6	printer_dcm
5-TP	alm	20	8	1	tape_checksum_
5-TP	v2pl1	1792	84	11	tdcm
5-TP	v2pl1	489	36	2	tdcm_status
5-TT	pl1	118	28	1	tty_unlock
5-TT	pl1	4153	285	7	tty_inter
5-TT	pl1	479	20	1	tty_con
5-TT	pl1	676	48	3	tty_free
5-TT	v2pl1	1883	30	1	tty_read
5-TT	v2pl1	2103	30	2	tty_write
5-TT	v2pl1	2146	60	8	tty_index
6-E	alm	106	56	3	emergency_shutdown
6-E	alm	18	10	1	check_trailer
6-E	alm	24	16	1	syserr
6-E	pl1	108	28	2	debug_check
6-E	pl1	21	24	1	call_pds
6-E	pl1	254	36	3	ring_0_peek
6-E	v2pl1	1030	52	3	copy_fdump
6-E	v2pl1	19	16	1	ring_zero_cleanup
6-E	v2pl1	253	22	1	verify_lock
6-E	v2pl1	660	124	3	trace
6-E	v2pl1	338	74	3	syserr_real
6-UI	alm	138	34	3	wire_stack
6-UI	alm	22	8	1	fm_checksum_
6-UI	alm	25	16	3	get_proc_id
6-UI	alm	501	74	18	privileged_mode_ut

Category	Language	Text Size (words)	Linkage Size (words)	Number of entries	Segment Name
6-UI	alm	61	16	1	absadr
6-UI	pl1	183	32	*	thread
6-UI	v2pl1	992	52	11	lock
6-US	alm	10	12	1	clock_
6-US	alm	14	8	2	unwinder_util_
6-US	alm	18	8	3	all_rings_util_
6-US	alm	206	8	3	condition_
6-US	alm	28	10	2	wired_utility_
6-US	alm	3818	42	3	pl1_operators_
6-US	alm	317	8	2	formline_
6-US	pl1	121	28	3	cv_bin_
6-US	pl1	139	32	3	unique_chars_
6-US	pl1	243	32	*	cv_dec_
6-US	pl1	49	26	2	unique_bits_
6-US	pl1	585	50	*	date_time_
6-US	pl1	338	40	*	object_info_
6-US	v2pl1	136	18	1	area_assign_
6-US	v2pl1	202	12	1	freen_
6-US	v2pl1	355	20	2	alloc_
6-US	v2pl1	365	16	2	area_
6-US	v2pl1	627	18	*	signal_
6-US	v2pl1	84	20	2	try_to_unlock_lock
7-N	alm	32	8	1	imp_status_driver
7-N	pl1	101	34	2	imp_get_buffer
7-N	pl1	103	24	1	imp_global_status
7-N	pl1	118	32	1	iom_imp_dcm_read
7-N	pl1	1238	132	14	iom_imp_dcm_init
7-N	pl1	1288	144	20	ncp_main_
7-N	pl1	146	24	2	imp_thread
7-N	pl1	1769	178	3	ncp_
7-N	pl1	182	44	3	imp_wakeup
7-N	pl1	193	30	1	iom_imp_dcm_write
7-N	pl1	194	38	3	imp_util_wired
7-N	pl1	194	44	3	imp_util
7-N	pl1	202	30	1	imp_write_service
7-N	pl1	211	56	3	imp_service
7-N	pl1	222	40	3	ncp_ring_
7-N	pl1	2429	244	3	iom_imp_status
7-N	pl1	255	60	3	imp_misc
7-N	pl1	2713	412	*	ncp_loop_
7-N	pl1	274	38	3	imp_get_wired_buffer
7-N	pl1	277	36	1	imp_global_queue
7-N	pl1	299	42	*	imp_mark_host
7-N	pl1	309	50	1	imp_read
7-N	pl1	317	58	3	imp_lock
7-N	pl1	32	26	1	imp_cleanup
7-N	pl1	406	56	2	imp_write
7-N	pl1	531	52	1	imp_input_processor
7-N	pl1	549	98	3	imp_init
7-N	pl1	612	132	21	imp_error
7-N	pl1	657	56	*	imp_order
7-N	pl1	726	78	*	ncp_util_
7-N	pl1	741	158	*	ncp_status_
7-N	pl1	773	72	3	imp_input_processor_int
7-N	pl1	887	88	7	imp_attach
7-N	pl1	37	30	1	imp_release_wired_buffer

Table VI - page 7

Category	Language	Text Size (words)	Linkage Size (words)	Number of entries	Segment Name
b-0	alm	113	b	1	old_treen_
o-0	alm	12	16	4	fast_hc_ipc_tv
o-0	alm	143	10	1	old_alloc_
o-0	alm	2574	14	13	pll_operators
o-0	alm	50	o	1	move_
o-0	alm	50	54	23	sss_active_tv_
o-0	alm	53	o	2	old_area_
o-0	alm	o	10	1	tty_read_tv
o-0	alm	o	12	2	tty_write_tv
o-0	pl1	102	42	o	usercode
o-0	pl1	1167	40	1	dc_pack
o-0	pl1	420	32	2	accept_alm_obj
o-0	pl1	255	42	1	list_dir
o-0	pl1	283	40	1	status
o-0	v2pl1	122	14	1	get_entry_name
o-0	v2pl1	1536	74	5	ex_acl
o-0	v2pl1	1704	oo	o	acl
Other	v2pl1	333	24	1	date_name_

SOME MULTICS SECURITY HOLES WHICH WERE CLOSED BY 6180 HARDWARE
by J. H. Saltzer, Phillippe Janson, and Douglas Hunt

This note is the second of a series* which describes design and implementation errors in Multics which affect its ability to protect information and provide service. The purpose of the series is to try to discuss what incorrectly laid groundwork permitted each trouble to creep in.

It is interesting (and comforting) to note that no security problem yet discovered has required any change in the original overall design of Multics; the problems have universally been at the level of detailed design errors or implementation slipups; the repairs have been conceptually simple readjustments to bring the design or implementation back to the originally intended one.

A fairly large number of security problems were fixed automatically by conversion from the Honeywell 645 to the Honeywell 6180, which has built-in argument validation hardware. As will be seen, replacement of a complex software package with a relatively simple hardware mechanism was remarkably effective, suggesting that it was a move in the right direction.

Unvalidated Gates

In the 645, the following gates to ring zero had no validation of arguments at all:

absentee_test_	(all entries)
hphcs_	(all entries)
phcs_	(all entries)
phnxhcs_	(all entries)
admin_gate_$guaranteed_eligibility_off	
admin_gate_$guaranteed_eligibility_on	

Argument validation consists of checking each argument to a gate entry to be sure it refers to an address to which the caller is permitted access. For example, if the ring zero program intends to write into the argument (e.g., an output value) then the caller of the entry should specify an address in which he is permitted to write. Failure to perform argument validation would mean that the caller could specify an address somewhere inside ring zero; if he did, the ring zero program could be used for unauthorized patching of the supervisor. It is slightly harder but still possible to exploit a gate which only reads its arguments.

* Previously issued memo in the series: see page 1 of this memo.

The unvalidated gates had one thing in common: they were all controlled by access control lists which limit their use to supposedly responsible individuals. This control was probably the chief rationalization for not putting in the extra effort required to specify the argument validation.

On the 6180, all arguments are automatically validated by hardware checks on the ring of origin of every argument. This approach eliminates both the extra (and sometimes neglected) effort needed to specify validation, and also any possibility of errors in that specification.

Incorrectly validated arguments

In the following entries, some argument was validated with more leniency than appropriate, permitting the user, typically, to cause the supervisor to write into an area in which the user has no access.

hcs_$get_seg_count	last argument unvalidated.
hcs_$get_entry_name	argument validated for wrong type.
hcs_$get_dbrs	argument validated for wrong usage.
hcs_$assign_channel	1st argument validated for wrong usage.
hcs_$check_device	2nd argument validated for wrong usage.
hcs_$get_search rule	argument validated for wrong usage.
hcs_$get_count_linkage	2nd argument validated for wrong usage.
hcs_$ipc_init	argument valudated for wrong usage.
hcs_$list_dir	2nd argument validated for wrong usage.
hcs_$make_ptr	1st argument validated for wrong usage.
hcs_$list_dir_acl	3rd argument validated for wrong usage.
hcs_$set_dtd	3rd argument validated for wrong usage.
hcs_$status	entire argument spec is wrong.
imp_dim_gate_$imp_read_order	3rd argument validated for wrong usage.
imp_dim_gate_$imp_write_order	3rd argument validated for wrong usage.
netp_$ncp_priv_status	3rd argument validated for wrong usage.
netp_$ncp_priv_order	3rd argument validated for wrong usage.
net_$ncp_status	3rd argument validated for wrong usage.
net_$ncp_order	3rd argument validated for wrong usage.
hcs_$acl_list	5th argument validated for wrong usage.

This list represents the accumulation of errors over several years of specifying argument validation for about 150 user-callable gates. When an argument is validated for "wrong usage" it typically means that the gate specification says that the gate only reads the argument, when the gate actually writes into it. Thus, the validator checks only to make sure that the user can read data at the specified address. If the user provides a pointer, say, to some location in the "sys_info" segment, in which he has read-only permission, the gate, which can write into "sys_info" by virtue of its ring-zero location, would then overwrite some item there.

Again, the value of the automatic hardware argument validation feature of the 6180 is clear: the opportunity for an incorrect software-declared specification is completely eliminated.

Unvalidatable arguments

In the following entries, some entry could not be checked by the automatic validator, since the correct method of validation depends on the value of some other argument.

hcs_$acl_list	3rd argument used as both input and output.
hcs_$ex_acl_list	3rd argument used as both input and output.
hcs_$ex_acl_delete	3rd argument meaning depends on 4th argument.
hcs_$initiate_seg_count	6th argument meaning depends on another argument.
hcs_$list_dir_acl	4,5th arguments meaning depend on the value of 3rd argument.
hcs_$replace_sall	3rd argument unvalidatable.
hcs_$replace_dall	3rd argument unvalidatable.

The problem in each case here was deeper than in the previous one: the particular choice of arguments lead to impossiblity of validation, and therefore to no validation at all. For example, suppose that the third argument is an input argument for some values of the first argument, but is an output value for others. Then a protection specification which says that the third argument must be writable would cause some correct programs, which intentionally provided a read-only third argument, to be declared illegal. If, when these entries were first introduced, their documentation had specified that the argument in question must be writable whether or not it is actually written into by the supervisor, then the trouble could have been avoided (at the cost of an additional obscurity in the user interface). Unfortunately, an after-the-fact change to require writeability might cause some correct user programs to stop working, so compatibility prevents correction.

Again, the automatic argument validation hardware of the 6180 provides a solution. Since every reference to an argument is separately checked, only if the argument is actually used as an output argument will it be checked for writeability.

EPL argument validation trap

The argument validator did not completely check out some of the more complex specifiers of arguments provided by EPL (the first Multics PL/I compiler) programs. Thus, a user could construct an argument descriptor which indicated that an EPL specifier was in use, and thereby induce the argument validator to allow the call to go unchecked. This problem was basically one of historical compatibility: the EPL specifier format and organization was designed before the implications of argument validation had been considered. When it became clear that certain argument types were hopelessly complex to validate, an attempt was made to prohibit (by edict) the use of those types of arguments in supervisor entries. After the later PL/I compiler eliminated the need for a restriction, some gates were installed which utilized the forbidden argument types. The argument validator, unfortunately, provided a default of "acceptable" for EPL arguments of unvalidatable type, so it turned out that one could call the new entries with programs written in EPL, which was still an available compiler. The alternatives of changing the default to "unacceptable" would have effectively denied access to the new gates for those users not yet ready to rely upon a new unseasoned, PL/I compiler. Thus, through a series of design slipups, errors in judgement, and bad practices, this protection bypass got into the system.

The 6180 argument validation hardware again automatically performs the appropriate access checking at argument usage time, independent of the format of the structure passed as an argument.

ECT terminate bug

The design of the Inter Process Communication (IPC) event channel table (ECT) had the following flaw: when the user-ring IPC created an ECT, it then called a ring-zero entry to inform the ring-zero part of IPC of the location of the ECT. The pointer in question was stored by the ring-zero part of IPC in a ring-zero data base, for future use in passing IPC messages back to the user. The user could now terminate the segment containing the ECT, and initiate some other segment (to which he had only read access in the user ring) with the same segment number as the former ECT. Then, the ring zero part of the IPC, using its stored pointer, would write the user's messages in a place the user had no business writing into.

With the 6180 hardware, the pointer passed by the user to the ring-zero part of the IPC facility, and stored there, contains the ring number of the user's ring. Thus all reference made by ring-zero IPC using that pointer will be validated as though they came from the user ring. If a segment for which the user did not have write access is substituted, the attempt of the ring-zero procedure to write in it will fail.

Exploitation of user-ring master-mode procedures

The 645 processor had a "master-mode" property, which bypassed all protection checks; certain procedures such as the fault interceptor and signaller had to operate in master-mode, yet in the ring of the user causing the fault or receiving the signal. To prevent exploitation, the hardware permitted calls to a master-mode procedure only to an entry point at location zero in the segment; the procedure was expected to very carefully examine the circumstances of its entry to insure that it was not being exploited.

Upon review of the standard entry sequence code actually being used, it was discovered that the design did not prevent exploitation at all. Three distinct problems were found, each of which could be exploited in several ways. First, the entry sequence was designed on the assumption that index register one had been set to indicate which of several actual entry points to the segment was desired. The entry sequence correctly assumed that the caller might place an out-of-bounds value in index register one, so it checked to make sure that the value was within reasonable limits. Unfortunately, if the value was out of bounds, it called out to the system trouble-handling procedure, which proceeded to "crash" the system. Thus, any user could cause a crash by transferring to location zero of the signaller, with an appropriate value in index register one. The second problem is that the call to the system trouble handler was done by an indirect transfer out through the linkage section of the master-mode procedure -- but this call occurred before verification that the linkage pointer had been set to the currect value. Thus, the user could plant a special value in the linkage pointer, transfer to location zero of the signaller, and cause the master-mode procedure to transfer anywhere he wished -- including into the middle of another master-mode procedure. Again, by preparing registers in advance, and choosing

carefully the code sequence to transfer into, one could develop an exploitation. Finally, the third problem is that safe-storing of the processor registers was done assuming that the register value in the stack base register did not need to be checked, since it was locked. Unfortunately, a 1971 modification to the system resulted in the stack base register being unlocked, so the user could, by loading the stack base register and transferring to a legal entry point of the signaller, cause it to safe-store the processor register almost anywhere.

Although the concept of securing a master-mode procedure still seems viable, the implementation is apparently very fussy. By checking the Multics System Programmers' Manual it can be established that the first two problems have existed at least since 1967, and probably earlier. It was precisely because of uneasiness about the securing of master-mode segments that the 6180 was designed without a master-mode, and with consistent and builtin hardware call and fault facilities.

Execute instruction user special protection checks

On the 645 processor, the checking of permission was special cased when an "execute" instruction was encountered, since the time of decoding of the instruction to be executed is delayed to a time when most instructions are in the midst of execution.

Apparently as a result of a field change, one of the special cased checks was accidentally disabled if the execute instruction was located in an odd location and it addressed an offset of zero in another segment. In this situation, write permission was not checked, so one could write into a read-only segment.

Here we have an example of the danger of special cases -- they tend to cover rare occurrences, which means that routine operation does not exercise them. It also points out the recertification problem: even if a design is originally sound, every later modification should be accompanied with a recertification.

SOME RECENTLY REPAIRED SECURITY HOLES OF MULTICS

by J. H. Saltzer and D. Hunt

This note is the third of a series* which describes design and implementation errors in Multics which affect its ability to protect information and provide service. The purpose of the series is to try to discuss what incorrectly laid groundwork permitted each trouble to creep in.

It is interesting (and comforting) to note that no security problem yet discovered has required any change in the original overall design of Multics; the problems have universally been at the level of detailed design errors or implementation slipups; the repairs have been conceptually simple readjustments to bring the design or implementation back to the originally intended one.

Reused address

Following a system crash, the salvager may discover that a single disk or drum page is being used by two or more page tables, a situation which should never occur intentionally, but may appear if a crash occurs while updating a page table value. In the original design, the page in question was awarded to the first page table encountered by the salvager, and later users of that page were assigned new pages containing zeroes. Since there is no way to tell which of the multiple users was the legitimate one, the present, safer design gives all users of a reused page distinct pages of zeroes. This improved design helps reduce the chance of one user seeing another user's data because of a system crash. Ideally, one would make the storage space which holds a page larger than the page itself, and store a copy of the segment unique identifier with each page when it is assigned to a segment. Then, since pages are identifiable, lost or multiply-used pages could be returned to their proper owners with less chance of accidental interchange.

This problem illustrates an issue which is as yet not very systematically approached in large systems: the initial design almost always assumes perfectly functioning hardware and software, and as experience is gained about which failures are most common, patches are added to protect. The design of the

---

* Previously issued: memos are reprinted on page 1 and 22 of this memo.

second CTSS file system included forward and backward pointers with every record of a file; the system always checked the back pointers to see that they contained the expected values. As a result, parts of user files were almost never interchanged -- a distinct improvement over the first CTSS file system which used forward pointers alone, and in which it was a common occurrence to find someone else's data in your file. Unfortunately, this particular CTSS lesson did not get transferred to Multics, probably because of the extra overhead that might have been involved in drum management.

Operator login window

When bootloading Multics, the operator dialed a telephone number to log in the "initializer" console, which controls all system operation. A hostile user, with careful timing, could dial the number and take over the system as it comes up. The design was adopted so that system initialization could be performed from any available terminal; it was originally intended that the operator supply a password, but for some reason that intent was never implemented. The design was recently changed to permit use of a terminal which is permanently wired to the system; security is higher, but when that terminal breaks, system operation may be awkward. The awkwardness can be eliminated by having several available hardwired terminals.

FSDCT update problem

The "file system device configuration table" (FSDCT) contains a bit for every storage block in every secondary storage device. A "one" means the block is unused, a "zero" means it is used. If several devices become completely used, a page of the FSDCT may become filled with zeroes. Since it is an important table, it is frequently backed up by copying it out to secondary storage. The procedure invoked for this copying is the standard page removal procedure, which has been designed to discard pages of zeroes rather than writing them out. The routines which read the FSDCT from secondary storage at system initialization time (before the standard paging program works) was a non-standard one which did not know that pages of zeroes were given special treatment; a system crash resulted whenever the system was initialized. In principle, at least, a user with a very large storage allotment could exploit this bug by creating many segments just before a system shutdown. The system would shut down with an FSDCT containing blank

pages, and all future attempts to bootload the system would fail. The bug was fixed by revising the FSDCT reading procedure to correctly recognize the blank pages during initialization.

This is a category of bug which does not permit the exploiter to read information, but merely to deny use of the system to other legitimate users. The particular problem illustrates the effect of first using a special trick for efficiency, followed by later use of an old procedure for a new purpose without reviewing its operation for special tricks.

Login table overflow

The list of logins during a single bootload of Multics was stored in a single segment with no overflow procedure. A single user, by logging in several thousand times, could overflow the segment, making further logins by authorized users impossible.

This is another example of a "denial-of-use" bug, but one which could be rapidly recovered from by reinitializing the system. Its origin lies in the period between 1968 and 1970 when a combination of pressure to get going and also a short average "system up" time made programmed provisions for table overflow look like a non-essential luxury. It has been long since fixed by adding an overflow procedure, but its origin is instructive since there may be yet unsuspected protection bugs with the same origin.

Page control magic number

An old hardware bug trap places magic numbers in core where a page is to be read in, then after reading the page checks the numbers. If still there, it assumes the page didn't come in, and reports a page read error to the user. If a user places contrived names containing the magic bit patterns strategically in a directory to which he has only append access, he can effectively delete other entries in the directory.

The trap has been left in the system, but it has been placed under strict operations control by requiring a special "debug" card in the configuration deck loaded by the system operator before lootload; operation with the debug card in place is done only with special authorization, and leaves an audit trail.

Retriever acl-setting bug

The retriever, used to obtain old copies of files from backup tapes, used to work as follows:

1. Create a new empty segment in the user's directory, with an access-control-list permitting access to anyone.
2. Copy the data from the tape into the new segment.
3. Read the appropriate access-control-list from the tape.
4. Replace the initial access-control-list with the one read from the tape.

If an error of any kind occurred after completion of step 2, the retriever would exit, leaving the data reloaded but unprotected; the user received no warning of the condition. As a result, an explorer of the directory hierarchy would typically discover several files to which he had access but should not have.

The problem was repaired by making the initial access-control-list grant access to the retriever process only; any errors after that point result in a fail-safe inaccessibility of the segment. Since the user who requested the retrieval will usually try to immediately use his retrieved segment, its inaccessibility will tend to be discovered quickly, and a locksmith can be called upon to adjust the situation.

This problem is a good example of design which did not take into account all the implications of an error encountered in an otherwise acceptable sequence.

Process directory record overflow

If the user generates too much storage (more than 500 pages) in his process directory, an error is signalled to him. In the original design, the signaller used the wrong stack, crashing the system. This bug could be exploited to deny service to others at the user's whim. It was repaired by having the signaller use the correct stack. It is a good example of the effect of complexity (the need for several possible stacks) compounded with the difficulty of testing unused and limit conditions. Basically, the handlers for rare and unusual conditions tend to be poorly tested simply because normal use, which uncovers most bugs in today's systems, does not exercise them.

Locked stack base problem

In the design of the 645, a provision was made for the supervisor to lock the value of any base register. This feature was included primarily because it was planned to handle faults and interrupts using a stack, and it was not certain at the time whether or not use of a stack was possible unless the stack base register (containing the stack segment number) was locked against user tampering. For several years, Multics operated with a locked stack base register whose value was changed by a master-mode procedure as part of the ring-switching operation.

The fault and interrupt interceptors were coded assuming a locked stack base at three points, although after the ring design was complete, it became clear that the user could, in principle, be safely allowed to modify the stack base register.

With the evolution of the design of the PL/I compiler, it became apparent that the extra flexibility of allowing the stack base register to be user changeable was quite handy, so the stack base register was unlocked. Unfortunately, no one followed through with the three one-line changes to the fault and interrupt interceptors required to eliminate their dependence on a locked stack base register. As a result, one could load the stack base register with the segment numbers of one of the ring-zero stacks, and then wait for the next fault or interrupt, which would go to an interceptor which incorrectly assumed that because the stack base register had the expected value, the stack pointer register must also be loaded correctly. The result was possible overwriting of a ring zero data storage area at the direction of the user.

The problem was fixed by adding the three one-line checks mentioned. The underlying trouble here seems to be a failure to follow through all the implications of a change in a fundamental ground rule; clearly such changes are dangerous and must be approached with all possible caution. (see also REC-46, discussion of user-ring master-mode procedures.)

New ring stack bug

The system has an internal procedure, named "append_branch", which creates a new segment, and a utility named "makeseg" which either creates a new segment (by calling "append_branch") or returns a pointer to an old

one if it already exists. Since "append_branch" requires many arguments to describe the newly created segment, and "makeseg" supplies useful defaults for most of the arguments, there is a tendency among system programmers to call "makeseg" rather than "append_branch", even when use of an old segment would be incorrect. In the case of the procedure which creates stacks for newly entered rings, the user could create a segment with the stack name of a previously unused inner ring, but with ring brackets allowing him to read and write the stack contents. Then, upon calling a procedure in the inner ring, stack creation would be automatically triggered. The stack creating program called "makeseg", and thus would receive a pointer to the previously planted stack rather than an indication of an error. The inner ring procedure would then proceed, oblivious to the fact that its stack was then accessible to programs in outer rings.

The problem was fixed in moving to the 6180, since the stack creation strategy had to be modified anyway; procedure append_branch is now used. We have here an example of how a particular combination of too many conveniences in one utility program can lead to sloppy consideration of the implications of using it.

Patterns of Security Violations: Multiple References to Arguments

by Harry C. Forsdick and David P. Reed

1. Introduction

A large class of potential holes in the security of an operating system is characterized by the use of an argument more than once. On the surface, this situation appears to be harmless: multiple references may be inefficient, but they seem to be functionally equivalent to a single reference. But, are they? If the value of an argument could change between one reference and the next, the possibility of an error in the logic of the program using the argument exists. The assumption made by the author of the program that an argument could only be altered by the program or agents of the program is violated. How could an argument change in this invalid way? A simple conceptual scheme on a multiple process system is for one process to execute the call, supplying the arguments and a second process which has access to the values of the arguments, to perform, at the appropriate time, the alteration on the arguments. Whether or not a multiple argument reference leads to a breach of security depends on how the information gained from each reference is used. If the results of a test on one reference to an argument determine how the information of a second reference is used, then a exploitable hole in the system probably exists. More specific conclusions on the correctness of multiple references to an argument depend on the semantics of the particular program under analysis. Richard Bisbey of the Information Sciences Institute of USC brought this subject to our attention. He described the multiple referencing of arguments as a general pattern for a class of security holes and cited several instances of this pattern in Multics.

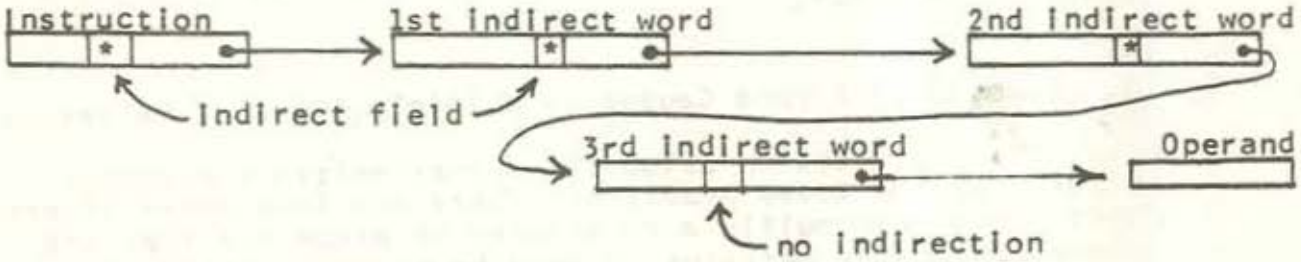
With these ideas as motivation, the Multics gate entrances to ring 0 were examined to determine if such multiple references to arguments were being made and if so, the implications of such flaws. Of the approximately 170 entrypoints to ring 0 through the hcs_ gate, about 50 were found to make multiple references to their arguments. Nine of these instances were potentially serious breaches of security in the Multics system. All of these breaches are easily fixed by copying arguments and then

referencing the local copies.

2. How to Change the Value of an Argument

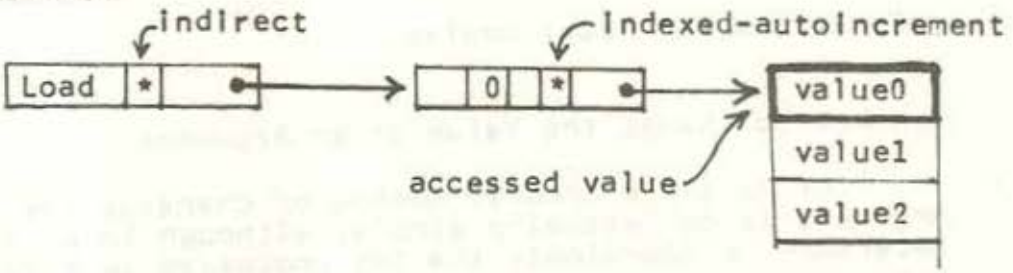
The multiple process method of changing the value of an argument is conceptually simple, although in practice, it is necessary to coordinate the two processes so that the argument gets changed at the proper time. This task is often impossible to accomplish except by chance. A slightly more complex mechanism however, makes the alteration of an argument trivial. The combination of indirect and indexed - autoincrement addressing and the ability to cascade these modes of addressing allows a programmer to set up an argument list so that each reference to an argument accesses a different value. On the H6180, indirect then Tally (IT) address modification is one of the kinds of indirect addressing and the Increment Address - Decrement Tally - Continue (IDC) variation on the IT modifier is an example of indexed - autoincrement addressing.

First, consider indirect addressing. Typically, there is a field in an instruction which can specify that the operand address points to a cell (the "indirect word") which contains the actual address of the operand. In addition, with cascading, a field in the indirect word can specify that the indirection process should continue at least one more level. For example, the diagram below depicts three levels of indirection:

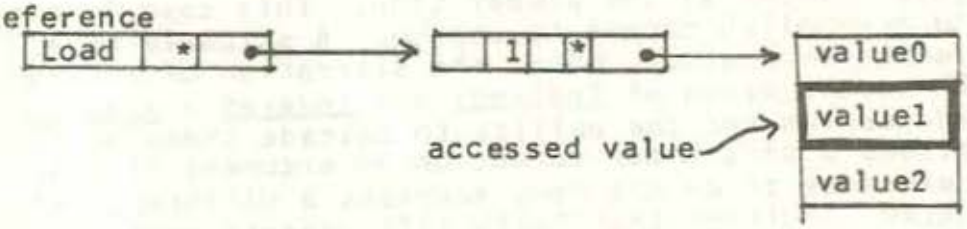


For the indexed-autoincrement mode, there are two additional fields in indirect words: the indexed-autoincrement field and the count (tally). When an indirect word with the indexed-autoincrement addressing mode is accessed, the count is added to the address and used as the effective address of the indirect word. In addition, the count field is incremented by 1. Thus, each time an indirect reference is made through an indirect word with the indexed-autoincrement addressing mode, the effective address is one location higher. This is very useful in accessing tables -- in our particular case, tables of values for a single argument. For example, the diagram below depicts two consecutive references to an argument. The indirect word is part of the argument list set up by the calling procedure. In the first reference, the count is zero and thus the value accessed is the first value in the array of values.

First Reference



Second Reference



The count is automatically incremented by one so that on the second reference, the value accessed will be the second member on the array.

If arguments are accessed by indirection (as they are in Multics) it is quite easy for a (malicious) programmer to set up an argument list so that each reference to an argument accesses a different cell. A number of machines (for example H6180, UNIVAC 1108) have the addressing features similar to the ones described above and thus systems running on these machines are susceptible to the problem of arguments changing values at unexpected but predictable times.

3. Classes of Errors Caused by Multiple Argument References.

The last Section established that multiple argument references can cause problems. There are four types of errors that arise from multiple references to arguments that are characterized by patterns of reading and/or setting the arguments. The illustrations below are stated in terms of double references, although the discussion applies equally well to any number of multiple references.

1. Double Reads: In this class of error, an argument is read twice. The value read the first time is tested and the result of that test determines how the value read the second time is used. The following program fragment illustrates this type of error:

```
if argument = 'pds' then switch = 0;
:
:
if  switch  = 0 then ....;
                else .... (reference to argument) ....;
```

The value obtained by the second reference to argument could very

well be 'pds', a state that is inconsistent with the original if statement.

2. Setting then Reading: Another common class of error occurs when a procedure initializes an output argument to a certain value and then relies on the integrity of that value. The scheme outlined in Section 2 works equally well for reading or setting arguments. Thus, it is possible for a user to cause a called procedure to use a value that is outside of its control. The following program fragment illustrates this type of error:

```
a_ename = 'mailbox';
:
:
call delentry$dfile(dirname,a_ename,code);
```

Between the points a_ename was set and used, its value could be changed to any value the user desired.

3. Setting twice: A slightly less obvious, yet potentially equally damaging error arises when an output argument is set twice. Damage results in situations where the value to which the argument is first set is to be hidden from the calling procedure by storing the second value. Again, since the scheme of Section 2 works equally well for reading and writing a history of argument values can be developed. This history is a potential privileged information leak. The following program illustrates this point:

```
argument_code = error_table$entry_not_found;
:
:
argument_code = error_table$no_access_to_file;
    /* Hide existance or non-existance of file from user. */
```

4. Passing an Argument: A "delayed" error can arise when an argument to one procedure is passed directly without copying to another procedure. This is because the value of the argument resides in an address space that is not protected (the user's address space). In Multics, the scheme described in Section 2 does not cause a problem because an entry in an argument list for an argument to the calling procedure points directly to the value of the argument. Thus, there can be no malicious addressing modifiers in the argument list. The more general multiple process scheme, however, is still effective in changing the value of the argument. For example, if procedure A is called with argument X by a user procedure, and A in turn calls B supplying X (without copying) as an argument, then the value of X can be changed by the multiple process scheme during the time B is running. This problem is made more serious by the tendency for argument validations to be dropped (for efficiency reasons) in

procedures that are internal to the protected part of a system.

5. Multiple References to Pointer Qualified Arguments: Quite often a pointer to an argument is passed to a procedure when the actual argument is a complex data structure. Again, the multiple process scheme can cause the actual data item to be altered during the running of a called routine. Copying the pointer into a local variable and performing references through this local copy does not solve the problem since the actual value of the argument can be changed by the multiple process scheme.

4. Methods of Recognizing Multiple References

In a large system it is very difficult to discover instances of the errors ourlined in Section 3. Two alternative methods of attack were taken in out study of Multics. One technique is to perform an analysis of the text of all procedures that are interfaces between the critically sensitive part of the operating system (ring 0 gates in Multics) and user programs. This analysis is aided by the cross reference listing produced by the PL/I compiler. Certain patterns in the cross reference listing for arguments indicate that multiple references are being made. The main advantage to this approach is that if done correctly, it will yield all instances of multiple argument references. The main disadvantage is that it is a time consuming task.

There are two defects in the cross reference technique. First, all references are listed together; thus it is impossible to tell by looking at the list which kind of reference (read, write, appearance in an argument list) occurred. The inability to distinguish in the cross reference listing between argument list appearances and reads and writes makes the analysis more difficult. The second defect of the cross reference technique is more serious. The appearance of a reference to a name in the text of a PL/I program does not guarentee that there will be a corresponding reference to the value of the name in the instructions emitted by the compiler. There could be zero or more references depending on optimizations performed by the compiler and the form of the actual reference. As an example of the last exception, the statement

```
x = convert(argument,z);
```

doesn't actually reference the value of the argument. The value of z is converted to a value whose type is the same as the type of argument and stored into x. Similarly, a reference to the length of a string does not reference the string, but rather the descriptor of the string. Thus, searching the cross reference list for multiple references can cause false alarms. On the other hand, the cross reference list provides no help in spotting

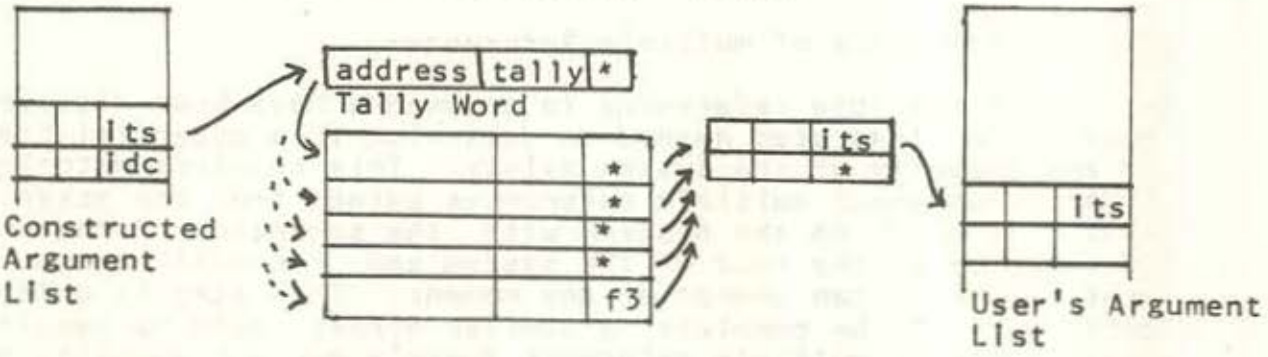
references to arguments that are contained within loops.

It is conceivably possible to mechanize this process so that multiple references to arguments could be discovered by an automatic analysis. This task would fit in easily in the framework of the PL/I compiler since all of the necessary information is already available within the compiler.

The second technique for discovering multiple argument references involves monitoring the actual use of arguments passed to interfaces and noting any arguments that were referenced more than once. The mechanism used to exploit multiple references to arguments noted in Section 2 can also be used to detect multiple references to arguments at runtime. While all multiply referenced arguments cannot be detected in this way, many which can be exploited via the autoincrement mechanism will be found. Since these are particularly easy to exploit, detection of them is quite useful.

In order to detect these bugs, a set of special transfer vectors were substituted for the ring 0 and ring 1 gates in several users' processes. These transfer vectors constructed a new argument list which made use of the autoincrement features of Multics indirect addressing to keep a count of references to arguments via the pointers in the argument list. This argument list, which ultimately referenced the original argument list via a series of indirections, was passed to the real ring 0 or ring 1 gate. Upon return, the transfer vector code observed the number of references to each argument, and recorded the maximum number of argument references in any call in a metering data base which had one entry per argument per entry point.

For those interested, the argument list constructed is detailed below. It should be noted that this technique can only work if the number of argument references can be bounded and small (i.e., references to arguments do not appear in loops). Unfortunately, this was not the case for tty_write, tty_read, and tty_order. Consequently, these entry points were not measured by this method after the initial tests.



There are several deficiencies in using this scheme to detect multiple references. First of all, it is necessary to exercise all possible control paths of the system procedure in order to find all of the cases of multiple references (so some holes may pass unnoticed). Secondly, this technique produces many false alarms, since the code produced by the PL/I

compiler may produce multiple indirections through the argument for one logical reference (this may or may not be a bug). Also, structure or array arguments may have subparts, all of which are singly referenced, but through the same argument pointer. Another problem is that PL/I sometimes copies argument pointers by indirection upon entry to a multiple entry point procedure (the case occurs if the same name appears in different positions in several formal parameter lists). This results in only a single reference being detected by this technique, even though multiple references may be made. The last problem is that arguments which are passed on to internal routines will not be caught, since PL/I indirects through the argument list once to get the address of the argument which is passed on. Even if the argument is referenced multiply by the internal routine which receives it, this will not be done via the indirect chain provided to the external routine by the transfer vector, and will not be counted by this technique.

Most of the bugs which were found in the current system by the auditing method were also found by the monitoring method. This suggests that the latter technique might be useful in attempting to prevent possible bugs in the system from being exploited, by crashing the user's process if an argument is referenced more than once. (This could be accomplished by causing a fault on the second reference by using a fault tag 3 indirect word as the second entry in a two element array of indirect words referenced by the idc autoincrement mode.) Certainly, such a firewall has its costs, both in runtime efficiency, and in the fact that all innocent multiple argument references must be purged from the system, as well as the security holes, in order for the firewall to work. Nevertheless, this may well be worthwhile in attempting to prevent retrogression in the security of the system for some users with high security requirements.

5. The Semantics of Multiple References

Once multiple references to arguments have been discovered, there is a final step needed to determine if a potential breach of the security of the system exists. This requires matching the information about multiple references gained from the essentially syntactic check on the program with the semantics of the program in relation to the rest of the system and the basic assumption that arguments can change at any moment. This step is quite difficult. To be complete, a similar effort would be required to justify that a multiple reference doesn't cause a security hole as to justify that the program <u>is</u> secure. But, shortcuts can be taken: knowledge of the meaning assigned to arguments helps in isolating serious problems from harmless mistakes.

Of all of the steps in the technique for discovering errors due to multiple argument references, this is the most difficult step to mechanize. A very large amount of knowledge about the operation of the system must be used to determine whether or not

a multiple reference is a serious error. The major benifit of searching for the pattern of multiple references is that areas of the program text which deserve close analysis are isolated.

6. Results of Applying this Approach to Multics.

An analysis of the Multics ring 0 gate entrances was performed. First, multiple references to arguments were discovered using both the cross reference listing technique and the monitor technique. Next, each entrypoint that had arguments that were multiply referenced was analyzed to determine the effect of the multiple reference. A list of the entry points tested and the results of those tests are found in Appendix 1. Numerous multiple argument references were uncovered. In most of these cases we were able to conclude with a high level of confidence that no errors result from these references. In a number of other cases, however, serious breaches in security were discovered.

The simplest and most glaring error was due to a multiple argument reference in "stop_process." By exploiting the multiple reference in the manner previously described, any process in the system could be stopped (including the initializer process). A less selective denial of service existed in "status_" and "status_long"; by setting up a certain form of argument list, these routines could be made to lock a lock that would never be unlocked. This would eventually cause the system to crash. It is possible to direct "tty_write" to send an unending stream of characters to a terminal. This has the effect of tying up the entire system and causing the appearance of a crash.

Other errors were found that were either deemed less serious or less obvious how to exploit. Because of a multiple reference to an argument in "add_inacl_entries" it is possible for a user to specify the initial access control list for any ring on any directories that he may create. This seems like a serious error, but it is difficult to see how to exploit it. In "printer_dcm" it seems possible, once a printer has been seized, to address any other printer. In "tdcm_message", multiple argument references make it possible to print inconsistent messages on the operator's console. Finally, assuming that it is possible to get past the "hphcs_" gate, it appears possible to set up inconsistent information in tables that record the state of tape drives by a call to "tdcm_add_drive".

One additional error due to a multiple argument reference is now known. At first we had classified the entrypoint "sfblock" as being in the class of entrypoints that did not have multiple references. A subsequent communication from Richard Bisbey pointed out a fairly subtle error in this entry to the supervisor. A portion of one of the arguments contains an index into a bit string stored into the PDS (an important ring 0 data base), and is first validated to be within range. It is then used to select a bit in the bit string to be set to one. If the second reference gives an out of bounds index, then any bit in the PDS may be set. Both of the multiple reference detection

techniques had failed to find this error. The monitor technique failed because the argument is referenced via a generated pointer; the auto-increment technique for exploiting such holes will not work for this instance. The cross reference listing technique probably failed due to human error.

Several direct conclusions come out of our experience with Multics. First, each of the multiple reference detection techniques discovered multiple references that the other did not uncover. In addition, both missed at least one instance of a multiple reference. Tedium accounted for the missed occurances in the cross reference listing technique; an automated version of this method would presumably not suffer from this limitation. In the monitor method, multiple references were missed because some program paths were not taken. Second, even when all multiple references have been uncovered, one must be very conservative in analyzing programs for correctness. Further, when such programs are modified, there is a strong chance that harmless multiple references may lead to serious holes; such programs will need to be audited on each new installation. In many cases this is an extremely tedious task for which people are not well suited. To be entirely sure that a multiple reference is harmless, all paths that a program may take must be traced. Clearly there is a need to develop algorithms which would perform the analysis mechanically.

All of the security holes reported above have been fixed in the current Multics system.

7. Solutions to the Problem.

In the past there have been a number of different reasons for copying arguments. Most of these are characterized by the need to avoid a fault (directed faults: segment, page, no access, ring violation; or indirect address fault: linkage, f1, f3, illegal procedure) while a lock is locked. In May, 1967 a protocol similar to the one described below was detailed in MSPM BD.9.02. The suggestion was made that all arguments to a procedure be copied and that only these copies should be used in the procedure. As various improvements in the system have occurred, some of the reasons for copying arguments have been eliminated and some programmers have ceased to copy arguments. The results of this work show that because of the difficulty in analyzing the effect of multiple references to arguments, all arguments should be copied and validated upon procedure invocation. To be entirely safe, the following pattern of coding should be followed for all ring 0 interfaces:

```
F:  procedure(a_arg1, a_arg2, ... , a_argn);

    copy the values of all input and input/output arguments
    into local variables.

    validate local copies with respect to semantics
    associated with them in this procedure.

        :
        :   use local copies
        :

    set output arguments to values of corresponding
    local variables.

    return

    end;
```

By using this conservative coding style, a procedure can be more strongly isolated from its callers. In effect, we are making a better (by no means perfect) simulation of separate domains by following suitable restrictions in programming style. It should be noted that there are situations where it is difficult to adhere to this style because of efficiency considerations. For example, it would be very inefficient to copy an argument that is a large structure occupying many words of storage. Just as there are syntactic patterns for recognizing bugs in programs, the inverses of these patterns appear to be guides for secure programming.

The general idea of patterns of errors seems to be a powerful tool that can be used in an analysis of a system. In a very short time we have discovered several serious holes in the security of Multics. The success of this error pattern resulted from its simplicity. The main obstacle in discovering other patterns is not so much the nature of the error but rather the suitable simple pattern for which to search. For example, one of the recurring types of errors reported in RFC's 5, 46 and 47 and in the Multics Change Requests is overflowing the capacity of a table. Because of the flexibility of the PL/I language, there are many ways to implement tables. It would be difficult to come up with a general pattern that matched all of these ways because of the many degrees of freedom in the PL/I language. The conclusion is obvious: What we need are more highly structured languages which require a programmer to identify the objects being used (for example the language "CLU" being developed in the Computation Structures Group of Project MAC at MIT). In this way, simple patterns for complex errors can be developed.

Appendix

Classification of Entry Points in hcs_

Of the 170-odd entrypoints in the hardcore gate hcs_, some 50 have multiply referenced arguments which were found by the auditing and online monitoring techniques. We may classify these further into five classes:

1. Those which are probably not security holes. To the best of our knowledge, with the way the system is currently structured, these multiple references do not cause any problems. Of course, we would feel even safer if all arguments were copied and the copies referenced.

2. Multiple references which cause the procedure to be fragile, but which probably do not cause security violations. By fragile, we are trying to dramatize the fact that the multiple references to arguments cause the procedure to be very dependent on the current order in which tasks are carried out. Alterations in the procedure are very likely to upset this delicate balance.

3. Multiple references that have not been explored to the depth necessary to assign them to one of the other classes.

4. Multiple references which look as if they produce holes in the system, but we can't think of a way to exploit the hole.

5. Multiple references which cause holes which we know how to use to penètrate the system.

The following list of entrypoints tells which arguments, if any, are multiply referenced. The notation 'entrypoint (1,3)' means that the first and third arguments of entrypoint are referenced more than once. If any arguments are referenced more than once, remarks are made about which of the above five classes the references belong to.

Summary of Results

A summary of the results obtained in our study is presented in the following table.

Number of entry points examined in hcs_.	170
Number of entry points with multiple references.	51
Classification of multiple references:	
Type 1 -- Probably O.K.	23
Type 2 -- Fragile, but probably O.K.	8
Type 3 -- Don't know, lack of information	3
Type 4 -- Hole without obvious exploitation	8
Type 5 -- Hole with known exploitation	9
Untested entry points	3

Entrypoint	Args referenced more than once	Type, Remarks
accept_alm_obj	(1, 2)	1 -- Probably O.K.
acl_add		
acl_add1	(3, 5)	1 -- Probably O.K. Arg 3 validated after 2nd reference, arg 5 is an array whose elements are referenced once each.
acl_delete		
acl_list		
acl_replace		
add_acl_entries		
add_dir_acl_entries		
add_dir_inacl_entries	(5)	4 -- Hole, without obvious exploitation. Can operate on any ring initial acl, since argument is validated before copying.
add_inacl_entries	(5)	4 -- See add_dir_inacl_entries.
append_branch		
append_branchx		
append_link		
appendl		
assign_channel		
assign_linkage	(1)	1 -- Probably O.K. This program could run in the user ring.
block		
chname		
chname_file		
chname_seg		
cpu_time_and_paging_		
del_dir_tree		
delentry_file		
delentry_seg		

delete_acl_entries		
delete_channel		
delete_dir_acl_entries		
delete_dir_inacl_entries (5)	4	-- See add_dir_inacl_entries.
delete_inacl_entries (5)	4	-- See add_dir_inacl_entries.
ex_acl_delete		
ex_acl_list		
ex_acl_replace		
fblock (1, 2)	2	-- Fragile, but probably O.K.
fs_get_brackets (3)	1	-- Probably O.K. Array whose elements are referenced once each.
fs_get_call_name		
fs_get_dir_name		
fs_get_mode		
fs_get_path_name		
fs_get_ref_name		
fs_get_seg_ptr		
fs_move_file		
fs_move_seg		
fs_search_get_wdir (1)	1	-- Probably O.K. Referenced twice in copy of pointer using old version 2 pointer copy.
fs_search_set_wdir		
get_alarm_timer		
get_author		
get_bc_author		
get_count_linkage		
get_defname_		
get_dir_ring_brackets (3)	1	-- Probably O.K. Array elements referenced once each.
get_entry_name		
get_initial_ring		
get_ips_mask		
get_link_target (4)	1	-- Probably O.K. Return value, insensitive.
get_linkage (2)	1	-- Probably O.K.
get_lp (1, 2)	1	-- Probably O.K.
get_max_length		
get_max_length_seg		
get_page_trace		
get_process_usage (1)	1	-- Probably O.K.
get_rel_segment		
get_ring_brackets (3)	1	-- Probably O.K. Array elements referenced once each.
get_safety_sw		
get_safety_sw_seg		
get_search_rules		
get_seg_count		
get_segment		
get_usage_values		
get_user_effmode (5)	1	-- Probably O.K.
high_low_seg_count		

initiate	
initiate_count	
initiate_search_rules (7)	1 -- Probably O.K. Twice referenced in copy operation.
initiate_seg	
initiate_seg_count	
ioam_list (1)	3 -- Don't know, haven't looked at it close enough.
ioam_release	
ioam_status	
ipc_init (6)	1 -- Probably O.K. Twice referenced in copy operation.
level_get	
level_set	
link_force	
list_acl (3)	2 -- Fragile, but probably O.K. User can cause fault, but no locks locked.
list_dir	
list_dir_acl (3)	2 -- Fragile, but probably O.K. See list_acl
list_dir_inacl (3)	2 -- Fragile, but probably O.K. See list_acl.
list_inacl (3)	2 -- Fragile, but probably O.K. See list_acl.
make_ptr	
make_seg (1, 2, 5)	2 -- Fragile, but probably O.K. Can cause strange KST state with blank name.
makeunknown	
mask_ips	
pre_page_info	
printer_attach (2)	4 -- Hole without obvious exploitation. Event channel saved in user area, then referenced.
printer_order	Not checked. No listing available.
printer_write_special	Not checked. No listing available.
printer_detach (1)	5 -- Hole. Can cause inconsistent attachment states, since device index is validated, then used.
printer_write (1, 2, 3)	5 -- Hole. Can write on different printer than the one assigned.
proc_info	
quota_get (2)	1 -- Probably O.K.
quota_read	
quota_move	
read_events (1, 2)	1 -- Probably O.K.
replace_acl	
replace_dir_acl	
replace_dir_inacl (6)	4 -- Hole without obvious exploitation. See

		add_dir_inacl_entries.
replace_inacl (6)		4 -- Hole without obvious exploitation. See add_dir_inacl_entries.
reset_ips_mask		
reset_working_set		
rest_of_datmk_		
set_alarm		
set_alarm_timer		
set_automatic_ips_mask		
set_backup_dump_time		
set_backup_times		
set_bc		
set_bc_seg		
set_copysw		
set_cpu_timer		
set_dates		
set_dir_ring_brackets	(3)	1 -- Probably O.K. Array elements referenced once each.
set_dtd		
set_ips_mask		
set_lp		
set_max_length		
set_max_length_seg		
set_pl1_machine_mode		
set_safety_sw		
set_safety_sw_seg		
set_ring_brackets	(3)	1 -- Probably O.K. See set_dir_ring_brackets.
set_timer		
sfblock (1)		5 -- Hole. Uncopied value used when copied value available!!
star_		
star_list_		
status		
status_	(4, 5)	5 -- Hole. User's argument controls whether lock is locked, and then whether it is unlocked. Can leave lock locked.
status_long	(4, 5)	5 -- Hole. See status_.
status_minf		
status_mins		
status_seg_activity		
stop_process (1)		5 -- Hole. Can stop any process. arg used after validation.
tdcm_attach		All tdcm entries use a segment as argument. It is not clear whether changes to this segment can cause problems.
tdcm_detach		
tdcm_locall		
tdcm_message	(2)	4 -- Hole without obvious exploitation. Can possibly

		cause message inconsistent with system's idea of tape name.
tdcm_promote		
tdcm_reset_signal		
tdcm_set_signal		
tdcm_mount_bit_get (1)		1 -- Probably O.K.
terminate_file		
terminate_name		
terminate_noname		
terminate_seg		
total_cpu_time_		
trace_marker		
truncate_file		
truncate_seg		
try_to_unlock_lock		
tty_abort (2)		3 -- Don't know effect of multiple reference. Not sure whether this is a problem or not.
tty_attach (2, 4, 5)		2 -- Fragile, but probably O.K. Finally copies second argument inside second level call to ioam_. Other args O.K.
tty_detach (3, 4)		1 -- Probably O.K.
tty_detach_new_proc (3, 4)		1 -- Probably O.K.
tty_event (2, 3, 4)		2 -- Fragile, but probably O.K. See tty_attach.
tty_index (4, 5)		5 -- Hole. Code is referenced twice in dn355$get_devx. Could return information which might be sensitive about allowed device id's.
tty_order (2, 3)		3 -- Don't know whether this multiple reference is a hole or not.
tty_read (3, 5, 6)		5 -- Hole. Perhaps hard to exploit.
tty_state		
tty_write (3, 4, 5, 6, 7)		5 -- Hole. Arg 3 referenced in a loop. Can cause the system to appear crashed.
unmask_ips		
unsnap_service (1, 2, 3)		1 -- Probably O.K. This program need not be in ring 0.
usage_values		
virtual_cpu_time_		
wakeup (4)		1 -- Probably O.K.

A Two-level Implementation of Processes for Multics.

September 8, 1976 21:23
R. Frankston

This is a description of an implementation of Multics Processes using multiple levels of abstraction. The implementation is being done in conjunction with David Reed and is based on the model described in his Master's Thesis titled **Processor Multiplexing in a Layered Operating System.**

This draft contains many implementation details, some of which have been modified in actually writing the code and will be described in a later memo. Some sections are only superficial and are meant as a guide for later revisions and extensions. Warning: Since this document is being modified as design changes are being made without a complete rewrite there may be inconsistencies in the descriptions.

If you have comments, suggestions or questions either see me personally or send mail to Frankston.CompSys@MIT-Multics or RMF@MIT-MC.

Table of Contents

Two-level Process Implementation

Introduction

The description of the implementation below is concerned with relatively narrow issues involved in actually coding algorithms which implement the model described in David Reed's thesis. The implementation includes some arbitrary decisions necessary for the embodiment of the algorithms. This description assumes familiarity with the current Multics system. David Reed's thesis should be consulted for a fuller discussion of the issues involved. To make the document at least somewhat readable for a wider audience as well as to reduce the problem of the proliferation of strange abbreviations there is a glossary on page 42.

The key difference between the current Multics implementation and the multilevel one is that a distinction is made between scheduling decisions (i.e. traffic control) that involve policy and those that don't. For the ones that don't involve policy the decision is relatively trivial -- the next processor available to run will be run, a relatively cheap operation. In order to achieve this simplicity the primitive level, level one, consists of a fixed number of virtual processors that are considered at higher levels to be always assigned to a processor. In fact physical processors are a relatively expensive and therefore scarce resource requiring the basement of the implementation to, in fact, multiplex the virtual processors on physical processors on a first-come, first-served basis within a predetermined priority assignment.

The advantages of the two level approach to traffic control include:

i. The system is simplified since one can view a Multics process as being built upon the relatively simple semantics of a virtual processor as opposed to the complex semantics of the current traffic control and interrupt structure.
ii. The implementation of the system primitives for process coordinations can be more efficient than the current ones because of the simplified environment in which they run.
iii. By improving the structuring of the system, the system an become more understandable and thereby more reliable.
iv. Robustness is enhanced by isolating Virtual Processor multiplexing within the PAM. One can assign properties such as encachability to individual processors. Since the PAM does all storing and restoring of physical processor states it can be responsible for all the complexity of maintaining such states.
v. By handling the fault within the PAM outside of the virtual processor, the VP itself need not be capable of handling page faults thereby simplifying the semantics and removing special restrictions which require the wiring of the descriptor segment. Further more faults due to processor failures can be handled by another VP that does not use the particular feature. For example, the can be a process that does not rely on the cache so that it can diagnose cache failures.

vi. By separating processor multiplexing from scheduling the implementation of the policy portions of the scheduler are simplified by separating them out and are infrequent enough to remove the need for the efficiency of assembly language programming.

The current implementation plan consists of three parts:

1. A basic level one system without paging.
2. Level one with paging.
3. A full Multics system with the second level traffic controller.

At present a basic version of level 1 has been debugged and run. It is described on page page 40.

The Processor Assignment Manager and primitives

This basement (level zero) program (corresponding to the GPP algorithm in the thesis) is referred to as the Processor Assignment Manager (PAM)[1]. The PAM is to be considered as part of the physical processor -- there exists one logical instance of the PAM per processor. In addition to the function of multiplexing the physical processors, the PAM also serves to enhance the basic 68/80 processor by rationalizing its operation so as to provide a better basis for the other levels of implementation.

The PAM is entered whenever an interrupt or fault occurs. The currently executing virtual processor is unbound from the physical processor by saving its state in its Virtual Processor Table Entry (VPTE). As part of saving the state of the process the metering information is updated and a check is made to see if the process has exceeded its limit for CPU usage. The next step in processing depends on the reason for entering the PAM.

External interrupts are transformed into events that can be serviced by processes awaiting their occurance. If an internal interrupt (fault) can be handled by the VP itself, the fault information is saved in a communications area in the VPTE, the VP is marked as being unable to process further faults and its state is modified to execute its fault handler. If the fault cannot be handled by the VP, the VP is marked as *unsafe* and the Virtual Processor Coordinator (described below) is expected to do further processing. One fault is handled specially; the mme4 executed in a priviliged segment is treated as a `callp` operation by the PAM and serves to extend the capabilities of the physical processor. `callp` is described in more detail below. When the PAM has finished the interrupt processing, it places the VP into a new state. If nothing that affects the ability to run the VP has occured, it is placed in the *runnable* state.

The states that a VP may be in are:

> *running* indicates that the VP state is currently being interpretted by a physical processor and that the version in the VPTE is therefore invalid.
>
> *runnable* indicates that the VP may be assigned to a physical processor as soon as there are no higher priority runnable VPs. A VP enters the *runnable* state when it is unbound from a physical processor, but may continue to execute.
>
> *unsafe* indicates that the VP cannot be run without further handling by the Virtual Processor Coordinator. A VP enters the unsafe state if it takes a fault it cannot handle or does something the PAM does not expect. *Currently this state is not used, instead the VP is simply placed in the stopped state for examination by the level two traffic controller.*

---

[1] For historical reasons this module is also referred to as the Processor Binding Manager (PBM).

stopped indicates that a VP is no longer runnable and will not be handled further by the VPC. Once a VP enters this state it is eligible for unbinding by the level two traffic controller. Furthermore that is the only operation that may be performed on it. A VP enters this state when it exceeds its resource limits, or otherwise requires higher level processing to continue. The level two traffic controller explicitly places a VP in this state when it wishes to unbind it so that the L2TC my modify its state. *Stopped* VP's are kept on a queue for action by the L2TC.

awaiting is a state the VP enters when it goes blocked waiting for an eventcount to be advanced.

VPC blocked is a special state indicating the VPC is waiting for something to do. The VPC may only be in this state, *runnable* or *running*.

After placing the VP in its new state the PAM can do some standard processing including processing requests for clearing the cache and possibly deleting the CPU on which it is running. (Some of this standard processing is done earlier in the sequence than indicated in this description in order to minimize the time between entering the PAM and performing the function.)

Once the PAM has finished its processing, it then searches the VPT for next *runnable* VP. It places the VP in the *running* state to indicate that no other processor may examine the VPTE state. After checking to make sure that the VP may indeed run on the available CPU, it then loads the VP's state in effect binding it to the processor and running the VP

The support of the virtual processors is split between the PAM and a dedicated VP; the Virtual Processor Coordinator. This support includes the handling of faults and interrupts and mapping them into the appropriate functions. It also includes the support of the extended operations described in the section on VPI and on the CALLP operator. The VPC runs in a Virtual Processor so that it may take advantage of the process environment to simplify its implementation. The details of the VPC operation are given in a later section of this memo. The VPC is made runnable whenever an event occurs that requires its attention. The VPC is always the highest priority process so that it runs as soon as it is made runnable. Events requiring the VPC include the transition of a process to the *unsafe* or *stopped* states, the occurance of an interrupt or the transmission of a message to the VPC via `callp` as described below

Other dedicated VP's perform functions such as interrupt handling and page fault handling. A key dedicated processor is the policy module for scheduling user processes. This process is referred to as the level two traffic controller. Because of the limited number of virtual processors the level two scheduler must multiplex these processors. The details of this operations are not relevent for this memo. What is important is how a user (or level two) process is bound to a virtual processor and later unbound. This is similar to the function performed by the PAM and is done via the `VP1$bind` and `VP1$unbind` primitives.

PAM Details

There are a number of details associated with actually accomplishing the functions required of the PAM. These are discussed in the relatively unordered sections below. Detailed knowledge of the 68/80 processor is assumed. This information is contained in the GMAP manual, the 6180 processor manual and the Multics debuggers handbook. None of them fully or accurately described the current 68/80 processor.

General flow through the PAM.

i. The PAM is entered via the interrupt or fault vector.

ii. The control unit state and processor registers are saved. The current value of the real time clock is saved.

iii. Any requests to clear the cache of an associative memory are honored. This is described below under heading of connect fault processing.

iv. Virtual CPU time is computed. If there is a process awaiting the realtime event count, it is is notified.

v. Any special processing associated with the particular fault or interrupt is done.

vi. The virtual processor that was executing is placed in a new state. Normally it is placed into the *runnable* state unless the fault handling changes the process' characteristics. If the resource limit for virtual CPU time has been exceeded the process is placed into the *stopped* state.

vii. If the CPU is to be deleted, it notes that it in fact has been deleted and then goes to sleep here. The interrupt indicating that it has been added back continues from this point after intializing the processor state.

viii. The VPT is locked. If there is a pending wakeup for the VPC and the VPC is in the *VPC_blocked* state, it is made *runnable*.

ix. A virtual processor that is *runnable* and does not have any restriction against the current physical processor is placed in the *running* state.

x. The timer register is set as described below.

xi. The state of the virtual processor is loaded into the physical processor and begins execution.

Two-level Process Implementation

Operating Modes

Since the PAM is meant to act as an extension of the processor and form the basis for other mechanisms it operates in absolute mode so as not to depend on the correct functioning of the memory management software or hardware. This also removes the need to treat the descriptor segment specially (such as wiring the zeroth page) since the PAM is even more primitive than the levels relying on the appending hardware. When the PAM does use the appending hardware in implementing the `callp` operation, it is able to take faults in the same manner that any other hardware instruction might and processes them as if they had occured in an arbitrary hardware instruction. Since PAM processes interrupts by simply noting that the event took place and then restoring the processor state it operates inhibited.

Interrupt and Fault Handling

The 68/80 does not have any physical processor registers that can be used to distinguish between physical processors when addressing memory to store the machine state when an interrupt is taken. Furthermore there is only one address associated with each interrupt handler, without regard to the processor on which the interrupt is taken. Because interrupts are handled by processes, the processor need not be masked for interrupts at any time it is assigned a virtual processor. Therefore there is no need for complex masking strategies -- the processor can run with all interrupts unmasked at all times with the PAM using the inhibit bit to prevent interrupts.

Since any interrupt can be taken on any processor it is necessary to be able to save the machine state without regard to the processor it is taken on until sufficiently far into the PAM to enable the program to determine which processor it is on and where the associated VPTE is for deassigning the virtual processor. The algorithm used was inspired by Andre Bensoussan's work and worked out in conjunction with Bob Mabee (of course Dave Reed contributed, but then his contributions are assumed throughout). There exist two tables with enough capacity to store SCU data for each processor that may be configured. There is a pointer with a delta modifier equal to the length of an SCU entry. The interrupt vector is initialized to store the SCU data using an AD modifier. Thus when the interrupt occurs an address is obtained to store the current data and the pointer is updated in storage in an atomic operation so that if any other processor takes a fault it will not interfere. Control is then transferred to a common disambiguating routine that operates under a lock. The lock itself is grabbed using the `sznc` instruction which does not require the use of registers. The rest of the registers are then stored, the processor id is determined and thus the per processor storage address to which the registers are transfered. The pointer to the SCU table is then reset to point to the beginning of the other table and the first table is scanned from its beginning using the AD modify. Each entry is checked to see if it belongs to the currently running physical processor. If it does, then the data is simply copied out into per processor storage. If it does not, the data is then copied into the new table, again using an AD modifier to grab and reserve a slot. When this processing is done, the lock is released (via an `stcl`) and the next processor looping on the lock can repeat the operation with SCU tables switched.

Fault processing is similar to interrupt processing except that we can have a separate fault vector

for each processor to save the need for having to determine dynamically the identity of the processor on which the handler is running. The processing for both faults and interrupts is the same once we have copied the machine conditions into per processor storage.

Faults while in the PAM.

When the PAM is processing a `callp` request or a page fault, a further fault may be taken. In order to handle these a separate fault handler is used that assumes the fault is expected and that the PAM is in a "good" state. The handler does not save any registers and assumes that control registers (pointers to the VPT entry and the perprocessor information) are intact. The detailed handling depends on the PAM state. If a `callp` operation is being performed then the machine conditions are set to indicate that the fault occured while processing the `callp` operation itself and the fault is processed as if it had occured at the beginning of the operation. For page faults a message is sent to the page fault process for the fault (which must be on the descriptor segment) and the machine conditions are set to continue with the appending cycle when the descriptor segment becomes available.

The descriptor segment.

It should be noted that by operating in absolute mode, the PAM avoids dependence upon the descriptor segment. Current Multics takes advantage of appending mode by using the fact that the descriptor segment can be used to address different memory in the PRDS for each processor. The elaborate scheme described above is complicated by not having this mechanism available but as a consequence removes the requirement that descriptor segments be different on each processor and allows processes to share descriptor segments. This can be of great importance in permitting many small process with a single descriptor segment. The idle process is a simple example of a process sharing a single descriptor segment.

Details of callp implementation.

The `callp` is supposed to look like a normal machine instruction that may take faults. It is first validated to make sure that the instruction was executed in a priviliged segment (maybe just the VP program's segment?). If not, it is treated as a standard (mme4) fault and reflected back to the virtual processor. If the instruction is acceptable, the pam state is set to indicate that the `callp` is being processed and a copy of the machine conditions is saved. The operation number in the `A-register` is then examined. If it is invalid the virtual processor is made *unsafe* and the VPC is notified (this should never occur).

The specific processing is done according to the request. Typically it would involve copying the data pointed to by `pointer register 0` into VPTE or copying the data from the VPTE. The detailed operation of each `callp` is described in the section on the `callp` operator.

When the processing is done, the PAM continues by placing the virtual processor into the *runnable* state and resetting the callp-in-progress flag. The PAM then continues as for any other fault.

If a fault occurs while the callp is being processed, the fault conditions are reset to those at the beginning of the callp instruction with the exception of the data address being referenced which is taken from the new SCU data associated with the fault. When (and if) the callp is restarted after the fault, it will begin from the beginning of the instruction. This allows the fault handling program to use the callp operation itself and not have restrictions on using the communications area in the VPTE.

Page fault processing.

The SCU data is examined to determine the type of fault. A message is sent to the page fault process consisting of the ASTE entry pointer, a unique segment id (in case the AST entry is deactivated), the descriptor segment AST entry pointer, the page number and a eventcounter associated with the fault. The process is then left awaiting this event, ready to continue address evaluation.

Processing the connect fault

The processing of the connect fault is very simple -- it is ignored. Its purpose is to force a processor to enter the PAM. It achieves its effect since whenever the PAM is entered it performs standard housekeeping functions. In particular a connect fault is issued after a message is left when clearing the cache or when adding/deleting a processor.

Clearing the Cache

The table of pending clears has one entry per processor. When the PAM wants to clear the cache in other processors, it places in each table entry the appropriate instruction. It does this via a stacq instruction to make sure that it is replacing a nop. If it does find an instruction other than a nop, it assumes that another processor has left a instruction and loops attempting to execute the instruction in its entry and leaving an instruction for the other processor. It makes sure the other processor enters the PAM by issuing a connect to the other processor.

Process addition and deletion.

When a processor is added, after some initialization, it enters the code to scan the VPT and find work to do. When a processor is being deleted, it checks for he request immediately priori to scanning the VPT for more work to do and disables itself. In either case an eventcount is incremented and the VPC is notified of the change.

Making the VPC runnable and processing the VPT

Whenever there is an event that requires the VPC's attention, a wakeup-waiting flag associated with the VPC is set using the **stcl** instruction. The last part of the PAM locks the VPT. The wakeup-waiting switch is cleared with an **sznc** instruction. If it was set, then the VPC is placed in the *runnable* state from the *VPC_blocked* state, using the **sznc** instruction.

The VPT is then scanned for the first (and therefore highest priority) process that is in the *runnable* state. One will always be found since there is always a lowest priority idle process available. When the entry is found, it is placed in the *running* state. A check is made to see if the process has a restriction against the current processor and if so, makes it again *runnable* and continues the scan. Otherwise the VPT is unlocked and the virtual processor is run.

Running the VP

This is the final part of processing that is done after a VP has been found in the VPTE and has been placed into the *running* state. The appropriate pointers are set in the per processor tables for storing fault data and referencing the VPTE, the clock time is saved for computing virtual CPU time and the registers are loaded. If the VP is being run on a different processor than it had last time, the cache for the current processor is cleared. Final processing is done with separate code per processor so that the appropriate SCU data may be restored. The VP is then off and running.

Process Signals (IPS)

The process signalling mechanism corresponds to the current IPS mechanism. It is implemented by setting a flag in the VPTE to indicate that an interrupt is pending. When the virtual processor is to be run a check is made to see if the flag is set and faults are permitted. If so a fault is simulated. If faults are not permitted, the action is deferred until the flag is reset to indicate that it is safe for the virtual processor to take faults again. The details of using this signal are discussed in the section on notification.

The interrupt pending flag is set by the L2TC. If a running process is to be interrupted, it is first stopped, the flag is set and then it is rebound to a VP. The choice of this method is motivated by a desire to minimize primitives available for accessing the VPTE. A tradeoff can be made between number of such primitives and the frequency with which the L2TC must unbind a VP in order to access parts of its description.

Special machine state information

This section explains how history registers, fault registers, alarm register are managed. In addition there is software state information such as the VP state which is discussed elsewhere. This will not be addressed at the moment since it is more a matter of retaining current Multics details without requiring a major changes for the PAM. Note, however, that since the PAM is aware of the VPs, it is feasible, possibly, to control history register handling on a per-VP basis (and therefore on a per process basis.

Virtual CPU time measurement and limits

Associated with each processor running a VP is the clock time at which the currently running virtual processor started running (the PAM was last exited). When the PAM is entered the starting time is subtracted from the clock time at which the virtual processor stopped (the PAM was entered) to determine how long the VP has been executing. This value is added to the value accumulating the in the VPTE. A check is then made against the VCPU limit for the VP. If the limit has been exceeded, the process is stopped for deassignment by the level two traffic controller.

As a refinement to this scheme is an estimate of the overhead involved in invoking the PAM before the clock is read on entry and after the is read on exit. This can be subtracted from the VCPU in an attempt to isolate the charge for a processor from that of running the PAM.

Timer register setting and usage by PAM

The timer register is used to make sure that the PAM gets invoked periodically so as to enforce quantum length restrictions (i.e. virtual time quota) and to make sure the VPC gets invoked so that it can advance the real time eventcount. For simplicity the PAM is run at least every 50(?) milliseconds. The alternative would be to calculate the minimum of the virtual time limit for the process being bound and the time the VPC is to be run. This would be more complicated and the additional resolution is not necessary.

Other processes

Proper operation of the PAM depends on two kinds of VP's. The first is the Virtual Processor Coordinator that is described in great detail below. It is always the highest priority virtual processor and is made runnable whenever there is something requiring its attention and therefore run immediately. Second are the lowest priority processors -- the idle processors. There is one idle processor for each physical processor. Since the idle VP is lowest priority it is run only if there is nothing else for the physical processor to do. The idle processors are quite cheap since they can share a descriptor segment or run in absolute mode without a descriptor segment. Other than that no special consideration need be given to the idle process.

The callp operator

As noted above the `callp` instruction is used to extend the operation of the virtual processor. It is implemented within the PAM. It takes an operation number in the `A-register` and a data pointer, if any, in `pointer register zero`. Like any other normal instruction, it may take faults. When the fault occurs the machine conditions are set to restart the execution of the instruction from the beginning so that there is no need to save partial state information associated with copying information into the VPTE buffers.

The operations are:

1: AWAIT takes a list of eventcount names and values (as described below under `VP1$await` and places the process in the *awaiting* state until one of the named events is notified. It is possible for one of the awaited events to be advanced while the process is being placed in the *awaiting* state. It is therefore necessary to make sure that the none of the eventcounts has passed the awaited value after the process is in the *awaiting* state. Since the process is no longer considered *running* it is necessary that no faults occur. In order to prevent faults the *absa* is used to get the address in primary memory of the counter value for each eventcount. A fault can occur during this operation in which case the normal page fault processing is done and the await is restarted from the beginning. This pointer can then be used to reference the value while the process is *awaiting*. We are assured that no fault will occur since primary memory addresses are being used for the reference and the virtual memory support is not invoked. We are assured that the address is valid since any other processor that is updating the page tables cannot assume all references to the page frame are completed until it receives an acknowledgement form the other processors. The processor performing the await will not give this acknowledgement until it finishes processing the await request.

   The real time clock is a special eventcount in that the minimum value of all such events must be stored so that the timer can be set to notify the event at the specified real time.

2: WAKE VPC is used when a change is made to a VPT entry that requires VPC attention. For example, when a message is queued for the VPC.

3: STOP is used to forcefully stop a specified process. If a process is in an atomic operation, but is to be stopped, a flag is set to indicate that it is to be stopped when the atomic operation count reaches zero.

4: BEGIN ATOMIC OPERATION is used when a process is executing a critical section of code. It increments an atomic operation counter in the VPTE.

5: END ATOMIC OPERATION decrements the atomic operation counter. If the count reaches zero and a stop is pending, the process is placed in the *stopped* state.

6: GET FAULT DATA copies fault data out of the process' state into pageable storage. Note that page faults are permitted during this operation since they are handled by another process. Segment faults are not permitted because they are handled by the faulting process and will require the use of the fault data area. Note that the atomic operation counter was incremented at the time of the fault and the process was marked as not being safe to take faults. The safe_to_take_fault_flag is reset by this operation. The atomic operation count must be decremented by restoring the processor state or explicitly ending the atomic operation.

7: RESTORE PROCESSOR STATE restores the machine conditions as specified and decrements the atomic operation counter. If this interface is not used the end atomic operation interface must be used to decrement the counter.

8: ADD CPU sends an ADD CPU message to the VPC.

9: DELETE CPU sends a DELETE CPU message to the VPC.

10: CLEAR CACHE used when an object loses encachability. Its parameters consist of a suboperation number and the page id for suboperation cache clearing by page. The suboperations are:

1. Clear PTW cache via a camp.
2. Clear SDW cache and PTW cache via cams and camp.
3. Clear PTW cache and memory cache by page - camp 4 + page id.
4. Clear memory cache, SDW cache and PTW cache with cams 4 and camp.

These are used by (1,3) page control, (2) segment control and (4) access control. They apply to all processors. The actual method by which the processors execute the instructions is explained in the section on PAM details.

11: VPC BLOCK is used by the VPC so as to cause checking of the VPC's wakeup waiting switch. It takes as a parameter the next real time before which the VPC is to be run.

The VP1 interface

The VP1 program provides a PL/I compatable interface to the `callp` instruction, the VPC and the VPT. It limits the operations the can be performed; no other interface exists. The use of the common segment name of VP1 is primarily for convenience; the entries are essentially independent.

A basic service provided by the VP1 routine is the management of assignment of level two processes (those managed by the level two traffic controller) to virtual processors. There are a number of semantic models that can be associated with this operation. The primary one is that of binding and unbinding. An alternative view is that one loads and unloads a processor state to and from a virtual processor much as one loads and unloads a process the current Multics implementation. A better understand of what is actually happening can be achieved by realizing that the bind operation is really taking a processor state description maintained by the level two TC which has no existence other than as an entry in a database and is creating a level one processor with an initial state for execution. The unload operation destroys this processor and returns a description of its final state. Key to the understanding is that the PAM does not enforce any continuity between the process description returned by an unbind operation and that provided to a bind operation. While the description is being maintained by the level two traffic controller, the L2TC is permitted to perform arbitrary operations on its description including fabricating new descriptions and discarding old ones.

VP1 communicates with the VPC via a communications queue. The queue is managed without the use of explicit locks. The `stacq` instruction is used to perform interlocking.

The information maintained in the VPTE consists of two parts -- that which is communicated via the VP1 interface and that which is internal to VP support. For convenience the portion that is passed through the interface is kept in the same format by the level two traffic controller as in the VPTE, but this is not necessary.

The Process_Description portion of the description is used to store information that maintains the identity of a Multics process as seen by the user.

```
declare 1 Process_Description based aligned, /* 16 words aligned!     */
          2 process_id bit(36),
          2 lock_id bit(36),
          2 excluded_processors aligned,
            3 excluded_processor(0:3) bit(1) unaligned,
            3 padding bit(32) unaligned,
          2 BAR bit(36),                /* For 6080 emulation         */
          2 DSBR bit(72),               /* Descriptor Segment Base Reg*/
          2 ring_alarm_word bit(36),
          2 PD_flags aligned,
            3 safe_to_take_faults bit(1) unaligned,
                                        /* Fault data can be copied?  */
            3 pending_process_interrupt bit(1) unaligned,
          2 resource_metering,          /* Metering and limits        */
            3 virtual_time_used fixed binary(71),
            3 virtual_time_limit fixed binary(71),
            3 memory_usage_meter_reference like meter_reference,
          2 processor_state,
            3 machine_conditions like mc;
```

The VP_Description contains information that is only available to the VP support and is not passed through the VP interface.

```
declare 1 VP_Description based aligned,
          2 next_VPTE like VPT_ptr aligned,
          2 VP_id bit(36),              /* Identification of this VP  */
          2 VP_state fixed bin,         /* runnable when bound        */
          2 VP_priority fixed binary,
          2 last_processor fixed bin(2), /* For cache maintainance"   */
          2 atomic_operation_count fixed bin(35), /* Initially zero   */
          2 pad16(10) bit(36) aligned,
          2 fault_conditions like processor_state,
                                        /* Communication with handler */
                    /* For simplicity I am putting the awaited events
                       in the VPTE.  Eventually they will be managed
                       separately by the VPC. */
          2 eventcounts,
            3 number_events fixed binary,
            3 event_names(4) like global_eventname aligned,
                     /* 4 = max_number_of_11_events                    */
          2 VPD_flags aligned,
            3 pending_stop bit(1) unaligned,
            3 padding bit(35) unaligned,
          2 pad8b(6) bit(36) aligned;

declare 1 VPT_ptr based aligned,        /* Pointer entry for VPT      */
          2 abs_ptr bit(18) unaligned,  /* For use in absolute mode   */
          2 rel_ptr bit(18) unaligned;  /* For use in appending mode  */
```

The VPTE itself contains both parts:

```
declare 1 VPTE based,
          2 VP_info like VP_Description,
          2 Process_info like Process_Description;
```

The awaiting_events_table is used in the interface between V1$await and callp/await.

```
declare 1 awaiting_events_table based,
          2 number_events fixed binary,
          2 events(max_number_of_ll_events),
            3 local_name pointer,     /* Only valid in owner's address
                                         space */
            3 global_name like global_eventname,
            3 value fixed binary(35); /* Value process is awaiting */

declare 1 global_eventname based aligned,
          2 segment_unique_id bit(36) unaligned,
          2 word_offset bit(18) unaligned,
          2 pad bit(18) unaligned;
```

VP1$bind

```
declare VP1$bind entry (bit(36),1 like Process_Description, fixed
                        binary(35));

call VP1$bind (VP_id, process_description, code);
```

The semantics of the bind operation has been discussed above. The caller of VP1$bind should set the appropriate flag in the ASTE to keep the descriptor segment of the specified process active. It initializes the values in VP_info as part of the transformation from the representation maintained by the L2TC and that in the VPTE. The process_state is *stopped*, the last processor is "-1" (i.e. none), and the atomic operation count is zeroed. It then uses the callp/load operation to load it into a free VPTE. The operation will fail if there are no VPTE slots available. It would be expected, however, that the second level TC will not call the primitive unless it knows that there is one available.

VP1$unbind

```
declare VP1$unbind entry (bit(36), 1 like Process_Description, fixed
                          binary(35));

call VP1$unbind (VP_id, process_description, code);
```

The semantics of unbinding has been discussed above. It issues a callp/unload operation

to request the contents of an stopped VPTE be returned. When this operation has been done the VPTE is available for a subsequent bind operation. It is expected that VP1$unbind would be used repeatedly to unbind all stopped virtual processors so that the associated process descriptions would be available to the level two traffic controller. Note that an eventcount is incremented any time a process is stopped so that by awaiting that event count the L2TC can immediately perform the unbind operation.

VP1$stop

```
declare VP1$stop entry (bit(36), fixed binary(35));

call VP1$stop (VP_id, code);
```

The stop entry is used to force a process associated with a VP to stop executing. The details a discussed in the description of the callp/stop operation. The VP1$stop operation is used whenever the level two traffic controller needs to manipulate the process' description. For example, to destroy a process, the L2TC would note that it wants a particular process destroyed. If it already has full control over the description, i.e. the process is not bound to a VP, it can perform the operation immediately. Otherwise it would issue a VP1$stop for the process. As soon as the process is stopped, the "stop process" eventcount would be incremented, VP1$next_stopped would locate the VP, and VP1$unbind would copy out the process description. For each process description returned by the VP1$unbind operation the L2TC would check the notes associated with the it and perform any necessary operations; in this case the process would get destroyed.

VP1$next_stopped

```
declare VP1$next_stopped entry (bit(36), fixed binary(35));

call VP1$next_stopped (VP_id,code);
```

This entry is used by the L2TC to get the id of the next available stopped VP. It is invoked in response to an advance on the *stopped* eventcount.

VP1$run

```
declare VP1$run entry (bit(36), fixed binary(35));

call VP1$run (VP_id, code);
```

This places a makes a stopped VP runnable. It is normally used after the VP1$bind operation.

VP1$await

```
declare VP1$await entry (1 (*), 2 pointer, 2 fixed binary(35), fixed
                         binary, fixed binary);

call VP1$await (events, number_events, advanced);
```

The parameters consists of a table of event names (pointers) and values to be awaited. The number parameter specifies the number (up until the maximum value) of events that are to be awaited. The index of the event which caused the return from awaiting is given as "advanced".

The table of event_counts is completed by filling the event name as derived by the VP interface from the segment id and the word address and passed to the callp/await operation. Note that there is a maximum for the number of entries in this table. The user level interface to VP1$await must permit an arbitrary number of event names to be specified while only passing a limited number of event names to VP1$await. The details of this are described in the section on notification.

VP1$advance

```
declare VP1$advance entry (1 like awaiting_events);

call VP1$advance (event_table);
```

As with VP1$await, the event_name is filled in. The await_value is, in this case also filled in after incrementing the associated counter with the new value. The table is then passed to callp/notify

VP1$add_cpu

```
declare VP1$add_cpu entry (fixed binary, fixed binary(35));

call VP1$add_cpu (cpu_number, code);
```

This entry interfaces to callp/add_cpu.

VP1$delete_cpu

```
declare VP1$delete_cpu entry (fixed binary, fixed binary(35));
call VP1$delete_cpu (cpu_number, code);
```

This entry interfaces to callp/delete_cpu.

VP1$crash_system

```
declare VP1$crash_system entry ();
call VP1$crash_system ();
```

Deletes all physical processors from the system, and forces one of the processors to execute a special debugging program.

VP1$clear

```
declare VP1$clear entry (fixed binary, bit(18), fixed binary(35));
call VP1$clear (suboperation, page_id, code);
```

Interfaces to callp/clear_cache to clear cache the specified associative memory.

VP1$begin_atomic_operation

```
declare VP1$begin_atomic_operation entry ();
call VP1$begin_atomic_operation ;
```

Interface to callp/begin_atomic_operation.

VP1$end_atomic_operation

```
declare VP1$end_atomic_operation entry ();

call VP1$end_atomic_operation ;
```

Interface to callp/end_atomic_operation.

VP1$get_fault_data

```
declare VP1$get_fault_data entry (1 like fault_conditions);

call VP1$get_fault_data (fault_conditions);
```

Interface to callp/get_fault_data.

VP1$restore_processor_state

```
declare VP1$restore_processor_state entry (1 like processor_state);

call VP1$restore_processor_state (processor_state);
```

Interface to callp/restore_processor_state.

VPC Operation

As noted above, the VPC is run whenever an event occurs that needs it attention. For example, a process leaving the runnable (or running) state, an interrupt event occuring or a message being sent from a process. In later implementation some of these occurances might bypass the coordinator, but for now it is assumed that all complicated low level operations involve the coordinator.

The basic operation of the VPC consists of three loops:

1. Scanning for processes by state, i.e. unsafe and exceeded limits.
2. Scanning for advanced interrupt cells. This means that there is an implicit, rather than an explicit advance done on the cells by the PAM.
3. Processing of explicit messages to the coordinator.

Note that each loop is entered only if an associated flag has been set to indicate that there may be work of the specified type to be performed. When the processing is done the VPC unbinds a set of physical processors so that they may adjust to the new state of the world. It is only necessary to unbind those processors that are running the "n" lowest priority processes where "n" is the number of processes that have been made runnable by the VPC.

In more detail, the processing consists of:

1. This loop scans the Virtual Processor Table (VPT) examining the state of each process that is found. Each *stopped* VP is removed from the chain of runnable processors and an eventcount is advanced to notify the level two traffic controller. Note that kernel processes should never be stopped. If an *unsafe* process is found, a debugging process should be notified or the system crashed. ????

2. Next the interrupt and fault counters are scanned for any that have been incremented by comparing against an earlier set stored in the VPC and the appropriate waiting processes are notified. (For the interim implementation with a single "interrupt side" processor there is an additional event counter to indicate that any interrupt has occured). As a special case of interrupt handling, the system clock can be interogated and compared with the value for the next timer event of interest.

3. Scan for messages from other processes.

    i. RUN. Places the specified VP into the runnable state and chains it into the queue of runnable VP's.

    ii. NOTIFY notifies processors that are AWAITing that counter.

iii. DELETE CPU. Leave a note for the specified processor to deconfigure itself and then unbind from any virtual processor it may be running it via a connect.

iv. ADD CPU. Leave a message telling a CPU to come to life and send a connect to it, forcing it to initialize itself.

A final note on locking. Normally the VPC looks at the VPT without setting a lock because it is the only process that may change the VPT. When it does change the VPC it loop locks to prevent conflicts with the PAM that may be searching the chain. The VPC itself is run whenever its wakeup-waiting switch is set by the PAM indicating that there may be work for it to do. This flag is reset whenever the VPC is placed in the runnable. Any events of interest that occur after this time will set the VPC wakeup-waiting switch in case it hasn't done all of its processing in its previous incarnation. Thus for example, if no paging communication buffer is available when the VPC looks and one becomes available while the VPC is running, no race condition arises because the VPC_run flag will be set anyway so that the VPC will be run again to make use of the buffer immediately after it unbinds to wait.

Also some efficiency considerations. As pointed out above it is possible to bypass some of the mechanism described above should the running of the VPC be considered too expensive. The VPC need not be expensive. Its operations are simple and it avoids the major expensive operation in PL/I, the full subroutine call. The only call it needs make is to an ALM procedure that is used for basic utility operations. This call only involves minimal housekeeping making it more efficient than a full PL/I call.

Modifications to page control

Unlike the current Multics, a page fault is not handled by the process taking the fault. This approach greatly simplifies the construction of a process because it removes the need to handle "awkward" situation such as a page fault occuring when the fault handler is copying fault data out of the VPTE. It also makes it possible to take a page fault on any page of the user's descriptor segment removing the necessity for wiring any pages of a process since the other requirement for wired pages -- external interrupt handling, is also removed by having interrupts handled by dedicated processes.

The page fault processing itself is simplified since the use of a process dedicated to this functions greatly reduces the locking problems associated with page fault handling. The modifications to page fault handling are minimal since page fault already runs in an environment that has little to do with its host process and is thus easily decoupled. Some consideration has been given to using the modified version of page control designed by Andy Huber and refined by Bob Mabee.

The PAM generates a message to the page fault process by extracting the relevent data from the SCU data. Faults on page zero of the descriptor segment are permitted. The messages is placed in a ring buffer. The format on an entry is:

```
declare 1 page_request based,
          2 pointer fixed binary,       /* In AMT or WMT */
          2 segment,
            3 astep pointer,            /* ASTE Entry */
            3 uid bit(36) aligned,      /* To make sure still same. */
          2 eventcount_index fixed binary;      /* To notify process */
```

The meter pointer is discussed in more detail below in the discussion of the Active Metering Table. When the request is queue the AMTE wire count is incremented. After the meter is incremented to charge for the processing, the wire count is decremented to release the meter. The event count is derived from the segment unique-id and the page number within the segment. This value is hashed into a wired table of page events. It is the index of this entry that is placed in the page request. The use of a preallocated table removes the problem of allocating wired storage. We can use a small table without limiting the number of outstanding page faults by not requiring that the assignments of eventcounts to paging operations be unique. There is no requirement that the event be unique, it is only a matter of efficiency. At worst, a processor may get a spurious notify, attempt to execute, and fault again.

The modifications to page control consist of:

Removal of the code that handles the fault directly as this is now done by the PAM.

Removal of the explicit interactions with pxss.

Removal of the code involved in locking the page table since this process has exclusive access to its databases.

Changing the references to metering data in the APT entries to use the AMT.

The Active Metering Table

Note: The discussion of the active metering table is included for completeness. The actual details of the mechanism are not yet fully worked out and the implementation of a layered system need not be dependent upon the current AMT design.

In a "real" system it is necessary to account for resource usage and to limit such usage against predetermined limits. In the current Multics system, many of the resource measurements are associated with processes. Since the processes are known to the lowest levels of the system, not even deactivated, the Active Process Table (APT) has become a repository for such information, or at least the resource measurement information.

In the multilevel system, only virtual processors exist at the lower levels. Since the processes assigned to this virtual processors do not exhibit the continuity of the present Multics processes it is necessary to develop a separate mechanism for measuring resource usage. Furthermore, if we look beyond just supporting the current measurements, a restructuring of the metering would permit the offering of improved mechanisms such as resource limits and shared meters at the base level; mechanisms which have been proposed in the past but which have not been implemented.

There are two primary components to resource measurement -- the long term and the short term. The long term measurements in current Multics are stored in the PDT (Project Definition Table) and consist of dollar usage and more detailed resource usage measurements. Short term measurements are maintained in the APT. Periodically the Answering Service copies measurements from short term to long term storage.

In the proposed Multics a similar mechanism is used except that the choice of short term meters is more explicit and not directly related to processes. At present we are mainly concerned with meters that must be available to ring zero[ii] -- those that correspond to the APT information. In addition, to simplify the design of page control, the meter (and limit) for storage system usage is also of interest. For the duration of its existence, each such meter resides in the Active Meter Table. It is only necessary for a meter to exist as such while the resource it is measuring may incur charges. For example, the meter of a process' processor usage can only be incremented while the processor is bound to a VP. Thus the level two traffic controller can create the meter at the time that it the process gets assigned to a VP and destroy it (after reading out the value) when the process is deassigned[iii]. In contrast a process can incur memory usage charges after the process has been

---

[ii] Need better term

[iii] In fact, the VCPU meter is a special case and is kept in the VPTE in the current PAM design; but could reasonably be incorporated into the AMT mechanism as soon as the operation of the AMT is better described, i.e. when I finish writing this section

deassigned from its VP. A third example of a meter is the storage quota meter. Since this meter must be accessible from page control when assigning additional pages to a segment, it seems logical to associate the information with the wired AST entry. Because the meter is actually shared by Multiple segments, it is actually kept separately in the AMT. Note that as a benefit of this aproach the quota limit is independent of the directory hierarchy and that storage system usage can be associated directly with accounts instead of just to superior quota.

Note that the meters described thus far share a special property - they must be accessible without taking a page fault; i.e. they must be wired. This is accomplished by maintaining a Wired Meter Table (WMT).

An entry in the Active Metering Table takes the form:

```
declare 1 AMTE based,
          2 id bit(72),
          2 value fixed binary(71),
          2 limit
            3 limit_set bit(1),
            3 value fixed binary(71),
          2 eventcount fixed binary(71),
          2 wire_count fixed binary;
```

When a meter is to be incremented (via amtm$add), the meter id is used to hash into the WMT and then the AMT to find the entry. If none is found, one is created in the AMT. To make the search more efficient, a meter_reference is used which contains a meter_index in addition to address the table entry. When the entry is found via the index, it is checked to make sure the meter_id in the entry matches that in the reference, if it does not, the hash search must be used and the index is updated to make the next reference more efficient.

```
declare 1 meter_reference based,
          2 index fixed binary,      /* Index in AMT or WMT */
          2 home fixed binary(1),    /* AMT or WMT          */
          2 id bit(72);
```

A meter may reside in either the AMT or the WMT, but not both in order to make limit checking work. When the wire count changes to or from zero the entry is moved. This move is not necessary if the meter is being created in one or the other, or is being read and cleared.

The AMT is managered by the active_meter_table_manager (amtm). The following entries are available.

```
declare amtm$set_limit entry (1 like amte, 1 like meter_reference,
                              fixed binary(35));

call amtm$set_limit (amte, meter_reference, code);
```

As noted above, meter entries are created when an attempt is made to use them. For entries such as page quotas, it is necessary to initialize the entries with a limit value. It is necessary for programs setting and using limits to cooperate such that programs do not check limits unless the limits have been set. For example, as part of activating a segment, a quota limit is set in the AMT. This entry is cleared when all segments sharing that limit are deactivated.

```
declare amtm$read entry (1 like amte, 1 like meter_reference, fixed
                         binary(35));

call amtm$read (amte, meter_reference, code);
```

Returns values for the specified meter entry. If the entry does not exist, zeros are returned for the values.

```
declare amtm$read_clear entry (1 like amte, 1 like meter_reference,
                               fixed binary(35));

call amtm$read_clear (amte, meter_reference, code);
```

Same as the read entry, except clears the value. This is the entry used to read a meter out so it can be updated in a higher level table. The AMT entry may be deleted if it is not wired and does not have a limit set.

```
declare amtm$read_clear_limit entry (1 like amte, 1 like
                                     meter_reference, fixed
                                     binary(35));

call amtm$read_clear_limit (amte, meter_reference, code);
```

This entry is similar to the previous but also clears the limit setting so that the entry may be deleted from the AMT if not wired.

```
declare amtm$add entry (fixed binary(71), 1 like meter_reference, fixed
                        binary(35));

call amtm$add (value, meter_reference, code);
```

Adds the specified value to the given meter. A code is returned if the value exceeds the meters limit. If the meter does not exist, it is created.

```
declare amtm$add_conditionally entry (fixed binary(71), 1 like
                                      meter_reference, fixed
                                      binary(35));

call amtm$add_conditionally (value, meter_reference, code);
```

This is like the add entry, except the meter value is left unchanged if the limit is exceeded.

```
declare amtm$wire entry (1 like meter_reference, fixed binary(35));

call amtm$wire (meter_reference, code);
```

The wire count for the specified meter is incremented. If the meter is already in the AMT, it is moved to the WMT.If it sdoes not exist at all, it is created in the WMT.

```
declare amtm$unwire entry (1 like meter_reference, fixed binary(35));

call amtm$unwire (meter_reference, code);
```

The wire count for the specified meter is decremented. If the count reaches zero, it is moved from the WMT to the AMT.

```
declare amte$unwire_read_clear entry (1 like amte, 1 like
                                      meter_reference, fixed
                                      binary(35));

call amte$unwire_read_clear (value, meter_reference, code);
```

Combines unwire and read_clear.

Notification and Events

The basic mechanism for coordinating processes in the proposed system is the event. More precisely, event counts are used to store state information about events.The eventcounts are discussed in detail in a CSR/RFC by Dave Reed and Raj Kanodia. When an event occurs the value of the eventcount associated with the event is *advanced.* A process interested in the occurance of the event can *await* this advance.

Eventcounts are identified by eventcount names. To the user an eventcount is simply a word in memory and thus its name is its address. To convert this into a system-wide address the segment number is replaced by the segment-unique id. The eventcount can then be referenced by the system-wide name in order to do a notification. The actual reference to the value of the eventcount within the process awaiting or advancing the primitive is done using the pointer for efficiency.

Eventcounts form a robust mechanism because, though a process may await a transition, the eventcounter itself always maintains its state for later examination. Since the counter is monotonically increasing the *await* operation can be implemented by simply comparing the current value of the counter with a previous value. If the previous value has not been surpassed the process can loop waiting for the change, or can go blocked. This block is actually implemented via the `callp/await` primitive described above. Complementary to going blocked is the mechanism for getting awakened. This is the notification mechanism.

The notification is performed by the VPC as a result of a `callp/notify` operation. This primitive is invoked by the `VP1$advance` interface. Note that only the advance interface is available outside the PAM. While this is not strictly necessary it does preserve the semantics of eventcounts. When the VPC gets a message to perform a notification, it scans the VPTEs which are in the *awaiting* state and places them in the runnable state. For efficiency, the VPC can actually check to make sure the value awaited has been reached since the value is copied into the VPTE, but this is not strictly necessary since the VPC can simply compare eventcount names.

Spurious notifies are not harmful since the `callp/await` primitive checks the values anyway before returning. `callp/await` also checks the eventcount values after putting the process into the *awaiting* state to prevent any loss of notifies sent just before the process entered the awaiting state.

Eventcounts associated with interrupts and page fault processing completion must be wired and preallocated. To simplify this a Wired Event Table is maintained. We can go further and require that all events originating at level one be in this table. Note that, unlike current IPC, the use of a wired table does not have the danger of overflowing since no messages are placed in the table, eventcounts are simply incremented.

We can take advantage of the restriction on level one originated requests when implementing the

level two primitive for event counts. Observe that there is a fixed maximum for the number of events upon which a process may wait. The user interface need not, and should not, have such a restriction. The level two traffic controller can implement its own await/notify mechanism similar to the lower level mechanism except using virtual memory to get around the restriction on the number of events.

A level one process (i.e. a kernel process) can simply use the VP1 event count interface (advance and notify) directly. For level 2 processes, there is a VP2 interface for these primitives. Since a level two process may have an arbitrary large number of events and may be unbind from a VP while awaiting, it is necessary for the level two interface to provide much of the functionality of the interface. To aid level two a special event count is provided that is advanced whenever a level one event count is advanced, the *outward_signal* counter. This is discussed in more detail in the description of the implementation of the level two traffic controller. Other event countes used for communicating with the level two traffic controller include the *stopped* event advanced whenever a VP is stopped and the *clock* event that is advanced at fixed intervals.

As described above eventcounts are passive in that they don't affect a process unless the process examines its value or *awaits* an *advance*. This is not sufficient to implement the current IPS mechanism. What is needed is a means of faulting a process so that it can examine eventcounts which it thinks are important. This consists of setting a process' pending interrupt flag while unbound at level two. When the process is to be run, the flag is examined by the PAM which will cause a fault to be simulated. Note that the fault itself doesn't tell the process what has happened; the process is simply told that something of immediate interest has occured. To give the effect of current IPS, there would be an eventcounter associated with the terminal I/O channel for quits, the real time clock and the virtual clock.

The Level Two Traffic Controller

The lowest levels of Multics described above do not provide all of the functionality of the current system. The implementation requires a second level of control that multiplexes the virtual processors among user processes. This second level is conceptually much like the lower level in that it multiplexes a limited number of processors to give the effect of a larger number. While the first level emphasises simplicity, the second level emphasises function. The second level removes restrictions on the number of processors provided and the number of events that can be observed. It is able to do so because it can make use of the virtual memory mechanisms for managing its databases. Note that the term process is used in the conventional Multics sense, of a user's address space and control point. The level two process is representation of the logical processor that executes a user's instructions.

The Implementation of old IPC and IPS

Basic to the design of any change to Multics is the requirement that the new mechanism provide an external interface that is compatable to any preexisting interface. The Interprocess Communications Mechanism of Multics is basic to many programs and must be supported.

IPC is relatively simple to implement and offers a subset of the facilities of the eventcount mechanism. Most significantly IPC lacks the access controls afforded by using normal memory words a means of communications and coordination. To implement IPC a per-process segment of eventcounts associated with IPC channels can be maintained. In addition a per-system segment could be used to transmit messages between users. An alternative is to provide each process with a segment for receiving its messages so that the access control can be used.

Much of the complexity of IPC comes from the requirements of wired programs and programs requiring a very high degree of efficiency. Since the wired programs will be converted to use eventcounts, the IPC implementation is greatly simplified. Similarly for programs using fast IPC channels, they can be converted to use eventcounts, though they can still operate using IPC during a transition period.

The implementation of IPS has been discussed in the section on notification. The mechanism has been generalized to separate the occurance of the signal from the message associated with it. Thus one is not limited to the signals currently defined in the APT entry. For example, the quit signal can be associated with the terminal as an I/O device without requiring that it have special significance as the process' controlling terminal.

The IPC facility offers an ability not offered by event counts alone -- the sending of mesages in addition to the wakeup. This can be accomplished by using the message segment facility accompanied by eventcounts within the message segments.

Implementation

Both top-down and bottom-up views of the implementation of the layered system are applicable. The top-down views entails examining the existing Multics implementation and determining what one must change to retain is functionality. The section on **initialization** examines the implementation from the bottom-up view. The following section on **transition** examines the implementation from the view of modifying and preserving the existing Multics system.

Initialization

The bottom-up view begins by recognizing that level one of the layered Multics is sufficient for supporting a simple operating system directly without the features provided by level two. In fact this is an environment that is much more sophisticated than BOS in that it permits the use of processes and programming in PL/I.

By making the first stage of implementation the programming of an environment consisting of just level 1 primitives. An environment can be brought up without requiring the modification of the existing Multics. Most importantly, such an implementation result in a running system that can support a set of debugging tools for the later software. The psychological value of having a completely running piece of software should not be ignored. The level implementation also provides a starting point for the initialization of Multics itself and is thus a necessary first step.

The level one implementation consists of relatively few programs:

1. A program to initialize the level one system within collection one. Associated with this is a program to generate a relocation dictionary for the PAM. In addition to initializing the PAM tables, the program also creates processes for the VPC, the idles processes and an interrupt side process.

2. The PAM.

3. The VPC.

4. An interrupt side process. In order to simplify implementation I/O programs will continue to run much as they do now except all programs that normally run in response to interrupts will run in a single processes in response to the correspond eventcount being advanced. The old interrupt handlers themselves should be able to run unchanged.

5. A debugger.

That is all that is strictly necessary. An additional nicety might be to implement the existing BOS within a process so that its functions can slowly be spread to multiple processes without the need to continue to support a second 68/80 operating system and without the alternative of rewriting all of the code from scratch.

Initialization consists of loading the kernel processes necessary to support the full level one environment and then the ones needed for level two. There is a discussion on page 39 of creating VP's as necessary as part of the operation. To fill out the level one environment the following functionality must be brought up:

1. Disk Control

2. Segment Control

3. Page Control

4. The Level 2 Traffic Controller

Once the level 2 traffic controller is brought up Multics is essentially running. An answering service process can be created to create user processes. Given that processes can be created easily, the answering service does not need the primacy it currently enjoys.

Transition

One question that must be considered if the implementation of the two level traffic controller is to be taken seriously is that of how to get from the current implementation of Multics to the new one. The difficulty is that a complete transition is necessary. This is not an insurmountable obstacle in that we have had such transitions in the past as in the case of the new storage system and earlier file system flag days. While the need to convert over completely is present, the difficulty is not comparable to that of a major change to the file system. Most of the Multics system will continue to operate as it presently does. The changes consist of

I. Changes requiring new software

1. A level one initialization program must be written.

2. The basic mechanisms of the PAM, VP1 and VPC must be implemented. The VPC would be implemented in PL/I.

3. The initialization path must be modified to build up a system from one running at unadorned level one to a full Multics environment.

4. The level two traffic controller must be implemented While it must acquire all of the functionality of pxss, the level two traffic controller function is less critical -- the vast majority of the scheduling decisions are made by the PAM and the VPC. Thus the initial implementation need not be highly optimized for demonstration of its feasibility.

5. A primitive version of the amtm must be implemented to support basic accounting functions.

II. Modifications to existing software

1. A replacement must be provided for IPC using events.

2. Page control must be removed to its own process. Much of the work has been done already. This task is simplified by the fact that the page control environment is already very constrained so as not to be dependent upon the process in which it is a parasite. This is discussed in detail on page 23.

3. The interrupt handlers must be moved to their own processes. As with page control, they already operate in a constrained environment and thus providing them with their own process will not deprive them of features and will simplify them by the removal of the need to do direct interrupt handling and will remove the need for separate interrupt side and user side components. As an interim implementation all interrupt side programs

can be written unchanged within a single processes with only `iom_manager` begin modified.

4. pxss would simply be removed from the system.

5. System initialization must be modified and possibly redone. Much of the existing software can be used. For example disk support must still be initialized. The initialization would, however, be done as part of setting up the disk control process.

6. Present H-Procs could be simplified by replacing them with kernel processors.

7. The accounting software must be supported.

Extensions

The thesis has been concerned mainly with presenting a clean model processor multiplexing. In actual implementation some additional issues can be considered. Some of this are simple extensions and others represent a different point of view on the part of the implementor.

I. Robustness

The layered implementation provides a much cleaner structure than the current Multics system. This structuring provides an environment in which the implementation of features to make the higher levels more robust by providing a low level in which the implementation of such support facilities is simplified

1. A Level 1 debugging process.

2. Ability to recover from trouble faults -- spare repair processes.

3. Ease of timeouts and error recovery by I/O processes.

4. Daemon kernel processes.

II. Taking advantage of the implementation

This section lists some ways of taking advantage of the existing software in implementing facilities on Multics.

1. Waiting on messages.

One can associate an event counter with each message segment (or mailbox) that gets advanced whenever a message gets placed in it. This is an effective and much more powerful replacement for IPC. Some of the advantages include the ability to have InterProcess (message) Communication with access control. There is also no limit to the number of processes that can be awaiting the message. Since the transmission of the message is via a segment in the hierarchy the problem of setting up and communicating IPC channel numbers is eliminated. One final advantage of the proposed implementation is that any process with access to await a message can specify immediate attention (i.e. an interrupt) when the value is changed.

These facilities can provide a basis for a number of features. It is possible to implement notification upon the receipt of mail. Alternatively a server can be awaiting messages and then create processes the handle them (i.e. potential processes).

III. Changes to the model

1. One of the basic assumptions in the model is that Virtual Processors at level 1 are neither created or destroyed. This assumption actually complicates the system by requiring that all uses of kernel processes be predetermined. In particular the initialization of the system must be carefully planned with respect to the use of VP's. This is similar to requiring that all Multics tables used in managing the system such as the AST be determined when the system is generated, as opposed to during initialization as is presently done.

   The reason for the restriction on VP's comes from two primary sources: the need for simplicity and the attempt to carefully structure management of memory. The simplicity argument is not one of absolute simplicity but a choice of what to simplify. One must pay the price of carefully preplanning use of these processors. In particular when one dynamically reconfigures the system to add a new device (logical or physical) and one needs to dedicate a virtual processor to its management, one cannot tolerate the lack of availability of such a processor, nor can one reduce the number of virtual processors managed by level 2 since that would change the level of multiprogramming of the system.

   While the requirement of a program that is able to assign primary memory addressable by the PAM might add additional complexity to the system, it does not affect the layering of memory memory management since it is not dependent the management of virtual memory. In fact in an ideal processor such a mechanism would be simply structured such that it can be shared by both the page frame allocation mechanism and the primary memory allocation interface. The 68/80 processor is a little more complicated in that the PAM is unable to easily address more than the first 256K of memory. But this requirement is already present for I/O buffer management. To summarize, this mechanism must exist anyway for performing I/O and fits within the structure of the memory management hierarchy so that it does not really add complexity to the system.

   Thus the ability to dynamically create virtual processors would simplify the implementation without affecting the layered model of the system.

The existing implementation

A test implementation of the basic level one portion of the two level system has been completed. It supports the functions of level 1 with the exception of paging and the handling of faults reflected to user processes.

It is a modification of collection one of Multics initialization. Interrupt and fault processing have been replaced by the PAM and the VPC. The VPI interfaces for "run", "await", "advance", "crash_system" and "clear_cache" are supported. The system spawns kernel processors (including the VPC and the idle processors).

The only I/O device supported is the console typewriter. The interrupt side processing for the I/O is performed in a processor dedicated to that function. The stopped (to indicate a processor entering the stopped state) and the clock events are supported. The idle processes share a descriptor segment.

The following changes were made to the system:

1. The PAM was implemented to handle all faults and interrupts.

2. The VPC was implemented to:

    a. Convert interrupts (as noted by the PAM) into notified events.

    b. Manage the clock event.

    c. Advance the stopped event when a VPT stops.

    d. Process run and notify messages.

3. `init_collections` was modified to call `init_basic_ll` and not to call `initialize_faults`. PVT initialization and tape initialization was also eliminated.

4. `init_basic_ll` was implemented to initialize the PAM and the VPT. It spawns the VPC and idle processors.

5. `create_kernel_process` was implemented to initialize a VPT entry.

6. `init_ll_get_segment` was implemented to create segments for processes' dsescriptor segment and pds.

7. The prds was eliminated.

8. privileged_master_mode_ut was modified to use the pam for entering BOS and for clearing the cache and associative memories.

9. init_sst (and the sst) was modified to remove masks was for inhibiting and generating interrupts.

10. pxss was eliminated. So was tc_data.

11. The fim and ii were replaced by stubs since at this point the system is unable to handle reflected faults. These routines will have to be redone. The same goes for emergency_shutdown and related programs.

12. The pds was cleaned up to remove unneeded storage for fault data in the header.

13. VP1 and VP_util were implemented to interface to the pam and to support the idle process.

14. run_basic_ll was implemented as a process to give periodic status messages. The moritician was implemented in a similar manner to monitor stopped processors. It uses status_report which, in turn, uses octal for typeouts.

15. interrupt_process_driver was implemented to manage the interrupt side process.

16. ocdcm_ was modified to use eventcounts to govern contention on locks.

17. A pxss was implemented to provide a write-around to addevent and notify primitives.

Glossary

Some suffixes are commonly associated with abbreviations. "E" is used to indicate an entry in table and "p" is used to designate a pointer.

AD The Add Delta modifier causes the effect address to be computed using an indirect word and increments the value of the word by a specified amount. It is of interest because it is atomic with respect to other instructions using the modifier.

AMT Active Meter Table.

amtm Active Meter Table Manager.

APT Active Process Table. The APT in current Multics would be replaced by three databases. At levels zero and one there is the VPT. The level two traffic controller maintains the APT, and for efficiency, an IPT.

AST Active Segment Table.

BOS Basic Operating System. This is a standalone operating system for the H68/80. It provides utility functions when the full Multics environment is not available. Such as when actually bootloading or debugging Multics.

callp "Call Processor", an instruction implemented using the faulting mme4 and interpretted by the PAM.

camp Clear Associative Memory PTWs.

cams Clear Associative Memory SDWs.

IPT Inactive Process Table. This is maintained by the level two traffic controller and corresponds to the APT, except that for reasons of locality the entries that are referenced infrequently are moved into the IPT.

L2TC Level Two Traffic Controller.

mme4 The Master Mode Entry 4 instruction simply causes a fault. The fault handler will interpret this to be a callp operation if the fault is taken while executing in a priviliged segment.

PAM Processor Assignment Manager.

PBM Processor Binding Manager; older term for PAM.

PDT Project Definition Table.

PTW Page Table Word

SDW Segment Descriptor Word

stcl Store Instruction Counter plus one. This instruction is used to set a flag to be tested with sznc. It is of interest because it does not affect registers, is atomic with respect to sznc and stores a nonzero value.

sznc Set Zero Negative and Clear. This instruction is used to test a flag set by **stcl**. It does not affect registers and rests the flag after test Since it is atomic with respect to **stcl** it is good for low level synchronization primitives.

VCPU Virtual Central Processing Usage. A measure of the time assigned and executing.

VP Virtual Processor.

VP1 The procedure that interfaces to the **call p** instruction.

VPC Virtual Processor Coordinator.

VPT Virtual Processor Table.

WET Wired Event Table

WMT Wired Meter table

FURTHER RESULTS WITH MULTI-PROCESS PAGE CONTROL

by R. F. Mabee

This memo updates performance measurements reported by Andy Huber in his recent thesis "A Multi-process Design of a Paging System", now available as MAC-TR 171. The PL/I code is brought up to date with NSS, and improved by removing many external subroutine calls from the critical page fault paths. This gives a performance improvement of about 30%. Many detailed measurements have been made; the results are used to determine where time is spent in both this and the standard page control.

This should be the final report on this project, as no further development is expected.

I. Review

In one chapter of his thesis, "A Multi-process Design of a Paging System", Andy Huber reports measurements made on two versions of Multics, one using his multi-process page control (MPPC) and the other using the standard page control. The former has two H-procs (fast system processes) that run the resource freeing functions of page control, and perform some operations for segment control (typically truncating a page table). Most of the code was rewritten in PL/I, except for the bulk store DIM, a piece of the fault handler, and the system interrupt handler, which are essentially unchanged. The results show comparatively poor performance by the MPPC in two respects:

1) The number of page faults (during a standardized benchmark run) is much higher.
2) The CPU time spent by the PC processes is excessive, doubling the time per page fault.

The increase in page faults can be attributed to the reduced size of the paging pool. The wired stacks, the RWS buffer, the increased size of the PL/I code, and the free core list reduce the paging pool by 10 to 20 pages. This could be cut in half by careful tuning of the algorithm, and becomes unimportant in systems with larger memory. Huber also points out that MPPC disconnects pages before writing them, while the standard PC leaves modified pages connected for an extra lap. If modified pages are more likely to be referenced than unmodified pages, then the standard PC will have fewer page faults.

The increased paging isn't very interesting, because it's readily explained and wouldn't much matter in more reasonable configurations.

For comparisons of CPU time, we adjust the sizes of the paging pools so that the metering run takes about the same number of page faults with each version of PC.

There are two special processes in MPPC: the core manager and the paging device (PD) manager. They perform functions that are mostly done at page fault time in the standard PC, so the MPPC should spend much less time in the page fault handler. Instead, the time is slightly higher (3%). This is the effect of using PL/I. Huber predicts a 40% improvement by replacing external calls with internal calls, with the resulting times shown in the last column of the table.

	Standard PC	MPPC	Predicted
Page fault handler	1973	2043	1226
PC processes	--	2641	1585

Table I. usec per fault. Adapted from Huber.

Three modifications should be made to these numbers for more accurate comparison. In both versions of PC, the fault time meter is updated about 500 usec too soon, before the bulk store read (if any) is posted. There is no question that the time should be accounted to the page fault handler; it's just a bug. Also, the time spent by the PC processes on operations other than page faults (primarily truncation) should be subtracted from the totals; by reasonable extrapolation from more recent measurements this amounts to 336 usec per fault. Thirdly, the cost of interrupt handling and of inter-process swapping (getwork time) should be included; again, these numbers are taken from recent runs. The corrected figures appear in the next table. Comparing the total times, we find MPPC just under twice as expensive.

	Standard PC	MPPC	Predicted
Page fault handler	2473	2543	1726
PC processes	--	2305	1383
Interrupts and getwork	445	684	684
	2918	5532	3824

Table II. usec per fault. Approximate corrections added.

II. Recent changes

For this new series of experiments I used version 28-10 of Multics, with both standard and MP page control subsystems. Among other changes since Huber's experiments was the introduction of NSS (New Storage System), with many consequent effects in page control. NSS resulted in a 200 usec improvement in page fault times for the standard PC, although no corresponding improvement was observed in MPPC. I believe this shows the benefit of the long, careful tuning process applied to standard PC; MPPC must compete without such tuning.

Page faults in the IPC benchmark have increased by 10% during this time, probably due mostly to online changes and only somewhat to reduced paging pool. As before, timing measurements are made with paging pools adjusted so the two versions of PC handle about the same number of faults during a standard metering run.

The final version of MPPC is optimized by embedding subroutines as internal procedures of the page_fault and core_manager programs so that most external calls and redundant assignments (i.e. "sstp = addr (sst$);") are avoided. If all of the external calls could have been removed, then the predictions in Table II would be realized. However, the calls to ALM subroutines (such as the bulk store DIM) couldn't be removed. Moreover, some of the calls that Huber counted to make his

predictions are executed only once in several page faults; in that case the cost per fault is proportionally lower, reducing possible optimization.

Six external calls were removed from page_fault, leaving only four calls, all involving ALM. Seven external calls were removed from core_manager, leaving four to or from ALM. However, three of the calls removed were executed only half the time (when a page must be written). If each external call costs 70 usec, the net gain is only 800 usec, or 14%. The rarer cases aren't optimized, on the grounds that a small improvement in an unusual case wouldn't affect the average times very much. Specifically, only PD reads, page creations, virtual writes, and PD writes not requiring PD allocation are optimized. This handles 84% of the cases.

As another optimization, the core_manager page removal algorithm is made more efficient, although complex, by starting writes for several pages before waiting on any. The overall results are shown in Table III.

	28-10 Standard	Original MPPC	Predicted by Huber	Observed by me
Fault handler	2531	2543	1756	2162
Core manager	--	1985	1191	1272
PD manager	--	320	192	312
Interrupts and getwork	445	684	684	684
	2976	5532	3823	4430

Table III. usec per fault. Results of optimizations.

III. Where the time goes

It is possible to attribute the total CPU time spent on a page fault to the various functions performed. The bulk store DIM alone accounts for about 500 usec per read or write in both systems, which

is surprisingly high. This apparently indicates that the I/O greatly slows the CPU by competing for memory cycles. Of course, this behavior should be unique to the test configuration combining MOS memory with bulk store. Depending on whether the CPU is locked out entirely or just slowed down, this effect may also be slowing down the rest of PC. Another 500 usec is spent (mostly by page$done) to report completion of the I/O. In the following table, the measured time for the standard PC page_fault is arbitrarily divided between freeing core and real page_fault in the proportion measured for the MPPC system. The unusual cases of page creation or forced write to disk are ignored.

	28-10 us/event	28-10 us/fault	MPPC us/event	MPPC us/fault
Real page_fault	482	482	1162	1162
Getwork awaiting core	--	--	637	54
DIM and page$done	1000	1000	1000	1000
Getwork awaiting disk	692	69	637	64
Interrupts, disk read	1921	192	2102	210
Getwork for pre-empt	692	69	637	50
Freeing core frame	297	297	715	715
DIM if must write	1000	557	1000	557
Getwork by core_manager	--	--	637	124
Freeing PD record	580	83	1400	200
DIM if must RWS	2000	112	2000	112
Getwork by pd_manager	--	--	637	56
Interrupts, RWS	1921	115	2102	126
		2976		4430

Table IV. Detailed breakdown of page fault cost.

The total CPU time per fault for MPPC is 1454 usec longer, or about 49%. Approximately 230 usec of the excess is spent in getwork when any process has to wait for a PC process to refill some free list, or when the PC process is done and goes to sleep. Perhaps an

equal amount (unmeasured) is spent in calls to perform the inter-process communication required for the PC processes. An estimated 300 usec represents the effect of less common paths that I didn't bother to optimize, and the cost of putting free frames on a separate list, and the cost of the extra metering done in this version. The rest of the excess (estimated at 700 usec) is directly caused by using PL/I to express the algorithms, which apparently increases the execution time of comparable operations by about 80%. (Note that Huber chose PL/I for ease of implementation, and not for performance.)

One important factor adding to the cost of PL/I is the frequent use of the pointer built-in function (to follow the many threads used by PC). In the ALM version this is done by one instruction, loading an index register. The PL/I compiler optimizes to shorten the generated code; this is not always best for execution speed. Furthermore, the ALM version optimizes register usage over a much larger scope. Mostly these are problems inherent in the use of PL/I, so (unless some gross bug is found) the best performance that might be achieved must still be 20% poorer (in total CPU time per fault) than the standard PC. It's worth noting that the interrupt times for MPPC are only slightly higher (181 usec). The system interrupt handler and disk DIM (both unchanged) use most of the time; the difference is in page$done, a very short procedure converted to PL/I for MPPC. Its execution time is around 400 usec, so the 80% PL/I overhead is still consistent.

In the test configuration, the page fault rate is somewhat less than 100 per second. Since the excess time for MPPC is 1454 usec per

fault, it should cost less than 145400 usec per second, or only 14% of the elapsed time for any run. However, overall system performance is not that much worse. In fact, the faulting process is delayed 369 usec less by the fault (from Table III), so it seems to run faster, and can respond to interactions faster (if it needs only a few new pages).

The PC processes sometimes run during time that would otherwise be idle. The benchmark results show this effect clearly if the working set estimator is enabled -- that reduces multiprogramming and increases idle time, so the MPPC system completes the benchmark in just 8% more elapsed time. (Tuning parameters: WSF = 1, Max Elig = 4; about 150 pages; 23% idle with standard PC.) The MPPC will provide faster service than the standard PC if there is enough idle time. If the PC processes always take what would otherwise be idle time, the page fault costs 369 usec less; if they never do, the fault costs 1454 usec more. At a point in between, the extra cost of MPPC is zero; this happens if the PC processes take idle time 80% of the time. Thus MPPC performs better than the standard PC if there is at least 80% idle time.

The paging function is exercised so heavily in the tiny test configuration that its cost is exaggerated in importance. A system with much larger main memory and no bulk store, which seems to be the right approach for Multics, might, for example, take only ten page faults per second per CPU. In this environment MPPC (minus the PD process) would cost only 4% of the total time, versus 2.8% for the standard PC. The reduction in the paging pool caused by maintaining a free list (in MPPC) would also be unimportant in such a configuration.

Since choosing the right page to evict would become relatively more important than doing it fast, alternative strategies should be tried, and for such experiments the modularity, readability, and PL/I-ness of MPPC make it ideal.

IV. Conclusions

First, the negative recommendations: MPPC as coded is not suitable for installation on a thrashing system like MIT-Multics. It is not ready for use anywhere because of glossed-over NSS issues, incomplete error handling, and just plain bugs. I have no intention of updating the code to more recent Multics releases than 28-10.

There are many positive results. The cost of the inter-process communication and swapping is not too bad (400 usec per fault?), and it could be made much lower by making the free lists longer. (The measurement runs were made with a maximum of 12 free cmes on the list. Because of the interaction with paging rate this size free list would be used only with paging pools from 500-1000.) The delay seen by a process when it faults is slightly reduced. The PL/I version of page control is available as a better base for experimentation and metering than the ALM version.

It turns out that the cost of using general-purpose processes and inter-process communication facilities, while small, is intrinsic. This cost would probably not be much reduced using another implementation of the process, such as Dave Reed's Virtual Processor, since a lot of the cost is in unavoidable overhead of process switching or of calls to perform IPC. Many of the IPC operations either implement a cross-process call to a specific routine, or merely

indicate that (say) the core_manager should be run sometime soon to free up more core frames. The latter function could be more cheaply implemented, at the expense of modularity, if the scheduler called the core_manager directly just before going idle. Of course, if the core_manager isn't a real process, it loses the ability to wait on I/O or on a lock.

By far and away, the biggest performance problem is the use of PL/I. It has already forced a non-modular design for the main programs, by imposing a stiff penalty for good design; it also handles the list-structured objects of page control very poorly. In order to obtain better performance, I would have to rewrite the programs to use constructs for which the code is known to be particularly good; that means picking out the machine language sequence I want first, then fooling the compiler into emitting it. It just isn't worth writing any program in higher-level language if its performance is so important and the language so poorly suited.

Let us momentarily suspend disbelief, to consider an ALM version of MPPC. It should execute similar functions at the same speed as the standard PC, so the extra cost is just the 400 usec presumed for IPC and swapping, or only an 8% increase in CPU time per fault. The delay at fault time becomes 1049 usec less (from Table IV), so overall performance is improved for any load up to 72% (i.e. more than 28% idle). In fact, if the IPC and swapping were optimized as previously suggested, the overall performance might be improved at any realistic load.

Even the ALM MPPC would cause some loss in throughput if there were no otherwise-idle time to give to the PC processes. In the face

of strong real-world emphasis on execution speed, it's sometimes hard to explain why the program with good organization and modularity, clearly expressed in higher-level language, is better than its assembly language predecessor. We have no way of measuring the intangible benefit of any such improvement or of weighing it against a known cost in CPU cycles or dollars. All we can fall back on is the general philosophy, "Good is better than evil, because it's nicer."