MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

January 17, 1972

MULTICS: THE FIRST SEVEN YEARS[*]

by F. J. Corbató
C. T. Clingen
and
J. H. Saltzer

Abstract

In 1964 planning began on the development of a prototype of a computer utility. The aspirations for this system, named Multics (for Multiplexed Information and Computing Service), were described in papers presented at the 1965 Fall Joint Computer Conference. Implicit in those papers was the expectation of a later examination of the development effort. From the present vantage point, however, it is clear that a definitive examination is beyond possibility in a single paper; only some of the possible topics can be discussed. First we will review the goals, history and current status of the appearance of the Multics system to its various classes of users. Finally we will describe some of the insights which have come out of the development activities.

PREPRINT

This memo is a preprint of a paper to be presented
at the 1972 Spring Joint Computer Conference,
May 16-18, 1972.

MULTICS: THE FIRST SEVEN YEARS*

by F. J. Corbató

Massachusetts Institute of Technology

Cambridge, Massachusetts

and

C. T. Clingen

Honeywell Information Systems Inc.

Cambridge, Massachusetts

and

J. H. Saltzer

Massachusetts Institute of Technology

Cambridge, Massachusetts

Paper #

(617) 864-6900, Ext. 6001

545 Technology Square - Room 513

02139

(617) 491-6300, Ext. 201

575 Technology Square

02139

(617) 864-6900, Ext. 6016

545 Technology Square - Room 519

02139

In 1964 planning began on the development of a prototype of a computer
utility. The aspirations for this system, named Multics (for Multiplexed
Information and Computing Service), were described in papers presented
at the 1965 Fall Joint Computer Conference. Implicit in those papers
was the expectation of a later examination of the development effort.
From the present vantage point, however, it is clear that a definitive
examination is beyond possibility in a single paper; only some of the
possible topics can be discussed. First we will review the goals, his-
tory and current status of the Multics project. This review will be
followed by a brief description of the appearance of the Multics system
to its various classes of users. Finally we will describe some of
the insights which have come out of the development activities.
(word count 135)

-1-

INTRODUCTION

In 1964, following implementation of the Compatible Time-Sharing System (CTSS)[1,2] serious planning began on the development of a new computer system specifically organized as a prototype of a computer utility. The plans and aspirations for this system, called Multics (for Multiplexed Information and Computing Service), were described in a set of six papers presented at the 1965 Fall Joint Computer Conference.[3-8] The development of the system was undertaken as a cooperative effort involving the Bell Telephone Laboratories (from 1965 to 1969), the computer department of the General Electric Company,* and Project MAC of M.I.T.

---

*

Subsequently acquired by Honeywell Information Systems Inc.

Implicit in the 1965 papers was the expectation that there should be a later examination of the development effort. From the present vantage point, however, it is clear that a definitive examination cannot be presented in a single paper. As a result, the present paper discusses only some of the many possible topics. First we review the goals, history and current status of the Multics project. This review is followed by a brief description of the appearance of the Multics system to its various classes of users. Finally several topics are given which represent some of the research insights which have come

out of the development activities. This organization has been chosen

in order to emphasize those aspects of software systems having the goals

of a computer utility which we feel to be of special interest. We

do not attempt detailed discussion of the organization of Multics;

that is the purpose of specialized technical books and papers. *

---

* For example, the essential mechanisms for much of the Multics system
are given in books by Organick[9] and Watson.[10]

GOALS

The goals of the computer utility, although stated at length

in the 1965 papers, deserve a brief review. By a computer utility

it was meant that one had a community computer facility with:

1) Convenient remote terminal access as the normal mode of

   system usage;

2) A view of continuous operation analogous to that of the

   electric power and telephone companies;

3) A wide range of capacity to allow growth or contraction with-

   out either system or user reorganization;

4) An internal file system so reliable that users trust their

   only copy of programs and data to be stored in it;

5) Sufficient control of access to allow selective sharing of

   information;

6) The ability to structure hierarchically both the logical

storage of information as well as the administration of the

system;

7) The capability of serving large and small users without

inefficiency to either;

8) The ability to support different programming environments

and human interfaces within a single system;

9) The flexibility and generality of system organization required

for evolution through successive waves of technological improve-

ments and the inevitable growth of user expectations.

In an absolute sense the above goals are extremely difficult

to achieve. Nevertheless, it is our belief that Multics, as it now exists,

has made substantial progress towards achieving each of the nine goals.*

Most importantly, none of these goals had to be compromised in any impor-

tant way.

---

*

To the best of our knowledge, the only other attempt to comprehen-

sively attack all of these goals simultaneously is the TSS/360

project at IBM. [11,12,13]

HISTORY OF THE DEVELOPMENT

As previously mentioned, the Multics project got underway

in the Fall of 1964. The computer equipment to be used was a modified

General Electric 635 which was later named the 645. The most signi-

ficant changes made were in the processor addressing and access control

logic where paging and segmentation were introduced. A completely
new Generalized Input Output Controller was designed and implemented
to accommodate the varied needs of devices such as disks, tapes and
teletypewriters without presenting an excessive interrupt burden to the
processors. To handle the expected paging traffic, a 4-million word
(36-bit) high-performance drum system with hardware queueing was developed.
The design specifications for these items were completed by Fall 1965,
and the equipment became available for software development in early
1967.

Software preparation underwent several phases. The first
phase was the development and blocking out of major ideas, followed
by the writing of detailed program module specifications. The resulting
3,000 typewritten pages formed the Multics System Programmers' Manual
and served as the starting point for all programming. Furthermore,
the software designers were expected to implement their own designs.
As a general policy PL/I was used as the system programming language
wherever possible to maximize lucidity and maintainability of the
system. [14,15] This policy also increased the effectiveness of system
programmers by allowing each one to keep more of the system within his
grasp.

The second major phase of software development, well under-
way by early 1967, was that of module implementation and unit checkout
followed by merging into larger aggregates for integrated testing.
Up to then most software and hardware difficulties had been anticipated

on the basis of previous experience. But what gradually became apparent
as the module integration continued was that there were gross discrep-
ancies between actual and expected performance of the various logical
execution paths throughout the software. The result was that an
unanticipated phase of design iterations was necessary. These design
iterations did not mean that major portions of the system were scrapped
without being used. On the contrary, until their replacements could
be implemented, often months later, they were crucially necessary to
allow the testing and evaluation of the other portions of the system.
The cause of the required redesigns was rarely "bad coding" since most
of the system programmers were well above average ability. Moreover
the redesigns did not mean that the goals of the project were compro-
mised. Rather three recurrent phenomena were observed: 1) typically,
specifications representing less-important features were found to be
introducing much of the complexity, 2) the initial choice of modularity
and interfacing between modules was sometimes awkward and 3) it was
rediscovered that the most important property of algorithms is simplicity
rather than special mechanisms for unusual cases.*

---

*

"In anything at all, perfection is finally attained not when there

is no longer anything to add, but when there is no longer anything

to take away..."

    -- Antoine de Saint-Exupéry, _Wind_, _Sand_ _and_ _Stars_

Quoted with permission of Harcourt Brace Jovanovich, Inc.

The reason for bringing out in detail the above design iteration experience is that frequently the planning of large software projects still does not properly take the need for continuing iteration into account. And yet we believe that design iterations are a required activity on any large scale system which attempts to break new conceptual ground such that individual programmers cannot comprehend the entire system in detail. For when new ground is broken, it is usually impossible to deduce the consequent system behavior except by experimental operation. Simulation is not particularly effective when the system concepts and user behavior are new. Unfortunately one does not understand the system well enough to simplify it correctly and thereby obtain a manageable model which requires less effort to implement than the system itself. Instead one must develop a different view:

1) The initial program version of a module should be viewed only as the first complete specification of the module and should be subject to design review before being debugged or checked out.

2) Module design and implementation should be based upon an assumption of periodic evaluation, redesign, and evolution.

In retrospect, the design iteration effect was apparent even in the development of the earlier Compatible Time-Sharing System (CTSS) when a second file system with many functional improvements turned out

to have poor performance when initially installed. A hasty design itera-
tion succeeded in rectifying the matter but the episode at the time
was viewed as an anomaly perhaps due to inadequate technical review of
individual programming efforts.

CURRENT STATUS

In spite of the unexpected design iteration phase, the Multics
system became sufficiently effective by late 1968 to allow system
programmers to use the system while still developing it. By October
1969, the system was made available for general use on a "cost-recovery"
charging basis similar to that used for other major computation facilities
at M.I.T. Multics is now the most widely used time-sharing system at
M.I.T., supporting a user community of some 500 registered subscribers.
The system is currently operated for users 22 hours per day, 7 days
per week. For at least eight hours each day the system operates with
two processors and three memory modules containing a total of 384k
(k = 1024) 36-bit words. This configuration currently is rated at a
capacity of about 55 fairly demanding users such that most trivial
requests obtain response in one to five seconds. (Future design itera-
tions are expected to increase the capacity rating.) Several times
a day during the off-peak usage hours the system is dynamically recon-
figured into two systems: a reduced capacity service system and an
independent development system. The development system is used for
testing those hardware and software changes which cannot be done under
normal service operation.

The reliability of the round-the-clock system operation described above has been a matter of great concern, for in any on-line real-time system the impact of mishaps is usually far more severe than in batch processing systems. In an on-line system especially important considerations are:

1) the time required before the system is useable again following a mishap,

2) the extra precautions required for restoring possibly lost files, and

3) the psychological stress of breaking the interactive dialogue with users who were counting on system availability.

Because of the importance of these considerations, careful logs are kept of all Multics "crashes" (i.e., system service disruption for all active users) at M.I.T. in order that analysis can reveal their causes. These analyses indicate currently an average of between one and two crashes per 24 hour day. These crashes have no single cause. Some are due to hardware failures, others to operator error and still others to software bugs introduced during the course of development. At the two other sites where Multics is operated, but where active system development does not take place, there have been almost no system failures traced to software.

Currently the Multics system, including compilers, commands, and subroutine libraries, consists of about 1500 modules, averaging

roughly 200 lines of PL/I apiece. These compile to produce some 1,000,000 words of procedure code. Another measure of the system is the size of the resident supervisor which is about 30k words of procedure and, for a 55 user load, about 36k words of data and buffer areas.

Because the system is so large, the most powerful maintenance tool available was chosen -- the system itself. With all of the system modules stored on-line, it is easy to manipulate the many components of different versions of the system. Thus it has been possible to maintain steadily for the last year or so a pace of installing 5 or 10 new or modified system modules a day. Some three-quarters of these changes can be installed while the system is in operation. The remainder, pertaining to the central supervisor, are installed in batches once or twice a week. This on-line maintenance capability has proven indispensible to the rapid development and maintenance of Multics since it permits constant upgrading of the user interface without interrupting the service. We are just beginning to see instances of user-written applications which require this same capability so that the application users need not be interrupted while the software they are using is being modified.

The software effort which has been spent on Multics is diffi-cult to estimate. Approximately 150 man-years were applied directly to design and system programming during the "development-only" period of Table I. Since then we estimate that another 50 man-years have been

System	Development Only	Development + Use	Use Only
CTSS	1960-1963	1963-1965	1965-present
Multics	1964-1969	1969-present	

Table I.   A comparison of the system development and use
periods of CTSS and Multics. The Multics development period
is not significantly longer than that for CTSS despite the
development of about 10 times as much code for Multics as
for CTSS and a geographically distributed staff. Although
reasons for this similarity in time span include the use of
a higher-level programming language and a somewhat larger
staff, the use of CTSS as a development tool for Multics
was of pivital importance.

devoted to improving and extending the system. But the actual cost of a single successful system is misleading, for if one starts afresh to build a similar system, one must compensate for the non-zero probability of failure.

THE APPEARANCE OF MULTICS TO ITS USERS

Having reviewed the background of the project, we may now ask who are the users of the Multics system and what do the facilities that Multics provides mean to these users. Before answering, it is worth describing the generic user as "viewed" by Multics. Although from the system's point of view all users have the same general characteristics and interface with it uniformly, no single human interface represents the Multics machine. That machine is determined by each user's initial procedure coupled with those functions accessible to him. Thus there exists the potential to present each Multics user with a unique external interface.

However, Multics does provide a native internal program environment consisting of a stack-oriented, pure-procedure, collection of PL/I procedures imbedded in a segmented virtual memory containing all procedures and data stored on-line. The extent to which some, all, or none of this internal environment is visible to the various users is an administrative choice.

The implications of these two views -- both the external interface

and the internal programming environment -- are discussed in terms
of the following categories of users:

- System programmers and user application programmers respon-
  sible for writing system and user software.

- Administrative personnel responsible for the management of
  system resources and privileges.

- The ultimate users of applications systems.

- Operations and hardware maintenance personnel responsible,
  respectively, for running the machine room and maintaining
  the hardware.

Multics as Viewed by System and Subsystem Programmers

The machine presented to both the Multics system programmer
and the application system programmer is the one with which we have
the most experience; it is the raw material from which one constructs
other environments. It is worth re-emphasizing that the only differentiation
between Multics system programmers and user programmers is embodied
in the access control mechanism which determines what on-line informa-
tion can be referenced; therefore, what are apparently two groups of
users can be discussed as one.

Major interfaces presented to programmers on the Multics
system can be classified as the program preparation and documentation
facilities and the program execution and debugging environment. They
will be touched upon briefly, in the order used for program preparation.

Program Preparation and Documentation. The facilities for program

preparation on Multics are typical of those found on other time-sharing

systems, with some shifts in emphasis. (See the Appendix.) For example,

programmers consider the file system sufficiently invulnerable to

physical loss that it is used casually and routinely to save all infor-

mation. Thus, the punched card has vanished from the work routine of

Multics programmers and access to one's programs and the ability to

work on them are provided by the closest terminal.

As another example, the full ASCII character set is employed

in preparing programs, data, and documentation, thereby eliminating

the need for multiple text editors, several varieties of text formatting

and comparison programs, and multiple facilities for printing informa-

tion both on-line and off-line. This generalization of user inter-

faces facilitates the learning and subsequent use of the system by

reducing the number of conventions which must be mastered.

Finally, because the PL/I compiler is a large set of programs,

considerable attention was given to shielding the user from the size

of the compiler and to aiding him in mastering the complexities of the

language. As in many other time-sharing systems, the compiler is

invoked by issuing a simple command line from a terminal exactly as

for the less ambitious commands. No knowledge is required of the user

regarding the various phases of compilation, temporary files required,

and optional capabilities for the specialist; explanatory "sermons"

diagnosing syntactic errors are delivered to the terminal to effect a self-teaching session during each compilation. To the programmer, the PL/I compiler is just another command.

Program Execution Environment. Another set of interfaces is embodied in the implementation environment seen by PL/I programmers. This environment consists of a directly addressable virtual memory containing the entire hierarchy of on-line information, a dynamic linking facility which searches this hierarchy to bind procedure references, a device-independent input/output[16] system,* and program debugging and metering facilities. These facilities enjoy a symbiotic relationship with the PL/I procedure environment used both to implement them and to implement user facilities co-existing with them. Of major significance is that the natural internal environment provided and required by the system is exactly that environment expected by PL/I procedures. For example, PL/I pointer variables, call and return statements, conditions, and static and automatic storage all correspond directly to mechanisms provided in the internal environment. Consequently, the system supports PL/I code as a matter of course.

---

* The Michigan Terminal System[17] has a similar device-independent input/output system.

The main effect of the combination of these features is to permit the implementer to spend his time concentrating on the logic

of his problem; for the most part he is freed from the usual mechanical

problems of storage management and overlays, input/output device quirks,

and machine-dependent features.

Some Implementation Experience. The Multics team began to be much

more productive once the Multics system became useful for software

development. A few cases are worth citing to illustrate the effective-

ness of the implementation environment. A good example is the current PL/I

compiler, which is the third one to be implemented for the project, and

which consists of some 250 procedures and about 125k words of object

code. Four people implemented this compiler in two years, from start

to first general use. The first version of the Multics program debugging

system, composed of over 3,000 lines of source code, was useable after

one person spent some six months of nights and weekends "bootlegging"

its implementation. As a last example, a facility consisting of 50 pro-

cedures with a total of nearly 4,000 PL/I statements permitting execution

of Honeywell 635 programs under Multics became operational after

one person spent eight months learning about the GCOS operating system

for the 635, PL/I, and Multics, and then implemented the environment.

In each of these examples the implementation was accomplished from

remote terminals using PL/I.

Multics users have discovered that it is possible to get

their programs running very quickly in this environment. They fre-

quently prepare "rough drafts" of programs, execute them, and then

improve their overall design and operating strategy using the results

of experience obtained during actual operation. As an example, again drawn from the implementation of Multics, the early designs and implementations of the programs supporting the virtual memory[18] made over-optimistic use of variable-sized storage allocation techniques. The result was a functionally correct but inadequately performing set of programs. Nevertheless, these modules were used as the foundation for subsequent work for many months. When they were finally replaced with modules using simplified fixed-size storage techniques, performance improvements of over an order of magnitude were realized. This technique emphasizes two points: first, it is frequently possible to provide a practical, useable facility containing temporary versions of programs; second, often the insight required to significantly improve the behavior of a program comes only after it is studied in operation. As implied in the earlier discussion of design iteration, our experience has been that structural and strategic changes rather than "polishing" (or recoding in assembly language) produce the most significant performance improvements.

In general, we have noticed a significant "amplifier" or "leverage" effect with the use of an effective on-line environment as a system programming facility. Major implementation projects on the Multics system seldom involve more than a few programmers, thereby easing the management and communications problems usually entailed by complex system implementations. As would be expected, the amplification

effect is most apparent with the best project personnel.

Administration of Multics Facilities and Resources

The problem of managing the capabilities of a computer utility with geographically dispersed subscribers leads to a requirement of decentralized administration. At the apex of an administrative pyramid resides a system administrator with the ability to register new users, confer resource quotas, and generate periodic bills for services rendered. The system administrator deals with user groups called projects. Each group can in turn designate a project administrator who is delegated the authority to manage a budget of system resources on behalf of the project. The project administrator is then free to deal directly with project members without further intervention from the system administrator, thereby greatly reducing the bottlenecks inherent in a completely centralized administrative structure.

Environment Shaping. In addition to having immediate control of such resources as secondary storage, port access, and rate of processor usage, the project administrator is also able to define or shape the environment seen by the members of his project when they log into the system. He does this by defining those procedures that can be accessed by members of his project and by specifying the initial procedure executed by each member of his project when he logs in. This environment shaping facility has led to the notion of a private project

subsystem on Multics. It combines the administrative and programming

facilities of Multics so that a project administrator and a few pro-

ject implementers can build, maintain, and evolve environments entirely

on their own. Thus, some subsystems bear no internal resemblance to

the standard Multics procedure environment.

For example, the Dartmouth BASIC[19] compiler executes in a

closed subsystem implemented by an M.I.T. student group for use by

undergraduate students. The compiler, its object code, and all support

routines execute in a simulation of the native environment provided

at Dartmouth. The users of this subsystem need little, if any, know-

ledge of Multics and are able to behave as if logged into the Dartmouth

system proper. Other examples of controlled environment subsystems

include one to permit many programs which normally run under the GCOS

operating system to also run unmodified in Multics. Finally, an APL[20]

subsystem allows the user to behave for the most part as if he were

logged into an APL machine. The significance of these subsystems is

that their implementers did not need to interact with the system adminis-

trator or to modify already existing Multics capabilities. The adminis-

trative facilities permit each such subsystem to be offered by its

supporters as a private service with its own group of users, each

effectively having its own private computer system.

Other Multics Users

Finally, we observe that the roles of the application user,

the system operators and the hardware maintainers as seen by the system
are simply those of ordinary Multics users with specialized access
to the on-line procedures and data. The effect of this uniformity
of treatment is to reduce greatly the maintenance burden of the
system control software. One example, of great practical importance,
has been the ease with which system performance measurement tools
have been prepared for use by the operating staff.

INSIGHTS

So far, we have discussed the status and appearance of the
Multics system. A further question is what has been learned in the
construction of Multics which is of use to the designers of other
systems. Having a bright idea which clearly solves a problem is not
sufficient cause to claim a contribution if the idea is to be part
of a complex system. In order to establish the real feasibility of
an idea, all of its implications and consequences must be followed
out. Much of the work on Multics since 1965 has involved following
out implications and consequences of the many ideas then proposed
for the prototype computer utility. That following out is an
essential part of proof of ideas is attested by the difficulties
which have been encountered in other engineering efforts such as the
development of nuclear fusion power plants and the electric automo-
bile. Not all proposals work out; for example, extended attempts to
engineer an atomic powered airplane suggest infeasibility.

Perhaps Multics' most significant single contribution to the state of the art of computer system construction is the demonstration of a large set of fully implemented ideas in a working system. Further, most of these ideas have been integrated without straining the overall design; most additional proposals would not topple the structure. Ideas such as virtual memory access to on-line storage, parallel process organization, routine but controlled information sharing, dynamic linking of procedures, and high-level language implementation have proven remarkably compatible and complementary.

To illustrate some of the areas of progress in understanding of system organization and construction which have been achieved in Multics, we consider here the following five topics:

1. Modular division of responsibility

2. Dynamic reconfiguration

3. Automatically managed multilevel memory

4. Protection of programs and data

5. System programming language

Modular Division of Responsibility

Early in the design of Multics a decision had to be made whether or not to treat the segmented virtual memory as a separately useable "feature", independent of a traditionally organized read/write type file system. The alternative, to use the segmented virtual memory as the file system itself, providing the illusion of direct

"in-core" access to all on-line storage, was certainly the less

conservative approach. (See figure 1.) The second approach,

which was the one chosen, led to a strong test of the ability of a

computing system to support an apparent one-level memory for an arbi-

trarily large information base. It is interesting that the resulting

almost total decoupling between physical storage allocation and data

movement on the one hand and directory structure, naming, and file

organization on the other led to a remarkably simple and functionally

modular structure for that part of the system. [18] (See figure 2.)

Another area of Multics in which a high degree of functional

modularity was achieved was in the area of scheduling, multiprogramming,

and processor management. Because harnessing of multiple processors

was an objective from the beginning, a careful and methodical approach

to multiplexing processors, handling interrupts, and providing inter-

process synchronizing primitives was developed. The resulting design,

known as the Multics traffic controller, absorbed into a single, simple

module a set of responsibilities often diffused among a scheduling

algorithm, the input/output controlling system, the on-line file manage-

ment system, and special purpose inter-user communication mechanisms. [21]

Finally, with processor management and on-line storage manage-

ment uncoupled into well-isolated modules, the Multics input/output

system was left with the similarly isolatable function of managing

streams of data flowing from and to source and sink type devices. [16]

Thus, this section of the system concentrates only on switching of the

streams, allocation of data buffering areas, and device control strategies.

Each of the divisions of labor described above represents an interesting result primarily because it is so difficult to discover appropriate divisions of complex systems.[*] Establishing that a certain proposed division results in simplicity, creates an uncluttered interface, and does not interfere with performance, is generally cause for a minor celebration.

---

[*] See Dijkstra[22] for a further discussion of this point.

Dynamic Reconfiguration

If the computer utility is ever to become as much a reality as the electric power utility or the telephone communication service, its continued operation must not be dependent upon any single physical component, since individual components will eventually require maintenance. This observation leads an electric power utility to provide procedures whereby an idle generator may be dynamically added to the utility's generating capacity, while another is removed for maintenance, all without any disruption of service to customers. A similar scenario has long been proposed for multiprocessor, multimemory computer systems, in which one would dynamically switch processors and memory boxes in and out of the operating configuration as needed. Unfortunately,

-23-

though there have been demonstrated a few "special purpose" designs,[*]
it has not been apparent how to provide for such operations in a general
purpose system. A recent thesis[24] proposed a general model for the
dynamic binding and unbinding of computation and memory structures to
and from ongoing computations. Using this model as a basis, the
thesis also proposed a specific implementation for a typical multi-
processor, multimemory computing system. One of the results of this
work was the addition to the operating Multics system of the capa-
bility of dynamically adding and removing central processors and memory
modules as in figure 3. The usefulness of the idea may be gauged
by observing that at M.I.T. five to ten such reconfigurations are per-
formed in a typical 24-hour operating day. Most of the reconfigurations
are used to provide a secondary system for Multics development.

---

[*]
    An outstanding example is the American Airlines SABRE system.[23]

Automatically Managed Multilevel Memory

By now it has become accepted lore in the computer system
field that the use of automatic management algorithms for memory sys-
tems constructed of several levels with different access times can
provide a significant reduction of user programming effort. Examples
of such automatic management strategies include the buffer memories

of the IBM system 370 models 155, 165, and 195[25] and the demand paging

virtual memories of Multics, IBM's CP-67[26] and the Michigan Terminal

System.[17] Unfortunately, behind the mask of acceptance hides a worri-

some lack of knowledge about how to engineer a multilevel memory system

with appropriate strategy algorithms which are matched to the load

and hardware characteristics. One of the goals of the Multics project

has been to instrument and experiment with the multilevel memory system

of Multics, in order to learn better how to predict in advance the

performance of proposed new automatically managed multilevel memory

systems. Several specific aspects of this goal have been explored:

- A strategy to treat core memory, drum, and disk as a three-
  level system has been proposed, including a "least-recently-
  used" algorithm for moving information from drum to disk.
  Such an algorithm has been used for some time to determine
  which pages should be removed from core memory.[27] The
  dynamics of interaction among two such algorithms operating
  at different levels are weakly understood, and some experi-
  mental work should provide much insight. The proposed
  strategy will be implemented, and then compared with the
  simpler present strategy which never moves things from drum
  to disk, but instead makes educated "guesses" as to which
  device is most appropriate for the permanent residence of a
  given page. If the automatic algorithm is at least as good

as the older, static one, it would represent an improvement in overall design by itself, since it would automatically track changes in user behavior, while the static algorithm requires attention to the validity of its guesses.

* A scheme to permit experimentation with predictive paging algorithms was devised. The scheme provides for each process a list of pages to be preloaded whenever the process is run, and a second list to be immediately purged whenever the process stops. The updating of these lists is controlled by a decision table exercised every time the process stops running. Since every page of the Multics virtual memory is potentially shared, the decision table represents a set of heuristics designed to separate out those which are probably not being shared at the moment.

* A series of measurements was made to establish the effectiveness of a small hardware associative memory used to hold recently accessed page descriptors. These measurements established a profile of hit ratio (probability of finding a page descriptor in the associative memory) versus associative memory size which should be useful to the designers of virtual memory systems. [28]

* A set of models, both analytic and simulation, was constructed to try to understand program behavior in a virtual memory.

So far, two results have been obtained. One is the finding
that a single program characteristic (the mean execution
time before encountering a "missing" page in the virtual
memory as a function of memory size) suffices to provide
a quite accurate prediction of paging and idle overheads.
The second is direct calculation of the distribution of
response times under multiprogramming. Having available
the entire response time distribution, rather than just
averages, permits estimation of the variance and 90-percen-
tile points of the distribution, which may be more meaning-
ful than just the average. A doctoral thesis is in progress
on this topic.

Although the immediate effect of each of these investigations
is to improve the understanding or performance of the current version
of Multics, the long-range payoff in methodical engineering using better-
understood memory structures is also evident.

Protection of Programs and Data

A long-standing objective of the public computer utility
has been to provide facilities for the protection of executing programs
from one another, so that users may with confidence place appropriate
control on the release of their private information. In 1967, a
mechanism was proposed[29] and implemented in software which generalized
the usual supervisor-user protection relationship. This mechanism,
named "rings of protection", provides user-written subsystems with
the same protection from other users that the supervisor has, yet

does not require that the user-written subsystem be incorporated into the supervisor. Recently, this approach was brought under intense review, with two results:

- A hardware architecture which implements the mechanism was proposed.[30] One of the chief features of the proposed architecture is that subroutine calls from one protection ring to another use exactly the same mechanisms as do sub-routine calls among procedures within a protection area. The proposal appears sufficiently promising that it is included in the specifications for the next generation of hardware to be used for Multics.

- As an experiment in the feasibility of a multilayered super-visor, several supervisor procedures which required protec-tion, but not all supervisor privileges, were moved into a ring of protection intermediate between the users and the main supervisor. The success of this experiment established that such layering is a practical way to reduce the quantity of supervisor code which must be given all privileges.

Both of these results are viewed as steps toward first, a more complete exploitation and understanding of rings of protection, and later, a less constrained organization of the type suggested by Evans and LeClerc[31] and by Lampson.[32] But more importantly, rings of pro-tection appear applicable to any computer system using a segmented virtual memory. Two doctoral theses are underway in this area.

System Programming Language

Another technique of system engineering metholodogy being explored within the Multics project is that of higher level programming language for system implementation. The initial step in this direction (which proved to be a very big step) was the choice of the PL/I language for the implementation of Multics. By now, Multics offers an extensive case study in the viability of this strategy. Not only has the cost of using a higher level language been acceptable, but increased maintainability of the software has permitted more rapid evolution of the system in response to development ideas as well as user needs. Three specific aspects of this experience have now been completed:

- The transition from an early PL/I subset compiler[14] to a newer compiler which handles almost the entire language was completed. This transition was carried out with performance improvement in practically every module converted in spite of the larger language involved. The significance of the transition is the demonstration that it is not necessary to narrow one's sights to a "simple" subset language for system programming. If the language is thoroughly understood, even a language as complex as the full PL/I can be effectively used. As a result, the same language and compiler provided for users can also be used for system implementation, thereby minimizing maintenance, confusion, and specialization.

•   Notwithstanding the observation just made, the time required
to implement a full PL/I compiler is still too great for
many situations in which the compiler implementation cannot
be started far enough in advance of system coding. For this
reason, there is considerable interest in defining a smaller
language which is easily compilable, yet retains the features
most important for system implementation. On the basis of
the experience of programming Multics in a subset of PL/I,
such a language was defined but not implemented, since it
was not needed. [33]

•   A census of Multics system modules reveals how much of the
system was actually coded in PL/I, and reasons for use of
other languages. Roughly, of the 1500 system modules, about
250 were written in machine language. Most of the machine
language modules represent data bases or small subroutines
which execute a single privileged instruction. (No attempt
was made to provide either a data base compiler or PL/I
built-in functions for specialized hardware needs.) Signi-
ficantly, only a half dozen areas (primarily in the traffic
controller, the central page fault path, and interrupt
handlers) which were originally written in PL/I have been
recoded in machine language for reasons of squeezing out the

utmost in performance. Several programs, originally in machine language, have been recoded in PL/I to increase their maintainability.

As with the earlier topics, the implications of this work with PL/I should be felt far beyond the Multics system. Most implementers, when faced with the economic uncertainties of a higher-level language, have chosen machine language for their central operating systems. The experience of PL/I in Multics when added to the expanding collection of experience elsewhere[34] should help reduce the uncertainty.

In a research project as large, long, and complex as Multics, any paper such as this must necessarily omit many equally significant ideas, and touch only a few which may happen to have wide current interest. It is the purpose of individual and detailed technical papers to explain these and other ideas more fully. The bibliography found in reference[35] contains over twenty such technical papers.

Immediate Future Plans

The Multics software is continuing to evolve in response to user needs and improved understanding of its organization. In 1972 a new hardware base for Multics will be installed by the Information Processing Center at M.I.T. for use by the M.I.T. computing community. This program compatible hardware base contains small but significant architectural extensions to the current hardware. The circuit technology used will be that of the Honeywell 6080 computer. The substantial changes include:

1) replacement of the high-performance paging drum initially

with bulk core and, when available, LSI memory, and

2) implementation of rings of protection as part of the paging

and segmentation hardware.

Wherever possible the strategy of using off-the-shelf standard equip-

ment rather than specially engineered units for Multics has been followed.

This strategy is intended to simplify maintenance.

CONCLUSIONS

There are many conclusions which could possibly be drawn

from the experience of the Multics project.   Of these, we con-

sider four to be major and worthy of note.   First, we feel it is clear

that it is possible to achieve the goals of a prototype computer

utility.   The current implementation of Multics provides a measure of

the mechanisms required.   Moreover, the specific implementation of

the system, because it has been written in PL/I, forms a model for

other system designers to draw upon when constructing similar systems.

Second, the question of whether or not the specific software

features and mechanisms which were postulated for effective computer

utility operation are desirable has now been tested with specific

user experience.   Although the specific mechanisms implemented sub-

sequently may be superseded by better ones, it is certainly clear that

the improvement of the user environment which was wanted has been

achieved.

Third, systems of the computer utility class must evolve

indefinitely since the cost of starting over is usually prohibitive and the many-year lead time required may be equally unacceptable. The requirement of evolvability places stringent demands on design, maintainability, and implementation techniques.

Fourth and finally, the very act of creating a system which solves many of the problems posed in 1965 has opened up many new directions of research and development. It would appear almost a certainty that increased user aspirations will continue to require intensive work in the areas of computer system principles and techniques.

In closing, perhaps we should take note that in the seven years since Multics was proposed, a great many other systems have also been proposed and constructed; many of these have developed similar ideas.[*] In most cases, their designers have developed effective implementations which are directed to a different interpretation of the goals, or to a smaller set of goals than those required for the complete computer utility. This diversity is valuable, and probably necessary, to accomplish a thorough exploration of many individually complex ideas, and thereby to meet a future which holds increasing demand for systems which embrace the totality of computer utility requirements.

---

[*] Some examples which have not already been mentioned include: the TENEX system of Bolt, Beranek and Newman, the VENUS system of Mitre Corp., the MU5 at Manchester University, RC-4000 of

Regnecentralen, 5020 TSS of Hitachi Corp., DIPS-1 of Nippon

Telephone, the Japanese National Computer Project, the PDP-10/50

TSS of Digital Equipment Corp., the BCC-500 of Berkeley Computer

Corp., I.T.S. of the M.I.T. Artificial Intelligence Laboratory,

Exec-8 of Univac, System 3 and 7 and the SPECTRA 70/46 of RCA,

Star-100 of CDC, UTS of Xerox Data Systems, the 6700 system

of Burroughs, and the Dartmouth Time-Sharing System.

REFERENCES

1.   Corbató, F. J., Daggett, M. M., and Daley, R. C., "An Experi-
mental Time-Sharing System", AFIPS Conf. Proc. 21, Spartan Books,
1962, pp. 335-344.

2.   Crisman, P. A., ed., "The Compatible Time-Sharing System: A
Programmer's Guide", 2nd ed., M. I.T. Press, Cambridge, Massa-
chusetts, 1965.

3.   Corbató, F. J., and Vyssotsky, V. A., "Introduction and Overview
of the Multics System", AFIPS Conf. Proc. 27 (1965 FJCC), Spartan
Books, Washington, D. C., 1965, pp. 185-196.

4.   Glaser, E. L., et al., "System Design of a Computer for Time-
Sharing Application", AFIPS Conf. Proc. 27 (1965 FJCC), Spartan
Books, Washington D. C., 1965, pp. 197-202.

5.   Vyssotsky, V. A., et al., "Structure of the Multics Supervisor",
AFIPS Conf. Proc. 27 (1965 FJCC), Spartan Books, Washington, D. C.,
1965, pp. 203-212.

6.   Daley, R. C., and Neumann, P. G., "A General-Purpose File System
for Secondary Storage", AFIPS Conf. Proc. 27 (1965 FJCC), Spartan
Books, Washington, D. C., 1965, pp. 213-229.

7.   Ossanna, J. F., et al., "Communication and Input/Output Switching
in a Multiplex Computing System", AFIPS Conf. Proc. 27 (1965 FJCC),
Spartan Books, Washington, D. C., 1965, pp. 231-241.

8.   David, E. E., Jr., and Fano, R. M., "Some Thoughts About the Social
Implications of Accessible Computing", AFIPS Conf. Proc. 27 (1965
FJCC), Spartan Books, Washington, D. C., 1965, pp. 243-247.

9. Organick, E. I., _The Multics System: An Examination of its Structure_, M.I.T. Press (in press), Cambridge, Massachusetts and London, England.

10. Watson, R. W., _Timesharing System Design Concepts_, McGraw-Hill Book Company, New York, 1970.

11. Comfort, Webb T., "A Computing System Design for User Service", _AFIPS Conf. Proc._ 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 619-626.

12. Lett, A. S., and Konigsford, W. L., "TSS/360: A Time-Shared Operating System" _AFIPS Conf. Proc._ 33 (1968 FJCC), Thompson Books, pp. 15-28.

13. Schwemm, R. E., "Experience Gained in the Development and Use of TSS/360", _AFIPS Conf. Proc._ 40 (1972 SJCC), AFIPS Press. (This volume.)

14. Corbató, F. J., "PL/I as a Tool for System Programming", _Datamation_ 15, 6 (May, 1969) pp. 68-76.

15. Freiburghouse, R. A., "The Multics PL/I Compiler", _AFIPS Conf. Proc._ 35 (1969 FJCC), AFIPS Press, 1969, pp. 187-199.

16. Feiertag, R. J. , and Organick, E. I., "The Multics Input-Output System", _ACM Third Symposium on Operating Systems Principles_, (October 18-20, 1971), pp. 35-41.

17. Alexander, M. T., "Organization and Features of the Michigan Terminal System" _AFIPS Conf. Proc._ 40 (1972 SJCC), AFIPS Press. (This volume.)

18. Bensoussan, A., Clingen, C. T., and Daley, R. C., "The Multics Virtual Memory", ACM Second Symposium on Operating System Principles, (October 20-22, 1969) Princeton University, pp. 30-42.

19. BASIC, Fifth Edition, Kiewit Computation Center, Dartmouth College (September, 1970).

20. APL/360 User's Manual, IBM form number GH20-0683-1 (March, 1970).

21. Saltzer, J. H., "Traffic Control in a Multiplexed Computer System", Sc.D. Thesis, M.I.T. Department of Electrical Engineering, 1966. Also available as Project MAC technical report TR-30.

22. Dijkstra, E. W., "The Structure of the 'THE'-Multiprogramming System", Comm. ACM 11, 5 (May, 1968), pp. 341-346.

23. Parker, R. W., "The Sabre System", Datamation 11, 9, September, 1965, pp. 49-52.

24. Schell, R. R., "Dynamic Reconfiguration in a Modular Computer System", Ph.D. Thesis, M.I.T. Department of Electrical Engineering, 1971. Also available as Project MAC technical report TR-86.

25. Conti, C. J., "Concepts for Buffer Storage", IEEE Computer Group News, March, 1969, pp. 9-13.

26. Meyer, R. A., and Seawright, L. H., "A Virtual Machine Time-Sharing System", IBM Systems Journal 9, 3, 1970, pp. 199-218.

27. Corbató, F. J., "A Paging Experiment with the Multics System", In Honor of P. M. Morse, M.I.T. Press, Cambridge, Massachusetts, 1969, pp. 217-228.

28. Schroeder, M. D., "Performance of the GE-645 Associative Memory While Multics is in Operation", ACM Workshop on System Performance Evaluation (April, 1971), pp. 227-245.

29. Graham, R. M., "Protection in an Information Processing Utility",

Comm. ACM 11, 5 (May, 1968) pp. 365-369.

30. Schroeder, M. D., and Saltzer, J. H., "A Hardware Architecture for

Implementing Protection Rings", ACM Third Symposium on Operating

Systems Principles, (October 18-20, 1971), pp. 42-54.

31. Evans, D. C., and LeClerc, J. Y., "Address Mapping and the Control

of Access in an Interactive Computer", AFIPS Conf. Proc. 30,

(1967 SJCC), Thompson Books, 1967, pp. 23-30.

32. Lampson, Butler W., "An Overview of the CAL Time-Sharing System",

Computer Center, University of California, Berkeley (September 5,

1969).

33. Clark, D. D., Graham, R. M., Saltzer, J. H., and Schroeder, M. D.,

"Classroom Information and Computing Service", M. I.T. Project MAC

Technical Report TR-80, (January 11, 1971).

34. Sammet, Jean E., "Brief Survey of Languages Used for Systems

Implementation", SIGPLAN Notices 6, 9, October, 1971,

pp. 1-19.

35. The Multiplexed Information and Computing Service: Programmers'

Manual, M. I.T. Project MAC, Rev. 10, 1972. (Available from the

M. I.T. Information Processing Center.)

APPENDIX: A CHECKLIST OF MULTICS FEATURES

Following is a checklist of currently available features and
facilities of Multics. Although many of the features are described
in cryptic and untranslated local jargon, one can at least obtain a
feel for the range of facilities now provided. Further information
on most of these features may be found in the Multics Programmers'
Manual.[35]

Interactive Time-Sharing Facilities

file editors

file manipulation (rename/move/delete)

personal command abbreviations

recursive command language

source language debugging with breakpoints

subroutine call tracer

can stop any running command or program

Programming Languages

PL/I

FORTRAN

BASIC*

APL

LISP

BCPL

ALM (assembly language/Multics)

Information Storage System

>configuration independent

>accessed through virtual memory (segments)

>access control lists by user and project

>links to segments of other users

>hierarchical directory (catalog) arrangement

>public library facilities

>sharing at all levels

>multiple segment names (synonyms)

>separate control of read, write, and execute

Programming Environment

>segmented virtual memory

>dynamic linking of procedures and data, or prelinking

>interprocess communication

>independent of configuration

>uniform error handling mechanism

>user definable protection rings

>microsecond calendar clock with interrupt

>program interrupt signal from console

Input and Output

>standard typewriter interface for device independence

>ASCII character set used throughout

>input characters converted to canonical form

>erase and kill editing on typed input

I/O streams switchable during execution

magnetic tape, printer, card punch, card reader

typewriter terminals: IBM 2741, 1050

                            Teletype 37, 33, 35

                            Dura, Datel, Execuport, Terminet-300

graphic support library (devices: ARDS, IMLAC, DEC 338)

ARPA network

interfaces at three levels:

   formatted data conversion

   bit stream control

   full device control

Management Facilities

passwords required for login

project may interpose authentication procedure

decentralized projects

accounting, billing, and quotas

on-line probing and account adjustment

operator or system initiated logout of users

unlisted and anonymous users

limited service system

dynamic reconfiguration of memories and processors

system performance metering for parameter adjustment

project-imposed starting procedure

Communication Facilities

       interuser mail

       help command; help files

       message of the day

       on-line error reporting and consultation service

       on-line user graffiti board

       operations message broadcast to logged-in users

Absentee Facilities

       priority/defer queues for printer, card punch

       queued translator facility

       general absentee job facility

Reliability Measures

       weekly file copies onto tape

       daily disk/drum copy onto tape

       incremental file copies onto tape, 1/2 hour behind use

       salvager to clean up files after system crash

       emergency shutdown entry to system

Maintenance Features

       on-line library change, no disruption of current users

       entire system source on-line, maintenance tools

       system checkout on small hardware configuration

on-line performance monitoring of multiprogramming

paging traffic

drum/disk usage

typewriter traffic

user performance feedback:

cpu time and paging load on each command

page trace always operating

subroutine call counters

Private Project Subsystems

project providable command interface

Dartmouth environment*

student environment

Miscellaneous Facilities

desk calculators

sort command

memorandum formatting and typing subsystem

user-provided list of programs to be automatically executed

when user logs in

GCOS environment

---

* The BASIC system and the Dartmouth environment were developed at Dartmouth College. They are used at M.I.T. by permission of Dartmouth College.

CAPTIONS FOR FIGURES

Figure 1. The entire storage hierarchy may be mapped into individual user process address spaces (see arrows) as if contained in primary memory. Illustrated are the sharing of a supervisor segment by user 1 and user 2 and private access to segment a and segment b. The necessary primary storage is simulated by a demand paging technique which moves information between the real primary memory and secondary storage.

Figure 2. Major lines of modular division in Multics. Solid lines indicate calls for services. Dotted lines indicate implicit use of the virtual memory.

Figure 3. Dynamic reconfiguration permits switching among the three typical operating configurations shown here, without currently logged-in users being aware that a change has taken place.

Root
directory

System
library
directory

Project
directory

Address map
for user 1

Virtual
processor
for user 1

Supervisor
segment

User 1
directory

User 2
directory

Address map
for user 2

Virtual
processor
for user 2

Segment
a

Segment
b

←——— Virtual memory storage system ———→

```
┌─────────────────────────────────────────────────────────────┐
│     User programs and command/subroutine library             │
└─────────────────────────────────────────────────────────────┘
```

General user
interface

```
                  ┌──────────────────┐    ┌──────────────────┐
                  │    Directory     │    │  User I/O device │
                  │  address space   │    │    control and   │
                  │   management     │    │    buffering     │
                  └──────────────────┘    └──────────────────┘
```

Virtual memory/
multi-process
interface

```
          ┌──────────────────┐         ┌──────────────────┐
          │  Drum, disk, core │         │ Processor multi- │
          │  demand paging    │────────▶│ plexing and process │
          │    controller     │         │  synchronization │
          └──────────────────┘         └──────────────────┘
```

a.

Central processor

Central processor

Memory

Memory

Memory

Service system

b.

Central processor

Central processor

Memory

Memory

Memory

Development system

Service system

c.

Off-line for maintenance

Central processor

Central processor

Memory

Memory

Memory

Service system