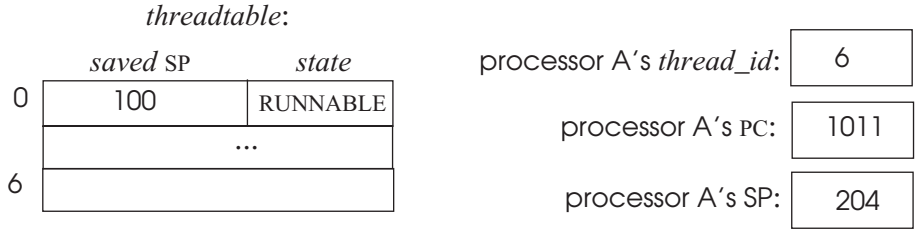
Finally, assume the following saved thread state for thread 0 and the follow current state for processor A:



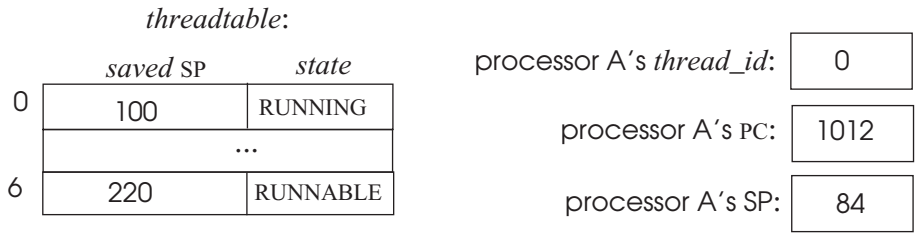
At some time in the past, thread 0 called YIELD and ENTER_PROCESSOR_LAYER stored the value of thread 0's stack pointer (100) into the thread table, and went on to run some other thread. Processor A is currently running thread 6: A's entry in the *processor_table* array contains 6, A's SP register points to the top of the stack of thread 6, and A's PC register contains address 1011, which holds the first instruction of YIELD (see line 11).

YIELD invokes the procedure ENTER_PROCESSOR_LAYER, following the procedure call convention of page 4–8, which pushes some values on thread 6's stack—in particular, the return address (1012)—and change A's SP to 220 (204 + 16). ENTER_PROCESSOR_LAYER knows that the current thread has index 6 by reading the processor's entry in the *processor_table* array. Line 17 saves thread 6's current top of stack (220) by storing processor A's SP into thread 6's entry into *threadtable*.

The statements at lines 21 through 24 choose which thread to run next, using a simple round-robin algorithm, and select thread 0. The scheduler invokes EXIT_PROCESSOR_LAYER to dispatch processor A to thread 0.

Line 32 loads the saved SP of thread 0 so that processor A can find the top of the stack at memory address 100. At the top of thread 0's stack will be the return address; this address will be 1012 (the line after the call to ENTER_PROCESSOR_LAYER into YIELD, line 12), since thread 0 entered ENTER_PROCESSOR_LAYER from YIELD. Thread 0 releases *threadtable_lock* so that another thread can enter ENTER_PROCESSOR_LAYER, and return from YIELD. Thread 0 returns from EXIT_PROCESSOR_LAYER following the procedure call convention, which pops off the return address from the top of the stack. The address that exit_processor_layer uses is 1012, because EXIT_PROCESSOR_LAYER uses the SP saved by ENTER_PROCESSOR_LAYER and thus returns to YIELD at line 12. YIELD returns control to the program in thread 0 that originally called YIELD.

At this point, the thread switch has completed and thread 0, rather than thread 6, is running on processor A, and state is as follows:



At some time in the future, the thread manager will resume thread 6, at the instruction at address 1012.

From this example we can see that a thread always releases its processor by calling ENTER_PROCESSOR_LAYER, and the thread always resumes right after the call to ENTER_PROCESSOR_LAYER. This stylized flow of control where a thread releases its processor always at the same point and resumes at that point is an example of what sometimes is called *co-routine*.

To ensure that the thread switch is atomic, the thread that invokes ENTER_PROCESSOR_LAYER acquires *threadtable_lock* and the thread that resumes using EXIT_PROCESSOR_LAYER releases *threadtable_lock* (line 33). Because the scheduler is likely to choose a different thread to run from the one that called YIELD, the thread that releases the lock is most likely a different thread from the one that acquired the lock. In essence, the thread that releases the processor passes the lock along to the thread that next receives the processor. Because thread switching follows this co-routine style of control, a thread knows that when it comes out of ENTER_PROCESS_LAYER, the lock will have been released.

Thread switching relies on a detailed understanding of the processor and the procedure call convention. In most systems the implementation of thread switching is more complex than the implementation in figure 5–23, because we made several assumptions that often don't hold in real systems: there is a fixed number of threads, all threads are runnable, and scheduling threads round-robin is an acceptable policy. In the next sections, we will eliminate some of these assumptions.

3. Creating and terminating threads

The example YIELD procedure supports only a fixed number of threads. A full-blown thread manager allows threads to be created and terminated on demand. To support a variable number of threads, we would need to modify the implementation of ALLOCATE_THREAD and extend the thread manager with the following procedures:

- EXIT_THREAD (): destroy and clean up the calling thread. When a thread is done with its job, it invokes EXIT_THREAD to release its state.

- DESTROY_THREAD (*id*): destroy the thread identified by *id*. In some cases, one thread may need to terminate another thread. For example, a user may have started a thread that turns out to have a programming error such as an endless loop, and thus the user wants to terminate it. For these cases, we might want to provide a procedure to destroy a thread.

For the most part the implementation of these procedures is relatively straightforward, but there are a few subtle issues. For example: if threads can terminate, we have to fix the problem that the previous code required at least as many threads as processors. To get at these issues, we detail their implementation. First, we create a separate thread for each processor (which we will call a *processor-layer thread*,or *processor thread* for short), which runs the procedure SCHEDULER (see figure 5–24). The way to think about this setup is that the

```
1     shared structure {            // each processor maintains the following information:
2           integer topstack;       // value of stack pointer
3           byte reference stack;// preallocated stack for this processor
4           integer thread_id;      // identity of thread currently running on this processor
5     } processor_table[7];
6     shared structure thread {     // each thread maintains the following information:
7           integer topstack;       // value of the stack pointer
8           integer state;          // RUNNABLE, RUNNING, or FREE
9           boolean kill_or_continue;// should this thread be terminated? initially set to CONTINUE
10          byte reference stack;// stack for this thread
11    } threadtable[7];

12    procedure YIELD () {
13          ENTER_PROCESSOR_LAYER(GET_THREAD_ID(), CPUID, CONTINUE); //goes to line 24
14    }
15
16    procedure SCHEDULER () {
17          while shutdown = FALSE do {
18                ACQUIRE (threadtable_lock);
19                for i from 0 until 7 do {
20                      if threadtable[i].state = RUNNABLE then {
21                            threadtable[i].state ← RUNNING;
22                            processor_table[CPUID].thread_id ← i;
23                            EXIT_PROCESSOR_LAYER(CPUID, i);
24                            if threadtable[i].kill_or_continue = KILL then {
25                                  threadtable[i].state ← FREE;
26                                  DEALLOCATE(threadtable[i].stack);
27                                  threadtable[i].kill = CONTINUE;
28                            }
29                      }
30                }
31                RELEASE (threadtable_lock);
32          }
33    }

34    procedure ENTER_PROCESSOR_LAYER(tid, processor, kill_or_continue) {
35          ACQUIRE (threadtable_lock);
36          threadtable[tid].kill_or_continue ← kill_or_continue;
37          threadtable[tid].state ← RUNNABLE;
38          threadtable[tid].topstack ← SP;              // save state: store yielding's thread SP
39          SP ← processor_table[processor].topstack; // dispatch: load SP of processor thread;
40    }

41    procedure EXIT_PROCESSOR_LAYER(processor, tid) { // transfers control to after line 14
42          processor_table[processor].topstack ← SP; // save state: store processor thread's SP
43          SP ← threadtable[tid].topstack;              // dispatch: load SP of thread;
44          RELEASE (threadtable_lock);
45    }
```

Figure 5–24: YIELD with support for dynamic thread creation and deletion. Control flow is not obvious because those procedures reload SP, which changes the place to which they return. The procedure called on line 13 returns control to line 24, and the procedure called on line 23 returns control to line 14. Figure 5–25 shows the control flow graphically.

SCHEDULER runs in the processor layer we, and it virtualizes its processor. The reason to have a processor thread per processor is because a thread in the thread layer (a *thread-layer thread*) cannot deallocate its own stack since it cannot call a procedure (e.g., DEALLOCATE or YIELD) on a stack that it has released. Instead, we set it up so that the processor-layer thread cleans up thread-layer threads. When starting the operating system kernel (e.g., after turning the computer on), the kernel creates processor-layer threads as follows:

```
procedure RUN_PROCESSORS {
        foreach processor do {
                allocate stack and set up a processor thread;
                shutdown ← FALSE;
                SCHEDULER ();
                deallocate processor thread stack;
                halt processor;
        }
}
```

This procedure allocates a stack and sets up a processor thread for each processor. This thread runs the scheduler procedure until some procedure sets the global variable *shutdown* to TRUE. Then, the computer restarts or halts.

We first revisit YIELD with this setup and then see how this generalization supports thread creation and deletion. Using a separate processor thread, switching a processor from one thread-layer thread to another actually requires two thread switches: one from the thread that is releasing its processor to the processor thread, and then one from the processor thread to the thread that is to receive the processor (see figure 5–25). In more detail, let's suppose, as before, that thread 0 calls YIELD on processor A, and thread 6 is runnable and has called YIELD earlier. Thread 0 switches to the processor thread by invoking ENTER_PROCESSOR_LAYER (line 13). The implementation of ENTER_PROCESSOR_LAYER is almost identical to ENTER_PROCESSOR_LAYER of figure 5–23: it acquires the lock *threadtable_lock* and saves the stack pointer in the calling thread's *threadtable* entry, but loads a new stack pointer from CPUID's *processor_table* entry and maintains a *kill* variable (which we will discuss below). When ENTER_PROCESSOR_LAYER returns, it will switch to the processor thread and resume at line 24 (right after EXIT_PROCESSOR_LAYER).

The processor thread will cycle through the thread table until it hits thread 6, which is runnable. The SCHEDULER sets thread 6's state to RUNNING (line 21), records that thread 6 will run on this processor (line 22), and invokes EXIT_PROCESSOR_LAYER, to switch the processor to thread 6 (line 23). EXIT_PROCESSOR_LAYER saves the scheduler's thread state into CPUID's entry in the *processor_table* and loads thread 6's state in the processor, which will resume execution at line 40 right at the end of ENTER_PROCESS_LAYER (because thread 6's last procedure call was to the procedure ENTER_PROCESSOR_LAYER). ENTER_PROCESSOR_LAYER will return to YIELD, which returns to the program of thread 6 that originally called YIELD.

With this setup of thread switching in place, we can return to creating and deallocating threads dynamically. To keep track if a *threadtable* entry is in use, we extend the set of possible states of each entry with the additional state FREE. Now we can implement ALLOCATE_THREAD as follows:

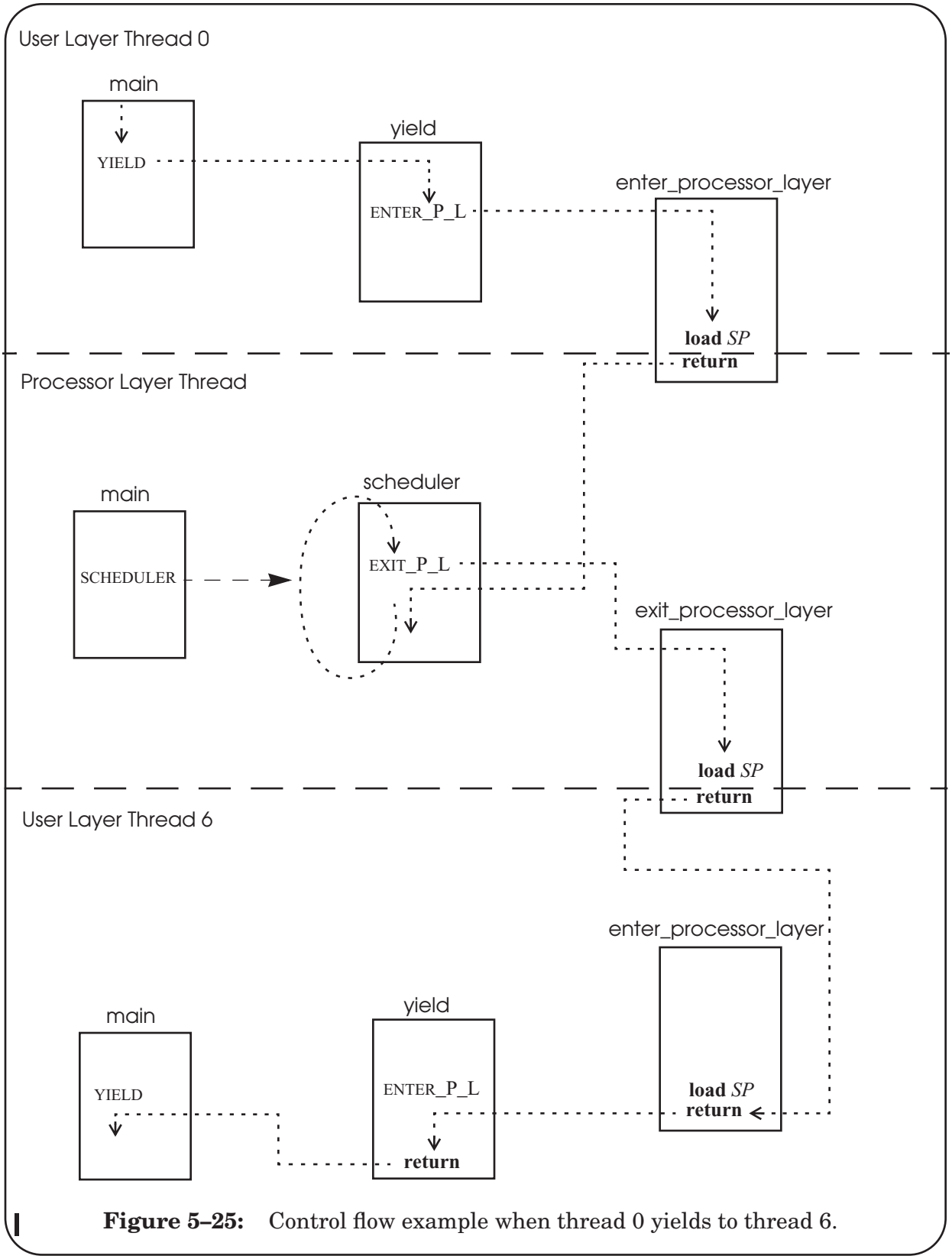


Figure 5–25: Control flow example when thread 0 yields to thread 6.

1. allocate space in memory for a new stack;

2. place on the new stack an empty frame containing just a return address and initialize that return address with the address of EXIT_THREAD;

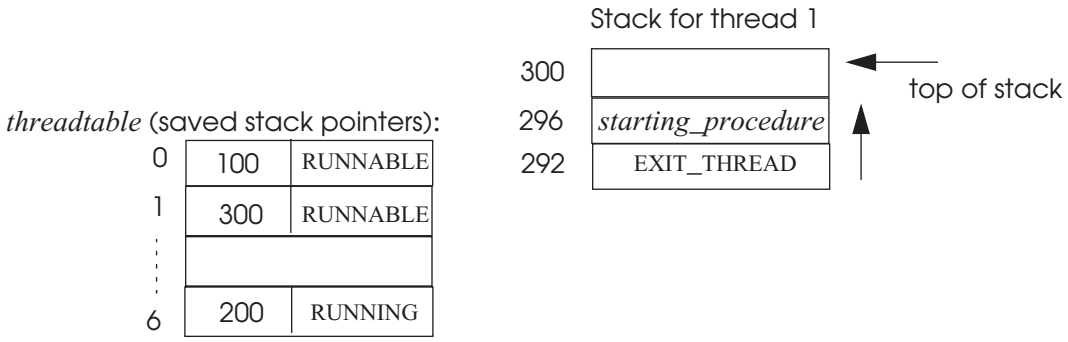
3. place on the stack a second empty frame containing just a return address, and initialize this return address with the address of *starting_procedure*;

4. find an entry in the thread table that is FREE and initialize that entry for the new thread in the thread table by storing the top of the new stack;

5. set the state of newly-created thread to RUNNABLE.

If the thread manager cannot complete these steps (e.g., all entries in the thread table are in use), then THREAD_ALLOCATE returns an error.

To illustrate this implementation, consider the following state for a newly-created thread 1:



Thread 1's stack is located at address 292 and its saved stack pointer is 300. With this initial setup, it appears that EXIT_THREAD called the procedure STARTING_PROCEDURE, and thread 1 is about to return to this procedure. Thus, when SCHEDULER selects this thread, its return statement will go to the procedure *starting_procedure*. In detail, when the scheduler selects the new thread (1) as the next thread to execute, it sets its stack pointer to the top of the new stack (300) in EXIT_PROCESSOR_LAYER. When the processor returns from EXIT_PROCESSOR_LAYER, it will set its program counter to the address on top of the stack (*starting_procedure*), and start execution at that location. The procedure *starting_thread* releases *threadtable_lock* and the new thread is running.

With this initial setup, when a thread finishes the procedure *starting_procedure*, it returns using the standard procedure return convention. Since the THREAD_CREATE procedure has put the address of the EXIT_THREAD procedure on the stack, this return transfers control to the first instruction of the EXIT_THREAD procedure.

The EXIT_THREAD procedure can be implemented as follows:

```
1    procedure EXIT_THREAD() {
2        ENTER_PROCESSOR_LAYER(GET_THREAD_ID(), CPUID, KILL);
3    }
```

EXIT_THREAD invokes ENTER_PROCESSOR_LAYER, passing as an argument a flag indicating that this thread wants to terminate. Based on this flag, ENTER_PROCESSOR_LAYER sets the *kill* variable for thread (line 36) and switches the processor to the processor thread. The processor thread checks the variable *kill* on line 24 to see if a thread is done and, if so, marks the thread entry as reusable (line 25) and deallocates its stack (line 26). Since no thread is using that stack, it is safe to deallocate it.

The implementation of DESTROY_THREAD is a bit tricky too, because the target thread to be destroyed might be running on one of the processors, so the calling thread cannot just free the target thread's stack; the processor running the target thread must do that. We can achieve that in an indirect way. DESTROY_THREAD just sets the *kill* variable of the target thread to TRUE and returns. When a thread invokes YIELD and enter the processor layer, the processor thread will check this variable and release the thread's resources. (Section 4 will show how to ensure that each thread running on a processor will call YIELD at least occasionally.)

The implementation described for allocating and deallocating threads is just one of many ways of handling creating and destroying threads. If one opens up the internals of half a dozen different thread packages, one will find half a dozen quite different ways to handle launching and terminating threads. The goal of this section was not to exhibit a complete catalog, but rather by illustrating one example in detail to dispel any mystery and expose the main issues that every implementation must address.

4. Enforcing modularity with threads: preemptive scheduling

The thread manager described so far switches to a new thread only when a thread calls YIELD. This scheduling policy, where a thread continues to run until it gives up its processor, is called *nonpreemptive scheduling*. It can be problematic because the length of time a thread holds its processor is entirely under the control of the thread itself. If, for example, a programming error sends one thread into an endless loop, no other thread will ever be able to use that processor again. Nonpreemptive scheduling might be acceptable for a single module that has several threads (e.g., a Web server that has several threads to increase performance), but not for several modules.

Some systems partially address this problem by having a gentlemen's agreement called *cooperative scheduling (*which in the literature sometimes is called *cooperative multitasking*): every thread is supposed to call YIELD periodically, for instance, once per 100 milliseconds. This solution is not very robust, since it relies on modules behaving well and not having any errors. If a programmer forgets to put in a YIELD, or the program accidently gets into an endless loop that does not include a YIELD, that processor will no longer participate in the gentlemen's agreement. If, as is common with cooperative multitasking designs, there is only a single processor, the processor may appear to freeze, since the other threads won't have an opportunity to make progress.

To enforce modularity among multiple threads, the operating system thread manager must ensure thread switching by using what is called *preemptive scheduling*. The thread manager must force a thread to give up its processor after, for example, 100 milliseconds. The thread manager can implement preemptive scheduling by setting the interval timer of a clock device. When the timer expires, the clock triggers an interrupt, switching to kernel mode in the processor layer. The clock interrupt handler can then invoke an exception handler, which