Chapter 4

Concurrency

Given that the operating system supports multiple processes, there may be various interactions among them. We will study three rather different types of interactions:

- Access to shared operating system data structures.

  This issue is concerned with internal operating system integrity. The problem is that several processes may request the operating system to take related actions, which require updates to internal operating system data structures. Note that there is only one operating system, but many processes. Therefore the operating system data structures are shared in some way by all the processes. Updates to such shared data structures must be made with care, so that data is not compromised.

- Deadlock due to resource contention.

  This issue is concerned with the resource management functionality of the operating system. Consider a scenario in which one application acquires lots of memory, and another acquires access to a tape drive. Then the first application requests the tape, and the second requests more memory. Neither request can be satisfied, and both applications are stuck, because each wants what the other has. This is called deadlock, and should be avoided.

- Mechanisms for user-level inter-process communication.

  This issue is concerned with the abstraction and services functionality of the operating system. The point is that multiple processes may benefit from interacting with each other, e.g. as part of a parallel or distributed application. The operating system has to provide the mechanisms for processes to identify each other and to move data from one to the other.

We'll discuss the first two here, and the third in Chapter 12.

4.1 Mutual Exclusion for Shared Data Structures

An operating system is an instance of concurrent programming. This means that multiple activities may be ongoing at the same time. For example, a process may make a system call, and while the system call is running, an interrupt may occur. Thus two different executions of operating system code — the system call and the interrupt handler — are active at the same time.

4.1.1 Concurrency and the Synchronization Problem

Concurrency can happen on a single processor

Concurrency does not necessarily imply parallelism.

In a *parallel* program, different activities actually occur simultaneously on different processors. That is, they occur at the same time in different locations.

In a *concurrent* program, different activities are interleaved with each other. This may happen because they really occur in parallel, but they may also be interleaved on the same processor. That is, one activity is started, but before it completes it is put aside and another is also started. Then this second activity is preempted and a third is started. In this way many different activities are underway at the same time, although only one of them is actually running on the CPU at any given instant.

The operating system is a concurrent program

An operating system is such a concurrent program. It has multiple entry points, and several may be active at the same time. For example, this can happen if one process makes a system call, blocks within the system call (e.g. waiting for a disk operation to complete), and while it is waiting another process makes a system call. Another example is that an interrupt may occur while a system call is being serviced. In this case the system call handler is preempted in favor of the interrupt handler, which is another operating system activity.

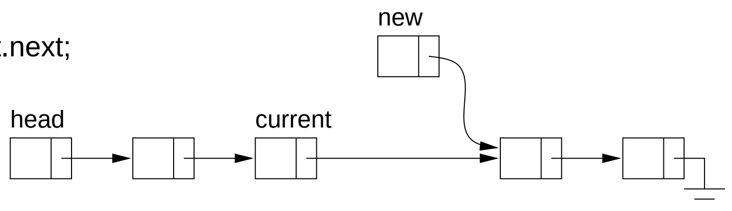
As operating system activities are often executed on behalf of user processes, in the sequel we will usually talk of processes that are active concurrently. But we typically mean operating system activities on behalf of these processes. On the other hand, user-level processes (or threads) may also cooperatively form a concurrent program, and exactly the same problems and solutions apply in that case as well.

Concurrent updates may corrupt data structures

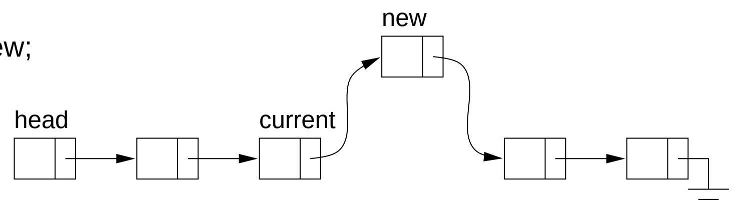
The problem with concurrency is that various operations require multiple steps in order to complete. If an activity is interrupted in the middle, the data structures on which it operated may be in an inconsistent state. Such a situation in which the outcome depends on the relative speed and the order of interleaving is called a *race condition*.

Consider adding a new element to a linked list. The code to insert `new` after `current` is trivial and consists of two statements:

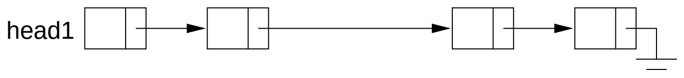
```
new.next = current.next;
```



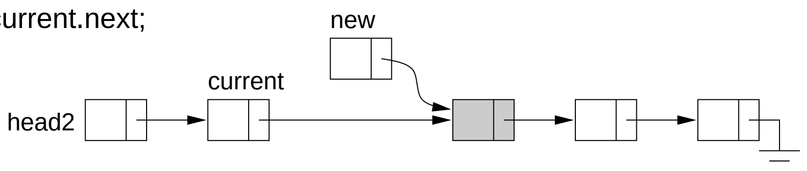
```
current.next = &new;
```



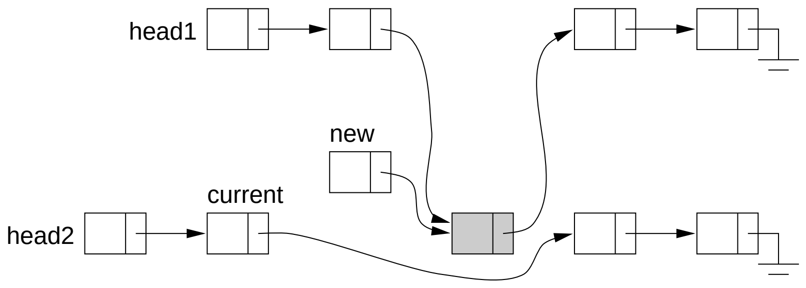
But what if the activity is interrupted between these two statements? `new.next` has already been set to point to another element, but this may no longer be valid when the activity resumes! For example, the intervening activity may *delete* the pointed element from the list, and insert it into another list! Let's look at what happens in detail. Initially, there are two lists, and we are inserting the new element into one of them:



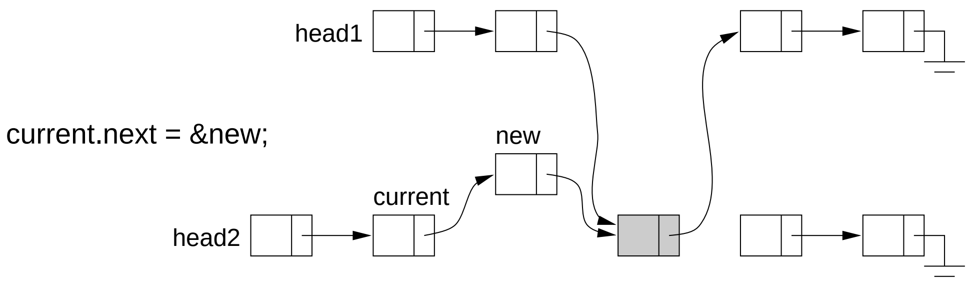
```
new.next = current.next;
```



Now we are interrupted. The interrupting activity moves the gray item we are pointing at into the other list, and updates `current.next` correctly. It doesn't know about our new item, because we have not made it part of the list yet:



However, we don't know about this intervention. Therefore we overwrite `current.next` and make it point to our new item:



As a result, the two lists become merged from the gray element till their end, and two items are completely lost: they no longer appear in any list! It should be clear that this is very very bad.

Exercise 64 *What would happen if the interrupting activity also inserted a new item after* `current`*, instead of moving the gray item? And what would happen if the interrupting activity inserted a new item after the gray item?*

Does this happen in real life? You bet it does. Probably the most infamous example is from the software controlling the Therac-25, a radiation machine used to treat cancer patients. A few *died* due to massive overdose induced by using the wrong data. The problem was traced to lack of synchronization among competing threads [8].

4.1.2 Mutual Exclusion Algorithms

The solution is to define mutually exclusive critical sections

The solution is that all the steps required to complete an action must be done *atomically*, that is, as a single indivisible unit. The activity performing them must not be interrupted. It will then leave the data structures in a consistent state.

This idea is translated into program code by identifying *critical sections*. Thus the code to insert an item into a linked list will be

```
begin_critical_section;
new.next = current.next;
current.next = &new;
end_critical_section;
```

It is up to the system, by means of implementing the code for `begin_critical_section` and `end_critical_section`, to ensure that only one activity is in the critical section at any time. This property is called *mutual exclusion*. Each activity, by virtue of passing the `begin_critical_section` code and being in the critical section, excludes all other activities from also being in the critical section.

Exercise 65 *Does the atomicity of critical sections imply that a process that is in a critical section may not be preempted?*

Mutual exclusion can be achieved by sophisticated algorithms

But how do you implement critical sections? In the 1960s the issue of how and whether concurrent processes can coordinate their activities was a very challenging question. The answer was that they can, using any of several subtle algorithms. While these algorithms are not used in real systems, they have become part of the core of operating system courses, and we therefore review them here. More practical solutions (that are indeed used) are introduced below.

The basic idea in all the algorithms is the same. They provide code that implements `begin_critical_section` and `end_critical_section` by using an auxiliary shared data structure. For example, we might use shared variables in which each process indicates that it is going into the critical section, and then checks that the others are not. With two processes, the code would be

```
process 1:                         process 2:

going_in_1 = TRUE;                 going_in_2 = TRUE;
while (going_in_2) /*empty*/;      while (going_in_1) /*empty*/;

critical section                   critical section

going_in_1 = FALSE;                going_in_2 = FALSE;
```

where `going_in_1` and `going_in_2` are shared memory locations that can be accessed by either process. Regrettably, this code is not very good, as it may create a deadlock: if process 1 sets `going_in_1` to `TRUE`, is interrupted, and then process 2 sets `going_in_2` to `TRUE`, the two processes will wait for each other indefinitely.

Exercise 66 *Except for the problem with deadlock, is this code at least correct in the sense that if one process is in the critical section then it is guaranteed that the other process will not enter the critical section?*

A better algorithm is the following (called Peterson's algorithm [9]):

```
process 1:                         process 2:

going_in_1 = TRUE;                 going_in_2 = TRUE;
turn = 2;                          turn = 1;
while (going_in_2 && turn==2)      while (going_in_1 && turn==1)
    /*empty*/;                         /*empty*/;

critical section                   critical section

going_in_1 = FALSE;                going_in_2 = FALSE;
```

This is based on using another shared variable that indicates whose turn it is to get into the critical section. The interesting idea is that each process tries to let the *other* process get in first. This solves the deadlock problem. Assume both processes set their respective `going_in` variables to `TRUE`, as before. They then both set `turn` to conflicting values. One of these assignments prevails, and the process that made it then waits. The other process, the one whose assignment to `turn` was overwritten, can then enter the critical section. When it exits, it sets its `going_in` variable to `FALSE`, and then the waiting process can get in.

The bakery algorithm is rather straightforward

While Peterson's algorithm can be generalized to more than two processes, it is not very transparent. A much simpler solution is provided by Lamport's bakery algorithm [7], which is based on the idea that processes take numbered tickets, and the one with the lowest number gets in. However, there remains the problem of assigning the numbers. Because we have more processes, we need more shared variables. The algorithm uses two arrays: `i_am_choosing[`N`]`, initialized to `FALSE`, and `my_ticket[`N`]` initialized to `0`. The code for process i (out of a total of N) is[1]

```
i_am_choosing[i] = TRUE;
for (j=0 ; j<N ; j++) {
    if (my_ticket[i] <= my_ticket[j])
        my_ticket[i] = my_ticket[j] + 1;
}
i_am_choosing[i] = FALSE;

for (j=0 ; j<N ; j++) {
    while (i_am_choosing[j]) /*empty*/;
    while ((my_ticket[j] > 0) &&
           ((my_ticket[j] < my_ticket[i]) ||
           ((my_ticket[j] == my_ticket[i]) && (j < i)))) /*empty*/;
}

critical section

my_ticket[i] = 0;
```

The first block (protected by `i_am_choosing[`i`]`) assigns the ticket number. Note that there are no loops here, so no process will be delayed, so tickets indeed represent the arrival order.

However, note that several processes may be doing this at the same time, so more than one process may end up with the same number. This is solved when we compare our number with all the other processes, in the second `for` loop. First, if we encounter a process that is in the middle of getting its ticket, we wait for it to actually get the

[1]Actually, this is a slight paraphrase.

ticket. Then, if the ticket is valid and smaller than ours, we wait for it to go through the critical section (when it gets out, it sets its ticket to 0, which represents invalid). Ties are simply solved by comparing the IDs of the two processes.

Exercise 67 *A simple optimization of this algorithm is to replace the loop of choosing the next ticket with a global variable, as in*

```
my_ticket[i] = current_ticket;
current_ticket = current_ticket + 1;
```

Note, however, that incrementing the global variable is typically not atomic: each process reads the value into a register, increments it, and writes it back. Can this cause problems? Hint: if it would have worked, we wouldn't use the loop.

There are four criteria for success

In summary, this algorithm has the following desirable properties:

1. *Correctness:* only one process is in the critical section at a time.
2. *Progress:* there is no deadlock, and some process will eventually get into the critical section.
3. *Fairness:* there is no starvation, and no process will wait indefinitely while other processes continuously sneak into the critical section.

   Exercise 68 *Does the danger of starvation imply that the system is overloaded?*

4. *Generality:* it works for N processes.

Exercise 69 *Can you show that all these properties indeed hold for the bakery algorithm?*

Details: A Formal Proof

After showing that the algorithms are convincing using rapid hand waving, we now turn to a formal proof. The algorithm used is the n-process generalization of Peterson's algorithm, and the proof is due to Hofri [3].

The algorithm uses two global arrays, `q[`n`]` and `turn[`$n-1$`]`, both initialized to all 0's. Each process p_i also has three local variables, `j`, `k`, and its index `i`. The code is

```
for (j=1; j<n; j++) {
    q[i] = j;
    turn[j] = i;
    while ((∃k≠i s.t.  q[k]≥j) && (turn[j] = i))
        /*empty*/;
}
critical section
q[i] = 0;
```

The generalization from the two-process case is that entering the critical section becomes a multi-stage process. Thus the Boolean `going_in` variable is replaced by the integer variable `q[i]`, originally 0 to denote no interest in the critical section, passing through the values 1 to $n-1$ to reflect the stages of the entry procedure, and finally hitting n in the critical section itself. But how does this work to guarantee mutual exclusion? Insight may be gained by the following lemmas.

Lemma 4.1 *A process that is ahead of all others can advance by one stage.*

Proof: The formal definition of process i being ahead is that `∀k≠i, q[k] < q[i]`. Thus the condition in line 4 is not satisfied, and `j` is incremented and stored in `q[i]`, thus advancing process i to the next stage. ▮

Lemma 4.2 *When a process advances from stage j to stage $j+1$, it is either ahead of all other processes, or there are other processes at stage j.*

Proof: The first alternative is a rephrase of the previous lemma: if a process that is ahead of all others can advance, than an advancing process may be ahead of all others. The other alternative is that the process is advancing because `turn[j]≠i`. This can only happen if at least one other process reached stage j after process i, and modified `turn[j]`. Of these processes, the last one to modify `turn[j]` must still be at stage j, because the condition in line 4 evaluates to true for that process. ▮

Lemma 4.3 *If there are at least two processes at stage j, there is at least one process at each stage $k \in \{1, \ldots, j-1\}$.*

Proof: The base step is for $j = 2$. Given a single process at stage 2, another process can join it only by leaving behind a third process in stage 1 (by Lemma 4.2). This third process stays stuck there as long as it is alone, again by Lemma 4.2. For the induction step, assume the Lemma holds for stage $j-1$. Given that there are two processes at stage j, consider the instance at which the second one arrived at this stage. By Lemma 4.2, at that time there was at least one other process at stage $j-1$; and by the induction assumption, this means that all the lower stages were also occupied. Moreover, none of these stages could be vacated since, due to Lemma 4.2. ▮

Lemma 4.4 *The maximal number of processes at stage j is $n-j+1$.*

Proof: By Lemma 4.3, if there are more than one process at stage j, all the previous $j-1$ stages are occupied. Therefore at most $n-(j-1)$ processes are left for stage j. ▮

Using these, we can envision how the algorithm works. The stages of the entry protocol are like the rungs of a ladder that has to be scaled in order to enter the critical section. The algorithm works by allowing only the top process to continue scaling the ladder. Others are restricted by the requirement of having a continuous link of processes starting at the bottom rung. As the number of rungs equals the number of processors minus 1, they are prevented from entering the critical section. Formally, we can state the following:

Theorem 4.1 *Peterson's generalized algorithm satisfies the conditions of correctness, progress, and fairness (assuming certain liveness properties).*

Proof: According to Lemma 4.4, stage $n-1$ can contain at most two processes. Consider first the case where there is only one process at this stage. If there is another process currently in the critical section itself (stage n), the process at stage $n-1$ must stay there according to Lemma 4.2. Thus the integrity of the critical section is maintained. If there are two processes in stage $n-1$, then according to Lemma 4.3 all previous stages are occupied, and the critical section is vacant. One of the two processes at stage $n-1$ can therefore enter the critical section. The other will then stay at stage $n-1$ because the condition in line 4 of the algorithm holds. Thus the integrity of the critical section is maintained again, and correction is proved.

Progress follows immediately from the fact that some process must be either ahead of all others, or at the same stage as other processes and not the last to have arrived. For this process the condition in line 4 does not hold, and it advances to the next stage.

Fairness follows from the fact that the last process to reach a stage is the one that cannot proceed, because the stage's `turn` cell is set to its ID. Consider process p, which is the last to arrive at the first stage. In the worst case, all other processes can be ahead of it, and they will enter the critical section first. But if they subsequently try to enter it again, they will be behind process p. If it tries to advance, it will therefore manage to do so, and one of the other processes will be left behind. Assuming all non-last processes are allowed to advance to the next stage, process p will have to wait for no more than $n-j$ other processes at stage j, and other processes will overtake it no more than n times each. ∎

To read more: A full description of lots of wrong mutual exclusion algorithms is given by Stallings [11, Sect. 5.2]. The issue of sophisticated algorithms for mutual exclusion has been beaten to death by Lamport [5, 6, 4].

An alternative is to augment the instruction set with atomic instructions

As noted, the problem with concurrent programming is the lack of atomicity. The above algorithms provide the required atomicity, but at the cost of considerable headaches. Alternatively, the hardware may provide a limited measure of atomicity, that can then be amplified.

The simplest example is the test_and_set instruction. This instruction operates on a single bit, and does two things atomically: it reads the bit's value (0 or 1), and then it sets the bit to 1. This can be used to create critical sections as follows, using a single bit called `guard`:

```
while ( test_and_set(guard) ) /*empty*/;
critical section
guard = 0;
```

Because the hardware guarantees that the test_and_set is atomic, it guarantees that only one process will see the value 0. When that process sees 0, it atomically sets the value to 1, and all other processes will see 1. They will then stay in the while loop until the first process exits the critical section, and sets the bit to 0. Again, only one other process will see the 0 and get into the critical section; the rest will continue to wait.

Exercise 70 *The compare_and_swap instruction is defined as follows:*

```
compare_and_swap( x, old, new )
   if (x == old)
      x = new;
      return SUCCESS;
   else
      return FAIL
```

where `x` *is a variable,* `old` *and* `new` *are values, and the whole thing is done atomically by the hardware. How can you implement a critical section using this instruction?*

4.1.3 Semaphores and Monitors

Programming is simplified by the abstraction of semaphores

Algorithms for mutual exclusion are tricky, and it is difficult to verify their exact properties. Using hardware primitives depends on their availability in the architecture. A better solution from the perspective of operating system design is to use some more abstract mechanism.

The mechanism that captures the abstraction of inter-process synchronization is the semaphore, introduced by Dijkstra [1]. Semaphores are a new type, that provides only two operations:

- the `P` operation checks whether the semaphore is free. If it is, it occupies the semaphore. But if the semaphore is already occupied, it waits till the semaphore will become free.
- The `V` operation releases an occupied semaphore, and frees one blocked process (if there are any blocked processes waiting for this semaphore).

The above specification describes the semantics of a semaphore. The most common way to implement this specification is by using an integer variable initialized to 1. The value of 1 indicates that the semaphore is free. A value of 0 or less indicates that it is occupied. The `P` operation decrements the value by one, and blocks the process if it becomes negative. The `V` operation increments the value by one, and frees a waiting process. This implementation is described by the following pseudo-code:

```
class semaphore
   int value = 1;
   P() {
      if (--value < 0)
         block_this_proc();
   }
   V() {
      if (value++ < 0)
         resume_blocked_proc();
   }
```

The reasons for calling the operations P and V is that they are abbreviations of the Dutch words "proberen" and "verhogen". Speakers of Hebrew have the advantage of regarding them as abbreviations for פנות and ותור. In English, the words *wait* and *signal* are sometimes used instead of P and V, respectively.

Exercise 71 *"Semáforo" means "traffic light" in Spanish. What is similar and what is different between semaphores and traffic lights? How about the analogy between semaphores and locks?*

Using semaphores, it is completely trivial to protect a critical section. If we call the semaphore mutex (a common name for mutual exclusion mechanisms), the code is

```
P(mutex);
critical section
V(mutex);
```

Thus semaphores precisely capture the desired semantics of begin_critical_section and end_critical_section. In fact, they more generally capture an important *abstraction* very succinctly — they are a means by which the process can tell the system that it is waiting for something.

Semaphores can be implemented efficiently by the operating system

The power of semaphores comes from the way in which they capture the essence of synchronization, which is this: the process needs to wait until a certain condition allows it to proceed. The important thing it that the abstraction does not specify *how* to implement the waiting.

All the solutions we saw so far implemented the waiting by burning cycles. Any process that had to wait simply sat in a loop, and continuously checked whether the awaited condition was satisfied — which is called *busy waiting*. In a uniprocessor system with multiple processes this is very very inefficient. When one process is busy waiting for another, it is occupying the CPU all the time. Therefore the awaited process cannot run, and cannot make progress towards satisfying the desired condition.

The P operation on a semaphore, on the other hand, conveys the information that the process cannot proceed if the semaphore is negative. If this is the case, the operating system can then preempt this process and run other processes in its place. Moreover, the process can be blocked and placed in a queue of processes that are waiting for this condition. Whenever a V operation is performed on the semaphore, one process is removed from this queue and placed on the ready queue.

And they have additional uses beyond mutual exclusion

Semaphores have been very successful, and since their introduction in the context of operating system design they have been recognized as a generally useful construct for concurrent programming. This is due to the combination of two things: that they capture the abstraction of needing to wait for a condition or event, and that they can be implemented efficiently as shown above.

An important advantage of the semaphore abstraction is that now it is easy to handle multiple critical sections: we can simply declare a separate semaphore for each one. More importantly, we can use the same semaphore to link several critical sections of code that manipulate the same shared data structures. In fact, a semaphore can be considered as a mechanism for *locking* data structures as described below.

Exercise 72 *Consider a situation in which we have several linked lists, and occasionally need to move an item from one list to another. Is the definition of a unique semaphore for each pair of lists a good solution?*

Moreover, the notion of having to wait for a condition is not unique to critical sections (where the condition is “no other process is executing this code”). Thus semaphores can be used for a host of other things.

One simple example comes from resource allocation. Obviously a semaphore can represent the allocation of a resource. But it also has the interesting property that we can initialize the semaphore to a number different from 1, say 3, thus allowing up to 3 processes “into” the semaphore at any time. Such *counting semaphores* are useful for allocating resources of which there are several equivalent instances.

Exercise 73 *A common practical problem is the producer / consumer problem, also known as the bounded buffer problem. This assumes two processes, one of which produces some data, while the other consumes this data. The data is kept in a finite set of buffers in between. The problem is to keep the producer from overflowing the buffer space, and to prevent the consumer from using uninitialized data. Give a solution using semaphores.*

Monitors provide an even higher level of abstraction

Another abstraction that was introduced for operating system design is that of monitors [2]. A monitor encapsulates some state, and provides methods that operate on that state. In addition, it guarantees that these methods are executed in a mutually exclusive manner. Thus if a process tries to invoke a method from a specific monitor, and some method of that monitor is already being executed by another process, the first process is blocked until the monitor becomes available.

A special case occurs when a process cannot complete the execution of a method and has to wait for some event to occur. It is then possible for the process to enqueue itself *within* the monitor, and allow other processes to use it. Later, when the time is right, some other process using the monitor will resume the enqueued process.

To read more: Additional issues in concurrent programming are covered in Sections 5.4 through 5.7 of Stallings [12]. Similar material is covered in Sections 6.4 through 6.9 of Silberschatz and Galvin [10].

4.1.4 Locks and Disabling Interrupts

So where are we?

To summarize, what we have seen so far is the following.

First, the mutual exclusion problem can be solved using a bare bones approach. It is possible to devise algorithms that only read and write shared variables to determine whether they can enter the critical section, and use busy waiting to delay themselves if they cannot.

Second, much simpler algorithms are possible if hardware support is available in the form of certain atomic instructions, such as test_and_set. However, this still uses busy waiting.

Third, the abstraction of semaphores can be used to do away with busy waiting. The P operation on a semaphore can be used to tell the system that a process needs to wait for a condition to hold, and the system can then block the process.

However, it is actually not clear how to implement semaphores within the operating system. The pseudo-code given on page 91 includes several related operations, e.g. checking the value of the semaphore and updating it. But what happens if there is an interrupt between the two? We need to do this in a critical section... Reverting to using busy waiting to implement this "internal" critical section may void the advantage of blocking in the semaphore!

The simple solution is to disable interrupts

In order to avoid busy waiting, we need a mechanism that allows selected operating system code to run without interruption. As asynchronous interruptions are caused by external interrupts, this goal can be achieved by simply disabling all interrupts. Luckily, such an option is available on all hardware platforms. Technically, this is done by setting the interrupt level in the PSW. The processor then ignores all interrupts lower than the level that was chosen.

Exercise 74 *Should the disabling and enabling of interrupts be privileged instructions?*

Example: classical Unix used a non-preemptive kernel

A rather extreme solution to the issue of mutual exclusion is to consider the whole kernel as a critical section. This was done in early versions of the Unix system, and was one of the reasons for that system's simplicity. This was accomplished by disabling all interrupts when in the kernel. Thus kernel functions could complete without any interruption, and leave all data structures in a consistent state. When they returned, they enabled the interrupts again.

Exercise 75 *The Unix process state graph on page 54 shows an arrow labeled "preempt" leaving the "kernel running" state. How does this fit in with the above?*

The problem with a non-preemptive kernel is that it compromises the responsiveness of the system, and prevents it from reacting in real time to various external conditions. Therefore modern versions of Unix do allow preemption of kernel functions, and resort to other means for synchronization.

Exercise 76 *Does the blocking of interrupts provide the required atomicity on multiprocessors?*

Locks can be used to express the desired granularity

Disabling interrupts can be viewed as locking the CPU: the current process has it, and no other process can gain access. Thus the practice of blocking all interrupts whenever a process runs in kernel mode is akin to defining a single lock, that protects the whole kernel. But this may be stronger than what is needed: for example, if one process wants to open a file, this should not interfere with another that wants to allocate new memory.

The solution is therefore to define multiple locks, that each protect a part of the kernel. These can be defined at various granularities: a lock for the whole file system, a lock for a system call, or a lock for a data structure. By holding only the necessary locks, the restrictions on other processes are reduced.

Note the shift in focus from the earlier parts of this chapter: we are no longer talking about critical sections of code that should be executed in a mutually exclusive manner. Instead, we are talking of resources (such as data structures), represented by locks, which need to be used in a mutually exclusive manner. This shifts the focus from the artifact (the code handling the data structure) to the substance (the data structure's function and why it is being accessed).

And locks can have special semantics

Once locks are in place, it is possible to further reduce the restrictions on system activities by endowing them with special semantics. For example, if two processes only need to read a certain data structure, they can do so concurrently with no problem. The data structure needs to be locked only if one of them modifies it, leading to possible inconsistent intermediate states that should not be seen by other processes.

This observation has led to the design of so called *readers-writers locks*. Such locks can be locked in two modes: locked for reading and locked for writing. Locks for reading do not conflict with each other, and many processes can obtain a read-lock at the same time. But locks for writing conflict with all other accesses, and only one can be held at a time.

Exercise 77 *Write pseudo-code to implement the three functions of a readers-writers lock: read_lock, write_lock, and release_lock. Hint: think about fairness and starvation.*

4.1.5 Multiprocessor Synchronization

But what about multiprocessor systems? The operating system code running on each processor can only block interrupts from occurring *on that processor*. Related code running on another processor will not be affected, and might access the same data structures.

The only means for synchronizing multiple processors is atomic instructions that are implemented by the hardware in some shared manner — for example, the atomicity may be achieved by locking the shared bus by which all the processors access memory.

The problem with using these atomic instructions is that we are back to using busy waiting. This is not so harmful for performance because it is only used to protect very small critical sections, that are held for a very short time. A common scenario is to use busy waiting with test_and_set on a variable that protects the implementation of a semaphore. The protected code is very short: just check the semaphore variable, and either lock it or give up because it has already been locked by someone else. Blocking to wait for the semaphore to become free is a local operation, and need not be protected by the test_and_set variable.

4.2 Resource Contention and Deadlock

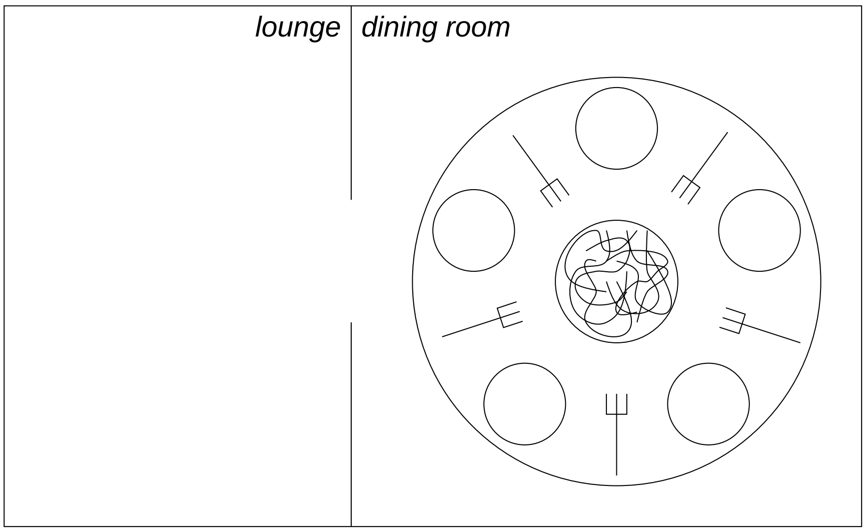
One of the main functions of an operating system is resource allocation. But when multiple processes request and acquire multiple resources, deadlock may ensue.

4.2.1 Deadlock and Livelock

The dining philosophers problem provides an abstract example

One of the classical problems in operating system lore is the following. Assume five philosophers live together, spending their time thinking. Once in a while they become hungry, and go to the dining room, where a table for five is laid out. At the center of the table is a large bowl of spaghetti. Each philosopher sits in his place, and needs two forks in order to shovel some spaghetti from the central bowl onto his personal plate. However, there is only one fork between every two philosophers.

The problem is that the following scenario is possible. All the philosophers become hungry at the same time, and troop into the dining room. They all sit down in their places, and pick up the forks to their left. Then they all try to pick up the forks to their right, only to find that those forks have already been picked up (because they are also another philosopher's left fork). The philosophers then continue to sit there indefinitely, each holding onto one fork, and glaring at his neighbor. They are deadlocked.



For the record, in operating system terms this problem represents a set of five processes (the philosophers) that contend for the use of five resources (the forks). It is highly structured in the sense that each resource is potentially shared by only two specific processes, and together they form a cycle. The spaghetti doesn't represent anything.

Exercise 78 *A more realistic problem is the one alluded to in Exercise 72. Consider a set of processes that need to manipulate a set of linked lists. Each list has a semaphore associated with it. To maintain consistency, each process needs to gain hold of all the necessary semaphores before performing an operation involving several lists. Is there a danger of deadlock? What can the processes do so as not to get deadlocked?*

A tempting solution is to have each philosopher relinquish his left fork if he cannot obtain the right fork, and try again later. However, this runs the risk that all the philosophers will put down their forks in unison, and then try again at the same time, resulting in an endless sequence of picking up forks and putting them down again. As they are active all the time, despite not making any progress, this is called *livelock* rather than deadlock.

Luckily, several solutions that do indeed work have been proposed. One is to break the cycle by programming one of the philosophers differently. For example, while all the philosophers pick up their left fork first, one might pick up his right fork first. Another is to add a footman, who stands at the dining room door and only allows four philosophers in at a time. A third solution is to introduce randomization: each philosopher that finds his right fork unavailable will relinquish his left fork, and try again only after a random interval of time.

Exercise 79 *Why do these solutions work?*

Exercise 80 *Write a program that expresses the essentials of this problem and one of its solutions using semaphores.*

However, these solutions are based on the specific structure of this problem. What we would like is a more general strategy for solving such problems.

4.2.2 A Formal Setting

The system state is maintained in the resource allocation graph

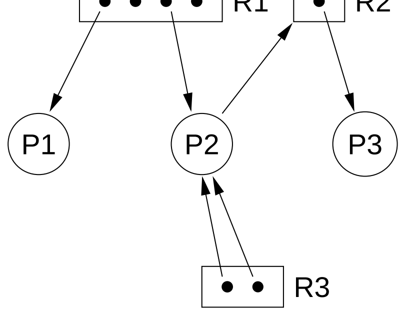
It is convenient to represent the system state — with respect to resource management — by the *resource allocation graph*. This is a directed graph with two types of nodes and two types of edges.

Processes are represented by round nodes.

Resource types are represented by square nodes. Within them, each instance of the resource type is represented by a dot.

Requests are represented by edges from a process to a resource type.

Allocations are represented by edges from a resource instance to a process.



For example, in this graph process P2 has one instance of resource R1 and two of R3, and wants to acquire one of type R2. It can't get it, because the only instance of R2 is currently held by process P3.

Exercise 81 *What are examples of resources (in a computer system) of which there is a single instance? A few instances? Very many instances?*

Exercise 82 *A computer system has three printers attached to it. Should they be modeled as three instances of a generic "printer" resource type, or as separate resource types?*

Cycles in the graph may imply deadlock

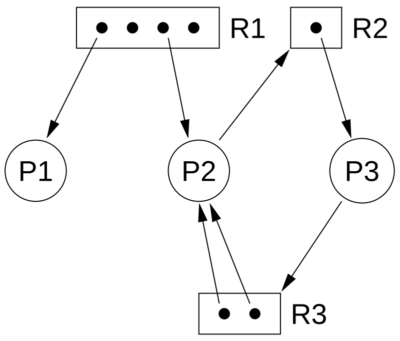
Assuming requests represent resources that are really needed by the process, an unsatisfied request implies that the process is stuck. It is waiting for the requested resource to become available. If the resource is held by another process that is also stuck, then that process will not release the resource. As a result the original process will stay permanently stuck. When a group of processes are stuck waiting for each other in this way, we say they are *deadlocked*.

More formally, four conditions are necessary for deadlock:

1. Resources are allocated exclusively to one process at a time, and are not shared.
2. The system does not preempt resources that are held by a processes. Rather, processes are expected to release resources when they are finished with them.
3. Processes may simultaneously hold one resource and wait for another.
4. There is a cycle in the resource allocation graph, with each process waiting for a resource held by another.

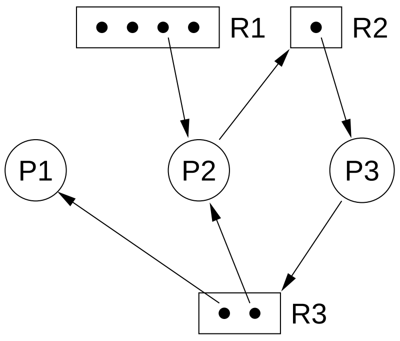
The first two are part of the semantics of what resource allocation means. For example, consider a situation in which you have been hired to run errands, and given a bicycle to do the job. It would be unacceptable is the same bicycle was given to someone else at the same time, or if it were taken away from under you in mid trip. The third is a rule of how the system operates. As such it is subject to modifications, and indeed some of the solutions to the deadlock problem are based on such modifications. The fourth may or may not happen, depending on the timing of various requests. If it does happen, then it is unresolvable because of the first three.

For example, the following graph represents a deadlocked system:



Process P3 is waiting for resource R3, but both instances are held by P2 who is waiting for R2, which is held by P3, and so on.

Note, however, that having a cycle in the graph is not a sufficient condition for deadlock, unless there is only one instance of each resource. If there are multiple instances of some resources, the cycle may be resolved by some other process releasing its resources. For example, in the following graph process P1 may release the instance of R3 it is holding, thereby allowing the request of P3 to be satisfied; when P3 subsequently completes its work it will release the instance of R2 it is holding, and it will be given to P2.



Exercise 83 *Under what conditions is having a cycle a sufficient condition? Think of conditions relating to both processes and resources.*

4.2.3 Deadlock Prevention, Avoidance, and Detection

There are several ways to handle the deadlock problem. Prevention involves designs in which deadlock simply cannot happen. Avoidance manages to stay away from deadlock situations by being careful. Detection allows deadlock and handles it after the fact.

Deadlock is prevented if one of the four conditions does not hold

Deadlock is bad because the involved processes never complete, leading to frustrated users and degraded service from the system. One solution is to design the system in a way that prevents deadlock from ever happening.

We have identified four conditions that are necessary for deadlock to happen. Therefore, designing the system so that it violates any of these conditions will prevent deadlock.

One option is to annul the "hold and wait" condition. This can be done in several ways. Instead of acquiring resources one by one, processes may be required to give the system a list of all the resources they will need at the outset. The system can then either provide the resources, or block the process. However, this means that processes will hold on to their resources for more time than they actually need them, which is wasteful. An alternative is to require processes to release all the resources they are currently holding before making new requests. However, this implies that there is a risk that the resources will be allocated to another process in the meanwhile; for example, if the resources are a system data structure, it must be brought to a consistent state before being released.

In some cases we can avoid the second condition, and allow resources to be preempted. For example, in Section 5.4 we will introduce swapping as a means to deal with overcommitment of memory. The idea is that one process is selected as a victim and swapped out. This means that its memory contents are backed up on disk, freeing the physical memory it had used for use by other processes.

Exercise 84 *Can locks be preempted? Can processes holding locks be swapped out?*

Prevention can be achieved by acquiring resources in a predefined order

A more flexible solution is to prevent cycles in the allocation graph by requiring processes to acquire resources in a predefined order. All resources are numbered in one sequence, and these numbers dictate the order. A process holding some resources can then only request additional resources that have higher numbers. A process holding a high-number resource cannot request a low-numbered one. For example, in the figure shown above, Process P2 holds two instances of resource R3. It will therefore not be allowed to request an instance of R2, and the cycle is broken. If P2 needs both R2 and R3, it should request R2 first, before it acquires R3.

Exercise 85 *With n resources there are $n!$ possible orders. Which should be chosen as the required order?*

Exercise 86 *Consider an intersection of two roads, with cars that may come from all 4 directions. Is there a simple uniform rule (such as "always give right-of-way to someone that comes from your right") that will prevent deadlock? Does this change if a traffic circle is built?*

Deadlock is avoided by allocations that result in a safe state

Another approach is deadlock avoidance. In this approach the system may in principle enter a deadlock state, but the operating system makes allocation decisions so as to avoid it. If a process makes a request that the operating system deems dangerous, that process is blocked until a better time. Using the cars at the intersection analogy, the operating system will not allow all the cars to enter the intersection at the same time.

An example of this approach is the banker's algorithm (also due to Dijkstra) [1]. This algorithm is based on the notion of *safe states*. A safe state is one in which all the processes in the system can be executed in a certain order, one after the other, such that each will obtain all the resources it needs to complete its execution. The assumption is that then the process will release all the resources it held, which will then become available for the next process in line. Such an ordering provides the operating system with a way to run the processes without getting into a deadlock situation; by ensuring that such an option always exists, the operating system avoids getting stuck.

Safe states are identified by the banker's algorithm

In order to identify safe states, each process must declare in advance what its *maximal* resource requirements may be. Thus the algorithm works with the following data structures:

- The maximal requirements of each process $\vec{M}_p$,
- The current allocation to each process $\vec{C}_p$, and
- The currently available resources $\vec{A}$.

These are all vectors representing all the resource types. For example, if there are three resource types, and three units are available from the first, none from the second, and one from the third, then $\vec{A} = (3, 0, 1)$.

Assume the system is in a safe state, and process p makes a request $\vec{R}$ for more resources (this is also a vector). We tentatively update the system state *as if* we performed the requested allocation, by doing

$$\vec{C}_p = \vec{C}_p + \vec{R}$$
$$\vec{A} = \vec{A} - \vec{R}$$

Where operations on vectors are done on respective elements (e.g. $\vec{X} + \vec{Y}$ is the vector $(x_1 + y_1, x_2 + y_2, ...x_k + y_k)$). We now check whether this new state is also safe. If it is, we will really perform the allocation; if it is not, we will block the requesting process and make it wait until the allocation can be made safely.

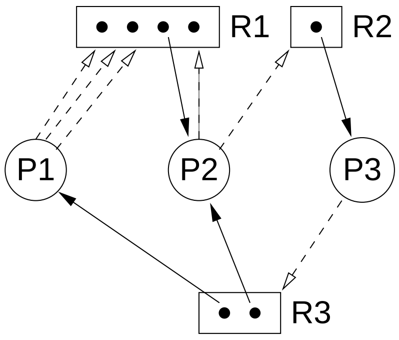
To check whether the new state is safe, we need to find an ordering of the processes such that the system has enough resources to satisfy the maximal requirements of each one in its turn. As noted above, it is assumed that the process will then complete and release all its resources. The system will then have a larger pool to satisfy the maximal requirements of the next process, and so on. This idea is embodied by the following pseudo-code, where P is initialized to the set of all processes:

```
while ($P \neq \emptyset$) {
    found = FALSE;
    foreach $p \in P$ {
        if ($\vec{M}_p - \vec{C}_p \leq \vec{A}$) {
            /* $p$ can obtain all it needs.          */
            /* assume it does so, terminates, and */
            /* releases what it already has.      */
            $\vec{A} = \vec{A} + \vec{C}_p$;
            $P = P - \{p\}$;
            found = TRUE;
        }
    }
    if (!  found) return FAIL;
}
return OK;
```

where comparison among vectors is also elementwise. Note that the complexity of the algorithm is $O(n^2)$ (where $n = |P|$), even though the number of possible orders is $n!$. This is because the resources available to the system increase monotonically as processes terminate, so if it is possible to execute any of a set of processes, the order that is chosen is not important. There is never any need to backtrack and try another order.

Exercise 87 *Show that the complexity is indeed $O(n^2)$.*

For example, consider the following resource allocation graph, where dashed edges represent potential future requests as allowed by the declared maximal requirements:



Assume process P1 now actually requests the allocation of an instance of resource R1. This request can be granted, because it leads to a safe state. The sequence of allocations showing this is as follows. There are enough available instances of R1 so that all of P1's requests can be satisfied. P1 will then terminate and release all its resources, including its instance of R3, which can then be given to P3. P3 will then release its instance of R2, which can be given to P2. Finally, P2 will acquire all the resources it needs and terminate.

However, if P2 requests an instance of resource R1, this cannot be granted, as it will lead to an unsafe state, and therefore might lead to deadlock. This can be seen by noting that if the request is granted, the system will not have sufficient resources to grant the maximal requirements of any of the processes. Therefore the first iteration of the algorithm will fail to find a process that can in principle obtain all its required resources first.

Exercise 88 *What can you say about the following modifications to the above graph:*

1. *P1 may request another instance of R3 (that is, add a dashed arrow from P1 to R3)*
2. *P3 may request an instance of R1 (that is, add a dashed arrow from P3 to R1)*

Exercise 89 *Consider a situation in which the resources we are interested in are locks on shared data structures, and several instances of the same program are being run. What is the result of using the banker's algorithm in this scenario?*

Exercise 90 *Avoidance techniques check each resource request with something like the Banker's algorithm. Do prevention techniques need to check anything?*

Deadlock detection and recovery is the last resort

The banker's algorithm is relatively flexible in allocating resources, but requires rather complicated checks before each allocation decision. A more extreme approach is to allow all allocations, without any checks. This has less overhead, but may lead to deadlock. If and when this happens, the system detects the deadlock (using any algorithm for finding cycles in a graph) and recovers by killing one of the processes in the cycle.

4.2.4 Real Life

While deadlock situations and how to handle them are interesting, most operating systems actually don't really employ any sophisticated mechanisms. This is what Tanenbaum picturesquely calls the ostrich algorithm: pretend the problem doesn't exist [13]. However, in reality the situation may actually not be quite so bad. Judicious application of simple mechanisms can typically prevent the danger of deadlocks.

Note that a process may hold two different types of resources:

Computational resources: these are the resources needed in order to run the program. They include the resources we usually talk about, e.g. the CPU, memory, and disk space. They also include kernel data structures such as page tables and entries in the open files table.

Synchronization resources: these are resources that are not needed in themselves, but are required in order to coordinate the conflicting actions of different processes. The prime example is locks on kernel data structures, which must be held for the duration of modifying them.

Deadlocks due to the first type of resources are typically prevented by breaking the "hold and wait" condition. Specifically, processes that attempt to acquire a resource and cannot do so simply do not wait. Instead, the request fails. It is then up to the process to try and do something intelligent about it.

For example, in Unix an important resource a process may hold onto are entries in system tables, such as the open files table. If too many files are open already, and the table is full, requests to open additional files will fail. The program can then either abort, or try again in a loop. If the program continues to request that the file be opened in a loop, it might stay stuck forever, because other programs might be doing the same thing. However it is more likely that some other program will close its files and terminate, releasing its entries in the open files table, and making them available for others.

Exercise 91 *If several programs request something in a loop until they succeed, and get stuck, is this deadlock? Can the operating system detect this? Should it do something about it? Might it take care of itself? Hint: think quotas.*

Deadlock due to locks can be prevented by acquiring the locks in a specific order. This need not imply that all the locks in the system are ordered in one grand scheme. Consider a kernel function that handles several data structures, and needs to lock them all. If this is the only place where these data structures are locked, then multiple instances of running this function will always lock them in the same order. This effectively prevents deadlock without requiring any cognizant global ordering of all the locks in the system. If a small number of functions lock these data structures, it is also easy to verify that the locking is done in the same order in each case. Finally, system designers that are aware of a potential deadlock problem can opt to use a non-blocking lock, which returns a fail status rather than blocking the process is the lock cannot be acquired.

4.3 Summary

Abstractions

The main abstraction introduced in this chapter is semaphores — a means of expressing the condition of waiting for an event that will be performed by another process. This abstraction was invented to aid in the construction of operating systems, which are notoriously difficult to get right because of their concurrency. It was so successful that it has since moved to the user interface, and is now offered as a service to user applications on many systems.

A closely related abstraction is that of a lock. The difference is that semaphores are more "low level", and can be used as a building block for various synchronization needs. Locks focus on the functionality of regulating concurrent access to resources, and ensuring mutual exclusion when needed. This includes the definition of special cases such as readers-writers locks.

Another abstraction is the basic notion of a "resource" as something that a process may need for its exclusive use. This can be anything from access to a device to an entry in a system table.

Resource management

Deadlock prevention or avoidance have a direct effect on resource management: they limit the way in which resources can be allocated, in the interest of not entering a deadlock state.

Mechanisms for coordination have an indirect effect: providing efficient mechanisms (such as blocking) avoids wasting resources (as is done by busy waiting).

Workload issues

Workload issues are not very prominent with regard to the topics discussed here, but they do exist. For example, knowing the distribution of lock waiting times can influence the choice of a locking mechanism. Knowing that deadlocks are rare allows for the problem to be ignored.

Hardware support

Hardware support is crucial for the implementation of coordination mechanisms — without it, we would have to use busy waiting, which is extremely wasteful. The most common form of hardware support used is the simple idea of blocking interrupts. More sophisticated forms of support include atomic instructions such as test_and_set.

Bibliography

[1] E. W. Dijkstra, "*Co-operating sequential processes*". In *Programming Languages*, F. Genuys (ed.), pp. 43–112, Academic Press, 1968.

[2] C. A. R. Hoare, "*Monitors: an operating system structuring concept*". *Comm. ACM* **17(10)**, pp. 549–557, Oct 1974.

[3] M. Hofri, "*Proof of a mutual exclusion algorithm – a 'class'ic example*". *Operating Syst. Rev.* **24(1)**, pp. 18–22, Jan 1990.

[4] L. Lamport, "*A fast mutual exclusion algorithm*". *ACM Trans. Comput. Syst.* **5(1)**, pp. 1–11, Feb 1987.

[5] L. Lamport, "*The mutual exclusion problem: part I — a theory of interprocess communication*". *J. ACM* **33(2)**, pp. 313–326, Apr 1986.

[6] L. Lamport, "*The mutual exclusion problem: part II — statement and solutions*". *J. ACM* **33(2)**, pp. 327–348, Apr 1986.

[7] L. Lamport, "*A new solution of Dijkstra's concurrent programming problem*". *Comm. ACM* **17(8)**, pp. 453–455, Aug 1974.

[8] N. G. Leveson and C. S. Turner, "*An investigation of the Therac-25 accidents*". *Computer* **26(7)**, pp. 18–41, Jul 1993.

[9] G. L. Peterson, "*Myths about the mutual exclusion problem*". *Inf. Process. Lett.* **12(3)**, pp. 115–116, Jun 1981.

[10] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Addison-Wesley, 5th ed., 1998.

[11] W. Stallings, *Operating Systems*. Prentice-Hall, 2nd ed., 1995.

[12] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice-Hall, 3rd ed., 1998.

[13] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*. Prentice-Hall, 2nd ed., 1997.