Synchronization
Keeping out of each other's way

Paul Krzyzanowski
Rutgers University

February 10, 2012 [original September 21, 2010]

1 Introduction: concurrency

Let's start off with some definitions. In this discussion, we will use the terms *processes* and *threads* interchangeably. The underlying assumption is that the processes or threads may have access to the same shared regions of memory. This sharing is automatic for threads but is also possible with processes on most operating systems if they explicitly create a shared memory segment.

A set of threads is **concurrent** if the threads exist at the same time. The threads may be running at the same time in multiprocessing environments or their execution may be interleaved through preemption. The execution of these threads may be interleaved *in any order*, as long as each thread of execution follows the order dictated by the program.

Threads are **asynchronous** if they require occasional synchronization and communication among themselves. For the most part, the execution of one thread neither speeds up nor slows down the execution of another.

Threads are **independent** if they have no reliance on one another. This means that they never communicate and that one thread does not generate any side effects that impact another thread.

Synchronous threads are threads that are kept synchronized with each other such that the order of execution among them is guaranteed.

Parallel threads run at the same time on separate processors.

In this section we examine the reasons for thread synchronization and how it can be realized.

2 The race is on!

Suppose two or more asynchronous processes or threads access to a common, shared piece of data. Any process can be preempted at any time. What problems can arise?

2.1 Example 1

Let's look at an example where you have a system-wide global variable called *characters* that counts all the characters received from all terminal devices connected to a computer. Suppose also that each process that reads *characters* uses the following code to count them:

```
characters = characters + 1;
```

and that the compiler generates the following machine instructions for this code:

```
read characters into register r1
increment r1
write register r1 to characters
```

As a concrete example, the x86–64 code generated by the gcc compiler for that one line of code is:

```
movl    _characters(%rip), %eax
addl    $1, %eax
movl    %eax, _characters(%rip)
```

Where *eax* is a CPU register and *_characters(%rip)* is used to create a position-independent reference to the memory location of the integer *characters*. When the code is loaded linked, *_characters(%rip)* gets replaced by a byte offset from the current instruction. *RIP* is the 64-bit instruction pointer and the operand is a RIP-relative address, an offset value.

Let's assume that *characters* is 137. Process A executes the first two instructions and is then preempted by process B. Process B now reads *characters*, which is still 137 because A did not get around to writing its result. Process B adds 1 and writes out the result: 138. Some time later, process A gets a chance to run again. The operating system restores A's state (CPU registers, program counter, stack pointer, and flags) from the point where it was preempted and allows it to continue execution. It already added 1 to 138 and and has the result in a register, so all it does is write the contents of register, 138, out to memory. We now lost count of a character!

2.2 Example 2

Let us suppose that your current checking account balance is $1,000. You withdraw $500 from an ATM machine while, unbeknownst to you, a direct deposit for $5,000 was coming in. We have these two concurrent actions:

Withdrawal	Deposit
1. Read account balance	1. Read account balance
2. Subtract 500.00	2. Add 5,000.00
3. Write account balance	3. Write account balance

Consider the possible outcomes:

$5500 The proper outcome, of course, is that your checking account will have $5,500 after both operations are finished. Either the *Deposit* sequence of operations takes place first or the *Withdrawal* sequence does. Either way, the answer is the same.

$500 Suppose that the system just completed step 1 of the *Withdrawal* operation. At that time, the thread gets preempted and the *Deposit* thread gets to run. It goes through its steps: reads the account balance ($1,000), adds $5000 ($6,000), and writes the result ($6,000). Then the *Withdrawal* thread is allowed to resume running. It already completed step 1, where it read $1,000 (not knowing that the value has been modified!), so it resumes operation with step 2 and subtracts $500 from $1,000, storing $500 in step 3. Your account balance is now $500. The actions of *Deposit* have effectively vanished.

$6,000 Now suppose that the operations are now scheduled in a slightly different order. The *Deposit* thread runs first, reads the account balance ($1,000) and adds $5,000 (giving $6,000). At this time, the thread is preempted and the *Withdrawal* thread is scheduled to run. In step 1, it reads the balance ($1,000, since the new balance was not yet written), subtracts $500, and stores the result, $500. When the *Deposit* thread is now given a chance to run, all it has left to do is step 3: store its result, $6,000. The effects of the *Withdrawal* thread have been obliterated.

2.3 Example 3

As a final example, let's consider a linked list. Multiple threads are running concurrently and add items to the list by adding a new node to the head of the list:

```
1. stuff = new Item;
2. stuff->item = whatever; /* new item contents */
3. stuff->next = head;  /* the new node points to the current head of the list */
4. head = stuff;    /* the new node becomes the head of the list */
```

Currently, we have two items on the list; see Figure 1.

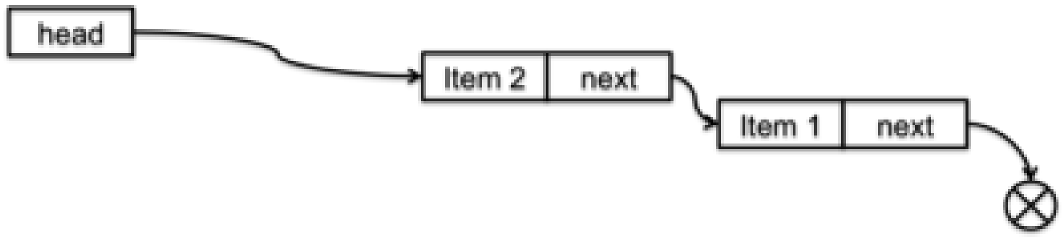

Figure 1: Figure 1. Current queue

Thread A now needs to add a new element to the list: *Item 3*. At more or less the same time, thread B also needs to add a new element to the list, *Item 4*. Thread A is currently running and executes the first three instructions. The head of the list has not yet been set to point to the new element. Figure 2 shows how things look at this point in time:

All is well so far. Now the operating system preempts thread A and gives thread B a chance to run. Thread B executes the same instructions and runs the entire sequence 1–4. It allocates a new list cell, sets its *next* pointer to the current head of the list, and then changes the global list head to point to this new cell. Fgure 3 shows what we have now.

Thread B may go on to do other things and is eventually preempted so that thread A can run. Thread A continues where it left off, at step 4. It sets the global list head to point to the new cell that it added to the list. Figure 4 shows us how the list looks now. Item 4 is effectively gone from the list since *head* does not point to it.

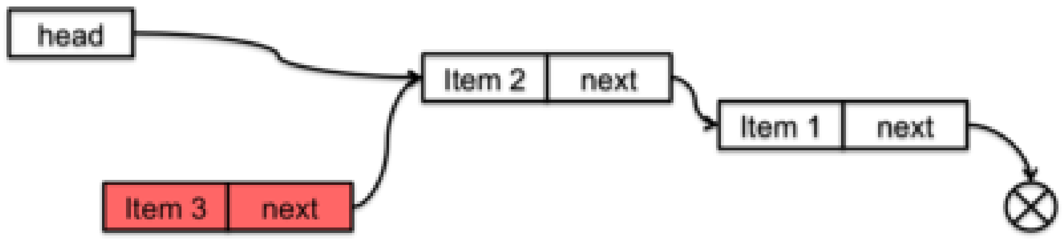

Figure 2: Figure 2. Adding Item 3

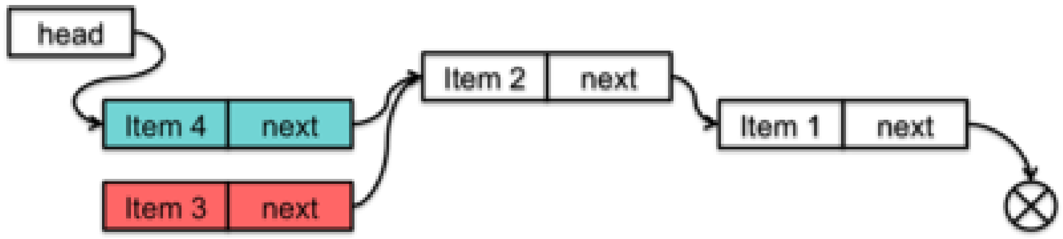

Figure 3: Figure 3. Adding Item 4

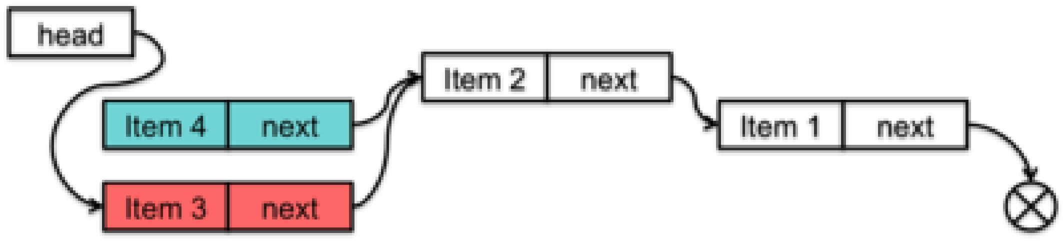

Figure 4: Figure 4. Messed up list

What happened is that thread A was oblivious to the fact that thread B snuck in and added a list element. Thread A overwrote the head of the list without taking into account the fact that thread B modified it.

2.4 Race conditions

The above examples illustrate how bad things can happen when multiple threads or processes access shared data. These bugs result from race conditions. A **race condition** is a bug where the outcome of concurrent threads is dependent on the precise sequence of the execution of one relative to the other. **Thread (or process) synchronization** deals with developing techniques to avoid race conditions.

We'll use *threads* and *processes* almost interchangeably in our discussion and the principles apply equally to both. Processes or threads on different processes, since they generally do not share the same memory with other processes, have to rely on **interprocess communication mechanisms** (**IPC**), such as explicitly setting up regions of shared memory, to share data and communicate. Threads within the same process, of course, share the same address space and hence the same static data and heap (all dynamically allocated memory that is not stack-based). This increases the likelihood that race conditions may arise.

3 Mutual exclusion

The above problems can be resolved by not allowing other threads or processes to run while a process that is manipulating shared data is active. This is a drastic approach and not a good one. First, it's not particularly fair. Because one process is running, others won't get a chance to run. Secondly, it is overbearing: it is possible, even likely, that this or other processes have a lot more useful work to do outside of touching shared data. Thirdly, it may not even be feasible. It is possible that the processes require communication with each other to accomplish their result.

What we can do instead is to identify the regions in a program that access shared memory (or other shared resources) and may cause race conditions to arise. These regions are called **critical sections**. A critical section is a sequence of code that reads shared data, uses and modifies it, and then writes it back out.

To avoid race conditions, only one thread should be allowed to access a shared resource at a time. Enforcing this condition is known as **mutual exclusion**. If a thread is in its critical section, any other thread that wants to access the shared resource must wait (be blocked).

When we introduce mutual exclusion into the system, we have to be careful that a thread will not be perpetually blocked because of a circular dependency. An example of this is if one thread in a critical section R but also needs to enter critical section S. However, another thread is in critical section S and needs to access critical section R. This is a circular dependency and the condition is known as **deadlock**. The two threads are deadlocked, each waiting for the other to free up a resource (i.e., let go of a critical section).

A related problem is **livelock**. Here, there is no strict circular dependency but threads may be communicating and constantly changing state yet not making any progress. In some ways, this is more insidious than deadlock because it is more difficult to detect as no thread is in a waiting state.

Starvation is the case where the scheduler never gives a thread a chance to run. One reason that a scheduler may do this is because the thread's priority is lower than that of other threads that are ready to run. Never getting a chance to run is bad. It's even worse when the thread is in a critical section since, by not running, it will not get a chance to release the critical section.

3.1 Locking

The logical approach for a programmer to deal with critical section is via **locks**: you **acquire** a lock before you enter a critical section and then **release** it when you exit the critical section. This ensures that only one thread can be in the critical section at the same time. Here is a representation of the withdrawal/deposit scenario with locks around the critical sections. We define a critical section that we lock with a lock variable that we call *transfer_lock*. To enter it, we call an *acquire* function. If any thread is already in the critical section, then we block until that thread releases it via *release*. This ensures that only one thread at a time can be in the code following *acquire*.

A solution for managing critical sections requires the following conditions to hold.

Withdrawal	Deposit
1. **Acquire(transfer_lock)**	1. **Acquire(transfer_lock)**
2. Read account balance	2. Read account balance
3. Subtract 500.00	3. Add 5000.00
4. Write account balance	4. Write account balance
5. **Release(transfer_lock)**	5. **Release(transfer_lock)**

1. Mutual exclusion. No threads may be inside the same critical sections simultaneously.
2. Progress. If no thread is executing in its critical section and some thread or threads want to enter the critical section, the selection of a thread that can do so cannot be delayed indefinitely. Specifically, if only one thread wants to enter, it should be permitted to do so. If more than one wants to enter, only one of them should be allowed to.
3. Bounded waiting: No thread should wait forever to enter a critical section.
4. No thread running outside its critical section may block others from entering a critical section.

A good solution for managing critical sections should also ensure that

- No assumptions are made on the number of processors. The solution should work even if threads are executing at the same time on different processors.
- No assumptions are made on the number of threads or processes. The solution should not be designed to only support, for example, two threads.
- No assumptions are made on the relative speed of each thread. We cannot assume any knowledge of when or if a thread will request a critical section again. It will be undesirable to have an algorithm where a thread has to wait unnecessarily for another thread.

4 Achieving mutual exclusion

Let's examine a few approaches that we may implement to try to achieve mutual exclusion among a number of concurrent threads.

4.1 Proposed solution #1: disable interrupts

Have each thread disable all interrupts just before entering its critical section and re-enable them when leaving. This ensures that the operating system will not get a timer interrupt and have its scheduler preempt the thread while it is in the critical section.

Disadvantage

This gives the thread too much control over the system. Moreover, it is something that can only be done if the thread is executing in kernel mode; user processes are disallowed from disabling interrupts. Consider what happens if the logic in the critical section is incorrect and interrupts are never enabled? The operating system will not get its timer interrupt and will not be able to preempt the thread and let any other process run. The system will have to be rebooted. Even if the thread does re-enable interrupts after the critical section, what happens if the critical section itself has a dependency on some other interrupt, thread, or system call? For example, the logic in the critical section may need to read data from a file on a disk but the operating system will not get the disk interrupt that tells it that data is ready. Finally, if you're on a multiprocessor system, disabling interrupts will only disable them on one processor, so the technique does even work on these systems.

Advantage

This is a simple approach and, on a uniprocessor system, is guaranteed to work. It is very tempting to use within the kernel for code that modifies shared data.

In short, this is clearly not a good idea for the general welfare of the operating environment. However, because of its extreme simplicity, this was a common approach to mutual exclusion in operating system kernels, at least before multiprocessors spoiled the fun.

4.2 Proposed solution #2: test & set locks

We can keep a shared lock variable and set it to 1 if a process is in a critical section and 0 otherwise. When a process wants to enter its critical section, it first reads the lock. If the lock variable is 0, the process sets the lock to 1 and executes its critical section. If the lock is 1, the process waits until it becomes 0.

Here's a sample snippet of code that uses a global variable called *locked*:

```
while (locked) ;    /* loop, waiting for locked == 0 */
locked = 1;     /* set the lock */
/* do critical section */
locked = 0;     /* release the lock */
```

Disadvantage

The really big problem with this approach is that a race condition exists for reading and setting the lock. Here's why. After the *while* loop sees that *locked* is zero, the thread could be preempted. Another thread that is waiting for the critical section will also see that *locked* is zero and will also exit its *while* loop and enter the critical section. When the first thread is given a chance to continue running, it is already out of the *while* loop and simply sets *locked=1*, unaware that somebody else already entered the critical section and set *locked* to 1.

Advantage

This solution is extremely simple to understand, implement, and trace. Unfortunately, it is a buggy, and hence invalid, solution. It's been used for operations such as locking a mailbox file to prevent another mail reading program from modifying the file. The insidious nature of race conditions is that the bugs may not present themselves even after a lot of intensive testing, so buggy solutions might appear to be flawless.

4.3 Proposed solution #3: lockstep synchronization

This attempt at a solution involves keeping a shared variable (*turn*) that tells you which thread's turn it is to execute a critical section. Each thread that may want to access the critical section is given a unique number and the *turn* variable cycles through these numbers. In the simple case of only two threads, we can use the numbers 0 and 1. For instance, thread 0 gets to run its critical section when *turn* is 0 and thread 1 gets to run its critical section when *turn* is 1.

```
int turn = 0;    /* 0 if thread 0's turn; 1 if thread 1's turn */
```

Thread 0

```
for (;;) {   /* forever */
    while (turn != 0) ;     /* wait for my turn */
    critical_section();     /* run critical section */
    turn = 1;               /* let Thread 1 run */
    other_stuff();
}
```

Thread 1

```
for (;;) {   /* forever */
    while (turn != 1) ;     /* wait for my turn */
    critical_section();     /* run critical section */
    turn = 0;               /* let Thread 0 run */
    other_stuff();
}
```

This solution works in avoiding race conditions but has a major disadvantage: it forces strict alternation between the threads. As soon as thread 0 finishes its critical section, it sets *turn* to 1, forcing the next entry into the critical section has to be by thread 1. If thread 0 happens to be faster than thread 1 or just needs to access the critical section more frequently, this scheme forces thread 0 to slow down (by waiting for its turn) to the same rate of critical section access as thread 1. It turns asynchronous threads into synchronous threads.

4.4 Software solutions

There have been a number of pure software solutions proposed to handle reliable entry into critical sections. We'll provide a brief overview of one of them, known as **Peterson's algorithm**, for the simple case of a two-thread system.

The algorithm relies on a shared array (one element per thread) and a shared variable. The array, *wants*, identifies which threads want to enter a critical section. The variable, *turn*, indicates whose turn it is. Everything is initialized to zero initially.

A thread calls a function, *acquire*, before running its critical section. This function sets an array element in the shared array (*wants*) that corresponds to its position (e.g., thread 0 sets element 0) to show that the thread wants to enter the critical section. It sets the shared variable (*turn*) to the thread number. Then it spins in a busy wait loop if it's the thread's turn but the other thread wants the critical section. Upon completing the critical section, the thread calls *release*.

```
acquire(int thread_id) {
    int other_thread = 1-thread_id;
    wants[thread_id] = 1;    /* we want the critical section */
    turn = thread_id;
    while (turn == thread_id && wants[other_thread]) ; /* wait */
}

release(int thread_id) {
    wants[thread_id] = 0;
}
```

Let's see how this works. Suppose that Thread 0 is calling *acquire* and Thread 1 is not. Here's what happens in *acquire* when Thread 0 calls it:

```
wants[0(thread)] = 1;
turn = 0(thread);
while (0(turn) == 0(thread) && 0(wants[other_thread, value=1])) ;
```

The while loop tells us that thread 1 does not want the critical section (*wants[1]* is 0), so we can continue into the critical section. Now suppose that Thread 1 wants to enter the critical section and calls *acquire*:

```
wants[1((thread))] = 1;
turn = 1(thread);
while (1(turn) == 1(thread) && 1(wants[other_thread, value=0])) ;
```

Thread 1 is now kept waiting because Thread 0 set its *wants* entry to 1 (true). When Thread 0 is done with its critical section, it sets *wants[0]* to 0 (false) and Thread 1 will jump out of the while loop.

Now let's see if race conditions can develop by examining the case when both threads try to enter the critical section at the same time. It doesn't matter which thread gets to set *wants* first because, even though it's a shared array, only one thread ever sets a particular entry; there is no contention to write to the same location. The variable *turn*, on the other hand, can be set by either thread. Suppose Thread 0 gets to set turn to its thread number (0) first.

Now if Thread 0 gets to set *turn* first, the following happens:

We see that the algorithm works by disallowing the last thread that set *turn* to proceed unless nobody else wants to enter the critical section.

Thread 0	Thread 1
wants[0] = 1;	wants[1] = 1;
turn = 0;	
	turn = 1;
while (turn==0 && wants[0]);	while (turn==1 && wants[0]);
test (1==0 && 1)	*test* (1==1 && 1)
proceed	*busy wait*

Thread 0	Thread 1
wants[0] = 1;	wants[1] = 1;
	turn = 1;
turn = 0;	
while (turn==0 && wants[1]);	while (turn==1 && wants[0]);
test (0==0 && 1)	*test* (1==1 && 1)
busy wait	*proceed*

The disadvantage with this and other algorithmic techniques is that they can be tricky to implement correctly, especially when there are more than two threads involved. The example presented only works for two threads. With higher-level languages such as C or C++, you need to rely on the *volatile* data type to ensure that reads and writes actually reach main memory and that the compiler does not optimize the code to use registers or delay writing to memory.

5 Help from the processor

Most processors have specific instructions that aid in programming critical sections. The key to these instructions is that the operations that they perform are **atomic**, meaning that they are indivisible. A system interrupt cannot come in and preempt the instruction after the first part of its operations finished, which was the problem with a solution such as a our pure software test & set solution.

Different processors offer different instructions that help with critical sections. We will look at three variations of these atomic instructions:

- Test-and-set
- Compare-and-swap
- Fetch-and-increment

5.1 Test and set

A test-and-set instruction reads the contents of a memory word into a register, writes a value of 1 into that location, and returns the previous contents of the memory. The crucial item here is that this operation is guaranteed to be atomic: indivisible and non-interruptible. This is what it does (indivisibly):

```
int test_and_set(int *x) {
    last_value = *x;
```

```
        *x = 1;
        return last_value;
    }
```

Test-and-set always sets a lock but it lets you know if it was already set at the time that you executed the instruction. If the lock was already set when you ran test-and-set then somebody simply got the lock before you did; you don't have it. If the lock was not set when you ran the instruction (i.e., the instruction returned a zero), then it is now set and you have the lock. Managing a critical section can look like this:

```
    while (test_and_set(&lock)) ; /* wait for the lock */
    /* do critical section */
    lock = 0;   /* release the lock */
```

5.2 Compare and swap

The compare-and-swap instruction compares the value of a memory location with an "old" value that is given to it. If the two values match then the "new" value is written into that memory location. This is the sequence of operations:

```
    int compare_and_swap(int *x, int old, int new) {
        int save = *x;    /* current contents of the memory */
        if (save == old)
            *x = new;
        return save;    /* tell the caller what was read */
    }
```

Think of *compare_and_swap(int *x, int A, int B)* as having the function of *set x to B only if x is currently set to A and return the value that x really contained.*

Remember our attempt at a software-only test-and-set lock? We'd loop while a memory location, *locked*, was not zero and then set it to one? The problem was that we had a race condition between those two operations: somebody else could come in and set the value of *locked* to 1 after we tested it. With compare-and-swap, we're telling it the value we last read from that memory location and what we'd like to set it to *provided that* somebody did not change the contents since we last read it. If they did, our new value will not be set since our idea of the old value was not correct. The return value tells us what the contents of the memory location were when the compare-and-swap instruction was executed. An example of how we control entry to and exit from a critical section using compare-and-swap is:

```
    while (compare_and_swap(&locked, 0, 1) != 0) ; /* spin until locked == 0 */
    /* if we got here, locked got set to 1 and we have it */
    /* do critical section */
    locked = 0; /* release the lock */
```

Here, we spin on compare_and_swap and keep trying to set *locked* to 1 (last argument) if *lock*'s current value is 0. Let's consider a few cases:

- If nobody has the region locked, then *locked* is 0. We execute compare_and_swap and tell it to set *locked* to 1 if the current value is 0. The instruction does this and returns the old value (0) to us so we break out of the while loop.

- If somebody has the region locked, then *locked* is 1. We execute compare_and_swap and tell it to set *locked* to 1 if the old value was 0. But the old (current) value is 1, so compare_and_swap does not set it (it wouldn't matter) and returns the value that was already in *locked*, 1. Hence, we stay in the loop.

- Let's consider the potential race condition when two threads try to grab a lock at the same time. Thread 1 gets to run *compare_ and_ swap* first, so the instruction sets *locked* to 1 and returns the previous value of 0. When Thread 2 runs the instruction, *locked* is already set to 1 so the instruction returns 1 to the thread, causing it to keep looping.

In summary, we're telling the *compare_ and_ swap* instruction to set the contents of *locked* to 1 *only if locked* is still set to 0. If another thread came in and set the value of *locked* to non-zero, then the instruction will see that the contents do not match 0 (*old*) and therefore will not set the contents to *new*.

5.3 Fetch and increment

Figure 5: Ticket

The final atomic instruction that we'll look at is fetch-and-increment. This instruction simply increments a memory location but returns the previous value of that memory location. This allows you to "take a ticket." Think of fetch-and-increment as one of those take-a-number machines at the deli counter in a supermarket:

Figure 6: Turn

```
int fetch_and_increment(int *x) {
    last_value = *x;
    *x = *x + 1;
    return last_value;
}
```

The return value, *last_ value* corresponds to the number on your ticket while the machine is now ready to generate the next higher number. To implement a critical section, we grab a ticket (*myturn*) and wait our turn. Our turn arrives when the turn-o-matic machine[1] shows our number and the deli clerk calls it (*turn* == *myturn*):

```
ticket = 0;
turn = 0;
...
myturn = fetch_and_increment(&ticket);
while (turn != myturn) ;  /* busy wait */
/* do critical section */
fetch_and_increment(&turn);
```

The variable *ticket* represents the ticket number in the machine. The variable *myturn* represents your ticket number and *turn* represents the number that is currently being served. Each time somebody takes a ticket via *fetch_ and_ increment*, the value of *ticket* is incremented. Each time somebody is done with their critical section and is ready for the next thread to get served, they increment *turn*.

6 Spin locks and priority inversion

One problem with any of the above solutions is that we end up with code that sits in a *while* loop (spins), waiting for conditions to allow it entry into the critical section. This constant checking is called **busy waiting** or a **spin lock**. The process is always busy running but it is really just waiting since it is making no process in execution. From a resource utilization point of view, this is not efficient. To an operating system, the thread is in a *ready* state and gets scheduled to run (use the CPU) even though it will not make any progress. It is a waste of CPU time.

[1]http://www.checkpointsystems.com/EUR/products-services/Meto-Solutions/Turnomatic%20-%20Customer%20Service%20Enhancement/Turn-O-Matic-Systems.aspx

What makes this situation even worse is that it is possible that the process that is currently holding the lock may not even be allowed to run or may be significantly delayed. Suppose we have a priority-based scheduler and a lower-priority process has obtained a lock and is in the critical section. Now suppose that a higher-priority process wants to get access to the critical section and is sitting in a spin lock. As far as the operating system is concerned, that higher-priority process is always ready to run and will be scheduled instead of the process that is in the critical section (the scheduler simply picks the higher priority thread or process). This situation is known as **priority inversion**. What really needs to happen for the higher priority process to make progress (i.e., to get the lock and get into the critical section) is for the lower priority process to get to run so that it can get out of the critical section and relinquish its lock.

One technique that may be used to avoid priority inversion is to increase the priority of any process in a critical section to the maximum priority level of any of the processes that are waiting for the lock. Once the process releases its lock, its priority goes back to its normal level. This approach is known as **priority inheritance**. The challenge with implementing this is that you need to ensure that there is some way for the operating system to know about whether a process is in a critical section, looping on a spin lock for that same critical section, or doing something completely unrelated.

Spin locks are a fundamental problem with pure software solutions. Without having a way for a thread to ask the operating system to suspend itself, all that a thread can do to stop itself from making progress is to loop while waiting for some condition to change. Ideally, we'd like to create operating system constructs for mutual exclusion that will let us block the process instead of having it spin.

7 OS mechanisms for synchronization

All of the above solutions to accessing a critical section have the drawback of busy waiting: if a thread cannot enter its critical section, it sits in a loop waiting until it can do so. This has two problems. First, it wastes CPU time since the thread is always ready to run and the operating system scheduler will schedule it to do so periodically. Secondly, and more seriously, is that cases can arise when a thread that has the lock is not allowed to run. Suppose the operating system uses a thread scheduler that always favors higher-priority threads. If a lower priority thread obtained a lock on a critical section, the higher priority thread can do nothing but loop and wait for the other thread to give up the lock. However, because this higher priority thread is always ready to run, the lower priority thread never gets a chance to run and give up its critical section. This situation is known as **priority inversion**.

It would greatly help us if we can get some support from the operating system to put a thread to sleep (block it) when it cannot enter a critical section and then wake it up when it is finally allowed access to the critical section.

We will now look at a few mechanisms that operating systems give us.

7.1 Semaphores

In 1963, Edsger W. Dijkstra[2] created a special variable type called a **semaphore** that allows one to manage access to critical sections EWD123[3]. A semaphore is an integer variable with

[2]http://en.wikipedia.org/wiki/Edsger_W._Dijkstra

[3]http://www.cs.utexas.edu/~EWD/ewd01xx/EWD123.PDF

two associated operations: **down** (also known as *p* or *wait*) and **up** (known as *v* or *signal*).

The following actions take place for up and down, all *indivisible*.

```
down(sem s) {
    if (s > 0)
        s = s - 1;
    else
        /* put the thread to sleep on event s */
}

up (sem s) {
    if (one or more threads are waiting on s)
        /* wake up one of the threads sleeping on s  */
    else
        s = s + 1;
}
```

The indivisibility of these operations is ensured by the operating system's implementation of semaphores and may involve lower-level mutual exclusion operations, such as spin locks. While we discussed spin locks a bad thing, they serve a purpose within the kernel for getting mutual exclusion locks to *small* sections of code. We just want to avoid them for potentially long-term operations.

Whenever a thread calls *down* on a semaphore, the semaphore's value is decremented and the thread can continue running. The value of a semaphore, however, cannot go below zero. If the value of a semaphore is zero and a thread calls *down* on that semaphore, the kernel will put that thread to sleep. Specifically, that thread will be in a *blocked* state and the thread ID will be placed on a list of any other threads that are blocked on that semaphore.

Whenever a thread calls *up* on a semaphore, the kernel checks to see whether there are any threads in the list of threads that are blocked on that semaphore. If so, one of the threads is moved to the *ready* state and can continue execution. The value of the semaphore does not change. If there are no threads blocked on the semaphore, the value of that semaphore is simply incremented. The *up* operation never blocks a thread.

One way to look at semaphores is to think of a semaphore as counting the number of threads that are allowed to be in a section of code at the same time. Each time a thread enters the section, it performs a *down*, allowing one fewer access (threads that will be allowed entry). When there are no more accesses allowed, any thread requesting access is forced to block. When a thread is done with its section, it performs an *up*, which allows a waiting thread to enter or, if none are waiting, increments the count of threads that are allowed to enter.

Semaphores are often initialized to 1 and are used by two or more threads to ensure that only one of them can enter a critical section. Such semaphores are known as **binary semaphores**. Here is an example of accessing a critical section using a binary semaphore:

```
sem mutex = 1;  /* initialize semaphore */
...
down(&mutex);
/* do critical section */
up(&mutex);
```

Producer-consumer example

Let us suppose that there are two processes: a *producer* that generates items that go into a buffer and a *consumer* that takes things out of the buffer. The buffer has a maximum capacity of N items. If a producer produces faster than the consumer consumes then it will have to go to sleep until there's enough room in the buffer. Similarly, if the consumer consumes faster than the producer produces, the consumer will go to sleep until there is something for it to consume. This scenario is formally known as the **Bounded-Buffer Problem**.

As we saw in our discussion on race conditions, we need to avoid race conditions to properly keep track of the items in the buffer in case we produce and consume something at the same time. To handle this, we'll use a binary semaphore that we'll call *mutex* (for mutual exclusion). We'll initialize it to 1 so that we can use it to guarantee mutual exclusion whenever we add or remove something from the buffer.

We want the producer to go to sleep when there is no more room in the buffer and for it to wake up again when there is free room in the buffer. To do this, we will create a semaphore called *empty* and initialize it to *N*, the number of free slots in the buffer. When we execute *down(&empty)*, the operation will decrement *empty* and return to the thread until *empty* is 0 (no more slots). At that point, any successive calls to *down(&empty)* will cause the producer to go to sleep.

When the consumer consumes something, it will execute *up(&empty)*, which will wake up the producer if it is sleeping on the semaphore and allow it to add one more item to the buffer. If the producer was not sleeping because there there is still room in the buffer, *up(&empty)* will simply increment empty to indicate that there is one more space in the buffer.

The *empty* semaphore was used to allow the producer to go to sleep when there was no more room in the buffer and wake up when there were free slots. Now we have to consider the consumer. We want the consumer to go to sleep when the producer has not generated anything for the consumer to consume. To put the consumer to sleep, we will create yet another semaphore called *full* and have it count the number of items in the buffer. Since there are no items in the buffer initially, we will initialize *full* to 0. Initially, when the consumer starts running, it will run *down(&full)*, which will put it to sleep. When the producer produces something, it calls *up(&full)*, which will cause the consumer to wake up. If the producer works faster than the consumer, successive calls to *up(&full)* will cause *full* to get incremented each time. When the consumer gets to consuming, successive calls to *down(&full)* will just cause *full* to get decremented until *full* reaches 0 (no more items to consume), at which point the consumer will go to sleep again.

```
sem mutex=1;    /* for mutual exclusion */
sem empty=N;    /* keep producer from over-producing */
sem full=0;     /* keep consumer sleeping if nothing to consume */

producer() {
    for (;;) {
        produce_item(&item);    /* produce something */
        down(&empty);           /* decrement empty count */
        down(&mutex);           /* start critical section */
        enter_item(&item);       /* put item in buffer */
        up(&mutex);             /* end critical section */
```

```
            up(&full);              /* +1 full slot */
        }
    }
    consumer() {
        for (;;) {
            down(&full);            /* one less item */
            down(&mutex);           /* start critical section */
            remove_item(&item);      /* get the item from the buffer */
            up(&mutex);             /* end critical section */
            up(&empty);             /* one more empty slot */
            consume_item(&item);     /* consume it */
        }
    }
```

Readers-writers example

Now we will look at a **Readers-Writers** example. Here, we have a shared data store, such as a database. Multiple processes may try to access it at the same time. We can allow multiple processes to read the data concurrently since reading does not change the value of the data. However, we can allow at most one process at a time to modify the data to ensure that there is no risk of inconsistencies arising from concurrent modifications. Moreover, if a process is modifying the data, there can be no readers reading it until the modifications are complete.

To implement this, we use a semaphore (*canwrite*) to control mutual exclusion between a writer or a reader. This is a simple binary semaphore that will allow just one process access to the critical section. Used this way, it would only allow one of anything at a time: one writer or one reader. Now we need a hack to allow multiple readers. What we do is to allow other readers access to the critical section by bypassing requesting a lock if we have at least one reader that has the lock and is in the critical section. To keep track of readers, we keep a count (*readcount*). If there are no readers, the first reader gets a lock. After that point, *readcount* is 1. When the last reader exits the critical section (*readcount* goes to 0), that reader releases the lock.

We also use a second semaphore, *mutex*, to protect the region where we change *readcount* and then test it.

```
sem mutex=1;        /* critical sections used only by the reader */
sem canwrite=1;     /* critical section for N readers vs. 1 writer */
int readcount = 0;  /* number of concurrent readers */

writer() {
   for (;;) {
      down(&canwrite);    /* block if we cannot write */
      // write data ...
      up(&canwrite);      /* end critical section */
   }
}

reader() {
   for (;;) {
      down(&mutex);
      readcount++;
      if (readcount == 1)
         down(canwrite);      /* sleep or disallow the writer from writing */
      up(&mutex);
      // do the read
      down(&mutex);
      readcount--;
      if (readcount == 0)
         up(writer);          /* no more readers! Allow the writer access */
      up(&mutex);
      // other stuff
   }
}
```

7.2 Event counters

An **event counter** is a special data type that contains an integer value that can only be incremented. Three operations are defined for event counters:

- **read(E)**: return the current value of event counter E
- **advance(E)**: increment E (this is an atomic operation)
- **await(E,v)**: wait until E has a value greater than or equal to v

Here is an example of the producer-consumer problem implemented with event counters. Both the consumer and producer maintain a sequence number locally. Think of the sequence number as the serial number of each item that the producer produces. From the consumer's point, think of the sequence number as the serial number of the next item that the consumer will consume.

The event counter *in* is the number of the latest item that was added to the buffer. The event counter *out* is the serial number of the latest item that has been removed from the buffer.

The producer needs to ensure that there's a free slot in the buffer and will wait (sleep) until the difference between the sequence number and *out* (the last item consumed) is less than the buffer size. The consumer needs to wait (sleep) until there is at least one item in the buffer; that is, *in* is greater than or equal to the next sequence number that it needs to consume.

```
#define N 4    /* four slots in the buffer */
event_counter in=0;    /* number of items inserted into buffer */
event_counter out=0;    /* number of items removed from buffer */

producer() {
    int sequence=0;
    for (;;) {
        produce_item(&item);     /* produce something */
        sequence++;              /* item # of item produced */
        await(out, sequence-N);  /* wait until there's room */
        enter_item(&item);       /* put item in buffer */
        advance(&in);            /* let consumer know there's one more item */
    }
}
consumer() {
    int sequence=0;
    for (;;) {
        sequence++;            /* item # we want to consume */
        await(in, sequence);   /* wait until that item is present */
        remove_item(&item);    /* get the item from the buffer */
        advance(&out);         /* let producer know item's gone */
        consume_item(&item);   /* consume it */
    }
}
```

The producer's sequence number mirrors the value of *in* and the consumer's sequence number mirrors the value of *out*. We can simplify the code a bit by getting rid of sequence numbers. The producer needs to ensure that there's a free slot in the buffer and will wait (sleep) until the difference between *in* and *out* is less than the buffer size. The consumer sleeps until there is at least one item ready (*in* > *out*). Here is the slightly simplified ode.

```
#define N 4    /* four slots in the buffer */
event_counter in=0;    /* number of items inserted into buffer */
event_counter out=0;   /* number of items removed from buffer */

producer() {
    for (;;) {
        produce_item(&item);    /* produce something */
        await(out, in-(N-1));   /* wait until there's room: (in-out) < N, or out >= in-N-1 *
        enter_item(item);       /* put item in buffer */
        advance(&in);           /* let consumer know there's one more item */
}
}
consumer() {
    for (;;) {
        await(in, out+1);     /* wait until an item is present (in > out) */
        remove_item(item);    /* get the item from the buffer */
        advance(&out);        /* let producer know item's gone */
        consume_item(item);   /* consume it */
    }
}
```

Note that there is no mutual exclusion protection around *enter_item* and *remove_item* as we had with semaphores. This is because the way the code is written, the consumer waits until the buffer is full and the producer waits until the buffer is empty.

7.3 Condition variables (monitors)

Controlling critical sections reliably (without deadlocking) is often difficult and error-prone. A **monitor** is an attempt to create a higher level synchronization primitive to make it easier to synchronize events. A monitor is a collection of procedures and data with the following characteristics:

- Processes may call the functions in a monitor but may not access the monitor's internal data structures.
- Only one process may be active in a monitor at any instant. Any other process that tries to access an active monitor will be suspended.

A monitor is a **programming construct**: part of the programming language rather than the operating system. The compiler has to understand how to compile code to implement monitors (for example, it may implement the monitor constructs using the operating system's semaphores). Two operations are supported by monitors:

wait(condition_variable) Blocks until condition_variable is "signaled." Another thread is allowed to run.

signal(condition_variable) Wakes up one thread that is waiting on the condition variable. This function is often called **notify** (e.g., in Java).

A separate condition variable has to exist for each reason that a process might need to wait.

Resource allocation with monitors

Here's a small example of how a monitor may be used to allow a process to grab exclusive access to some resource and then relinquish it when it's done:

```
monitor resource_allocator {
    condition is_free;
    int in_use = 0;

    get_resource() {
        if (in_use)
            wait(is_free);    /* block until free */
        in_use = 1;
    }
    return_resource() {
        in_use = 0;
        signal(is_free);    /* wake up a waiting process */
    }
}
```

If these were normal functions, we would have to worry about a race condition developing in *get_resource* between the time we check *in_use* and set it (another thread might be switched in at that point and try to get a resource). Because this is a monitor, we are assured that mutual exclusion holds for the entire monitor, so these problems do not arise.

Advantage

Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphores.

Disadvantage

The major drawback of monitors is that they have to be implemented as part of the programming language. The compiler must generate code for them. This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes. Since few languages have any understanding of concurrency, many languages do not support monitors. Some languages that do support monitors are Java, C#, Visual Basic, Ada, and Concurrent Euclid.

Producer-consumer example with Java

This illustrates a Java example of the producer-consumer problem. The *synchronized* keyword for the *consume_ item* method ensures that the method is a critical section.

```
class producer_consumer {
    private boolean item_ready = false;

    public synchronized int produce_item() {
        while (item_ready == true) {
            try {
                wait();    // block until signaled (notify)
            } catch (InterruptedException e) { } // ignore
        }
        item_ready = true;
        // generate the item
        notify();    // signal the monitor
        return item;
    }

    public synchronized void consume_item(int item) {
        while (item_ready == false) {
            try {
                wait();    // block until notified
            } catch (Interrupted Exception e) { } // ignore
        }
        // consume the item
        item_ready = false;
        notify();    // signal the monitor
    }
}
```

8 Interprocess communication

One drawback of monitors, semaphores, and global data structures to control mutual exclusion is that they assume that all concurrent threads or processors have access to common memory and share the same operating system kernel (which is responsible for putting processes to sleep and waking them up). Asynchronous processes do to things: the synchronize (e.g., wait on one another) and they exchange data.

While this is a fine assumption for threads and processes, these solutions don't work on distributed systems where each system has its own local memory and its own operating system since none of these primitives provide for the exchange of information between machines. Messages don't have this restriction. They allow us to synchronize processes (by waiting for messages) and exchange data among processes. Moreover, they can work across different threads and processes on the same as well as different machines.

8.1 Message passing

Message passing is a form of interprocess communication that uses two primitives:

send(*destination, message*) Sends a message to a given destination.

receive(*source, &message*) Receives a message from a source. This call could block if there is no message.

When we use a network, we should be aware of certain issues:

- Networks may be unreliable. What if a message gets lost? We can try to make communications reliable via acknowledgements and retransmissions. A receiver can be asked to send an acknowledgement message when it receives a message. If a sender doesn't receive the acknowledgement in a certain amount of time, then it will retransmit the message. However, if the acknowledgement gets lost, then the sender will send two messages. We may consider tagging each message with a sequence number and have the operating system keep track of missing, redundant, or out-of-order messages.
- Processes and machines need to be named unambiguously. How do you identify a process on a different machine?
- We trust our operating system to do the right thing and provide a protection structure to keep our data private when we want it to be private. With a network, things get more complicated. Can you trust other operating systems on the network? How can you ensure that the message you received really came from where you thought it did?

Produced-consumer example with message passing

The producer-consumer example uses a messaging channel as a queue between producer and consumer threads/processes. That on its own would give us no way to control the size of the buffer since the producer could just keep sending more and more messages even if the consumer is slow to consume them. What we use instead is a bi-directional message channel. A producer can add an item *only* in response to receiving an empty message from the consumer. Each received empty message represents a slot in the buffer. The producer receives one such message whenever it has to add another item to the buffer. If there isn't an empty message available then the producer blocks on *receive* and waits until the consumer consumes an item from the buffer and responds with an empty message.

```
#define N 4        /* number of slots in the buffer */

producer() {
    int item;
    message m;

    for (;;) {
        produce_item(&item);         /* produce something */
        receive(consumer, &m);       /* wait for an empty message */
        build_message(&m, item);     /* construct the message */
        send(consumer, &m);          /* send it off */
    }
}
consumer() {
    int item, I;
    message m;

    for (i=0; i<N; ++i)
        send(producer, &m);          /* send N empty messages */
    for (;;) {
        receive(producer, &m)        /* get a message with the item */
        extract_item(&m, &item)      /* take item out of message */
        send(producer, &m);          /* send an empty reply */
        consume_item(item);          /* consume it */
    }
}
```

Advantages of messages

The concept of messages is scalable from a single processor environment to a networked system of multiple processors, each with its own memory.

Disadvantages of messages

Even on a single processor system, it is possible that message passing may be slower than other methods due to the overhead of copying the message. Some systems, however, make local message passing efficient.

8.2 Messages: Rendezvous

The example we just looked at assumed that the receiver blocked whenever a message was not ready for it and that the sender could just send a message and not be blocked; the message would get sent out and buffered by the operating system (of the receiving process if we're going across a network).

A **rendezvous** is a variant of message passing that uses no message buffering. A sender blocks until a receiver reads the message. If a *send* is done before a *receive*, the sending process is blocked until a *receive* occurs. In an SMP (symmetric multiprocessor) environment, as opposed to a multi-computer environment, the message can be copied directly with no intermediate buffering. If a *receive* is done first, it blocks until a *send* occurs.

Advantage of rendezvous

It is easy and efficient to implement.

Disadvantage of rendezvous

It forces the sender and receiver to be tightly synchronized at these send/receive junctures.

8.3 Messages: Mailboxes

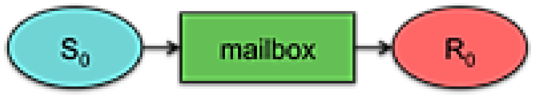

Figure 7: Mailbox: Single sender, single reader

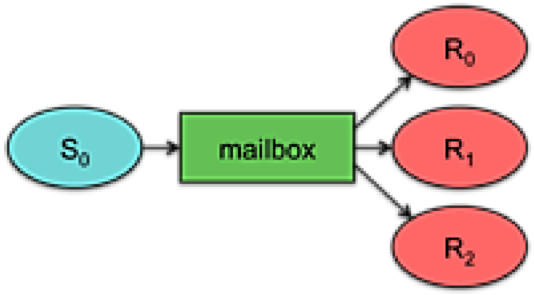

Figure 8: Mailbox: Single sender, multiple readers

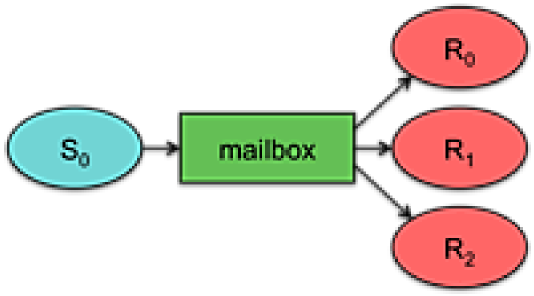

Figure 9: Mailbox: Multiple senders, single reader

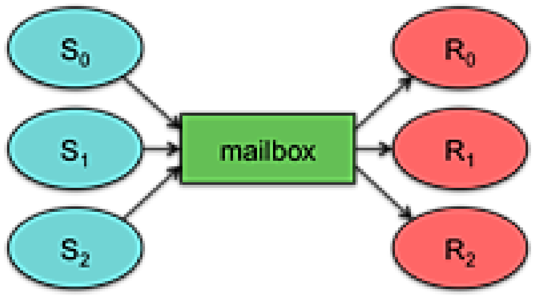

Figure 10: Mailbox: Multiple senders, multiple readers

Messages and rendezvous require the use of **direct addressing**, where the sending process must identify the receiving process. The receiving process can either specify the sender or may receive its identification as an incoming parameter from the *receive* operation. This messaging mechanism makes it easy to support a single sender/single receiver or multiple senders/single receiver models. If we want to support multiple receivers, say multiple processes on multiple computers that all want to grab messages for processing, things get complicated since the sending processes will have to have some way of figuring out which of several recipients should receive a given message.

A **mailbox** is an intermediate entity that is applied to messaging: a set of FIFO queues two which senders can send messages and from which readers can receive messages. It is an example of **indirect addressing** since neither the sender nor reader correspond directly with each other. Mailboxes make it easy to support multiple readers since they can all contact the same mailbox and extract items from the queue.

A process sends a message to a mailbox rather than to a specific other process and another process (or processes) reads messages from that mailbox. If the mailbox is full, a sending process is be suspended until a message is removed.

Advantage of mailboxes

They give us the flexibility of having multiple senders and/or receivers. Moreover, they do not require the sender to know how to identify any specific receiver. Senders and receivers just need to coordinate on a mailbox identifier.

Disadvantage of mailboxes

They may incur an extra level of data copying since data has to be first copied to the mailbox queue and then copied to the receiver. In the case of networked system, there is the question of where the mailbox should reside. Since senders and receivers can all be different computers, the mailbox becomes a distinct destination on the network: messages get sent to a mailbox and then retrieved from a mailbox. This act imposes an extra hop over the network for each message.

9 Deadlocks

Whenever we create an environment where threads compete for exclusive access to resources, we run the risk of deadlock.

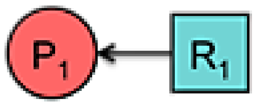

Figure 11: Assignment edge

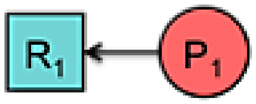

Figure 12: Request edge

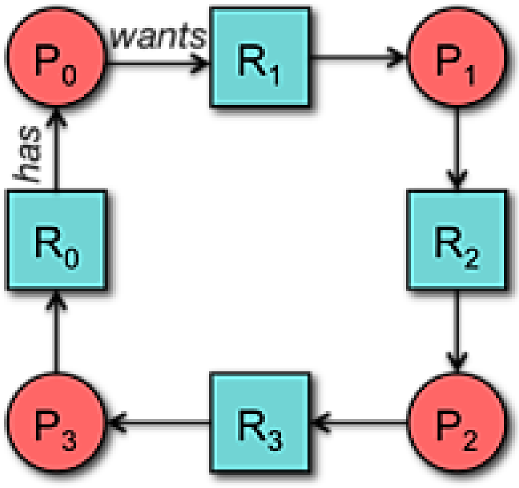

Figure 13: Resource allocation graph: deadlock!

A **deadlock** is a condition in a system where a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs. More formally, four conditions have to be met for a deadlock to occur in a system:

1. **Mutual exclusion**: a resource can be held by at most one process.
2. **Hold and wait**: processes that already hold resources can wait for another resource.
3. **Non-preemption**: a resource, once granted, cannot be taken away.
4. **Circular wait**: two or more processes are waiting for resources held by one of the other processes.

We can express resource holds and requests via a **resource allocation graph**. An **assignment edge** means that a resource R1 is allocated to process P1. Since resources are exclusive, this means that P1 has a lock on the resource. A **request edge** means that a resource R1 is being requested by process P1. If process P1 cannot be granted access to the resource R1, it will have to wait. Deadlock is present when the graph has cycles.

We can consider a few strategies for dealing with deadlock.

Deadlock prevention Ensure that at least one of the necessary conditions for deadlock cannot hold. This approach is sometimes used for transactional systems. Every transaction (process) is assigned a timestamp. For example, a transaction that wants to use a resource that an older process is holding will kill itself and restart later. This ensures that the resource allocation graph always flows from older to younger processes and cycles are impossible. It works for transactional systems since transactions, by their nature, are designed to be restartable (with rollback of system state when they abort). It is not a practical approach for general purpose processes.

Deadlock avoidance Provide advance information to the operating system on which resources a process will request throughout its lifetime. The operating system can then decide if the process should be allowed to wait for resources or not and may be able to coordinate the scheduling of processes to ensure that deadlock does not occur. This is an impractical approach. A process is likely not to know ahead of time what resources it will be using and coordinating the scheduling of processes based on this information may not always be possible.

Deadlock detection We allow deadlocks to occur but then rely on the operating system to detect the deadlock and deal with it. This requires that an operating system gets enough information to build a resource allocation graph. For instance, while an operating system will know which processes are blocked on a semaphore, it does not keep track of those that are not. After that, the operating system will need to decide what action to take. Do you kill any random process that is in the cycle? The youngest one? The oldest one? Or do you just warn the user? Operating systems avoid dealing with this.

Ignore the problem By far the most common approach is simply to let the user deal any potential deadlock.

10 References

- Edsger W. Dijkstra, Cooperating sequential processes[4]. Edsger W. Dijkstra's lecture notes on the mutual exclusion problem and semaphores.
- Abraham Silberschatz, Peter B. Galvin, & Greg Gagne, Operating System Concepts Essentials[5], Wiley, 1st edition, 2010
- Abraham Silberschatz, Peter B. Galvin, & Greg Gagne, Operating System Concepts[6], Wiley, 8th edition, 2008
- Andrew S. Tanenbaum, Modern Operating Systems[7], Prentice Hall, 3rd edition, 2007

[4]http://www.cs.utexas.edu/~EWD/ewd01xx/EWD123.PDF
[5]http://www.amazon.com/dp/0470889209/pkorg
[6]http://www.amazon.com/dp/0470128720/pkorg
[7]http://www.amazon.com/dp/0136006639/pkorg

- Intel Corporation, IA–32 Intel Architecture Software Developer's Manual[8], 2003
- Wikipedia. x86–64[9]
- Michael Chynoweth & Mary R. LeeIntel Corporation, Implementing Scalable Atomic Locks for Multi-Core Intel EM64T and IA32 Architectures[10]. Intel Corporation, November 2009

[8]http://flint.cs.yale.edu/cs422/doc/24547012.pdf
[9]http://en.wikipedia.org/wiki/X86-64
[10]http://software.intel.com/en-us/articles/implementing-scalable-atomic-locks-for-multi-core-intel-em64t-and-ia32-architectures/