OPERATING SYSTEMS FOUNDATIONS
MSIT 5170

University of Massachusetts Lowell
Department of Computer Science
Spring Semester 2020

Week #5 Lesson

1. The material in this lesson follows the reading assignment for ch 2.3 through the end of chapter 2 from the Tanenbaum text
2. Please read the text before you read through this lesson.
3. The remaining material in chapter 2 will focus on synchronization, scheduling and inter-process communication issues (IPC).
4. We continue to look at system level tools, and I encourage students to experiment with these tools. We will also build some simple applications in the Linux environment to explore some process and thread elements.

System Synchronization Requirements:

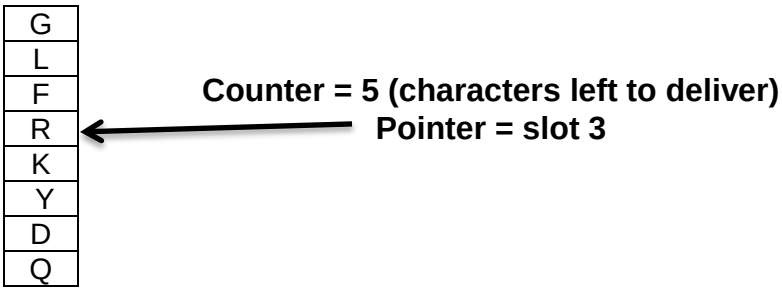
When we think about an operating system, we envision a collection of code and data that is required to maintain a coherent, stateful computing environment. Since the kernel's address space is visited by all of the threads in the system at one time or another and these threads are often interested in manipulating the kernel global data (with system calls like setpriority() or fork() for example), maintaining this data in a coherent form is a major challenge.

In this section of the book, we will examine the need for synchronization among threads that are accessing kernel code that will manipulate shared kernel global data. Remember, threads enter the kernel regularly, either voluntarily when making a system call or when coerced to handle some exception (like a clock or disk controller interrupt). In either case, the thread begins to execute kernel code that references or updates shared kernel data. Because we could have two different threads trying to run the same kernel code at the same time (if we had a system with two or more CPUs), the data these threads manipulate can become corrupted unless the kernel code is very careful about how the data is touched.

For example, suppose two threads from the same process both attempted to get the next character from an open file connection (channel). If we had a separate CPU for each of them to run on, then they could be running the exact same kernel code at the exact same instant, and this could lead to inconsistent results. The kernel code that performs a read system call and returns data to the caller depends on a kernel buffer that holds the requested data, and that buffer is controlled by a counter that specifies how many bytes are in the buffer, and a pointer that specifies where the next character will be taken out from the buffer. The kernel read code must check the counter to see if there are more characters in the buffer, must use the current buffer pointer to remove one character from the buffer to return to the caller, and then must update the pointer (to the next slot in the buffer) and decrement the character count (since a character has been logically removed). The pointer and the counter must be manipulated by only one thread at a time (an atomic update), or we could have erroneous results.

To understand the problem, you have to remember that to test or manipulate a data object, the object must first be brought from its RAM storage location into a CPU register (loaded from memory), where a CPU instruction can operate on it. If the data object is changed as a result of such an operation, then it must be moved back to RAM (stored into memory) after the change. The order of loads and stores on a computing system with multiple CPUs is non-deterministic. Each CPU needs a bus cycle to move an object between a CPU register and a memory location, but the CPUs can never be sure about what bus cycles they will get, since they are often asking at the same time, and the bus arbiter will assign the available bus cycles in a non-deterministic way. For the example above, 4 critical bus cycles will be needed for each thread to complete its read. A cycle will be needed by each thread to load the counter and check for more characters. If there are characters, then the counter value will be decremented, and the updated value will need a bus cycle to get back to memory. Each thread will then need to load the pointer value to remove a character from the buffer to return from the read call, and will then increment the pointer and use a bus cycle to store the updated value in memory.

A kernel buffer with characters to be returned to a read system call:

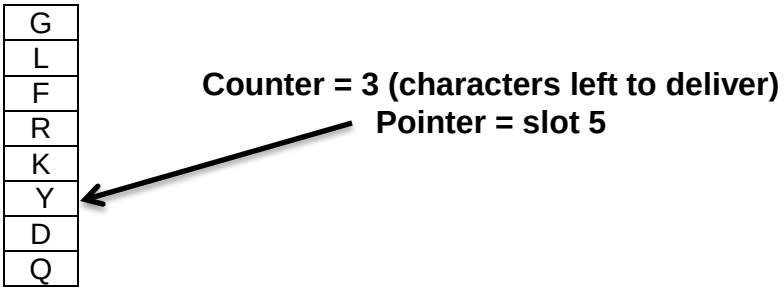


If the relative order of the bus cycles goes like this:

1. Thread A loads counter
2. Thread A decrements the counter (counter = 4) and stores it back to memory
3. Thread A loads the pointer
4. Thread A uses the pointer (return character R), increments the pointer to the next buffer slot and stores it back to memory (pointer now = 4, pointing at character K)

4. Thread B loads counter
5. Thread B decrements the counter (counter = 3) and stores it back to memory
6. Thread B loads the pointer
7. Thread B uses the pointer (return character K), increments the pointer to the next buffer slot and stores it back to memory (pointer now = 5, pointing at character Y)

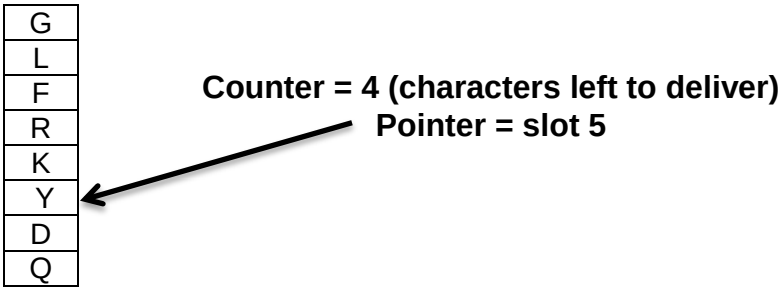
Then everything is fine, and each thread gets a discrete character from the buffer (thread A got R and thread B got K) and the buffer has been updated coherently as shown below:



But we could have had a different result if the bus cycles were given out in some other order:

1. Thread A loads counter
2. Thread B loads counter
3. Thread A decrements the counter (counter = 4) and stores it back to memory
4. Thread B decrements the counter (counter = 4) and stores it back to memory
   Counter has not been updated properly
5. Thread A loads the pointer
6. Thread A uses the pointer (return character R), increments the pointer to the next buffer slot and stores it back to memory (pointer now = 4)
7. Thread B loads the pointer
8. Thread B uses the pointer (return character K), increments the pointer to the next buffer slot and stores it back to memory (pointer now = 5)

The pointer update shows 3 more characters left (Y, D and Q), but the corrupted counter value thinks there are 4 characters left, which may lead to a buffer overflow, and the return of invalid data. Now the picture looks like this and is **inconsistent**:



Synchronization Mechanisms:

You might think that the previous example could only happen on multiple CPU systems which could have two or more threads running in true concurrent fashion, but this problem exists even in single CPU systems. When a thread is running on the only CPU in the system, it's reasonable to assume that any code it runs to update control data would happen atomically, but you must remember that a thread can be coerced to run an exception routine at any time (if the CPU it's running on just took an interrupt from a network controller for example), and if the exception code attempts to manipulate any of the data that the thread was working on at the time of the interrupt, that data could become corrupted in a fashion like that shown above.

In the case of the single CPU type system, the operating system uses a strategy of disabling interrupts before attempting to execute any code that might update data that an interrupt handler may also be interested in. If no exceptions can coerce the running thread to run contending exception code, then the thread will achieve the required atomic update to the potentially shared data. In single CPU systems, any code that has to run atomically begins by disabling interrupts, and finishes by restoring interrupts.

Spin Locks:

This approach does not solve the multi-CPU problem, since disabling interrupts is a CPU specific activity, and while a thread may be safe from contending exception code on its own CPU, we could find contending code running on some other CPU. In systems with multiple CPUs, we revert to using a locking mechanism on the data that is subject to contending update activity. The lowest level lock used in such systems is called a spin lock, and, as the book explains, spin locks can be implemented in simple load/store systems using software solutions like Peterson's algorithm. Even though software solutions are known to work, they require multiple machine instructions per execution and are thus very inefficient. Most hardware vendors have added a specialized machine instruction to their CPUs (often called a test-and-set instruction) to support the efficient implementation of spin locks. A test-and-set style instruction is specialized in the sense that it guarantees the CPU executing the instruction two back-to-back bus cycles that no contending CPU can get between. This means that such an instruction can load a memory location into the CPU on one cycle, and store a CPU register back into memory on the guaranteed next cycle. The book describes an Intel x86 version of this instruction called the exchange instruction:

XCHG Reg, Mem_location

The XCHG instruction guarantees the atomic exchange of the content of the CPU register **Reg** with the RAM location **Mem_location**, with two adjacent bus cycles and no possible intervention from any other bus master.

The spin lock depends on the Mem_location keeping the state of the lock. For example, let's say that Mem_location is **0** if unlocked, or **1** if locked. The exchange instruction is then used with a CPU register set to 1 against this Mem_location. After the instruction executes, the Mem_location will be 1, since the Reg content is deposited there, and it's the **value that is now in the Reg** that determines if we were successful in getting the lock. If the value in the Reg is still 1, then Mem_location was locked when we exchanged with it, and we will just do the entire operation over and over again (spin) until we see the Reg value become 0. If the value in the Reg is 0, then we now own the lock, since we know we put a 1 there no matter what, and if we got a 0 back, it means that the Mem_location was unlocked when we made out last exchange execution. The term spin lock lives up to its name, since a code path that needs such a lock will simply spin through try after try until it succeeds in getting the lock. When the holding code path is done doing its atomic updates, it releases the spin lock by simply storing a 0 value back into the Mem_location. Note that any other code path on a different CPU that needs the lock will be spinning until the current holder stores that 0 value back into the Mem_location.

Blocking Locks:

Spin locks are always at the base of any locking mechanism used on a multiple CPU platform, but they can only be used in limited ways. If a thread is attempting to get a busy spin lock, then that thread is burning off CPU cycles without getting any real work done. If the wait is expected

to be very short (a few hundred instructions worth of execution), then this may be reasonable, but when a wait is likely to last a long time, it may be better to force the waiting thread to give up its CPU and move from the **Run** state to the **Block** state until whatever it's waiting for happens. Spin locks leave the caller in the Run state, possibly wasting valuable CPU time as its spins waiting for the lock. Blocking locks force the calling thread to the Block state if they're not available, allowing the CPU to be used by some other productive thread. One of the most widely deployed types of blocking lock is an object called a semaphore. A semaphore is a data structure that contains two elements, a simple integer count field, and a queue. The count field is some value that ranges from 0 to some logical positive number (depending on the application). The queue is used to keep track of any threads that are currently blocked on the semaphore. There are two basic operations for a semaphore known as the wait and signal operations. The wait operation is a **conditional decrement** with the following semantics:

wait(sem) if **sem.count** != 0, then **decrement** sem.count and continue execution
else
add yourself to **sem.queue**, and block (yield the CPU to another thread)

the signal operation is a **conditional increment** with the following semantics:

signal(sem) if **sem.queue** is not empty, wake up a thread (move to **Ready** state)
else
increment the sem.count

As you can see, a simple lock uses a possible sem.count value of 0 or 1 (known as a binary semaphore), with the idea being that if the sem.count is currently 1, when a thread makes a **wait(sem)** call, that thread will **continue to execute**, but the sem.count has now become 0. A subsequent thread calling wait(sem) before the first thread calls signal(sem), will find a sem.count of 0 and will thus **block** on the **sem.queue.** When the first thread is finished with its atomic operations, it will call **signal(sem)** which, in this case, will wake up the thread on the sem.queue by moving it from the sem.queue (a Block state) to the **Ready** state. The semaphore components must be carefully synchronized themselves, and this accomplished (as with all synchronization primitives) using an underlying spin lock (remember, spin locks are at the bottom of all synchronization mechanisms for multi CPU systems). The **wait(sem)** function, for example, depends on a spin lock to access the semaphore safely, check its value and proceed to either decrement the sem.count and release the spin lock and allow the caller to continue execution, or, if the sem.count ==0, to safely place the calling thread on the semaphore queue, change its state to Block, unlock the spin lock and **context switch** the CPU to another thread.

Semaphores can also be used to protect multiple instances of a resource, by allowing the **sem.count** value to range between 0 and N, where N represents the instance count of some resource. The classic example of this type of use is a **ring buffer**, into which some producer thread places objects, and from which some consumer thread removes objects. Ring buffers are used extensively within an operating system to provide hysteresis between **producer and consumer** threads that may run at different speeds and at different times.

For example, a network interface controller (**NIC**) is connected to a network and listens for packets moving across the network that are addressed to it. When it discovers a packet on the network that carries its destination address, it copies the packet off of the network wire and into a packet buffer on the controller if that buffer is available (if not, the packet is simply discarded). Once the controller has captured the packet, it interrupts a CPU and coerces the current thread

that's running on that CPU to move into kernel space and run the **exception handler** for the NIC interrupt. This **interrupt handler code** will copy the packet from the controller's packet buffer into a slot in a ring buffer used by the driver to manage packets. Interrupt handlers must do the minimum amount of work required to service their device, and finish as quickly as possible, since they've basically hijacked some thread that has its own job to do, and must release this thread back to where it came from. This means that the interrupt handler will not process (rip) the packet (that's far too time consuming to do during an interrupt), but will clear the device buffer so the device can receive the next packet targeted to it, by attempting to move the just arrived packet from the controller's packet buffer into a slot in a memory based **ring buffer**. The interrupt handler (in conjunction with the NIC) is the ring buffer producer, and we now need a thread to remove the packet from the ruing buffer (a consumer) and extract its content to the intended endpoint. The consumer thread is usually a thread that is either processing a previously arrived packet, or is waiting in the Block state for the arrival of a packet to process. The producer and consumer threads synchronize their operation by using a ring buffer of N slots, a **producer semaphore** (Psem) whose **Psem.count** value can vary between 0 and N, a **consumer semaphore** (Csem)) whose **Csem.count** value can also vary between 0 and N and two pointers called **in_ptr** (used by the producer) and **out_ptr** (used by the consumer0).

```
#define N 100
some_object_type    ringbuffer[N];
int                 in_ptr=0, out_ptr=0;
sem_type            Psem = N, Csem = 0;  // Psem.count counts empty slots
                                         // Csem.count counts slots holding objects
```

PRODUCER CODE

```
void pro_funct (some_object_type *obj){
  wiat(Psem);    // wait for a space
  riingbuffer[in_ptr] = *obj;
  in_ptr = (in_ptr + 1) % N;
  signal(Csem);
  return;
}
```

CONSUMER CODE

```
some_object_type con_funct(void){
  some_object_type rtn_obj;
  wait(Csem);   // wait for an object
  rtn_obj = riingbuffer[out_ptr];
  out_ptr = (out_ptr + 1) % N;
  signal(Psem);
  return rtn_obj;
}
```

We start the producer semaphore at N so we can safely perform the **wait(Psem)** call N times before the calling producer would have to wait (out of empty slots). Each time the producer runs through its code, it calls **signal(Csem)**, to either awaken a waiting consumer thread, or increment the **Csem.count** of objects. We start the consumer semaphore at 0 (since there are no objects initially in the ring buffer), assuring that a consumer who arrives before any producer will have to wait on the **Csem.queue**. The semaphore implementation shown, guarantees that the number of producer executions can lead the number of consumer executions by no more than N, assuring that a producer cannot overflow the ring buffer (i.e. write into a slot that has not yet been consumed since the previous time it was written), nor can a consumer underflow the ring buffer (consume from a slot that has not been re-written since the last time it was consumed). The in and out pointers are incremented **MOD N** to insure their values can never be out of bounds, and must always remain between 0 and (N-1) (the valid subscripts to an N element array).

Other kinds of blocking synchronization primitives are discussed in the book (mutexes, monitors, message passing, etc.), but I will only test over semaphores on the first half exam.

Scheduling:

Now that we've looked at processes and threads in some detail, and we've looked at the states and state transitions of threads during their lifetimes, we need to consider the **Dispatch** arc in our state diagram in greater depth. Threads are dispatched to the **Run** state as a result of the **scheduling mechanisms** used by an operating system. Scheduling mechanisms are generally designed to meet various objectives, including:

- Maximizing system throughput
    - get the most work out of the system
    - keep as many system components productively executing as possible
    - schedule to get maximum cache utilization (affinity considerations)
- Minimizing execution latency
    - when an important thread wants to run, get it from the Ready state to the Run state quickly
- Providing predictability
    - limiting the variance in dispatching to promote fairness

A scheduling policy is a specific set of constraints that trades off one of these objectives to optimize for another. Most of the contemporary operating systems in use today divide the scheduling arena into two basic policies:

- Timesharing
    - generally focused on providing good response time and reasonable throughput
    - providing fairness by **adjusting thread priorities** based on thread behavior
    - providing predictability by limiting the amount of time a thread can hold a CPU before it's required to yield the CPU to a **Ready** peer
- Real Time
    - generally focused on minimizing latency
    - fairness is typically not of high importance, and real time thread priorities are generally **NOT** adjusted based on their behavior.
    - dispatching variance may be high with real time threads, since one thread may hold onto a CPU for a long time before yielding to a peer

Each thread is created with a scheduling **policy** and a scheduling **priority** within that policy. If the policy is timesharing, threads usually have a small range of relatively low priorities, and dispatching is done by selecting the highest priority thread in the **Ready** state to be the next thread to move to the **Run** state. Timesharing is also characterized by assigning a dispatched thread a **time-slice,** or **quantum** of execution time. When a thread in the **Run** state exhausts its **time-slice** it may have to yield the CPU to a peer, and wait for another turn back in the **Ready** state. One of the most common timesharing policies is known as Preemptive Highest Priority First/Round Robin (**PHPF/RR**), and operates with the following constraints:

- whenever a thread moves to the **Ready** state, its priority is checked against the currently executing thread
- if the newly arriving thread has a **higher priority** than the currently executing thread, then the currently executing thread will be **pre-empted,** and forced to **context switch** with the higher priority **Ready** thread
- when the currently executing thread has **exhausted its quantum**, it must look to the threads in the **Ready** state to see if there is one with the **same priority** as itself. If a peer exists, the running thread will be preempted back to the **Ready** state and its peer

will be moved to the **Run** state (**round robin** peer rotation). This forces threads with the same priority to share the CPU among themselves
- threads that are not getting much CPU time will **gradually have their priorities adjusted** upwards (**aging**), to help them compete with higher priority threads
- higher priority threads that are getting a lot of CPU time, will gradually have their priorities adjusted **downward** in an effort to provide more sharing (fairness)

In contrast to timesharing policies, consider the real time policy called **Real Time FIFO** (one of the policies available on Linux). This policy provides priorities that can be **much higher** than a timesharing priority, thereby assuring that when a thread with this policy is moved to the **Ready** state it will soon be running on a CPU since its high priority will force a preemption event with some lower priority timesharing thread currently using the CPU. Once a Real Time FIFO thread reaches a CPU, it will **hold** that CPU until it either finishes, blocks or is preempted by an even higher priority thread. It has no **timeslice** to force it to share with peers, and its priority is **NEVER** adjusted regardless of its behavior.

On single CPU systems, we generally consider **policy and priority** as the major factors driving scheduling decisions. We can envision a single **Ready** queue that's sorted by priority, and each time a **dispatch** operation must happen (each context switch), we simply select the thread at the head of this priority sorted ready queue and move it into the **Run** state. When we have a system with more than one CPU however, we must consider upon which CPU a thread **last executed**, and try and make sure that we run it on **this same CPU** again when it's next dispatched. When a thread runs on a given CPU, **it accumulates a cache footprint** there, and that footprint can provide a big **performance boost** if the thread runs in the presence of this footprint the next time it is dispatched. Giving preference to a specific CPU when we run a thread for a second, third, etc. time is called **affinity scheduling**, and is a growing consideration with most operating systems as the number of CPUs per system continues to increase.

Week #4 Summary:

Because operating systems include code and shared global data that are used and manipulated by all of the threads on a platform, there is a critical need for **synchronizing** access to these shared global data. If two threads are concurrently running the same code on separate CPUs, and that code manipulates shared data, the data may become **corrupted** if it is not treated **atomically** by each thread. Atomic execution can be assured by using an appropriate **synchronization primitive** such as a spin lock or semaphore

Spin locks are the lowest level primitive, and all higher level synchronization mechanisms (such as semaphores), use spin locks at the base of their implementation. Spin locks are ideal when there is little contention for the lock, and when no thread holds the lock for more than a very short time. If a thread needs to wait for a long period of time, **a blocking mechanism like a semaphore** is sensible. Semaphores may be used as simple locks (a binary semaphore), or as more elaborate multiple instance resource managers (counting semaphores).

Ring buffer sharing between a **producer** and **consumer** threads, provides a perfect application for counting semaphores. Since ring buffers are used extensively in virtually all operating systems (especially in the device drivers used in an operating system), it is important to understand how they are deployed, and how the semaphores used for keeping track of empty and full slots can avoid **overrun** and **underrun** by the contending threads.

Scheduling of threads to CPUs is done in many different ways from system to system, but we can generally categorize schedulers by their policy and priority range. The two main types of policies that we considered are **Timesharing** and **Real Time**, and it's important to understand the basic differences between them. While timesharing policies tend to put **fairness** high on the desirable attribute list, real time policies tend to focus on **low latency** dispatch and minimal system intervention (no dynamic priority adjustments).