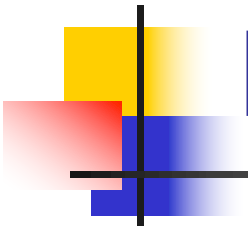
# *Synchronization Tools*

CS201 Lecture 6


Jason Hibbeler

University of Vermont

Spring 2019



# Motivation

Processes in the system can execute concurrently or in parallel

- are these synonymous?
- why or why not?

Furthermore, processes can share resources

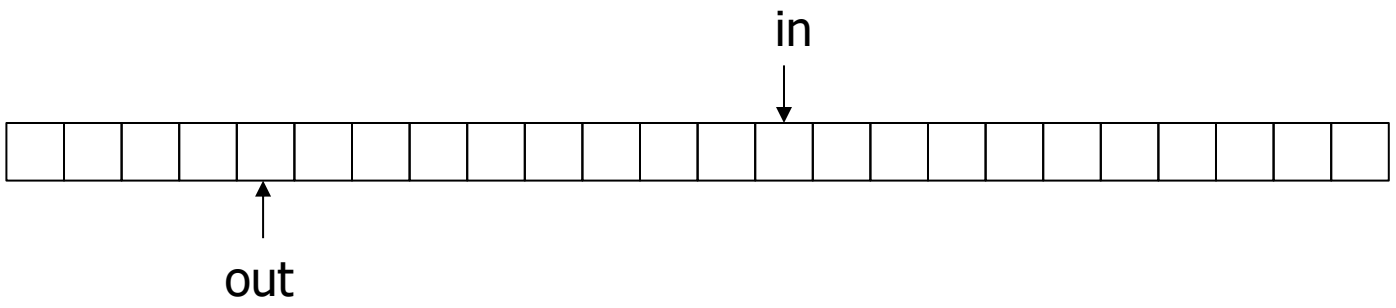
- such as memory

Let's look at an implementation of the producer-consumer model.

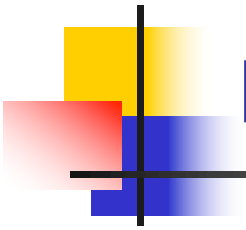
# Bounded (Circular) Buffer

Two tasks:

- a producer (writes to the buffer)
- a consumer (reads from the buffer)




and a third variable, counter: #unread items



# Producer/Consumer with Bounded Buffer

Here's the producer:

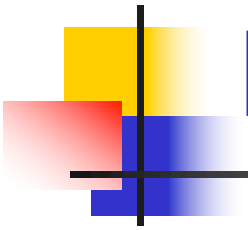
```
while (true) {
   while (counter == BUFFER_SIZE)
          /* do nothing */ ;

   buffer[in] = next_item_produced;
   in = (in + 1) % BUFFER_SIZE;
   ++counter;
}
```

Here's the consumer

```
while (true) {
   while (counter == 0)
          /* do nothing */ ;

   next_item_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   --counter;
}
```

Note that each of these seen separately is correct

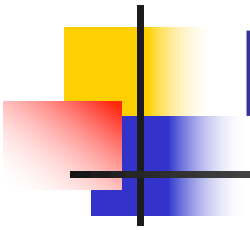
- observe that `counter` is a shared variable
- what happens to `counter` if these are executing concurrently?



# Motivation

Suppose the current value of counter is 10

- producer executes, after which the value of counter is 11
- then consumer executes, after which the value of counter is 10 again



# Motivation

But in a preemptive, multitasking environment, a process can be interrupted at any time

Let's analyze this by thinking about the individual machine instructions that implement "increment `counter`"

# Incrementing/Decrementing a Variable

*process 1*

++counter:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

*process 2*

--counter:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Now interleave these instructions:

```
register1 = counter          register1 has the value 10
register1 = register1 + 1    register1 has the value 11
register2 = counter          register2 has the value 10
register2 = register2 - 1    register2 has the value 9
counter = register1          counter has the value 11
counter = register2          counter has the value 9
```

# Race Condition

We've created a race condition due to fact that both processes are modifying the shared variable `counter`

- this is because the order in which the processes execute is not deterministic
- to prevent this, we need to insure that only one process at a time can access the shared variable, through **process synchronization**

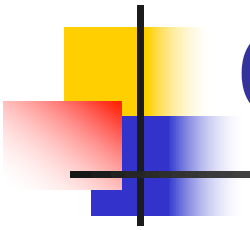
# The Critical-Section Problem

A characterization of processing in a multitasking environment

```
do {
    entry section
    critical section
    exit section
    remainder section
} while ( true );
```



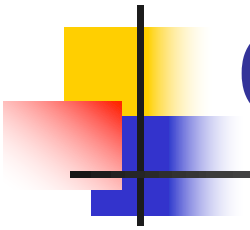
**requirement:**
when a process is in its critical section, then no other process on the system can be in its critical section



# Critical-Section Problem

The system (the OS) must enable processes to have this structure and must provide a solution that satisfies the three conditions:

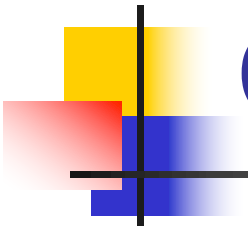
1. mutual exclusion
2. progress
3. bounded waiting



# Critical-Section Problem

The system (the OS) must enable processes to have this structure and must provide a solution that satisfies these conditions:

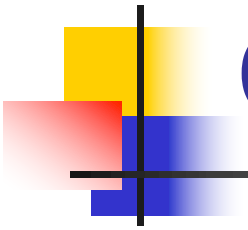
1. **mutual exclusion**: If process P is executing in its critical section, then no other processes in the system can be executing in their critical sections



# Critical-Section Problem

The system (the OS) must enable processes to have this structure and must provide a solution that satisfies these conditions:

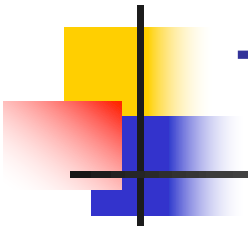
2. **progress**: if no process is executing in its critical section, and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.



# Critical-Section Problem

The system (the OS) must enable processes to have this structure and must provide a solution that satisfies these conditions:

3. **bounded waiting**: there exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

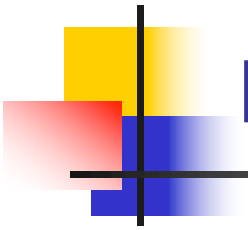


# The Kernel Itself

Note that the kernel maintains many data structures that are subject to race conditions

- list of open files, tables for memory allocation
- `next_available_pid`: if two different processes execute a fork, the the kernel's access (and modification) of `next_available_pid` must be subject to mutual exclusion

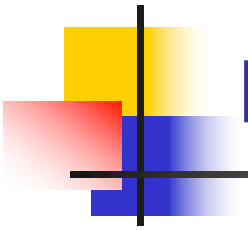
If a process, while running in kernel mode, needs to modify one of these data structures, then that modification needs to be performed in a critical section



# Preemptive vs. Nonpreemptive Kernels

If the kernel itself is not subject to preemption while running in kernel mode, then the kernel can safely modify its data structures on behalf of user processes (*i.e.*, race conditions are avoided).

But, if the kernel can be preempted while operating in kernel mode, then it is subject to race conditions.



# Preemptive vs. Nonpreemptive Kernels

Why not require the kernel to be nonpreemptive?

- on multi-core systems, two kernel-mode processes might be running in parallel on different cores and might simultaneously access a shared resource
- on a real-time system, we have to be able to preempt the kernel to handle a high-priority task

# Solution to the CSP: Peterson's Solution

A software-only solution, first published in 1981.

Here is the code for process $P_i$

```
while (true) {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j)
      ; // wait
   // here is the critical section
   flag[i] = false;
   // here is the remainder section
}
```

In software, this will work!

# Peterson's solution for P0 and P1

here is Process 0

```
while (true) {
  flag[0] = true; // I want to enter crit sect
  turn = 1; // but you can go if you want
  while (flag[1] && turn == 1)
   ; // I wait while you're in crit sect
  // critical section
  flag[0] = false; // I'm no longer waiting
  // remainder section
}
```

here is Process 1

```
while (true) {
  flag[1] = true;
  turn = 0;
  while (flag[0] && turn == 0)
    ; // wait
  // critical section
  flag[1] = false;
  // remainder section
}
```

- the variable `turn`: shows whose turn it is
  - turn == 0: must be true when P0 is in its critical section
  - turn == 1: must be true when P1 is in its critical section
- `flag[]`: shows that a process is ready to enter its critical section
  - so flag[0] == true means P0 is ready to enter its critical section
  - flag[1] == true means P1 is ready to enter its critical section

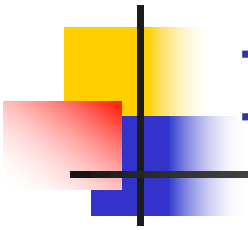
# Peterson's solution for P0 and P1

```
while (true) {
  flag[0] = true; // I want to enter crit sect
  turn = 1; // but you can go if you want
  while (flag[1] && turn == 1)
    ; // wait while you're in crit sect
  // critical section
  flag[0] = false; // I'm no longer waiting
  // remainder section
}
```

```
while (true) {
  flag[1] = true;
  turn = 0;
  while (flag[0] && turn == 0)
    ; // wait
  // critical section
  flag[1] = false;
  // remainder section
}
```

P0 and P1 are both accessing the same variable—will there be a race condition here?

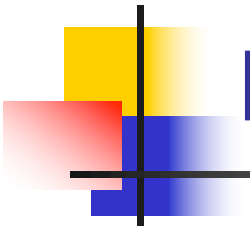
- no matter how the variable `turn` is set, it will have the value either 1 or 0
- and if it's set to 0, then P0 will get past the while loop and get into its critical section, and P1 will spin in the while loop
- and vice versa if `turn` ends up with the value 1



# Inadequacy of Software Solutions

Software solutions to the CSP won't work

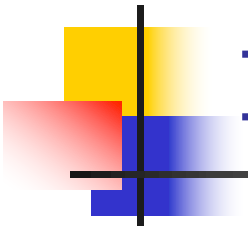
- the compiler can reorder instructions, causing them to be executed in a different order than in the source code (*out-of-order execution*)
- the CPU itself can process instructions in a different order than they appear in the code (related topic: Spectre and Meltdown—security vulnerabilities that exploit this)
- conclusion: we will need a noninterruptible hardware operation to enable our synchronization



# Parallel Execution

Parallel execution on different processors can cause a race condition

- when both processors modify the same variable
- in general, there is no way to predict the order in which execution streams on different processors will be processed



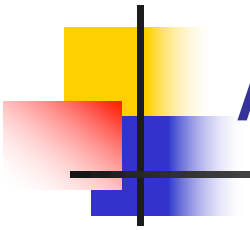
# Inadequacy of Software Solutions

We also have the problem that there's not a 1-to-1 correspondence between operations in the higher-level language and machine instructions

- such as the "++counter" statement from earlier

We need a way to prevent interrupts during the modification of a shared variable

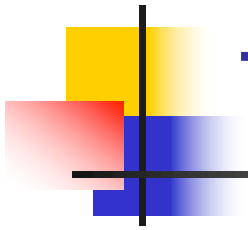
- it's not realistic to disable interrupts



# Atomic Hardware Instruction

We need a hardware instruction that will implement an operation visible in our high-level language

- and do this in an atomic (uninterruptible) fashion
- then we can solve the CSP
- one such atomic hardware instruction will let us swap the contents of two words atomically (i.e., without an interrupt)



# test_and_set()

```
boolean test_and_set(boolean *target) {
  boolean rv = *target;
  *target = true;
  return rv;
}
```

The key thing about this instruction is that it is executed *atomically*

- i.e., it can't be interrupted

# CSP Solution Using test_and_set()

```
// lock is a global variable, initialized to false

do {
  while ( test_and_set(&lock) )
    ; // do nothing

  // critical section

  lock = false;

  // remainder section

} while (true);

// so the code will spin in the while loop until a
// different process sets lock to false
```

we will break out of this loop only if lock was false to start with; after we break out, lock will be true

this means that anyone else who is calling test_and_set(&lock) in while loop like this will stay in the loop until we set lock to false

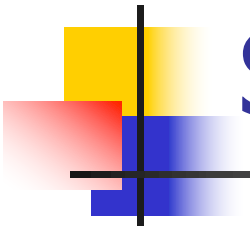
# compare_and_swap()

There is also another atomic instruction called compare_and_swap()

```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;
  if (*value == expected)
    *value = new_value;
  return temp;
}
```

# compare_and_swap()

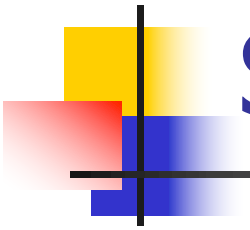
```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;
  if (*value == expected)
    *value = new_value;
  return temp;
}

// lock is initialized to 0
while (true) {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ; // do nothing
  // critical section
  lock = 0;
  // remainder section
}
```



# Solving the CSP

Does the solution using test_and_set() actually solve the critical-section problem?

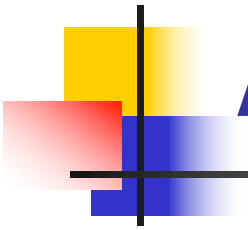
1. mutual exclusion - yes
2. progress - yes
3. bounded waiting - no, won't satisfy this condition



# Solving the CSP

A correct solution involving `compare_and_swap()` is more complex

- it's described in the book (Figure 6.9)
- understand why the simple solution doesn't solve the CSP



# Atomic Variables

Recall that the simple act of incrementing a variable can lead to a race condition

- as shown on Slide 5

Some systems provide an atomic operation on simple data structures (integers, booleans)

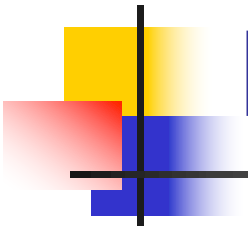
- increment(&lock)
- can implement this using a more primitive instruction, such as compare_and_swap()

# Busy Waiting

Busy waiting means that a processor is tied up until the desired condition comes true

- this structure is also called a spinlock
- is this a good thing? Why? Why not?

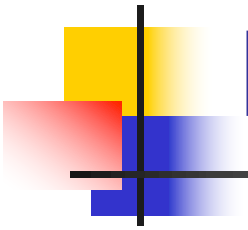

```
while ( test_and_set(&lock) )
  ; // do nothing
```



# Busy Waiting

Busy waiting means that a processor is tied up until the desired condition comes true

- this structure is also called a spinlock
- not necessarily a good thing in general (sorry, Martha)
- busy waiting requires the CPU
- so any process that is in the `while` is using CPU for no clear benefit
- but, in fact, busy waiting for threaded programs, when a thread has to wait only a short time, is efficient
  - because waiting on a resource (= being taking off of the CPU and moved to the waiting state) requires a context switch; and then another context switch when the wait is complete and the task is dispatched again to the CPU
  - so for a "short-duration" wait, a spinlock is OK



# Mutex Locks

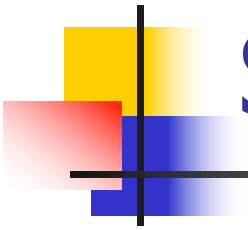
A mutex lock is an OS-provided structure and API to guarantee mutual exclusion

```
do {
   acquire_lock();
   // critical section
   release_lock();
   // remainder section
} while (true);
```

these functions are in the kernel

```
void acquire_lock() {
   while ( ! available )
      ; // busy wait
   available = false;
}
```

```
void release_lock() {
   available = true;
}
```



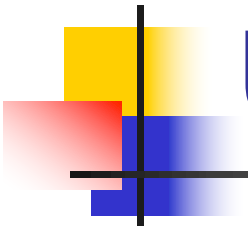
# Semaphores

A semaphore *S* is an integer variable that can be accessed only through two standard operations: wait() and signal()

```
void wait(S) {
  while ( S <= 0 )
    ; // busy wait
  --S;
}
```

```
void signal(S) {
  ++S;
}
```

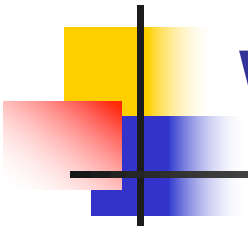
the OS implements these as atomic operations



# Use of Semaphores

In general, the semaphore can have a value between *0* and *n*

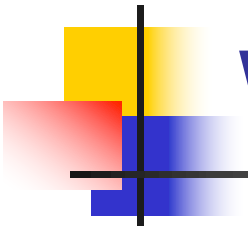
- a mutex lock is a special case of a semaphore: in the mutex lock, the variable *S* can have the value 0 or 1 (a binary semaphore)
- a counting semaphore: we can use a semaphore that has the values *0, 1, ..., n* to restrict access to a resources to *n* processes: initialize the semaphore to *n*; then each process does a wait() in order to gain access to the resources and does a signal() when it's done



# Weakness of Locks and Semaphores

As we've described things, it's possible to solve the CSP using locks or semaphores.

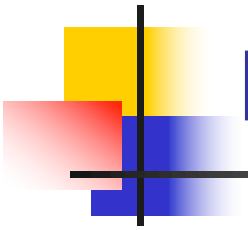
In particular, it's possible to ensure than only one process is executing in its critical section at any time



# Weakness of Locks and Semaphores

Even if we assume that we have a correct algorithmic solution to the CSP, software developer must correctly implement the code that protects the critical section!

- do software developers always write code correctly?
- suppose instead the programming language itself enabled the control over the critical section
- why would this be a great thing?
- and why do we care about critical sections?



# Monitors

Monitor: a high-level (language-level) synchronization construct

- an example of an abstract data type
- kind of like a class in which mutual exclusion is enforced

```
monitor monitor_name {
  // shared variable declarations
  function F1() { }
  function F2() { }
  …
  initialization_code() { }
}
```

only one process at a time can be active within the monitor

the book has quite a bit more about monitors in section 6.7, but this is enough

# Deadlocks and Starvation

If two or more processes are each waiting for an event that can only be caused be a different process, then the processes are **deadlocked.**

```
acquire_lock(L1);
acquire_lock(L2);
// do some work
release_lock(L1);
release_lock(L2);
```

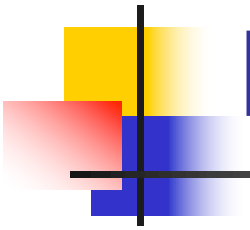
```
acquire_lock(L2);
acquire_lock(L1);
// do some work
release_lock(L2);
release_lock(L1);
```

**Starvation** is the situation in which a process waits indefinitely for a resources (such as a semaphore).

# Deadlock: Python Example

I'll show you a small python program with threads and locks

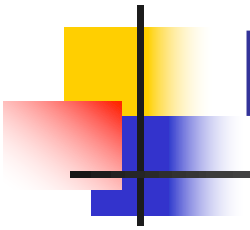
```
acquire_lock(L1);
acquire_lock(L2);
// do some work
release_lock(L1);
release_lock(L2);
```



# Priority Inversion

Section 6.8.2: just know that it's a situation in which a process with a lower priority can influence how long a higher-priority process must wait for a requested resources.

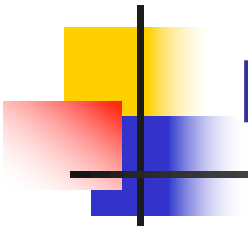
And this does not seem like behavior that we would want.



# Lock Contention

Contention: if a lock is available when a thread attempts to acquire it, then it is uncontended

- if it's not available, then it's contended
- high contention means that the program is spending a lot of time waiting for a lock rather than processing
- high contention $\Rightarrow$ poor performance



# Evaluation of Synchronization Schemes

In the end, we have to ask whether the scheme we employ solves the problem

- correctly: CSP conditions are enforced
- efficiently: code avoids excessive contention
- in a maintainable fashion: code is not too elaborate or difficult to understand