OS Lecture 10

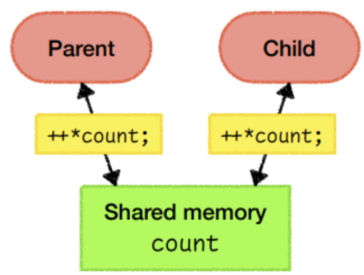
Recall: race conditions (Data Races)

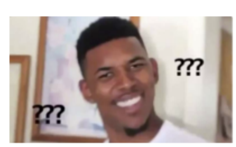
Out of sync?

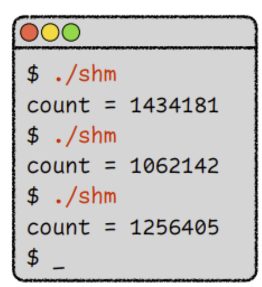
```
int main() {
  // create a 4-byte shared memory
  int *count = mmap(NULL, 4, PROT_READ | PROT_WRITE,
                  MAP_SHARED | MAP_ANONYMOUS, -1, 0);

  pid_t pid = fork();

  for (int i = 0; i < 1000000; ++i) {
    ++*count;
  }

  if (pid) {
    wait(NULL);
    printf("count = %d\n", *count);
  }
}
```

What does it output?

shm.c







```c
int main() {
  // create a 4-byte shared memory
  int *count = mmap(NULL, 4, PROT_READ | PROT_WRITE,
                    MAP_SHARED | MAP_ANONYMOUS, -1, 0);

  pid_t pid = fork();

  for (int i = 0; i < 1000000; ++i) {
    ++*count;
  }

  if (pid) {
    wait(NULL);
    printf("count = %d\n", *count);
  }
}
```

shm.c

```
$ gcc -S shm.c
$ cat shm.s
…
    movq    -16(%rbp), %rax
    movl    (%rax), %eax
    leal    1(%rax), %edx
    movq    -16(%rbp), %rax
    movl    %edx, (%rax)
…
```

```
$ gcc -S shm.c
$ cat shm.s
…
    movq    -16(%rbp), %rax
    movl    (%rax), %eax
    leal    1(%rax), %edx
    movq    -16(%rbp), %rax
    movl    %edx, (%rax)
…
```

```
a = count;
a = *a;          →   a = *count;
d = a + 1;       →   d = a + 1;
a = count;
*a = d;          →   *count = d;
```

What could go wrong?

	Parent	Child	count
Parent is running	a = *count; (a = 0)		0
	d = a + 1; (d = 1)		0
Context switch			
Child is running		a = *count; (a = 0)	0
		d = a + 1; (d = 1)	0
		*count = d; (*count = 1)	1
Context switch			
Parent is running	*count = d; (*count = 1)		1

This scenario is called a **race condition** (or, more specifically, a **data race**).

The results depend on the **timing** of the execution, i.e., the particular order in which the shared resource is accessed.

Race conditions are always bad…

- Worse yet, compiler optimizations may generate crazy output if your code has data races.
- What if you compile the previous code with "*gcc -O1*" and "*gcc -O2*"?
- To learn more about undefined behavior (a.k.a. "nasal demons"), read Schrödinger's Code.

Because the computation is **nondeterministic**, debugging is no fun at all.

- Heisenbug: bugs that disappear or change behavior when you try to debug.

Mutual exclusion

To avoid race conditions, we need **mutual exclusion**:

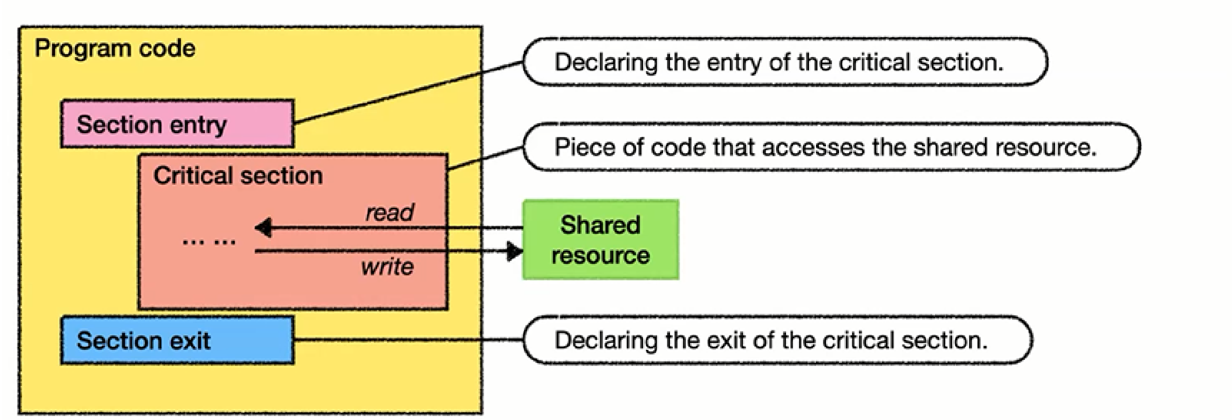
- If one process is accessing a shared resource,
- The other processes must be **excluded** from accessing the same thing.

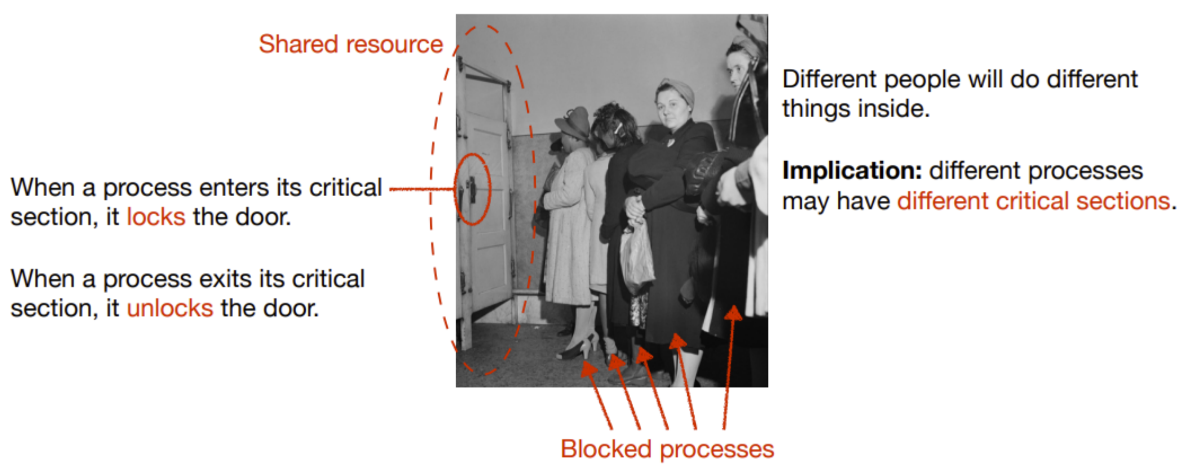
Note: race condition is a problem, and mutual exclusion is a requirement to avoid such a problem.

- However, mutual exclusion may hinder the performance of parallel computations.

Critical sections

A **critical section** (a.k.a. critical region) is a piece of code that accesses a shared resource.



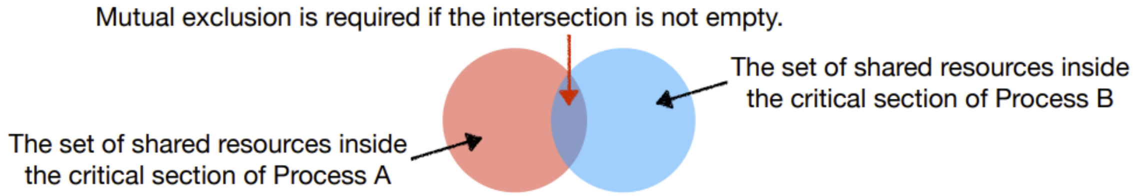


A critical section is a piece of code, not a shared resource.

A critical section should be as tight as possible.

- What would happen if you declare the entire program as a big critical section?

A critical section may access multiple shared resources.



- Requirements

No two processes may be simultaneously inside their critical sections.

- This is the mutual exclusion requirement: when one process is inside its critical section, any attempts to go inside the critical sections by other processes are not allowed.

No assumptions may be made about the speeds or the number of CPUs.

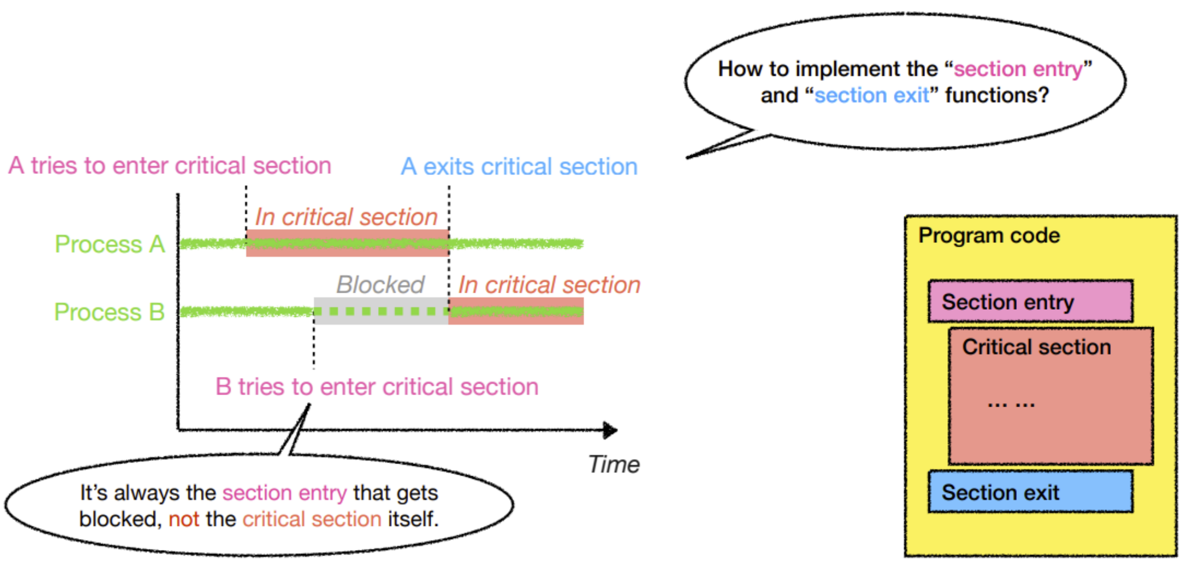
- The solution cannot depend on the time spent inside the critical section or assume the number of CPUs in the system.

No process running outside its critical section may block other processes.

- This ensures all processes can make progress. Otherwise, it may end up with a scenario where all processes are blocked but no process is inside its critical section.

No process should have to wait forever to enter its critical section.

- This guarantees bounded waiting, *i.e.*, no process will starve to death.



- Attempt #1: disabling interrupts

When interrupts are disabled, no context switch can occur.

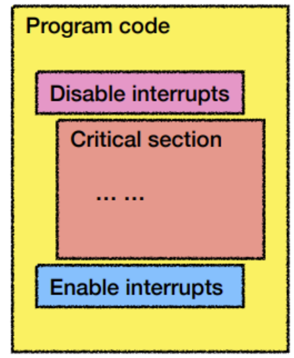
- When one process is in its critical section, the CPU will not be switched to another process.

Is it correct?

- Single-processor system: yes.
- Multiprocessor system: NO! Other CPUs can still access the shared resource.

Is it a good solution?

- Even on a single-processor system, it's a terrible idea to allow user processes to enable/disable interrupts at will.
- However, within the OS kernel itself, it's often convenient to disable interrupts for a few instructions.



- Attempt #2: using a "lock" variable

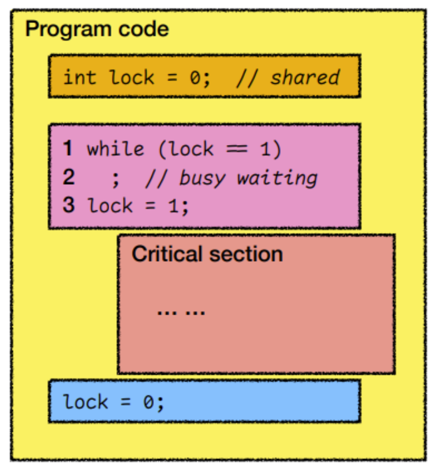
What about using a shared "`lock`" variable?

Is it correct?

- NO!
- It suffers from exactly the same race condition.

Is it a good solution (even if it worked)?

- Busy waiting (*a.k.a.* spinning) wastes CPU time.
- Priority inversion problem: a high-priority process may have to wait for a low-priority process to relinquish resource.



- Busy Waiting
  - The while loop keeps runing and running again. It's also called apinning
- Priority Inversion Problem
  - a high-priority process may have to wait for a low-priority process to relinquish resource.

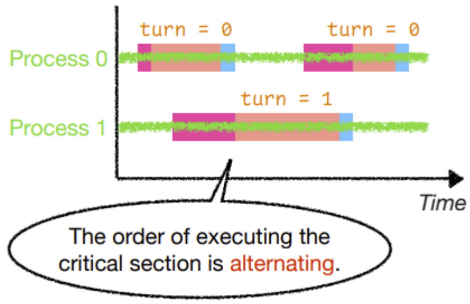
- Attempt #3: strict alternation

Use a shared variable to let processes take turns to enter the critical section.

Process 0

```
for (;;) {
  while (turn ≠ 0)
    ;  // busy waiting
  critical_section();
  turn = 1;
  noncritical_section();
}
```

```
int turn = 0;  // shared
```



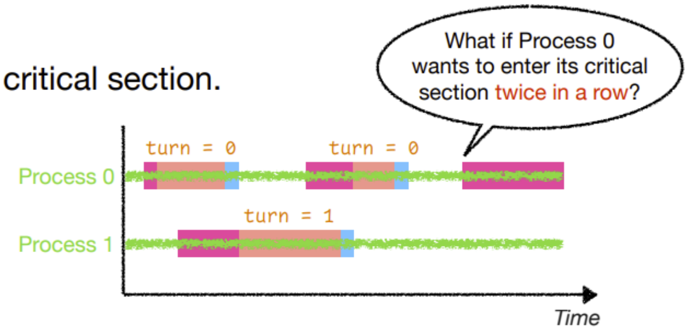
Process 1

```
for (;;) {
  while (turn ≠ 1)
    ;  // busy waiting
  critical_section();
  turn = 0;
  noncritical_section();
}
```

Processes take turns to enter the critical section.

Is it correct?

- It does avoid all races.
- However…



Is it a good solution?

- It violates Requirement #3 of being a good solution:

No process running outside its critical section may block other processes.

- Attempt #4: Peterson's algorithm

Critical sections

Attempt #4: Peterson's algorithm

Processes would act as a gentleman. *"If you want to enter, I'll let you first."*

This is an improved version of strict alternation.

```
int intent[2] = {0, 0};              // who wants to enter critical section?
int turn;                            // whose turn is it?
```

```
void section_entry(int process) {    // process is 0 or 1
  int other = 1 - process;           // the other process is 1 or 0
  intent[process] = true;            // I intend to enter critical section
  turn = other;                      // make it the other process's turn
  while (intent[other] && turn == other)   // I'm a gentleman!
    ;                                // busy waiting while it's not my turn
}
```

```
void section_exit(int process) {
  intent[process] = false;           // simply undo my intent
}
```

Process 0	Process 1
intent[0] = true turn = 1	
	intent[1] = true turn = 0
turn *is not* other	
In critical section	
	Busy waiting…
intent[0] = false	
	intent[0] *is* false
	In critical section
	intent[1] = false

- When the while loop runs, we can wait for the other process.
- When the while loop breaks, we can do our own work

Peterson's algorithm is a combination of using lock variables and taking turns.

Is it correct?

- Yes on early hardware, where `intent[]` and `turn` propagate immediately and atomically.
- Not anymore on modern hardware due to relaxed memory consistency models.

Is it a good solution?

- With a little hardware support, the solution could be much easier…

- Attempt #5: spinlocks

Modern CPUs all have instructions that guarantee **atomic operation**.

The simplest one is a test-and-set instruction.

Conceptually, it behaves as if this code snippet is executed without interruption:

```
int test_and_set(int *ptr, int new) {
  // this code executes atomically
  int old = *ptr;
  *ptr = new;
  return old;
}
```

- It returns the old value pointed to by `ptr` and simultaneously updates said value to `new`.

> “
>
> Atomic operation: The operation does a lot of things and can never be interrupted

```
int test_and_set(int *ptr, int new) {
  // this code executes atomically
  int old = *ptr;
  *ptr = new;
  return old;
}
```

How to implement `section_entry()` and `section_exit()` using the atomic `test_and_set` instruction?

```
int lock = 0;  // 0: available, 1: held
```

```
void section_entry(int *lock) {

  while (test_and_set(lock, 1) == 1)
    ;  // busy waiting

}
```

```
void section_exit(int *lock) {

  *lock = 0;

}
```

- We can use the `test_and_set()` to set the `lock` to 1 and check its old value.
  - If the value is already 1, it means someone else is in critical section, and we have to wait.
  - If it's 0, then we can enter the critical section

Spinlocks are built upon CPU instructions that guarantee atomic operation.

- **Example:** test-and-set, compare-and-swap, fetch-and-add, load-linked/store-conditional.

Is it correct?

- Yes on a preemptive scheduler. (Beware of the priority inversion problem!)

Is it a good solution?

- **Simplicity:** yes!
- **Fairness:** no guarantees. A process may spin forever under contention, leading to starvation.
- **Performance:**
  - Single-processor system: bad. With N processes contending for a lock, $^{N-1}/_{N}$ of CPU time is wasted.
  - Multiprocessor system: effective if [# of processes] ≈ [# of CPUs] and the critical section is short.

> If a scheduler is nonpreeptive, nothing can work

> Priority Inversion problem exists in this solution

Attempt #5.1: too much spinning?

```
int test_and_set(int *ptr, int new) {
  // this code executes atomically
  int old = *ptr;
  *ptr = new;
  return old;
}
```

Can you change one thing to make it less wasteful?

```
int lock = 0;  // 0: available, 1: held
```

```
void section_entry(int *lock) {

  while (test_and_set(lock, 1) == 1)
    sched_yield();  // give up the CPU

}
```

```
void section_exit(int *lock) {

  *lock = 0;

}
```

- When the lock is 1, instead of waiting forever, we can just give up the CPU to others

While better than spinning, this run-and-yield approach is still costly:

- Before the process holding the lock gets to run again, every other process still needs to run and yield.
- Context switching is expensive.

What if you can block instead of spinning?

It's time to introduce a new OS primitive.

- Attempt #6: semaphores

A **semaphore** is an object with a non-negative integer value.

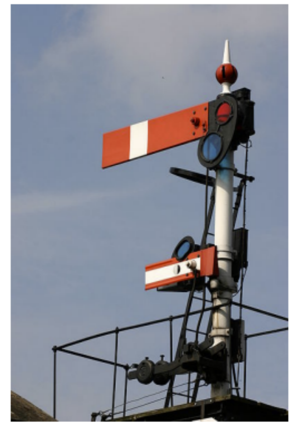
- The value must be initialized before being used.

There are two operations: `down()` and `up()`.

- See “`man 7 sem_overview`” for more details.

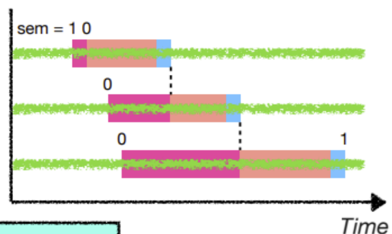
```
void down(semaphore *sem) {
  // atomic operation
  if (*sem > 0) {
    *sem = *sem - 1;
  } else {
    block on sem;
  }
}
```

```
void up(semaphore *sem) {
  // atomic operation
  if (some process is blocked on sem) {
    let one such process proceed
  } else {
    *sem = *sem + 1;
  }
}
```



```
void down(semaphore *sem) {
  // atomic operation
  if (*sem > 0) {
    *sem = *sem - 1;
  } else {
    block on sem;
  }
}
```

```
void up(semaphore *sem) {
  // atomic operation
  if (some process is blocked on sem) {
    let one such process proceed
  } else {
    *sem = *sem + 1;
  }
}
```



Program code

```
// init a binary semaphore
semaphore sem = 1;
```

```
down(&sem);
```

Critical section

… …

```
up(&sem);
```

> Binary semaphore: the value of it is either 0 or 1

- `up()`
  - If there're more than one process blocked, it can choose any of the process to proceed.
  - there's no specific order

We just used a binary semaphore to implement a **mutex** ("**mut**ual **ex**clusion").

Is it correct?

- Yes!

Is it a good solution?

- Yes!

Actually, semaphores are more powerful than guaranteeing mutual exclusion.

- Semaphores can be used to realize process synchronization, *i.e.*, to coordinate the set of processes so they can produce meaningful output.
- If the value of a semaphore is > 1, we call it a counting semaphore.

- Starvation?
    - Possible

Summary

- Today we introduced **6 different attempts for critical sections**
- The first three attempts didn't actually work
- **Peterson's algorithm** was a good attempt back in the days, but now it doesn't work anymore
- **Spinlocks** and **Semaphores** are the two approaches that are widely used today
    - **Spinlocks** are mainly used for short critical sections, as it uses busy waiting
    - **Semaphores** put the processes to the block state instead of busy waiting, so that it doesn't waste CPU power