CMPT 300
Operating System I
Synchronization Tools

Dr. Hazra Imran

Summer 2022

Mutex lock vs Spin lock

Criteria	Spinlock	Mutex
How it work?	• Test for the lock. • If available, enter the CS • If not, loop again and test the lock till you acquire the lock	• Test for the lock. • If available, enter the CS • If not, go to the wait queue • Once the current process is done, the next in the queue acquires the resource.

Mutex lock vs Spin lock

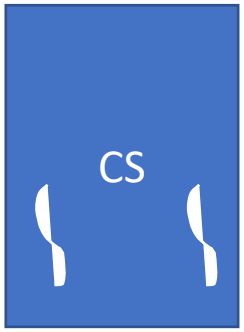
Criteria	Spinlock	Mutex
How it work?	• Test for the lock. • If available, enter the CS • If not, loop again and test the lock till you acquire the lock	• Test for the lock. • If available, enter the CS • If not, go to the wait queue • Once the current process is done, the next in the queue acquires the resource.
When to use?	• Used when the process should not be put to sleep like ISR (Interrupt service routines). • Spinlock causes the process to be uninterruptible. Thus, Spinlock is feasible only for a short wait.	• Used when putting process is not harmful like user space programs. • Use when there will be a considerable time before the process gets the lock.

Mutex lock vs Spin lock

Criteria	Spinlock	Mutex
How it work?	• Test for the lock. • If available, enter the CS • If not, loop again and test the lock till you acquire the lock	• Test for the lock. • If available, enter the CS • If not, go to the wait queue • Once the current process is done, the next in the queue acquires the resource.
When to use?	• Used when the process should not be put to sleep like ISR (Interrupt service routines). • Spinlock causes the process to be uninterruptible. Thus, Spinlock is feasible only for a short wait.	• Used when putting process is not harmful like user space programs. • Use when there will be a considerable time before the process gets the lock.
Drawbacks	• Processor is busy doing nothing till lock is acquired, wasting CPU cycles.	• Incur process context switch and scheduling cost.

Next step is

- T0 has lock held and gets interrupted.
- Timer interrupts goes off and T0 runs again… Releases lock.



- T1 tries to acquire the lock but finds that its held.
- It begins to spin…more spin
- T1 will be able to acquire lock now.

Spinning → waste of entire time slice doing NOTHING!!!

How can we develop a lock that doesn't needlessly waste time spinning on the CPU?

Semaphores (Sync tool + more powerful than lock)

- We looked at using locks to provide mutual exclusion.

- Locks work, but they have some drawbacks when critical sections are long
  - Spinlocks – inefficient (busy waiting)
  - Disabling interrupts – can miss or delay important events

- We want synchronization mechanisms that
  - Block waiters
  - Leave interrupts enabled inside the critical section

- There are two common high-level mechanisms
  - Semaphores: binary (mutex) and counting
  - Monitors: mutexes and condition variables

Semaphores

- A semaphore is a protected integer variable, which can only be manipulated by the atomic operations wait() and signal().

- **Terminology.** There are different terms used for describing the function calls that operate on semaphores:

P() (access a resource) V() (free a resource)	down() (access a resource) up() (free a resource)	semWait() (access a resource) semSignal() (free a resource)

- In general, the semaphore value is initialized to the number of available instances of a resource. For example, if there are two instances of a resource, we set semaphore=2.

- CS is the resource!!!

Semaphores

- The implementation of wait(s) and signal(s) must guarantee that:
  - If several processes simultaneously invoke wait(s) or signal(s), the operations will occur sequentially in some arbitrary order.

- If more than one process is waiting inside wait(s) for s to become > 0, one of the waiting processes is selected to complete the signal(s) operation. The selection can be at random, but a typical implementation maintains the blocked processes in a queue processed in FIFO order

Semaphores

```
semaphore s = 1;  /* shared globally */

/* each process or thread has code like this*/

while (true) {

  wait(s);

  /* critical section */

  signal(s);

  /* remainder */

}
```

- It is critical that the operations wait() and signal() execute atomically, however they are implemented.
- This is typically ensured by implementing semaphores in the OS kernel or by using assembly code.

Semaphores

Wait Operation

- The WAIT operation will wait until the specified semaphore has a value greater than 0, then it will decrement the semaphore value and return to the calling program.
- If the semaphore value is 0 when WAIT is called, conceptually execution is suspended until the semaphore value is non-zero.
- In a simple (inefficient) implementation, the WAIT routine loops, periodically testing the value of the semaphore, proceeding when its value is non-zero.

Signal Operation

- The SIGNAL operation increments the value of the specified semaphore.
- If there are many processes Waiting on that semaphore, exactly one of them may now proceed.

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

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

Semaphores

- When a process acquires a shared resource instance (e.g., enters its CS), the value of the semaphore is ***decremented*** (i.e., there is now one less resource instance available).

- When a process frees a resource (e.g., leaves its CS), the value of the semaphore is ***incremented*** (i.e., there is now one more resource instance available).

- A process is allowed to acquire a resource instance depending on the value of the semaphore:
  - If the semaphore value is ***greater than zero***, the process can acquire a resource. The value of the semaphore is then decremented as described above. *Note that this is done atomically with the acquisition.*

  - If the semaphore value is *zero*, there are no instances of the resource available (e.g., the CS is occupied.)

Clicker

s = 0 initially.

Process P1 executes the sequence: wait(s); wait(s); Signal(s);

Process P2 executes the sequence: Signal(s); Signal(s);

After both processes terminate, s = _____.

(A) -1 (B) 0 (C) 1

Clicker

s = 0 initially.

Process P1 executes the sequence: wait(s); wait(s); Signal(s);

Process P2 executes the sequence: Signal(s); Signal(s);

Which statement will terminate first?

(A) P1's first wait(s)

(B)P2's first signal (s)

Semaphores: Usage

Consider two concurrent processes: X and Y

Process X	Process Y
X1	Y1
X2	Y2
X3	Y3
X4	Y4
X5	Y5

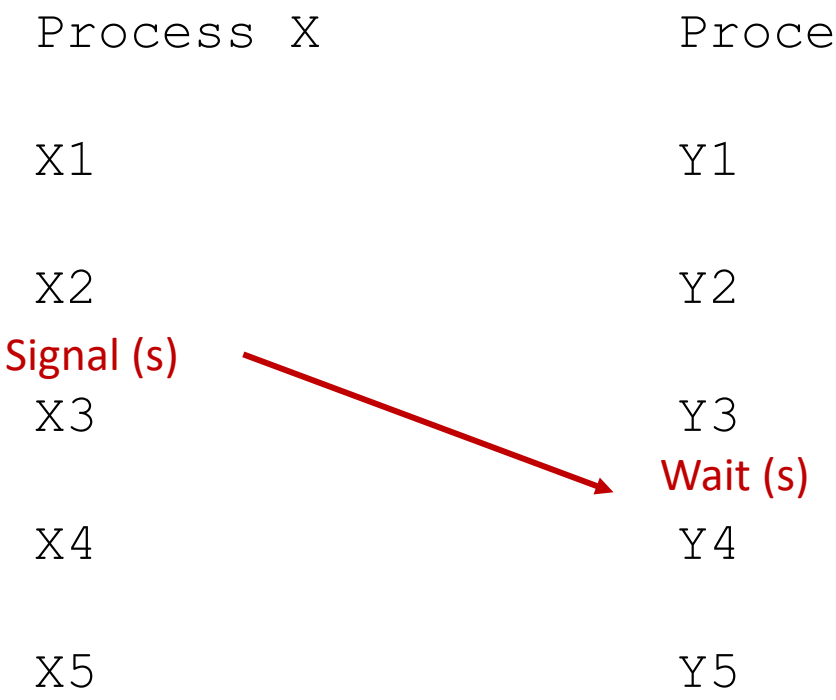
We want that

X2 in Process X should execute before Y4 in Process Y begins

Semaphores: Usage

Consider two concurrent processes: X and Y

Semaphore s =0



We want that

X2 in Process X should execute before Y4 in Process Y begins

How to do it?

- Declare semaphore s= 0
- Signal (s) at start of the arrow
- Wait(s) at end of the arrow

Semaphore Types

- **Counting semaphore:** integer value can range over an unrestricted domain
  - Can solve a wider range of synchronization problems

- **Binary semaphore:** integer value can range only between 0 and 1
  - Same as a mutex lock

Semaphore Details

- Implementations of **wait()** and **signal()** must guarantee that the same semaphore variable is not accessed by more than one process at the same time
    - wait and signal become CS (must be protected)

- With their use, we can still have the busy waiting problem. Just got shifted from application-level CS entry to semaphore's wait and signal commands, which are very short

- **Question:** So why not do the **busy waiting** in the user's critical section?
- Ans: User application may run for a long time, busy waiting can be costly

Using Semaphores: Mutual Exclusion

Semaphore lock; // initial semaphore value = 1

Process 1

```
wait (lock)
    Critical Section 1
signal (lock);
```

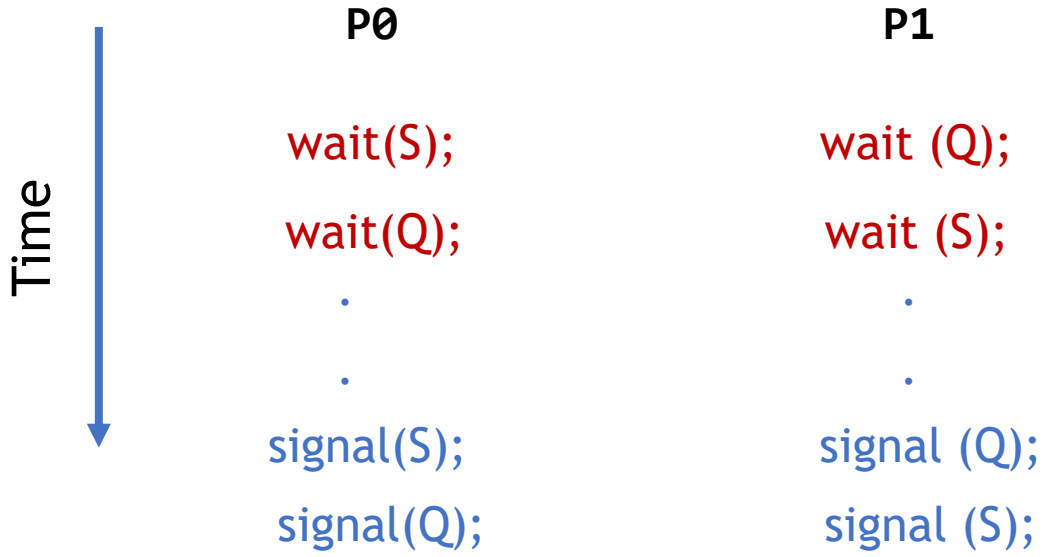
Process 2

```
wait (lock)
    Critical Section 2
signal (lock);
```

- Process 1 to execute wait () and lock is decremented to 0
- Process 2 waits until lock becomes > 0, which happens only when the first process executes signal
- Value of semaphore indicates number of waiting processes
  - lock = 0 means 1 process may be waiting
  - lock = 1 means no process is waiting

Semaphores

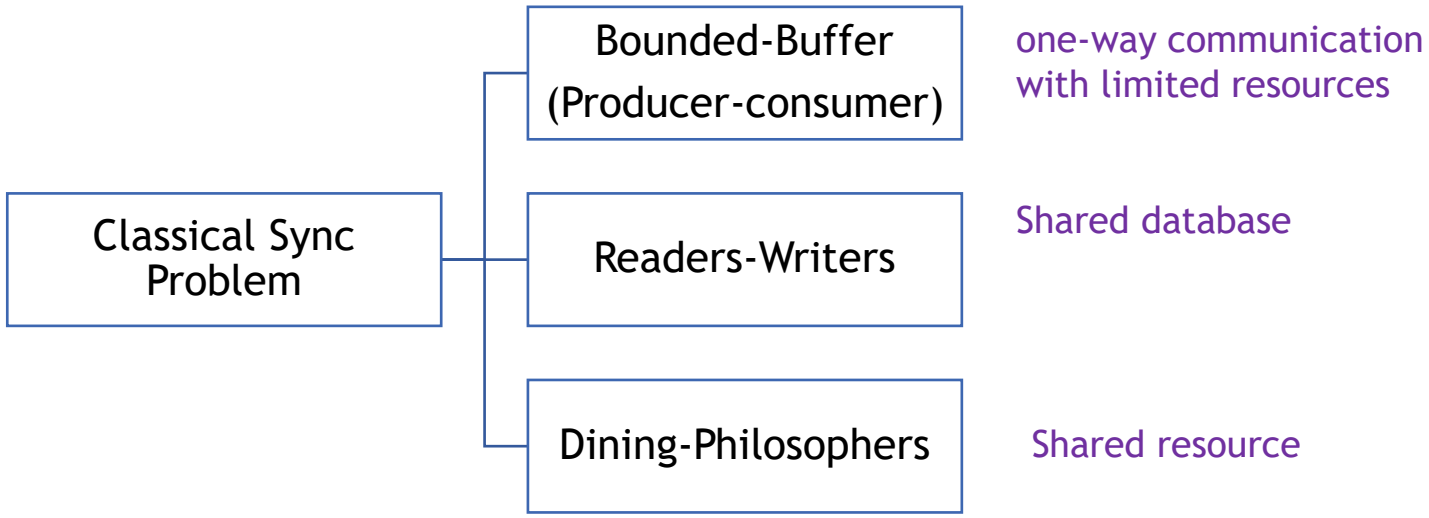
Let S and Q be two semaphores initialized to 1



Question: What could go wrong here?

Classic Synchronization Problems

These problems are abstractions that can be used to model many resource sharing problems



Bounded-Buffer Problem

- The "bounded buffer" producer-consumer problem is a classic problem.
- There is a buffer, usually a circular buffer in memory.
- There are one or more producers, placing items into the buffer
- There are one or more consumers, taking items out of the buffer

Three important considerations:

1. Producers and consumers can't both act on the buffer simultaneously. Only one process should be able to access the structure at a time.
2. A producer should not be able to place a new item into an already-full buffer
3. A consumer should not be able to retrieve an item from an empty buffer

Avoid busy waiting

Bounded-Buffer Problem

Data Structure:

- Semaphore **mutex** initialized to the value 1
  - Used to protect the buffer data structure from being accessed by more than one process
- Buffer of size ***n***
- Semaphore **full** initialized to the value 0
  - Counts how many items are in the buffer
  - Initially all slots are empty. Thus full slots are 0
- Semaphore **empty** initialized to the value n
  - Counts how many open spaces are in the buffer
  - Initially all slots are empty

Clicker

When the consumer pauses, the producer can fill all slots.

(A) Yes (B) No

Bounded-Buffer Problem

Semaphore mutex initialized to the value 1
Semaphore full initialized to the value 0
Semaphore empty initialized to the value n

Producer

```
do {
...
/* produce an item in
next_produced */
...
wait(empty);
wait(mutex);
...
/* add next produced to the
buffer */
...
signal(mutex);
signal(full);
} while (true);
```

Ensure there is empty slot

Protect access

Chance to other process

Increment #items: consumer's wait(full) can end

Consumer

```
do {
wait(full);
wait(mutex);
...
/* remove an item from buffer to
next_consumed */
...
signal(mutex);
signal(empty);
...
/* consume the item in next
consumed */
...
} while (true);
```

Ensure there is at least one item

Increment available slots: producer's wait(empty) can end

Bounded-Buffer Problem – Case 1

Producer

```
do {
...
/* produce an item in next_produced */
...
wait(empty);
wait(mutex);
...
/* add next produced to the buffer */
...
signal(mutex);
signal(full);
} while (true);
```

Consumer

```
do {
wait(full);
wait(mutex);
...
/* remove an item from buffer to next_consumed */
...
signal(mutex);
signal(empty);
...
/* consume the item in next consumed */
...
} while (true);
```

Bounded-Buffer Problem – Case 1

Producer

```
do {
...
/* produce an item in next_produced */
...
wait(empty);
wait(mutex);
CS /* add next produced to the buffer */

signal(mutex);
signal(full);
} while (true);
```

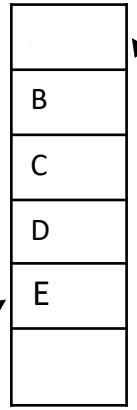
Consumer

```
do {
wait(full);
wait(mutex);
/* remove an item from buffer to next_consumed */
...
signal(mutex);
signal(empty);
...
/* consume the item in next consumed */
...
} while (true);
```

Empty = 2
Mutex = 1
Full = 4

1

Empty = ~~2~~ 1
Mutex = ~~1~~ 0
…Produce item (let say E)…
Mutex = ~~0~~ 1
Full = ~~4~~ 5



N=6

Check the total of empty and full… that's 6

Empty = 1
Mutex = 1
Full = 5

1

Full = ~~5~~ 4
Mutex = ~~1~~ 0
… enter in CS and remove the item (lets say A)
Mutex = ~~0~~ 1
Empty = ~~1~~ 2

Bounded-Buffer Problem – Case 2

Producer

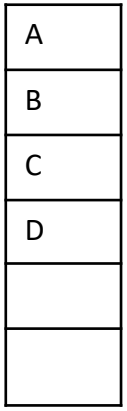
```
do {
...
/* produce an item in next_produced */
...
wait(empty);
wait(mutex);
CS /* add next produced to the buffer */

signal(mutex);
signal(full);
} while (true);
```

Consumer

```
do {
wait(full);
wait(mutex);
/* remove an item from buffer to next_consumed */
...
signal(mutex);
signal(empty);
...
/* consume the item in next consumed */
...
} while (true);
```

Empty = 2 Mutex = 1 Full = 4



N=6

Empty = ~~2~~ 1
PRODUCER PREEMPT
Consumer comes
Full = ~~4~~ 3
Mutex = ~~1~~ 0
Now Consumer is in the CS. It will consume the item.
Now, CONSUMER PREMPT
Ques Will Producer be able to perform wait(mutex) ?
Ans: No, as producer try to perform wait(mutex) then producer will get block.

Consumer will come again. Will perform signal(mutex). So now Mutex will be 1. At this time Producer can come in (if for some reason consumer preempt again).
Consumer will perform signal (empty). Now consumer finishes.
Now control will be back to producer. Producer's PCB has information that wait(empty) is already done. So now, Producer will start from wait(mutex).

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers: only read the data set; they do ***not*** perform any updates
  - Writers: can both read and write

- Problem:
  - Allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time

- Several variations of how readers and writers are considered … all involve some form of priorities

Readers-Writers Problem

Data: 100
One Reader is already in CS

Which case(s) will be problematic?

A. R –W

B. W- R

C. W-W

D. R-R

Readers-Writers Solution

Shared data:

- Data set
- Binary Semaphore **rw_mutex** initialized to 1
  - 1 = no readers/writers;
  - 0 = a writer or some number of readers
- Integer **read_count** initialized to 0
  - Number of processes actively reading the data set
- Binary Semaphore **mutex** initialized to 1
  - Protects read_count from being accessed/modified by more than one process