CSCI 4210 OPERATING SYSTEMS

Semaphores and Synchronization
David Goldschmidt (goldsd3@rpi.edu)
Summer 2023

SEMAPHORES

A *semaphore* is an operating system construct that serves as a mechanism for synchronizing access to one or more shared resources

We associate a semaphore with a shared resource

We define a semaphore as a non-negative integer variable for which:
- A value of zero indicates the shared resource is not available
- A value greater than zero indicates that that number of shared resources is available

Two atomic uninterruptable operations are:
- P() blocks until a resource is available, then acquires the resource
- V() relinquishes a resource

ATOMIC P() AND V() OPERATIONS

Each operation must execute without interleaving with any other operation on semaphore S

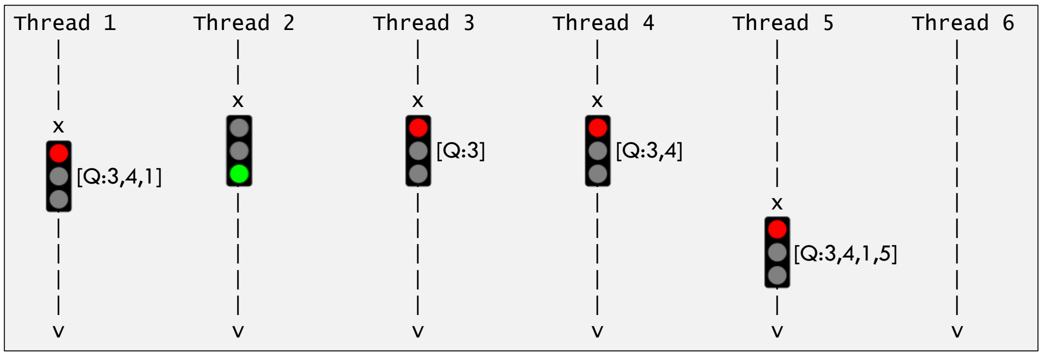
```
/* P() blocks until a resource is available, then acquires the resource */
P( semaphore S )
{
  while ( S == 0 ) { /* busy wait */ }
  S--;
}

/* V() relinquishes a resource */
V( semaphore S ) { S++ }
```

MUTUAL EXCLUSION

With *mutual exclusion*, only one thread may run its critical section of code at any given time

- Other threads that want to run their critical sections will be queued up and have to wait
- e.g., five of the threads below want to run their critical sections at point x in their execution:



BINARY SEMAPHORES

A *binary semaphore* is used to guarantee mutual exclusion

The only possible semaphore values are zero and one

Protect critical sections as follows:

```
/* initialize the semaphore to 1 since there is exactly one resource/lock */
semaphore mutex = 1;

/* each thread executes the code below:
...
P( mutex );
  execute_critical_section();
V( mutex );
```

COUNTING SEMAPHORES

A *counting semaphore* (or *general semaphore*) is used to control access to a finite number of (N) resources

Possible semaphore values are 0, 1, 2, ..., N

Initialize and use as follows:

```
/* initialize the semaphore to N, where N is the number of finite resources */
semaphore empty_array_slots = 20;

/* each thread executes the code below: */
...
P( empty_array_slots );
  ...
```

PRODUCER/CONSUMER PROBLEM

The shared resource is a buffer of finite size N

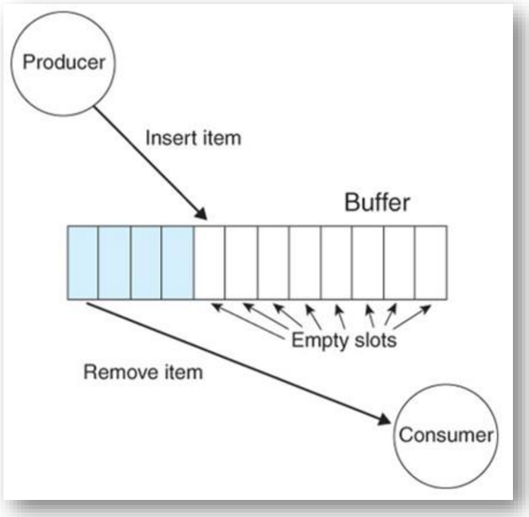
Producers add new items to the buffer

Consumers remove items from the buffer

How could synchronization problems occur?

Synchronization is required even with only one producer and one consumer

Scale this up to multiple consumers and multiple producers



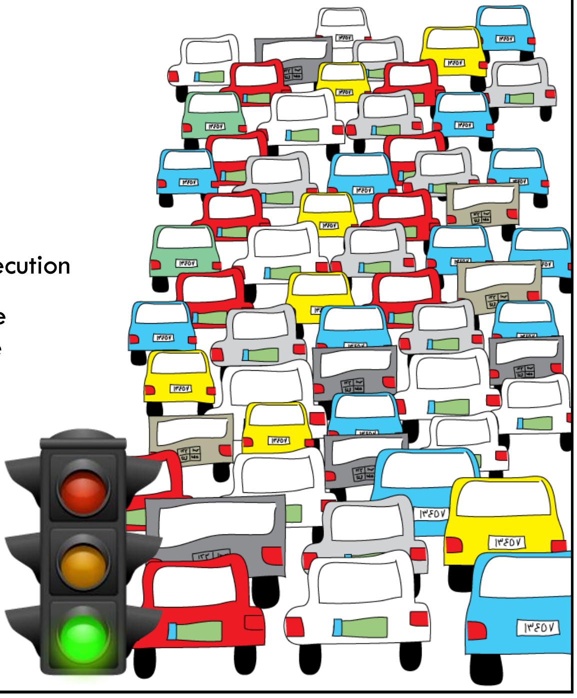
DEADLOCK

A group of threads is in a state of *deadlock* when no thread can make any further progress in its execution

All threads are blocked on a P() operation, but the requested resource(s) will never become available

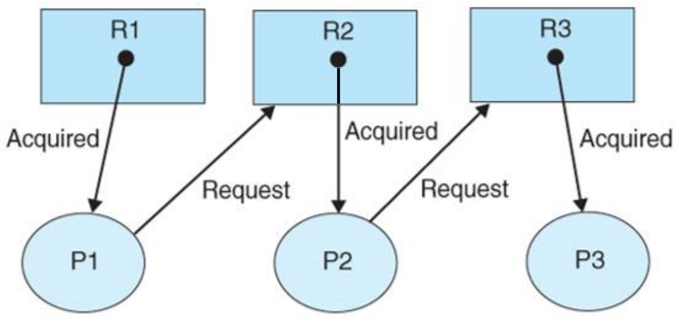
Deadlock requires four conditions:

- *Mutual exclusion*: threads require mutually exclusive access
- *Hold and wait*: after acquiring a resource via P(), a thread will hold on to that resource indefinitely (forever!)
- *No preemption*: once a thread has obtained a lock, we cannot preempt that lock
- *Circular wait*: a cycle exists in the corresponding resource allocation graph…



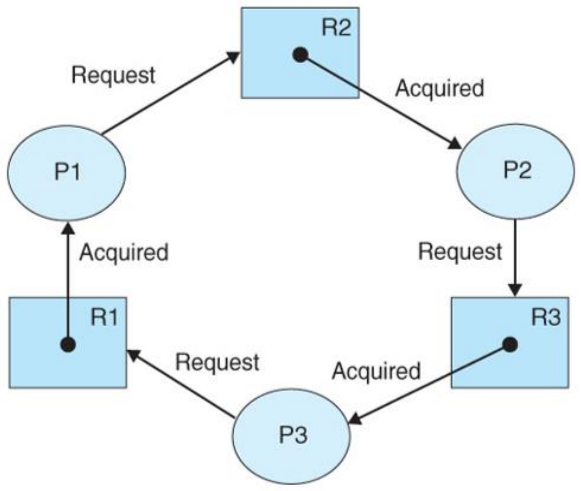
RESOURCE ALLOCATION GRAPHS

A *resource allocation graph* is a directed graph in which directed edges represent processes (e.g., P1, P2, P3) requesting and acquiring resources (e.g., R1, R2, R3)

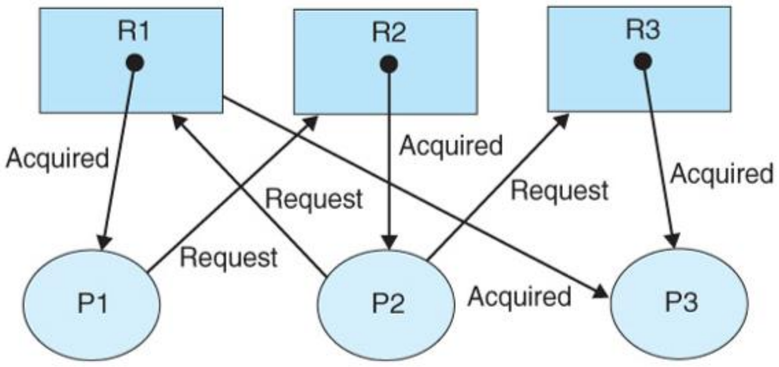


DEADLOCK

A cycle in our resource allocation graph indicates we have deadlock



DO WE HAVE DEADLOCK?


R1
R2
R3
Acquired
Acquired
Acquired
Request
Request
Request
Acquired
P1
P2
P3