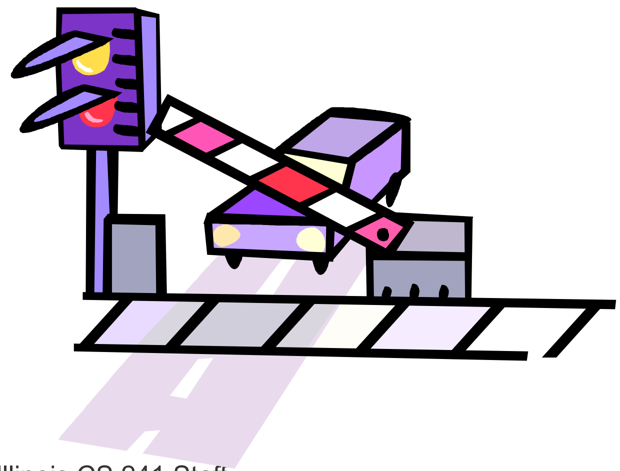
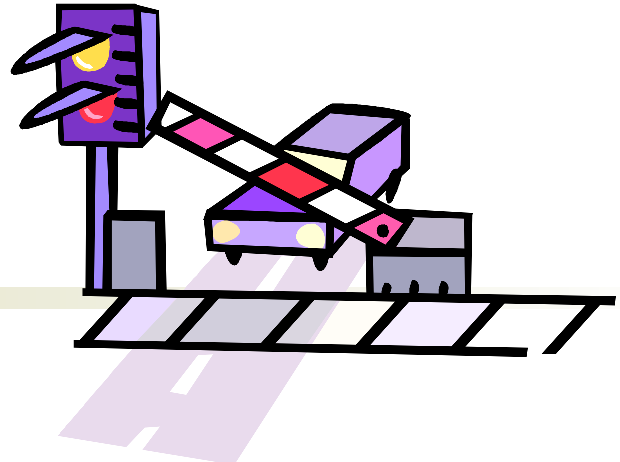
Achieving Synchronization



Overview



- Last lecture
  - Why do we need synchronization?
  - Solution: Critical Regions
- This lecture: achieving synchronization
  - Software-only synchronization
  - Hardware support: test-and-set
  - OS Support: sempahores

From last time...

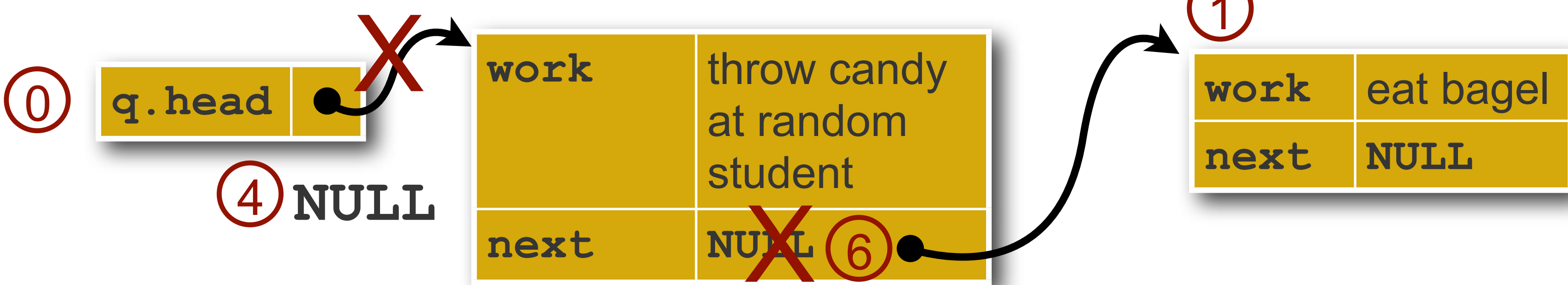
Things going Horribly Wrong

Producer thread:

```
while (true) {
① Create new work W;
② Find tail of q;
⑥ tail = W;
}
```

Consumer thread:

```
while (true) {
③ work = head of q;
④ remove head from q;
⑤ do_work(work);
}
```



I’ll never get to eat my bagel. :-(
(Could something worse happen?)

A simpler example

- We just saw that processes / threads can be preempted at arbitrary times.
  - The previous example might work, or not.
- What if we just use simple operations?

Shared state:	Thread 1:	Thread 2:
`int x=0;`	`x++;`	`x++;`

Are we safe now?

This could happen...

Thread 1	Thread 2	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
`x  = r1`		1		1
	`r2 = x`		1	1
	`r2 = r2+1`		2	1
	`x  = r2`		2	2

But this could happen too!

Thread 1	Thread 2	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
	`r2 = x`	1	0	0
	`r2 = r2+1`	1	1	0
`x  = r1`		1	1	1
	`x  = r2`	1	1	1

Race condition: results depend on timing!

Introducing: Critical Region (Critical Section)

```
Process {
   ...
   ENTER CRITICAL REGION
   Access shared variables;
   LEAVE CRITICAL REGION
   ...
}
```

Critical Region Requirements

- Mutual Exclusion
- Progress
- Bounded Wait

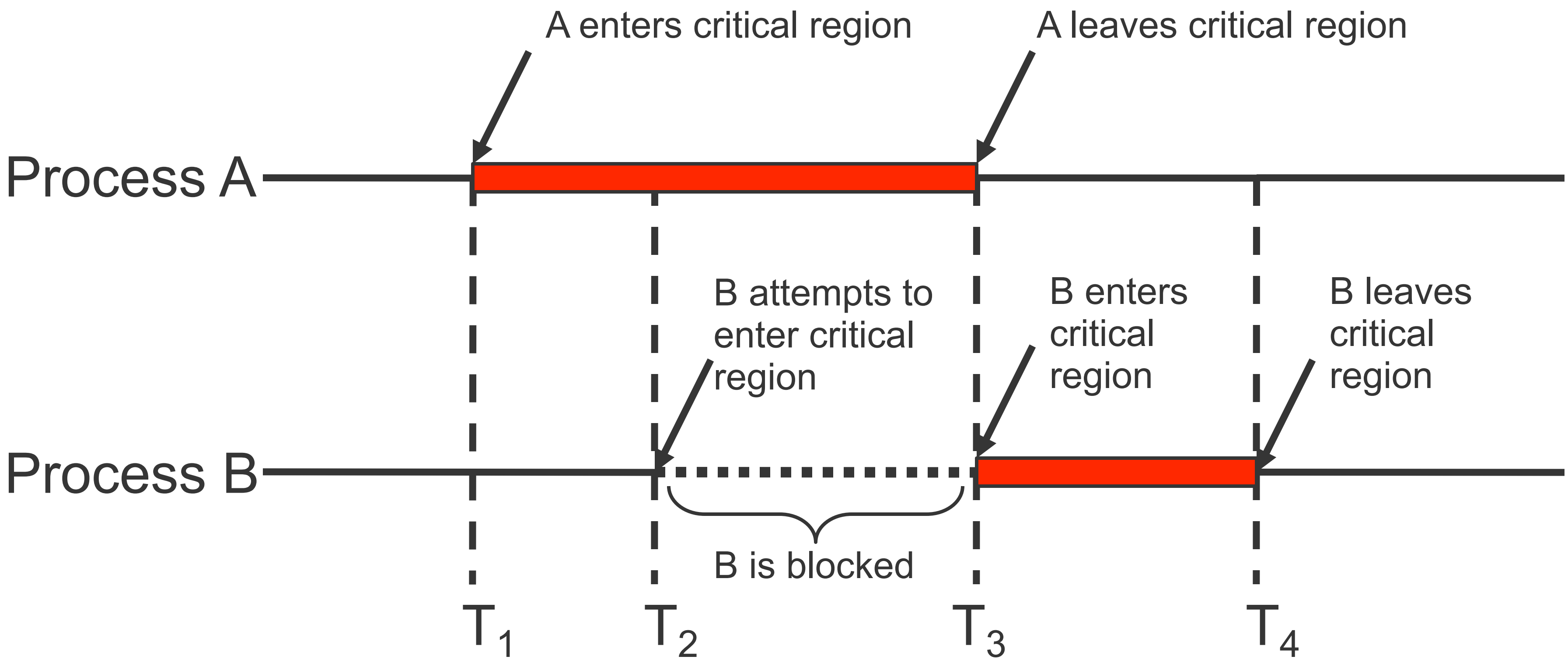

Bounded Wait
Mutual Exclusion
Progress


Bounded Wait
Mutual Exclusion
Progress
Can't wait forever!


Bounded Wait
Mutual Exclusion
Progress
Can't wait forever!
Are there door locks?


Bounded Wait
Mutual Exclusion
Progress
Can't wait forever!
Are there door locks?
Well, Did **you** see anybody go in?

Mutual exclusion using Critical Regions



How to implement a critical region

Mutual Exclusion solutions

- Software-only candidate solutions (Two-Process Solutions)
  - Lock Variables
  - Turn Mutual Exclusion
  - Other Flag Mutual Exclusion
  - Two Flag Mutual Exclusion
  - Two Flag and Turn Mutual Exclusion
- Hardware solutions
  - Disabling Interrupts; Test-and-set; Swap (Exchange)
- Semaphores

Lock Variables

```
...
while (lock) {
   /* spin spin spin spin */
}
lock = 1;
/* Entering critical section
access shared variable;
/* Leaving critical section
lock = 0;
...
```

Problem: Multiple processes could concurrently proceed past the while (lock) statement and violate mutual exclusion.

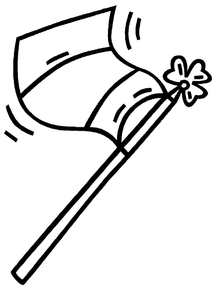
Turn-based Mutual Exclusion with Strict Alternation

```
…
while (turn != my_process_id) {
   /* wait your turn */
}
access shared variables;
turn = other_process_id;

…
```

Problem: If the other process is not interested in CS, this process cannot make progress.

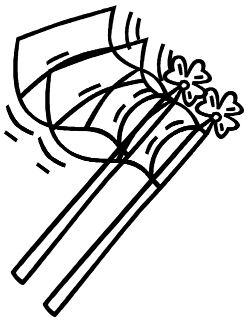
Other Flag Mutual Exclusion



```
int owner[2] = {false, false};
...
while (owner[other_process_id]) {
      /* wait your turn */
}
owner[my_process_id] = true;
access shared variables;
owner[my_process_id] = false;
...
```

Problem: No mutual exclusion – both processes can proceed past while() statement and into CS.

Two Flag Mutual Exclusion



```
int owner[2] = {false, false};
…
owner[my_process_id] = true;
while (owner[other_process_id]) {
    /* wait your turn */
}
access shared variables;
owner[my_process_id] = false;
…
```

Problem:
Could deadlock

Two Flag and Turn Mutual Exclusion

```
int owner[2]={false, false};
int turn;
…
owner[my_process_id] = true;
turn = other_process_id;
while (owner[other_process_id] and
       turn == other_process_id) {
    /* wait your turn */
}
access shared variables;
owner[my_process_id] = false;
…
```

Peterson’s Solution

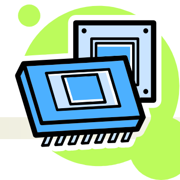
Discussion

- In uni-processors
  - Concurrent processes cannot be overlapped, only interleaved
  - A process runs until it invokes a system call, or is interrupted
  - To guarantee mutual exclusion, hardware support could help by allowing the disabling of interrupts

```
while(true) {
     /* disable interrupts */
     /* critical section */
     /* enable interrupts */
     /* remainder */
}
```

  - What’s the problem with this solution?

Discussion



- In multi-processors
  - Several processors share memory
  - Processors behave independently in a peer relationship
  - Interrupt disabling will not work
  - We need hardware support where access to a memory location excludes any other access to that same location
  - The hardware support is based on execution of multiple instructions atomically (test and set)

On to hardware-assisted solutions...

Test and Set Instruction

```
boolean Test_And_Set(boolean* lock) {
  atomic {
    boolean initial;
    initial = *lock;
    *lock = true;
    return initial;
  }
}
```

atomic = *executed in a single shot without any interruption*

Note: this is more accurate than the textbook version

Using Test_And_Set for Mutual Exclusion

```
Pi {
  while(1) {
    while(Test_And_Set(lock)) {
      /* busy-wait */
    }
    ... Critical Section ...
    lock = 0;
    ... Other work ...
  }
}
```

```
void main () {
    lock = 0;
    parbegin(P1,…,Pn);
}
```

Clean, simple, and works, but has performance loss because of **busy waiting**.

Semaphores

- Fundamental principle:
  - Two or more processes want to cooperate by means of simple signals
- Special variable type: **`semaphore`**
  - A special kind of “int” variable
  - Can’t just modify or set or increment or decrement it

Semaphores

- Before entering critical section
  - `semWait(s)`
    - receive signal via semaphore `s`
    - “down” on the semaphore
    - Executed
- After finishing critical section
  - `semSignal(s)`
    - transmit signal via semaphore `s`
    - “up” on the semaphore
- Implementation requirements
  - `semSignal` and `semWait` must be atomic

Semaphores

- Different notation can be used
  - **`semSignal`**
    - **`V`** - verhogen (“increment”)
    - **`signal`**
    - **`up`**
  - **`semWait`**
    - **`P`** - proberen (“test”)
    - **`wait`**
    - **`down`**

Semaphores vs. Test_and_Set

Semaphore

```
semaphore s = 1;
Pi {
  while(1) {
    semWait(s);
    ... Critical Section ...
    semSignal(s);
    ... other work...
  }
}
```

Test_and_Set

```
lock = 0;
Pi {
  while(1) {
    while(Test_And_Set(lock))
      { /* busy-wait */ }
    ... Critical Section ...
    lock = 0;
    ... Other work ...
  }
}
```

Inside a Semaphore

- Avoid busy waiting by suspending
  - Block if `s == False`
  - Wakeup on signal (`s == True`)
- Multiple process waiting on `s`
  - Keep a list of blocked processes
  - Wake up one of the blocked processes upon getting a signal

- Semaphore data structure

```
typedef struct {
    int count;
    queueType queue;
    /* queue for processes
       waiting on s */
} SEMAPHORE;
```

Inside a Semaphore

```
typedef struct {
    int count;
     queueType queue;
} SEMAPHORE;
```

semSignal and **semWait** must be atomic. (Q: how can we implement *that*?)

```
void semWait(semaphore s) {
  s.count--;
  if (s.count < 0) {
    place P in s.queue;
    block P;
  }
}
```

```
void semSignal(semaphore s) {
  s.count++;
  if (s.count ≤ 0) {
    remove P from s.queue;
    place P on ready list;
  }
}
```

Binary Semaphores

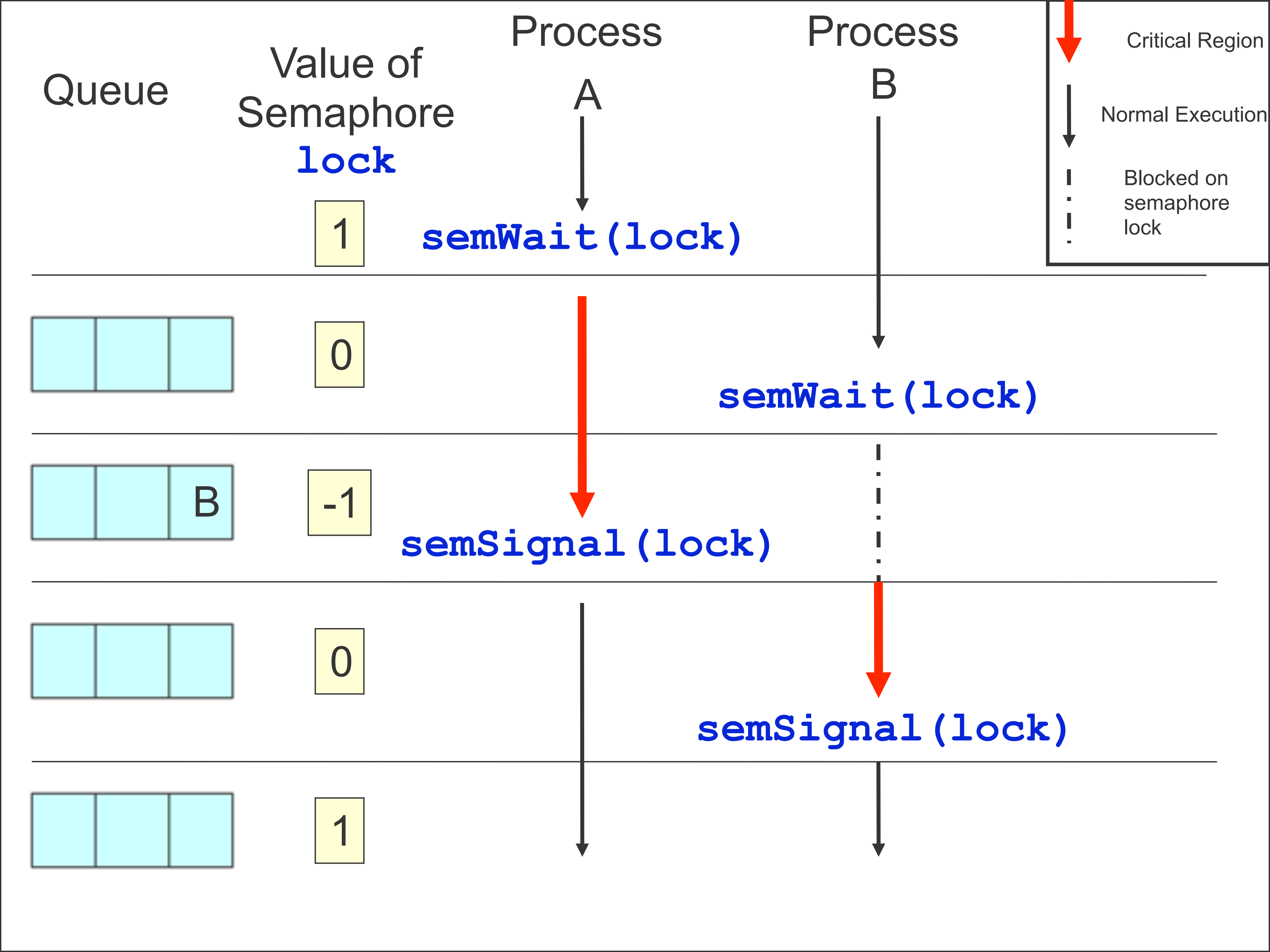
```
typedef struct bsemaphore {
   enum {0,1} value;
   queueType queue;
 } BSEMAPHORE;
```

```
void semWaitB(bsemaphore s) {
  if (s.value == 1)
    s.value = 0;
  else {
    place P in s.queue;
    block P;
  }
}
```

```
void semSignalB(bsemaphore s){
  if (s.queue is empty())
    s.value = 1;
  else {
    remove P from s.queue;
    place P on ready list;
  }
}
```

Mutual Exclusion Using Semaphores

```
semaphore s = 1;
Pi {
  while(1) {
    semWait(s);
    ... Critical Section ...
    semSignal(s);
    ... Other work ...
  }
}
```


Queue
Value of Semaphore lock
Process A
Process B
Critical Region
Normal Execution
Blocked on semaphore lock
1
semWait(lock)
0
semWait(lock)
B
-1
semSignal(lock)
0
semSignal(lock)
1

Summary

- Software-based mutual exclusion
  - Tricky
  - Busy-waiting
- Hardware solution: test-and-set
  - Simpler, cleaner, but still busy-waits
- Semaphores
- Next time: Using semaphores; other solutions