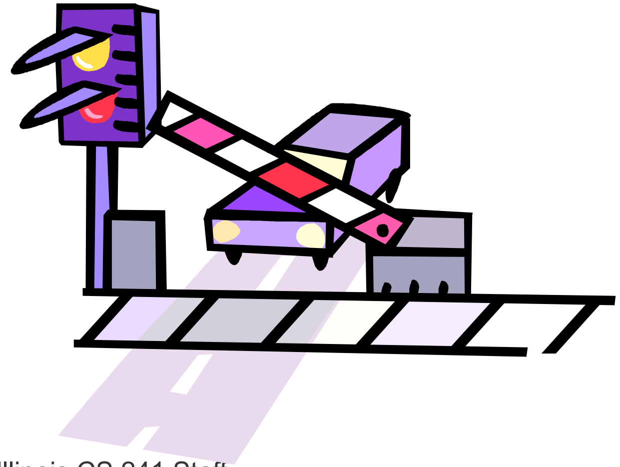
Semaphores



Recall: synchronization techniques so far

- Want mutual exclusion, progress, and bounded wait

Technique	Inconvenient?	Busy-waits?
Software-only (Peterson's)	Yes	Yes
Test-and-set	No	Yes
Semaphores	No	Essentially no

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

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;
```

Simplest case: 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;
  }
}
```

General case

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

`semSignal` and **`semWait`** must be atomic. So 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;
  }
}
```

Making the operations atomic

- Isn't this exactly what semaphores were trying to solve? Are we stuck?!
- Solution: resort to test-and-set

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

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

Making the operations atomic

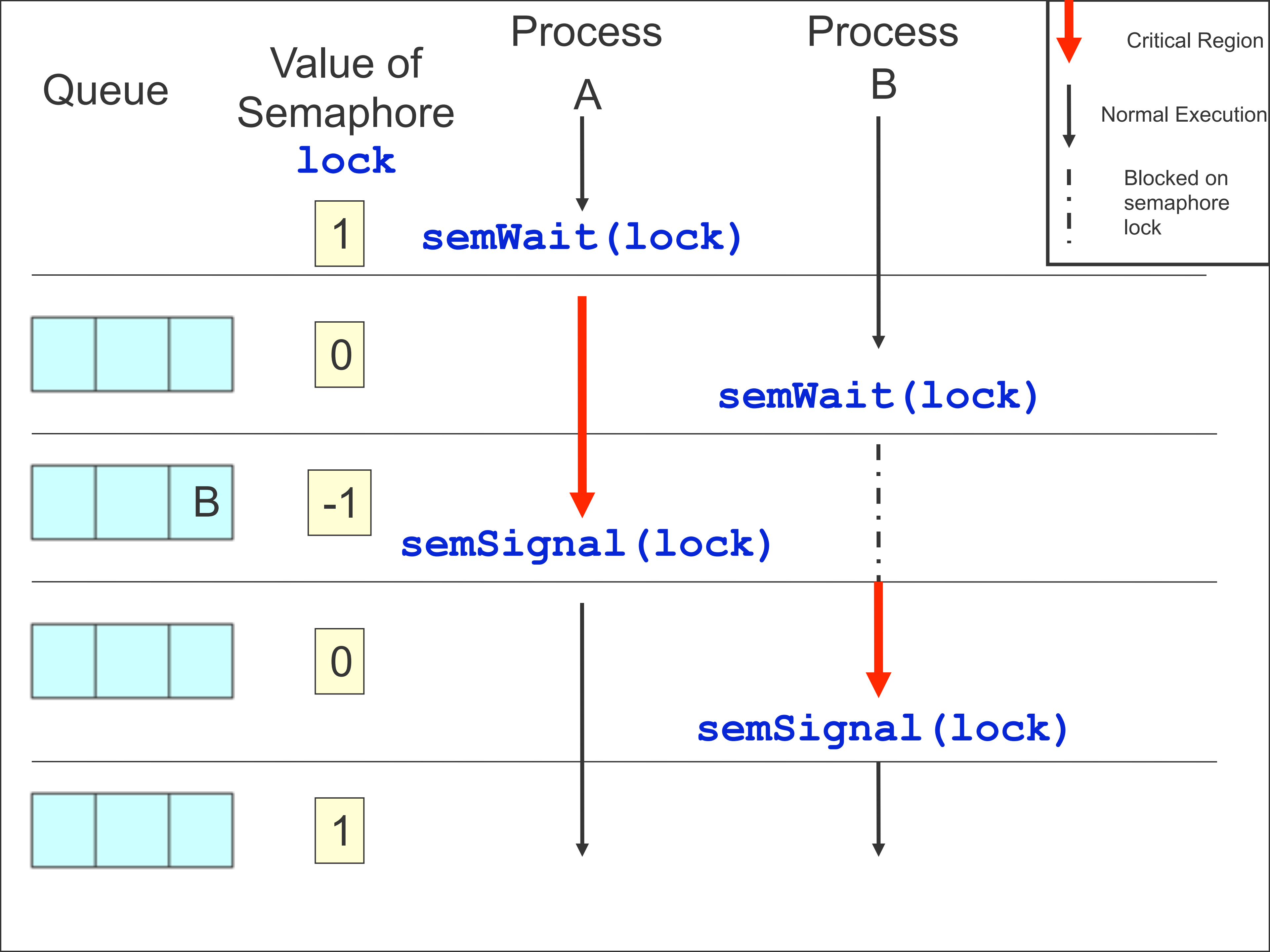
- Busy-waiting again!
- Then how are semaphores better than just using test_and_set?

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

- T&S: busy-wait during critical section
- Sem.: busy-wait just during semWait, semSignal: very short operations!

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

Semaphore Example 1

```
semaphore s = 2;
Pi {
  while(1) {
    semWait(s);
    /* Critical Sec. */
    semSignal(s);
    /* remainder */
    }
}
```

- What happens?
- When might this be desirable?

Semaphore Example 1

```
semaphore s = 2;
Pi {
   while(1) {
      semWait(s);
      /* Critical Sec. */
      semSignal(s);
      /* remainder */
   }
}
```

- What happens?
  - up to 2 processes can enter CS
- When might this be desirable?
  - allow up to 2 processes simultaneously in CS
  - e.g., limit number of processes reading a variable
  - Will see an example of why we might use s > 1 in a later lecture

Semaphore Example 2

```
semaphore s = 0;
Pi {
   while(1) {
      semWait(s);
      /* Critical Sec. */
      semSignal(s);
      /* remainder */
   }
}
```

- What happens?
- When might this be desirable?

Semaphore Example 2

```
semaphore s = 0;
Pi {
   while(1) {
      semWait(s);
      /* Critical Sec. */
      semSignal(s);
      /* remainder */
   }
}
```

- What happens?
  - No one can enter CS! Ever!
- When might this be desirable?
  - Never!

Semaphore Example 3

```
semaphore s = 0; /* shared */
P1 {
    /* do some stuff */
    semWait(s);
    /* do some more stuff */
}
```

```
P2 {
    /* do some stuff */
    semSignal(s);
    /* do some more stuff */
}
```

- What happens?
- When might this be desirable?

Semaphore Example 3

```
semaphore s = 0; /* shared */
P1 {
    /* do some stuff */
    semWait(s);
    /* do some more stuff */
}
P2 {
    /* do some stuff */
    semSignal(s);
    /* do some more stuff */
}
```

- What happens?
  - P1 waits until P2 signals
  - if P2 signals first, P1 does not wait
- When might this be desirable?
  - Having a process/thread wait for another process/thread

Be careful!

Mutual exclusion violation

```
semSignal(s);
critical_section();
semWait(s);
```

Certain deadlock!

```
semWait(s);
critical_section();
semWait(s);
```

Possible deadlock

```
semWait(s);
critical_section();
```

Mutual exclusion violation

```
critical_section();
semSignal(s);
```

Deadlock again!

```
semWait(s); semWait(s);
critical_section();
semSignal(s); semSignal(s);
```

POSIX Semaphores

- Named Semaphores
  - Provides synchronization between unrelated process and related process as well as between threads
  - Kernel persistence
  - System-wide and limited in number
  - Uses **`sem_open`**
- Unnamed Semaphores
  - Provides synchronization between threads and between related processes (shared memory)
  - Thread-shared or process-shared
  - Uses **`sem_init`**

POSIX Semaphores

- Data type
  - Semaphore is a variable of type `sem_t`
- Include `<semaphore.h>`
- Atomic Operations

```
int sem_init(sem_t *sem, int pshared,
   unsigned value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

Initialization

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned
    value);
```

- Initialize an unnamed semaphore
- Returns
  - 0 on success
  - -1 on failure, sets `errno`
- Parameters
  - `sem`: Target semaphore
  - `pshared`:
    - 0: only threads of the creating process can use the semaphore
    - Non-0: other processes can use the semaphore
  - `value`: Initial value of the semaphore

You cannot make a copy of a semaphore variable!!!

Sharing Semaphores

- Sharing semaphores between threads within a process is easy: use `pshared==0`
  - Not shared across processes
- A non-zero `pshared` allows any process that can access the semaphore to use it
  - e.g., processes with shared memory
  - Places the semaphore in the global (OS) environment

sem_init can fail

- On failure
  - sem_init returns -1 and sets errno

errno	cause
EINVAL	Value > sem_value_max
ENOSPC	Resources exhausted
EPERM	Insufficient privileges

```
sem_t semA;
if (sem_init(&semA, 0, 1) == -1)
  perror("Failed to initialize semaphore semA");
```

Semaphore Operations

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

- Destroy an semaphore
- Returns
  - 0 on success
  - -1 on failure, sets `errno`
- Parameters
  - `sem`: Target semaphore
- Notes
  - Can destroy a `sem_t` only once
  - Destroying a destroyed semaphore gives undefined results
  - Destroying a semaphore on which a thread is blocked gives undefined results

Semaphore Operations

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

- Unlock a semaphore
- Returns
  - 0 on success
  - -1 on failure, sets `errno` (`== EINVAL` if semaphore doesn't exist)
- Parameters
  - `sem`:
    - Target semaphore
    - sem > 0: no threads were blocked on this semaphore, the semaphore value is incremented
    - sem == 0: one blocked thread will be allowed to run
- Note: `sem_post()` is reentrant with respect to signals and may be invoked from a signal-catching function

Semaphore Operations

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
```

- Lock a semaphore
  - Blocks if semaphore value is zero
- Returns
  - 0 on success
  - -1 on failure, sets `errno` (`== EINTR` if interrupted by a signal)
- Parameters
  - `sem`:
    - Target semaphore
    - sem > 0: thread acquires lock
    - sem == 0: thread blocks

Semaphore Operations

```
#include <semaphore.h>
int sem_trywait(sem_t *sem);
```

- Test a semaphore's current condition
  - Does not block
- Returns
  - 0 on success
  - -1 on failure, sets `errno` (`== AGAIN` if semaphore already locked)
- Parameters
  - `sem`:
    - Target semaphore
    - sem > 0: thread acquires lock
    - sem == 0: thread returns

Example: bank balance

- Want shared variable ***balance*** to be protected by semaphore when used in:
  - **decshared** – a function that decrements the current value of *balance*
  - **incshared** – a function that increments the *balance* variable.

Example: bank balance

```
#include <errno.h>
#include <semaphore.h>

static int balance = 0;
static sem_t balance_sem;

int initshared(int val) {
   if (sem_init(&balance_sem, 0, 1) == -1)
      return -1;
   balance = val;
   return 0;
}
```

Example: bank balance

```
int decshared() {
   while (sem_wait(&balance_sem) == -1)
      if (errno != EINTR)
         return -1;
   balance--;
   return sem_post(&balance_sem);
}
int incshared() {
   while (sem_wait(&balance_sem) == -1)
      if (errno != EINTR)
         return -1;
   balance++;
   return sem_post(&balance_sem);
}
```

Summary

- Semaphores
- Semaphore implementation
- POSIX semaphores
- Programming with semaphores

- Next time: solving real problems with semaphores & more