CS 134:
Operating Systems
More Synchronization

CS 134:
Operating Systems
More Synchronization

Overview

Overview
More Synchronization
Monitors
Simpler Mechanisms

The Story So Far...

Mutual Exclusion

- Basic idea?

Semaphores

- Basic idea?

The Story So Far...
Mutual Exclusion
- Basic idea?
Semaphores
- Basic idea?

Fairness

Just how fair do we need to be. . . ?

Our Take. . .

Fairness

Just how fair do we need to be. . . ?

Our Take. . .

Fairness

Just how fair do we need to be. . . ?

Our Take. . .

No one likes semaphores!

- Too low-level
- Too much freedom (& too strange)
- Too hard to get right

Need an alternative. . .

Fairness

Just how fair do we need to be. . . ?

Our Take. . .

No one likes semaphores!

- Too low-level
- Too much freedom (& too strange)
- Too hard to get right

Need an alternative. . .

Monitors

Monitors were devised as an alternative to semaphores

- High-level synchronization construct, based on classes
- Only one task can be running “inside” the class at a time

Declare classes like this:

```
monitor class MyClass {
    public:
        /* method declarations only
    private:
        /* private data and private methods */
};
```


2013-05-19

CS34
└ More Synchronization
  └ Monitors
    └ Monitors

Monitors

Monitors were devised as an alternative to semaphores

- High-level synchronization construct, based on classes
- Only one task can be running “inside” the class at a time

Declare classes like this:

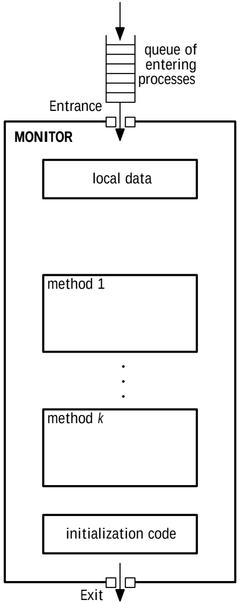
```
monitor class MyClass {
    public:
        /* method declarations only
    private:
        /* private data and private methods */
};
```

Monitors

Basic idea:

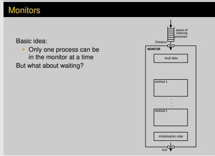
- Only one process can be in the monitor at a time

But what about waiting?



2013-05-19

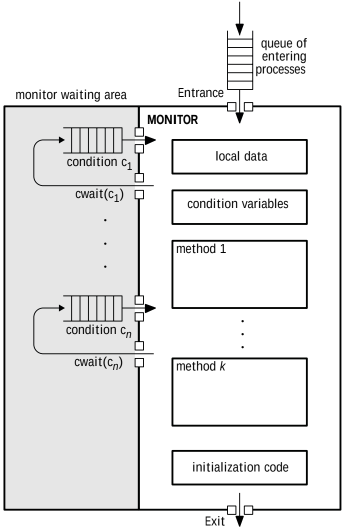
CS34
└More Synchronization
  └Monitors
    └Monitors



Monitors

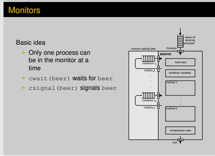
Basic idea

- Only one process can be in the monitor at a time
- `cwait(beer)` waits for `beer`
- `csignal(beer)` signals `beer`



2013-05-19

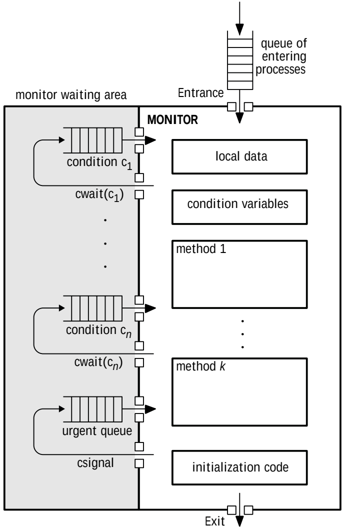
CS34
└More Synchronization
 └Monitors
  └Monitors



Monitors

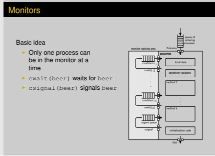
Basic idea

- Only one process can be in the monitor at a time
- `cwait(beer)` waits for `beer`
- `csignal(beer)` signals `beer`



2013-05-19

CS34
More Synchronization
Monitors
Monitors



Equivalence Claims

How could we show that

- Semaphores aren't "more powerful" than monitors?
- Monitors aren't "more powerful" than semaphores?

Equivalence Claims

How could we show that

- Semaphores aren't "more powerful" than monitors?
- Monitors aren't "more powerful" than semaphores?

Minimalism. . .

In NP-completeness, you learn `SAT`, and then the simpler `3−SAT`, which is equivalent.

Can we imagine something “less” than semaphores?

Minimalism. . .

In NP-completeness, you learn SAT, and then the simpler 3−SAT, which is equivalent.

Can we imagine something “less” than semaphores?

Binary Semaphores

Basic idea?

Binary Semaphores

Basic idea?

A binary semaphore is similar to test-and-set. If it's nonzero, one one process can set it to zero and continue past `bsem_dec`. If it's zero, `bsem_inc` sets it nonzero and wakes at least one process waiting on it. Multiple calls to `bsem_inc` with no intervening `bsem_dec` will have no effect. However, it is illegal to do that: you can't call `bsem_inc` unless the semaphore value is currently zero.

Semaphores from Binary Semaphores

Assume the following binary semaphore operations:

```
struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);
```

Data to implement semaphores...?

Semaphores from Binary Semaphores

Assume the following binary semaphore operations:

struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);

Data to implement semaphores...?

Semaphores from Binary Semaphores

Assume the following binary semaphore operations:

```
struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);
```

Data to implement semaphores...?

```
struct sem {
  volatile int count;    // Semaphore count

  struct bsem* wait;     // Wait here...
```


Semaphores from Binary Semaphores

Assume the following binary semaphore operations:

struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);

Data to implement semaphores...?

struct sem {
  volatile int count;   // Semaphore count

  struct bsem* wait;   // Wait here...

Semaphores from Binary Semaphores

Assume the following binary semaphore operations:

```
struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);
```

Data to implement semaphores...?

```
struct sem {
  volatile int count;    // Semaphore count

  struct bsem* wait;     // Wait here...
  struct bsem* mutex;    // Protects count
```


Semaphores from Binary Semaphores

Assume the following binary semaphore operations:

struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);

Data to implement semaphores...?

struct sem {
volatile int count; // Semaphore count

struct bsem* wait; // Wait here...
struct bsem* mutex; // Protects count

Semaphores from Binary Semaphores

Assume the following binary semaphore operations:

```
struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);
```

Data to implement semaphores...?

```
struct sem {
  volatile int count;    // Semaphore count

  struct bsem* wait;     // Wait here...
  struct bsem* mutex;    // Protects count
  volatile int waiting; // How many waiting
```


Semaphores from Binary Semaphores

Assume the following binary semaphore operations:

struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);

Data to implement semaphores...?

struct sem {
  volatile int count;   // Semaphore count

  struct bsem* wait;   // Wait here...
  struct bsem* mutex;   // Protects count
  volatile int waiting; // How many waiting

Semaphores from Binary Semaphores

Assume the following binary semaphore operations:

```
struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);
```

Data to implement semaphores...?

```
struct sem {
  volatile int count;    // Semaphore count
    // +val = sem count, -val = wait count
  struct bsem* wait;     // Wait here...
  struct bsem* mutex;    // Protects count
```


Semaphores from Binary Semaphores
Assume the following binary semaphore operations:
struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);
Data to implement semaphores...?
struct sem {
volatile int count; // Semaphore count
// +val = sem count, -val = wait count
struct bsem* wait; // Wait here...
struct bsem* mutex; // Protects count

Semaphores from Binary Semaphores

Assume the following binary semaphore operations:

```
struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);
```

Data to implement semaphores...?

```
struct sem {
  volatile int count;   // Semaphore count
    // +val = sem count, -val = wait count
  struct bsem* wait;    // Wait here...
  struct bsem* mutex;   // Protects count

};
```


Semaphores from Binary Semaphores

Assume the following binary semaphore operations:

struct bsem* bsem_create (int count);
void bsem_dec (struct bsem* s);
void bsem_inc (struct bsem* s);

Data to implement semaphores...?

struct sem {
  volatile int count;   // Semaphore count
    // +val = sem count, -val = wait count
  struct bsem* wait;   // Wait here...
  struct bsem* mutex;   // Protects count

};

Semaphores from Binary Semaphores (cont.)

Initialization:

```
struct sem* sem_init(int count)
{
  struct sem* s = malloc(sizeof(struct sem));
  assert(s != NULL && count >= 0);
  s->count = count;
  s->mutex = bsem_create(1); // Ordinary mutex
  s->wait = bsem_create(0);  // Mostly locked,
                             // briefly unlocked
  return s;
}
```


Semaphores from Binary Semaphores (cont.)
Initialization:
struct sem* sem_init(int count)
{
  struct sem* s = malloc(sizeof(struct sem));
  assert(s != NULL && count >= 0);
  s->count = count;
  s->mutex = bsem_create(1); // Ordinary mutex
  s->wait = bsem_create(0); // Mostly locked,
                             // briefly unlocked
  return s;
}

Semaphores from Binary Semaphores (cont.)

Is this code okay?

```
void sem_dec(struct sem* s)
{
  bsem_dec(s->mutex);
  --(s->count);
  if (s->count < 0) {
    bsem_inc(s->mutex);
    bsem_dec(s->wait);
  }
  else {
    bsem_inc(s->mutex);
  }
}
```

```
void sem_inc(struct sem* s)
{
  bsem_dec(s->mutex);
  ++(s->count);
  if (s->count <= 0) {
    bsem_inc(s->wait);

  }

  bsem_inc(s->mutex);

}
```


There is a race after `sem_dec` calls `bsem_inc` on the mutex; we could sleep on `s->wait` even though `s->count` has become nonzero. We need to ensure that there is exactly one `bsem_inc` per wait. For example:

1. Process 1 decs and stops after mutex release
2. Process 2 decs and stops after mutex release
3. Process 3 incs and bumps wait
4. Process 4 incs and re-bumps wait (illegally)
5. Process 1 continues and passes through wait
6. Process 2 continues and waits forever

Semaphores from Binary Semaphores (cont.)

Does this version fix the problem?

```
void sem_dec(struct sem* s)
{
  bsem_dec(s->mutex);
  --(s->count);
  if (s->count < 0) {
    bsem_inc(s->mutex);
    bsem_dec(s->wait);
  }
  bsem_inc(s->mutex);
}
```

```
void sem_inc(struct sem* s)
{
  bsem_dec(s->mutex);
  ++(s->count);
  if (s->count <= 0) {
    bsem_inc(s->wait);
  }
  else {
     bsem_inc(s->mutex);
  }
}
```


Semaphores from Binary Semaphores (cont.)

Does this version fix the problem?

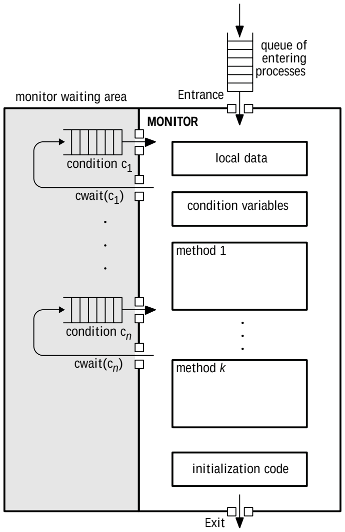
The assumption here is that if `sem_dec` waits, `sem_inc` would grab the mutex on its behalf and bump `wait`. So even if somebody else gets in between the release of the mutex and the wait, they will necessarily allow us to pass through. In our previous scenario:

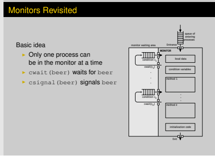
1. Process 1 decs and stops after mutex release
2. Process 2 decs and stops after mutex release
3. Process 3 incs and bumps wait
4. Because the mutex is still held, process 4 can't proceed. Instead, one of process 1 & 2 will continue and pass through the wait.
5. Process 1 continues and releases mutex.
6. Process 4 incs and bumps wait
7. Process 2 can now continue and pass through wait.

Monitors Revisited

Basic idea

- Only one process can be in the monitor at a time
- `cwait(beer)` waits for `beer`
- `csignal(beer)` signals `beer`



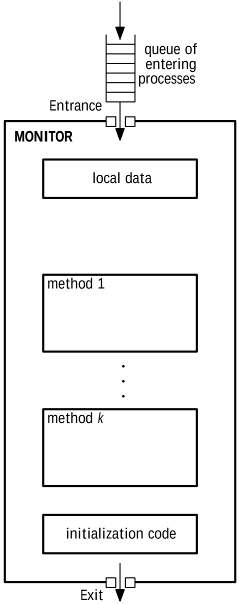


Monitors without Condition Variables

Basic idea

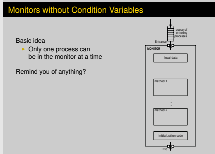
- Only one process can be in the monitor at a time

Remind you of anything?



2013-05-19

CS34
More Synchronization
Simpler Mechanisms
Monitors without Condition Variables



Mutexes / Locks

Like binary semaphores, but with *ownership* rules:

- You “acquire” the lock
- You “hold” the lock
- You “release” the lock

Someone else can’t release it for you.

Mutexes / Locks

Like binary semaphores, but with *ownership* rules:
- You “acquire” the lock
- You “hold” the lock
- You “release” the lock

Someone else can’t release it for you.

Mutexes / Locks

```
void task(const int i)
{
  for ( ; ; ) {
    lock_acquire(ourlock);
    critical_section_actions(i);
    lock_release(ourlock);
    other_actions(i);
  }
}
```

Class Exercise

Is `bool lock_tryacquire(lock)` useful?

2013-05-19

CS34
└More Synchronization
  └Simpler Mechanisms
    └Mutexes / Locks

Mutexes / Locks

void task(const int i)
{
  for (; ;) {
    lock_acquire(ourlock);
    critical_section_actions(i);
    lock_release(ourlock);
    other_actions(i);
  }
}

Class Exercise

Is bool lock_tryacquire(lock) useful?

Class Exercise

Can you implement semaphores using mutexes?

Can you implement mutexes using semaphores?

What do mutexes remind you of?

Class Exercise
Can you implement semaphores using mutexes?
Can you implement mutexes using semaphores?
What do mutexes remind you of?

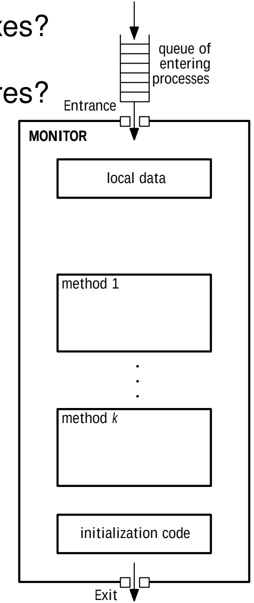
Class Exercise

Can you implement semaphores using mutexes?

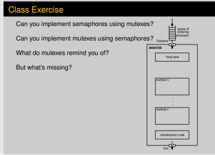
Can you implement mutexes using semaphores?

What do mutexes remind you of?

But what's missing?



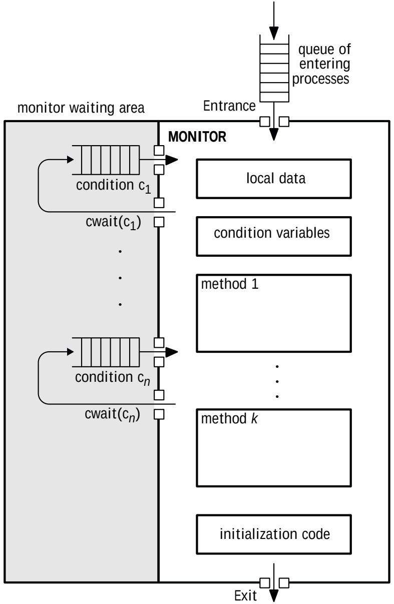
2013-05-19



Condition Variables (for Mutexes)

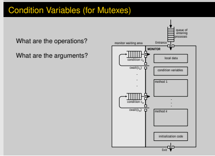
What are the operations?

What are the arguments?



2013-05-19

CS34
More Synchronization
Simpler Mechanisms
Condition Variables (for Mutexes)

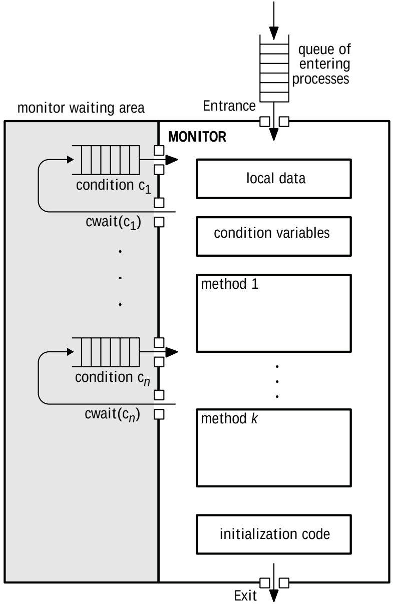


Condition Variables (for Mutexes)

What are the operations?

What are the arguments?

Do you need to hold the lock when you `cond_signal` or `cond_broadcast`?



2013-05-19
CS34
More Synchronization
Simpler Mechanisms
Condition Variables (for Mutexes)