Operating Systems!

Concurrency: **Synchronization**

Prof. Travis Peters

Montana State University

CS 460 - Operating Systems

Fall 2020

https://www.cs.montana.edu/cs460

Some diagrams and notes used in this slide deck have been adapted from Sean Smith's OS courses @ Dartmouth. Thanks, Sean!

Today

- Announcements
  - Yalnix! Form teams *ASAP!* ☺

    Travis has already helped a few "Free Agents" form team members! Great to have a group to discuss things (e.g., Accountability, PA2, Studying, Yalnix CP1)

  - *Exam 1:* ***Friday [10/02/2020] – due @ 11:59 PM (MST)***
    - **Read over the exam coversheet BEFORE Friday!**
  - *PA2 due* ***Sunday [10/04/2020] @ 11:59 PM (MST)***
    - **USE the PA2 starter files!**
  - *Office Hours*
    - **Extra Office Hours Tuesday (8-11am) + Thursday (8-11am)**
    - **NO OFFICE HOURS FRIDAY**

Today *(cont.)*

- *Last Time: Interleaving, Race Conditions, Naive Synchronization, …*
- Agenda
  - OS Synchronization
  - ***Locks***
  - ***Condition Variables***
  - Semaphores
  - Classic Concurrency Scenarios
  - Examples
  - Strategies
  - Real-World Gotchas

OS Synchronization: *Can We Do Better?*

TOCTOU

- **"Spin Lock"**
  - Want "mutual exclusion"
  - use TestAndSet instruction for atomic test/update
  - spin in loop (doing *nothing*) until the lock is free

```
while( TestandSet(flag) ) {}
/* critical section */
flag = false;
```

- **Any Problems?** *(Why might this be bad for "efficiency", "convenience", etc.?)*

Mutual Exclusion & Synchronization

- **Software Approaches**
  - Assume mutual exclusion at the memory access level
  - Decker's Algorithm, Peterson's Algorithm

- **Hardware Support**
  - Disabling Interrupts
  - Special Instructions (Compare&Swap, Exchange)

- **Programming Language Mechanisms**
  - *Mechanisms*: Semaphores, Monitors, Message Passing, Condition Variables, Mutexes (Locks), *…oh my!*
  - *Famous problems: Producer/Consumer, Dining Philosophers*

Mutexes (“Locks”)

A simple mechanism for **mut**ual **ex**clusion!

… watch out for variations in semantics (re: “Binary Semaphores”)

Locks: *What Are They?*

Typically used for mutual exclusion

A lock has two states:

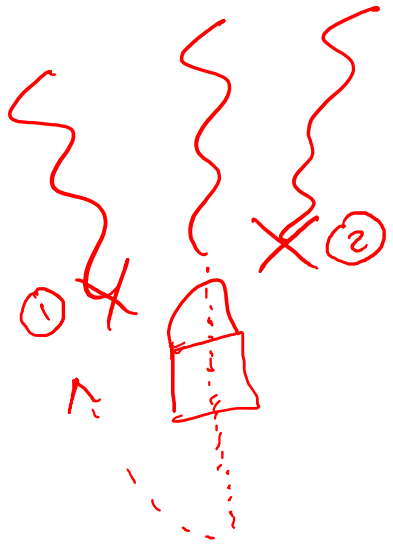
- **locked** – AKA "busy" (with exactly one "owner")
- or **unlocked** - AKA "free"

A lock has two (main) operations:

- **LockInit()** – sets up the lock for use
- **Acquire()** – block the caller until the caller gets the lock
- **Release()** – let the lock go (but the caller must already have the lock).
  ➔ *If someone was blocked trying to Acquire the lock, then let them have it!*

Programming w/ Locks

A Typical Sequence



1. **Create** and **initialize a mutex variable**

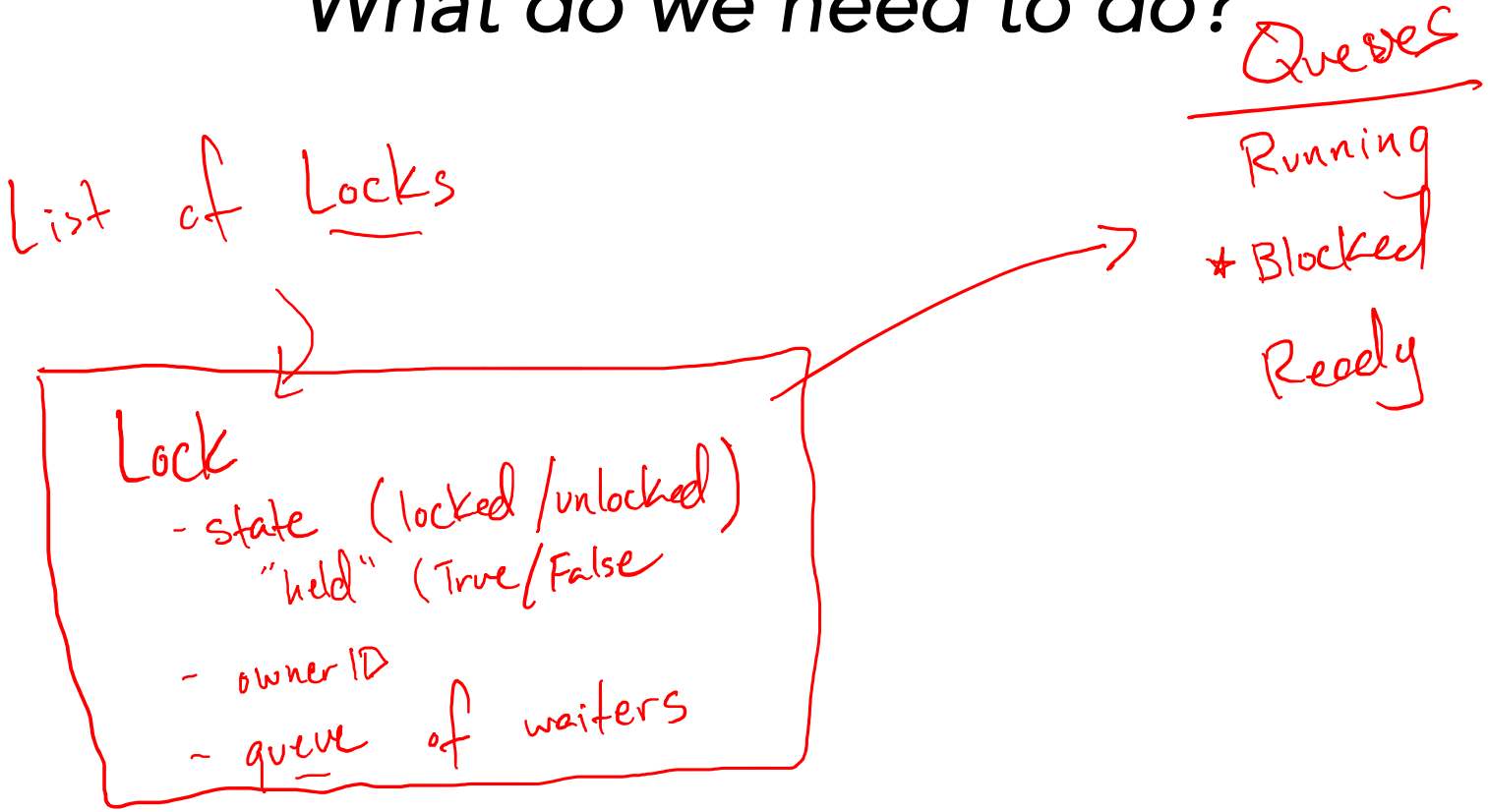
2. **Create** and start **several threads**
   - Several threads attempt to lock (**Acquire**) the mutex *(only 1 succeeds and "owns" the lock)*
   - The lock "owner" performs some set of actions that require **mutual exclusion**
   - The "owner" unlocks (**Release**) the mutex
   - The next thread waiting on the lock is awoken and repeats the process

3. **Wait for all threads** to finish their work and do any **cleanup** such as **(destroy) the mutex**

Implementing Locks

Suppose we want to implement the lock synchronization primitives as syscalls.

What do we need to do?

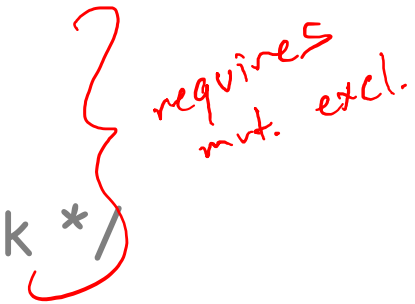


Using Locks: Initialization

```
// In pthreads, a lock is a pthread_mutex_t.

// You can initialize one statically:
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Or dynamically:
rc = pthread_mutex_init(&mutex, NULL);
```

Using Locks: Acquire & Release

```
// Precondition: Suppose you have some mutex that has been initialized...
rc = pthread_mutex_lock(&mutex);
if (rc) {
    printf("hey, it failed!\n");
    exit(-1);
}

/* after you've used the lock */

rc = pthread_mutex_unlock(&mutex);
if (rc) {
    printf("hey, it failed!\n");
    exit(-1);
}
```



When to Use Locks?

When you need mutual exclusion!

- we establish a lock for a set of conflicting critical sections
- we ensure that each *actor* **acquires** the lock before proceeding into their section
- we ensure that each *actor* **releases** the lock afterwards

Demo: sync_lock.c

Locks In The Real World...

You only need to know pure locks in this class... but it is good to know that alternatives often exist!

What should happen if caller that already holds the lock tries to acquire it again?

Other edge cases...

- *What should happen if a caller tries to **release a lock they don't own?***
- *What should happen if a caller tries to **release a lock that isn't locked?***
- *Could we implement "a **non-blocking acquire?**"*
  (get the lock if it's unlocked, but return – with some kind of info/error – if it's not?)

Condition Variables

Simple synchronization based on a logical condition!

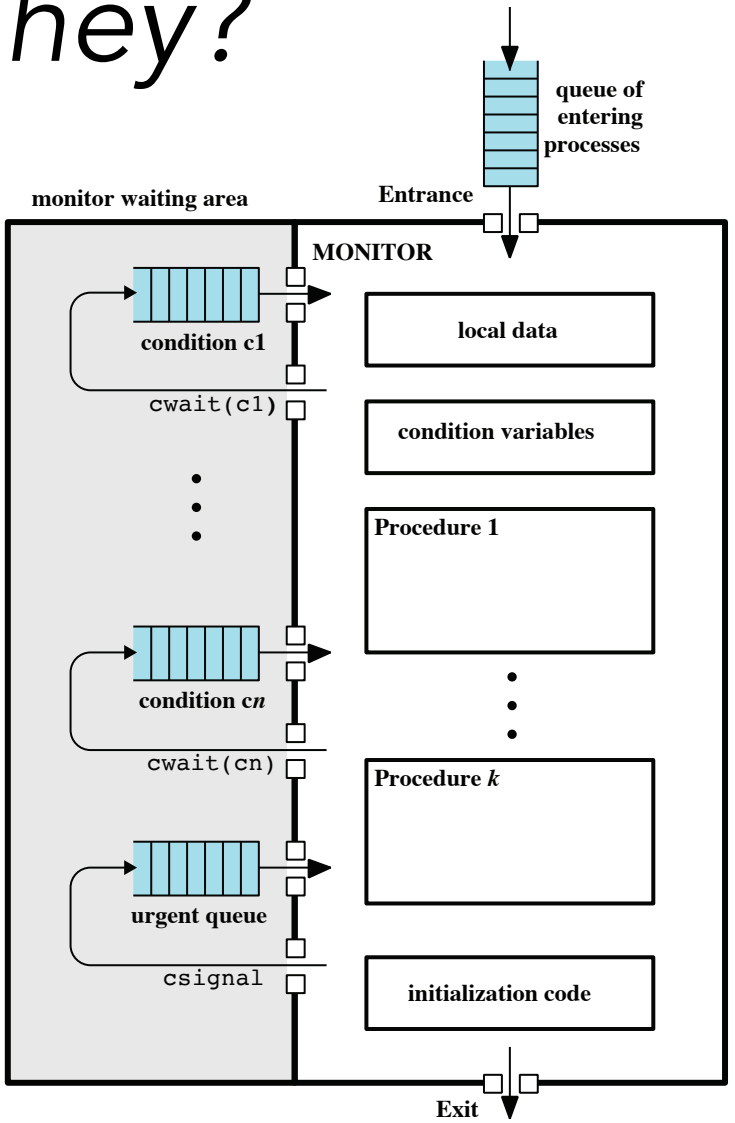
Condition Variables: *What Are They?*

The Main Idea:

- A **monitor** is a set of **mutually exclusive** routines coordinated by **condition variables**

Operations on Condition Variables:

- **CVarInit()** – initialize one
- **CVarWait()** – block (+release lock) until someone signals
- **CVarSignal()** – let a waiter run
- **CVarBroadcast()** – *(sometimes)* let *ALL* waiters run



NOTE: Inside of CVarWait(), the waiter reacquires the lock before returning.

Condition Variables: *What Are They?*

What does "let a waiter run" mean?

There are two main styles:

- **Hoare-style**: The signaled waiter runs right away

```
if ( !condx ) { /* cvar_wait; */ }
```

- **Mesa-style**: The signaled waiter is put back on the ready queue

```
while ( !condx ) { /* cvar_wait; */ }
```

Implementing Condition Variables

Suppose we want to implement the condition variable synchronization primitives as syscalls.

What do we need to do?

Using Condition Variables: Initialization

```
// In pthreads, a cond. var. is a pthread_cond_t.
// ... you'll also need a pthread_mutex_t (see next slide)

// You can initialize one statically:
pthread_cond_t cvar1 = PTHREAD_COND_INITIALIZER;

// Or dynamically:
rc = pthread_cond_init(&cvar1, NULL);
```

Using Condition Variables: Wait/Signal

```
// Precondition: Suppose you have some mutex that has been initialized…
pthread_cond_wait(&cvar1, &mutex); // rc always 0; see the man page

/* after you're done, signal waiters… */

pthread_cond_signal(&cvar1);
// vs.
pthread_cond_broadcast(&cvar1);
```

Demo: sync_cvar.c

When to Use Condition Variables?

When you need non-trivial rules to determine who should block/proceed

Some scenarios:

- to signal real, transient conditions
- to coordinate methods that share resources (hence require mutual exclusion), but need to wait for each other
- when you need to wait for multiple conditions...

Programming w/ Condition Variables

A Typical Sequence

```
// The waiter
acquire lock;
while (my condition is NOT true)
    wait on the cvar;

// Now I have the lock! And the condition is true!
*do the critical section *

release lock;
```

Semaphores

Synchronization via counting, waiting, and signaling!

Semaphores: *What Are They?*

The Main Idea:

- Signaling variables / an integer value used for signaling

Operations on Semaphores:

- **SemInit()** – set up a semphore with a specified initial value
- **SemUp()** – AKA V() or Signal() or Post()
  - increment value
  - if value is **less than or equal to zero**, transmit a signal to wake a waiter!
- **SemDown()** –AKA P() or Wait()
  - decrement value
  - if value is **less than or equal to zero**, block the caller! *(else, continue)*

Implementing Semaphores

Suppose we want to implement the semaphore synchronization primitives as syscalls.

What do we need to do?

Using Condition Variables: Initialization

```
#include <semaphore.h> // <<< Note the need for a new include!

// A semaphore is a sem_t.
sem_t sem;

rc = sem_init(&sem,                 // the semaphore
              0,                    // must be 0 on Linux; see the man page
              INITIAL_VALUE);       // initial value of semaphore
if (rc) {
    printf("hey, it failed!\n");
    exit(-1);
}
```

Using Condition Variables: Wait/Signal

```
// "SemDown"
sem_wait(&sem); // rc is always 0; see man page

/* after you're done, signal waiters... */

// "SemUp"
rc = sem_post(&sem);
if (rc) {
    printf("hey, it failed!\n");
    exit(-1);
}
```

Demo: sync_sem.c

Reflection

- How does a semaphore differ from a lock?
  - *Capacity:* A semaphore has memory: you can "release it" 3 times, which will then let three more people acquire it.
  - *Cross-entity:* You don't need to "have" the semaphore in order to release it.

- How does a condition variable differ from a semaphore?
  - The condition variable does not "remember" unused signals.
  - The semaphore does not require mutual exclusion between the two callers.

Reflection

When to use Semaphores?

Typically:

- when you have a resource with **capacity**, or
- when you want an action by **one actor to wake up another one**, in some asymmetric way

NOTE:

While you can use semaphores, it is almost always easier to end up with clear and correct code if you use locks and cvars instead.

Reflection

Why might a semaphore be the right sync. mechanism to use in PA2?

Why might a semaphore ***NOT*** be the right sync. mechanism to use?

NOTE: While they are powerful/flexible, it is often difficult to produce a correct program using semaphores...