Announcements

Project 3: Shared memory segments

- Linux: use shmget and shmat across server + client processes
  - semaphores for locks; catch ctrl-C to do clean-up
  - Can work with a project partner (request new one if desired)
  - No videos
- Xv6: Implement combination of shmgetat() – Watch video!
  - No partner for this part
- Due Wed 11/02 by 9:00 pm

Class feedback for mid-course evaluations

- Receive email about survey to fill out until this Friday

Midterm 2: 11/09 Wednesday 7:15-9:15 in Humanities 3650

- Fill out form on web page if have conflict – deadline 2 weeks before

Today's Reading: Chapter 31

Fastest Sorters

- 301 : Craig Barabas
- 302 : Gerald Anders
- 303 : William Yang
- 304 : Zachary Wachtel

UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

CLASSIC SYNCHRONIZATION PROBLEMS

Questions answered in this lecture:

What are Monitors and why do people like them?

How to solve Dining Philosophers synchronization problem?

How to provide Reader/Writer Locks?

Priority to Readers vs Priority to Writers

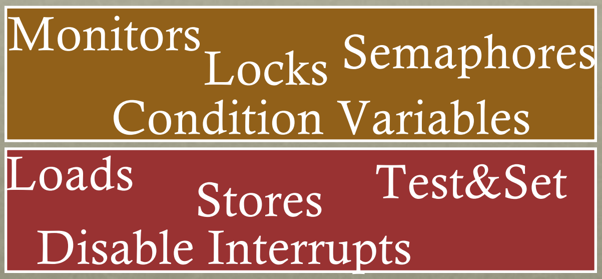
If time: Condition Variable review for Exam

SYNCHRONIZATION

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right



MONITORS

Motivation

- Users can inadvertently misuse locks and semaphores
  - (e.g., never unlock a mutex)

Idea

- Provide language support to automatically lock and unlock monitor lock when in critical section
  - Lock is added implicitly; never seen by user
- Provide condition variables for scheduling constraints (zero or more)

Examples

- Mesa language from Xerox
- Java: Acquire monitor lock when call **synchronized** methods in class

```
synchronized deposit(int amount) {
     // language adds lock.acquire()
     balance += amount;
     // language adds lock.release()
}
```

SEMAPHORE OPERATIONS

Allocate and Initialize

```
sem_t sem;
sem_init(sem_t *s, int initval) {
  s->value = initval;
}
```

User cannot read or write value directly after initialization

Wait or Test (sometime P() for Dutch word)

Waits until value of sem is > 0, then decrements sem value

Post or Signal or Increment (sometime V() for Dutch)

Increment sem value, then wake a single waiter (so it can check)

wait and post are atomic

SEMAPHORES

Semaphores are equivalent to locks + condition variables

- Can be used for both mutual exclusion and ordering

Semaphores contain **state**

- How they are initialized depends on how they will be used
- Init to 1: Mutex
- Init to 0: Join (1 thread must arrive first and post)
- Init to N: Number of available resources

Sem_wait(): Waits until value > 0, then decrement (atomic)

Sem_post(): Increment value, then wake a single waiter (atomic)

Can use semaphores in producer/consumer relationships (previous lecture) and for reader/writer locks

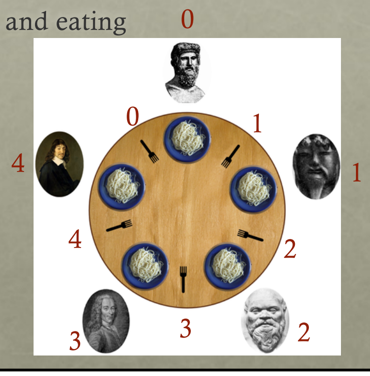
DINING PHILOSOPHERS

Problem Statement

- **N** Philosophers sitting at a round table
- Each philosopher shares a chopstick (or fork) with neighbor
- Each philosopher must have both chopsticks to eat
- Neighbors can't eat simultaneously
- Philosophers alternate between thinking and eating

Each philosopher/thread **i** runs following code:

```
while (1) {
       think();
       take_chopsticks(i);
       eat();
       put_chopsticks(i);
}
```



DINING PHILOSOPHERS: ATTEMPT #1

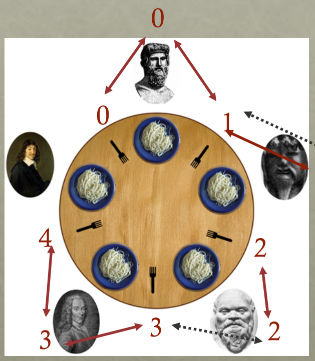
Two neighbors can't use chopstick at same time

Must test if chopstick is there and grab it atomically

- Represent each chopstick with a semaphore
- Grab right chopstick then left chopstick

Code for 5 philosophers:

```
sem_t chopstick[5]; // Initialize each
take_chopsticks(int i) {
      wait(&chopstick[i]);
      wait(&chopstick[(i+1)%5]);
}
put_chopsticks(int i) {
      signal(&chopstick[i]);
      signal(&chopstick[(i+1)%5]);
}
```



DINING PHILOSOPHERS: ATTEMPT #1

Two neighbors can't use chopstick at same time

Must test if chopstick is there and grab it atomically

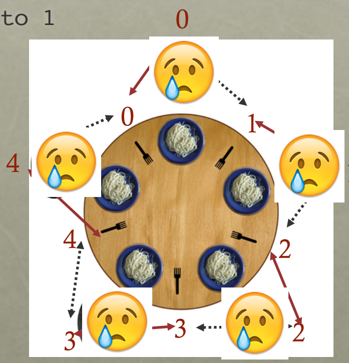
- Represent each chopstick with a semaphore
- Grab right chopstick then left chopstick

Code for 5 philosophers:

```
sem_t chopstick[5]; // Initialize each to 1
take_chopsticks(int i) {
      wait(&chopstick[i]);
      wait(&chopstick[(i+1)%5]);
}
put_chopsticks(int i) {
      signal(&chopstick[i]);
      signal(&chopstick[(i+1)%5]);
}
```

What is wrong with this solution???

Deadlocked!



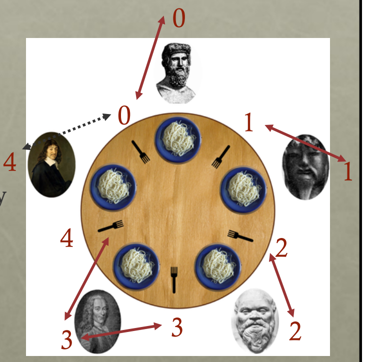
DINING PHILOSOPHERS: ATTEMPT #2

Approach

  Grab lower-numbered chopstick first, then higher-numbered

Code for 5 philosophers:

```
sem_t chopstick[5]; // Initialize to 1
take_chopsticks(int i) {
      if (i < 4) {
              wait(&chopstick[i]);
              wait(&chopstick[i+1]);
      } else {
              wait(&chopstick[0]);
              wait(&chopstick[4]);
}
```



Philosopher 3 finishes take_chopsticks() and eventually calls put_chopsticks();

Who can run then?

What is wrong with this solution???

DINING PHILOSOPHERS: HOW TO APPROACH

Guarantee two goals

- **Safety:** Ensure nothing bad happens (don't violate constraints of problem)
- **Liveness:** Ensure something good happens when it can (make as much progress as possible)

Introduce state variable for each philosopher i

```
state[i] = THINKING, HUNGRY, or EATING
```

Safety:

No two adjacent philosophers eat simultaneously

```
for all i: !(state[i]==EATING && state[i+1%5]==EATING)
```

Liveness:

Not the case that a philosopher is hungry and his neighbors are not eating

```
for all i: !(state[i]==HUNGRY &&
        (state[i+4%5]!=EATING && state[i+1%5]!=EATING))
```

```
sem_t mayEat[5]; // how to initialize?
sem_t mutex; // how to init?
int state[5] = {THINKING};
take_chopsticks(int i) {
     wait(&mutex); // enter critical section
     state[i] = HUNGRY;
     testSafetyAndLiveness(i); // check if I can run
     signal(&mutex); // exit critical section
     wait(&mayEat[i]);
}
put_chopsticks(int i) {
     wait(&mutex); // enter critical section
     state[i] = THINKING;
     test(i+1 %5); // check if neighbor can run now
     test(i+4 %5);
     signal(&mutex); // exit critical section
}
testSafetyAndLiveness(int i) {
     if(state[i]==HUNGRY&&state[i+4%5]!=EATING&&state[i+1%5]!=EATING) {
            state[i] = EATING;
            signal(&mayEat[i]);
     } }
```

Dining Philosophers: Example Execution

Take_chopsticks(0)

Take_chopsticks(1)

Take_chopsticks(2)

Take_chopsticks(3)

Take_chopsticks(4)

Put_chopsticks(0)

Put_chopsticks(2)

READER/WRITER LOCKS

Protect shared data structure; Goal:

Let multiple reader threads grab lock with other readers (shared)

Only one writer thread can grab lock (exclusive)

- No reader threads
- No other writer threads

Two possibilities for priorities – different implementations

1) No reader waits unless writer in critical section
    - How can writers starve?
2) No writer waits longer than absolute minimum
    - How can readers starve?

Let us see if we can understand code…

VERSION 1

Readers have priority

READER/WRITER LOCKS

```
typedef struct _rwlock_t {
    sem_t lock;
    sem_t writelock;
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 1);
    sem_init(&rw->writelock, 1);
}

```

READER/WRITER LOCKS

```
void rwlock_acquire_readlock(rwlock_t *rw) {
      sem_wait(&rw->lock);
      rw->readers++;
      if (rw->readers == 1)
         sem_wait(&rw->writelock);
      sem_post(&rw->lock);
}
void rwlock_release_readlock(rwlock_t *rw) {
      sem_wait(&rw->lock);
      rw->readers--;
      if (rw->readers == 0)
         sem_post(&rw->writelock); ]
      sem_post(&rw->lock);
}
rwlock_acquire_writelock(rwlock_t *rw) {  sem_wait(&rw->writelock);  }
rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

```
T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T6: acquire_readlock()
T1: release_readlock()
T6: release_readlock()
T4: acquire_readlock()
T5: acquire_readlock()  // ???
T3: release_writelock()
// what happens???
```

VERSION 2

Writers have priority

Three semaphores

- Mutex
- OKToRead (siimilar to myEat[] in Dining Philisphers)
- OKToWrite

How to initialize?

Reader Process

```
Sem_wait(&mutex);
If (ActiveWriters +
    WaitingWriters==0) {
     sem_post(OKToRead);
     ActiveReaders++;
} else WaitingReaders++;
Sem_post(&mutex);
Sem_wait(OKToRead);
// Do read
Sem_wait(&mutex);
ActiveReaders--;
If (ActiveReaders==0 &&
WaitingWriters > 0) {
     Sem_post(OKToWrite);
     ActiveWriters++;
     WaitingWriters--;
}
Sem_post(&mutex);
```

Writer Process

```
Sem_wait(&mutex);
If (ActiveWriters +
ActiveReaders +
WaitingWriters==0) {
     sem_post(OKToWrite);
     ActiveWriters++;
} else WaitingWriters++;
Sem_post(&mutex);
Sem_wait(OKToWrite);
// Do write
Sem_wait(&mutex);
ActiveWriters--;
If (WaitingWriters > 0) {
     Sem_post(OKToWrite);
     ActiveWriters++;
     WaitingWriters--;
} else while(WaitingReaders>0)
{
         sem_post(OKToRead);
         ActiveReaders++;
         WaitingReaders--;
     }
Sem_post(&mutex);
```

SEMAPHORES

Semaphores are equivalent to locks + condition variables

- Can be used for both mutual exclusion and ordering

Semaphores contain **state**

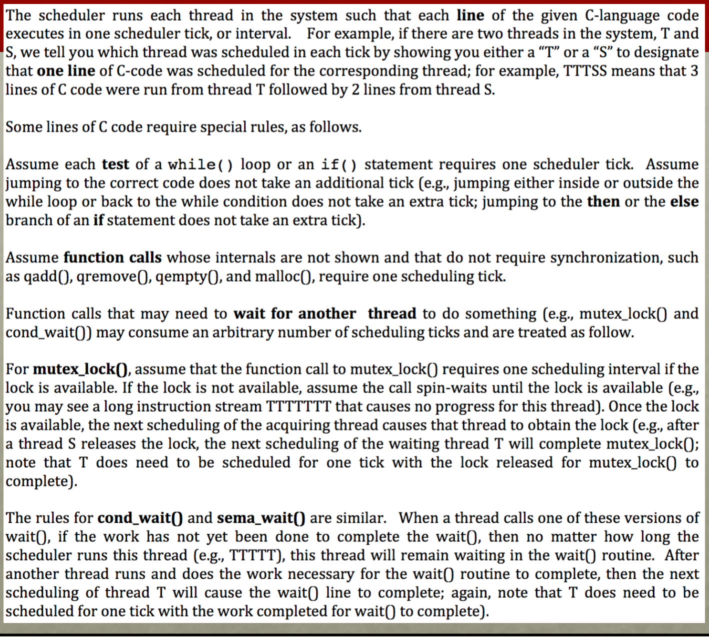
- How they are initialized depends on how they will be used
- Init to 1: Mutex
- Init to 0: Join (1 thread must arrive first, then other)
- Init to N: Number of available resources

Sem_wait(): Waits until value > 0, then decrement (atomic)

Sem_post(): Increment value, then wake a single waiter (atomic)

Can use semaphores in producer/consumer relationships and for reader/writer locks

CV REVIEW



The scheduler runs each thread in the system such that each **line** of the given C-language code executes in one scheduler tick, or interval. For example, if there are two threads in the system, T and S, we tell you which thread was scheduled in each tick by showing you either a "T" or a "S" to designate that **one line** of C-code was scheduled for the corresponding thread; for example, TTTSS means that 3 lines of C code were run from thread T followed by 2 lines from thread S.

Some lines of C code require special rules, as follows.

Assume each **test** of a `while()` loop or an `if()` statement requires one scheduler tick. Assume jumping to the correct code does not take an additional tick (e.g., jumping either inside or outside the while loop or back to the while condition does not take an extra tick; jumping to the **then** or the **else** branch of an **if** statement does not take an extra tick).

Assume **function calls** whose internals are not shown and that do not require synchronization, such as qadd(), qremove(), qempty(), and malloc(), require one scheduling tick.

Function calls that may need to **wait for another thread** to do something (e.g., mutex_lock() and cond_wait()) may consume an arbitrary number of scheduling ticks and are treated as follow.

For **mutex_lock()**, assume that the function call to mutex_lock() requires one scheduling interval if the lock is available. If the lock is not available, assume the call spin-waits until the lock is available (e.g., you may see a long instruction stream TTTTTTT that causes no progress for this thread). Once the lock is available, the next scheduling of the acquiring thread causes that thread to obtain the lock (e.g., after a thread S releases the lock, the next scheduling of the waiting thread T will complete mutex_lock(); note that T does need to be scheduled for one tick with the lock released for mutex_lock() to complete).

The rules for **cond_wait()** and **sema_wait()** are similar. When a thread calls one of these versions of wait(), if the work has not yet been done to complete the wait(), then no matter how long the scheduler runs this thread (e.g., TTTTT), this thread will remain waiting in the wait() routine. After another thread runs and does the work necessary for the wait() routine to complete, then the next scheduling of thread T will cause the wait() line to complete; again, note that T does need to be scheduled for one tick with the work completed for wait() to complete).

```
void thread_join() {
        Mutex_lock(&m);          // p1
        while (done == 0)        // p2
            Cond_wait(&c, &m);   // p3
        Mutex_unlock(&m);        // p4
}
void thread_exit() {
        Mutex_lock(&m);          // c1
        done = 1;                // c2
        Cond_signal(&c);         // c3
        Mutex_unlock(&m);        // c4
}
```

After the instruction stream “P” (i.e., after scheduler runs one line from parent), which line of the parent’s will execute when it is scheduled again?

p2. P will have acquired lock and finished mutex_lock().

Assume the scheduler continues on with “C” (the full instruction stream is PC). Which line will child execute when it is scheduled again?

C1. C must wait to acquire the lock since it is currently held by P.

After PPP (full is PCPPP), which line for parent next?

p3. Since done = 0, P will execute p2 and p3; it is stuck in p3 until signaled.

After CCC (full is PCPPPCCC), which line for child next?

c4. C finishes c1, c2, c3. Next time it executes c4. CV signaled but mutex still locked

After PP (full is PCPPPCCCPP), which line for parent next?

p3. P cannot return from cond_wait() until it acquires lock held by C; P stays in p3

After CC (full is PCPPPCCCPPCC, which line for child next?

Beyond. C executes c4 and then code beyond c4.

After PPP (full PCPPPCCCPPCCPPP), which line for parent next?

Beyond. P finishes p3, rechecks p2, then p4. Next time beyond p4.