Computer Science 162
Discussion Section
Week 3

Agenda

- Project 1 released!
- Locks, Semaphores, and condition variables
- Producer-consumer
  - Example (locks, condition variables)
  - Student exercise
- Dining philosophers problem
  - In-class exercise

Note: Referenced slides from Jonathan Walpole, Henri Casanova, CERCS Intro. Thread Lecture

Definitions

- Synchronization: using atomic operations to ensure cooperation between threads
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- Critical Section: piece of code that only one thread can execute at once
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing.

Where are we going with

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
  – Everything is pretty painful if only atomic primitives are load and store
  – Need to provide primitives useful at user-level

Implementing Locks with test&set

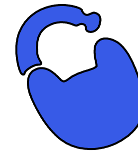
- Simple solution:

```
int value = 0; // Free
Acquire() {
   while (test&set(value)); // while busy
}
Release() {
   value = 0;
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value



```
int guard = 0;
int value = FREE;

Acquire() {
    // Short busy-wait time
    while (test&set(guard));
    if (value == BUSY) {
        put thread on wait queue;
        go to sleep() & guard = 0;
    } else {
        value = BUSY;
        guard = 0;
    }
}
```

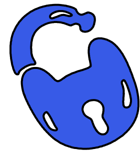
```
Release() {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
        take thread off wait queue
        Place on ready queue;
    } else {
        value = FREE;
    }
    guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Better Locks using test&set

- Compare to “disable interrupt” solution

```
int value = FREE;
```



```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

- Basically replace
  - `disable interrupts` → `while (test&set(guard));`
  - `enable interrupts` → `guard = 0;`

Examples of Read-Modify-Write

- ```
  test&set (&address) {     /* most architectures */
      result = M[address];
      M[address] = 1;
      return result;
  }
  ```

- ```
  swap (&address, register) { /* x86 */
      temp = M[address];
      M[address] = register;
      register = temp;
  }
  ```

- ```
  compare&swap (&address, reg1, reg2) { /* 68000 */
      if (reg1 == M[address]) {
          M[address] = reg2;
          return success;
      } else {
          return failure;
      }
  }
  ```
  ```

Implementing Locks with test&set

- Simple solution:

```
int value = 0; // Free

Acquire() {
   while (test&set(value)); // while busy
}

Release() {
   value = 0;
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock

Semaphores

- An abstract data type that can be used for condition synchronization and mutual exclusion
- Condition synchronization
  - wait until invariant holds before proceeding
  - signal when invariant holds so others may proceed
- Mutual exclusion
  - only one at a time in a critical section

Semaphores

- An abstract data type
  - containing an integer variable (S)
  - Two operations: Wait (S) and Signal (S)

- Alternative names for the two operations
  - *Wait(S) = Down(S) = P(S)*
  - *Signal(S) = Up(S) = V(S)*

Classical Definition of Wait and Signal

```
Wait(S)
  {
   while S <= 0 do noop;  /* busy wait!
  */
   S = S - 1;           /* S >= 0 */
  }
Signal (S)
  {
   S = S + 1;
  }
```

Blocking implementation of semaphores

Semaphore S has a value, S.val, and a thread list, S.list.

Wait (S)

```
S.val = S.val - 1
If S.val < 0                      /* negative value of S.val */
  {  add calling thread to S.list;  /* is # waiting threads */
     block;                       /* sleep */
  }
```

Signal (S)

```
S.val = S.val + 1
If S.val <= 0
  {  remove a thread T from S.list;
     wakeup (T);
  }
```

Using Semaphores for Mutex

```
semaphore mutex = 1     -- unlocked
```

```
1 repeat
2   wait(mutex);
3   critical section
4   signal(mutex);
5   remainder section
6 until FALSE
```

Thread A

```
1 repeat
2   wait(mutex);
3   critical section
4   signal(mutex);
5   remainder section
6 until FALSE
```

Thread B

Using Semaphores for Mutex

```
semaphore mutex = 0     -- locked
```

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread A

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

Thread B

Using Semaphores for Mutex

```
semaphore mutex = 0     --locked
```

Thread A

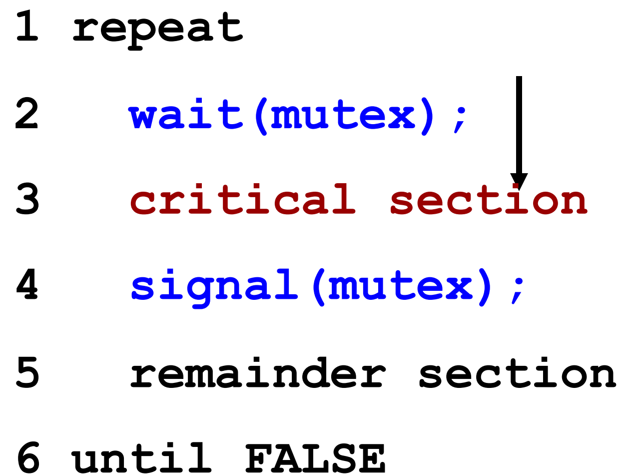
```
1 repeat
2   wait(mutex);
3   critical section
4   signal(mutex);
5   remainder section
6 until FALSE
```

Thread B

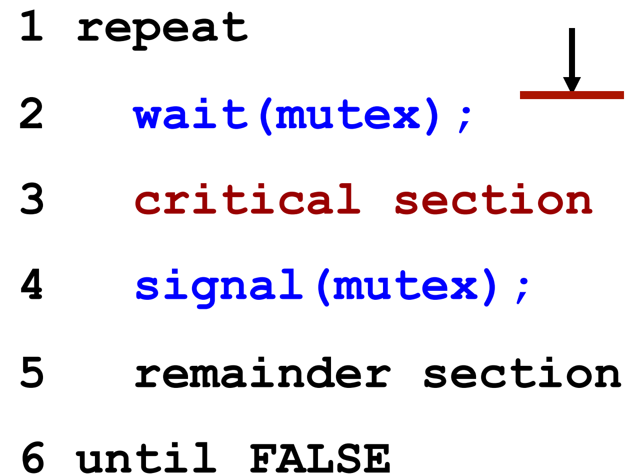
```
1 repeat
2   wait(mutex);
3   critical section
4   signal(mutex);
5   remainder section
6 until FALSE
```

Using Semaphores for Mutex

```
semaphore mutex = 0     -- locked
```



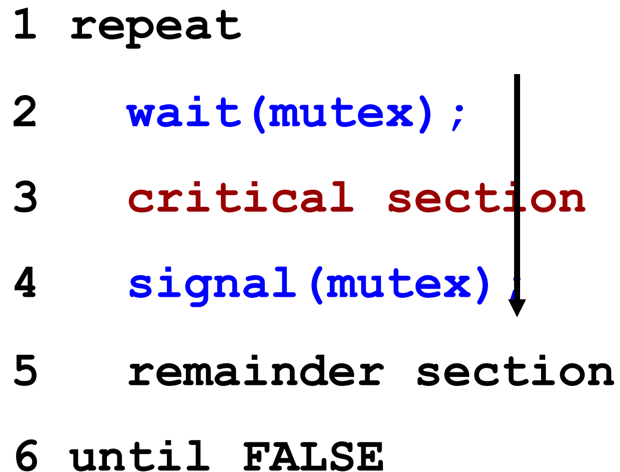
Thread A



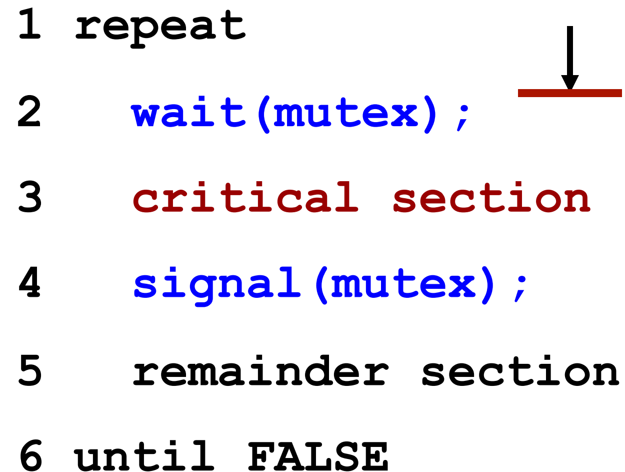
Thread B

Using Semaphores for Mutex

```
semaphore mutex = 0     -- locked
```



Thread A



Thread B

Using Semaphores for Mutex

```
semaphore mutex = 1     -- unlocked
```

This thread can now be released!

```
1 repeat
2   wait(mutex);
3   critical section
4   signal(mutex);
5   remainder section
6 until FALSE
```

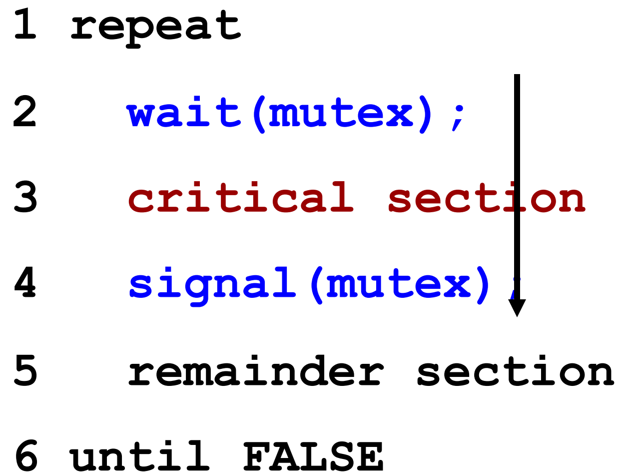
Thread A

```
1 repeat
2   wait(mutex);
3   critical section
4   signal(mutex);
5   remainder section
6 until FALSE
```

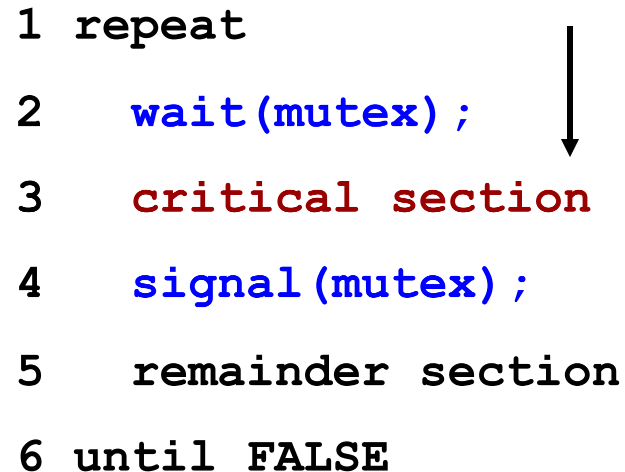
Thread B

Using Semaphores for Mutex

semaphore **mutex = 0 -- locked**



Thread A



Thread B

To block or not to block?

- Spin-locks do *busy waiting*
  - wastes CPU cycles on uni-processors
  - Why?
- Blocking locks put the thread to *sleep*
  - may waste CPU cycles on multi-processors
  - Why?
  - … and we need a spin lock to implement blocking on a multiprocessor anyway!

Condition Variables

- Mutexes are used to control access to shared data
  - only one thread can execute inside a `Lock` clause
  - other threads who try to `Lock`, are blocked until the mutex is unlocked
- Condition variables are used to wait for specific events
  - free memory is getting low, wake up the garbage collector thread
  - 10,000 clock ticks have elapsed, update that window
  - new data arrived in the I/O port, process it
- Could we do the same with mutexes?
  - (think about it and we' ll get back to it)

Condition Variable Example

```
Mutex io_mutex;
Condition non_empty;
...
Consumer:
Lock (io_mutex) {
   while (port.empty())
     Wait(io_mutex, non_empty);
   process_data(port.first_in());
}

Producer:
Lock (io_mutex) {
   port.add_data();
   Signal(non_empty);
}
```

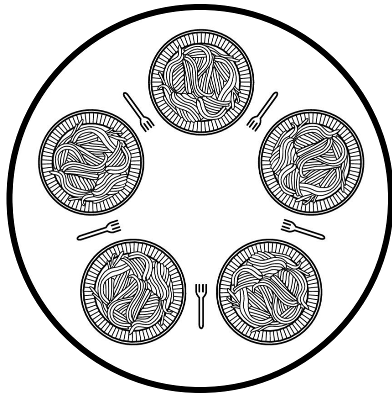
Condition Variables Semantics

- Each condition variable is associated with a single mutex
- `Wait` *atomically* unlocks the mutex and blocks the thread
- `Signal` awakes a blocked thread
  - the thread is awoken inside `Wait`
  - tries to lock the mutex
  - when it (finally) succeeds, it returns from the `Wait`
- Doesn't this sound complex? Why do we do it?
  - the idea is that the "condition" of the condition variable depends on data protected by the mutex

Extra

Dining philosophers problem

- Five philosophers sit at a table
- One fork between each philosopher



Each philosopher is modeled with a thread

```
while(TRUE) {
  Think();
  Grab first fork;
  Grab second fork;
  Eat();
  Put down first fork;
  Put down second fork;
}
```

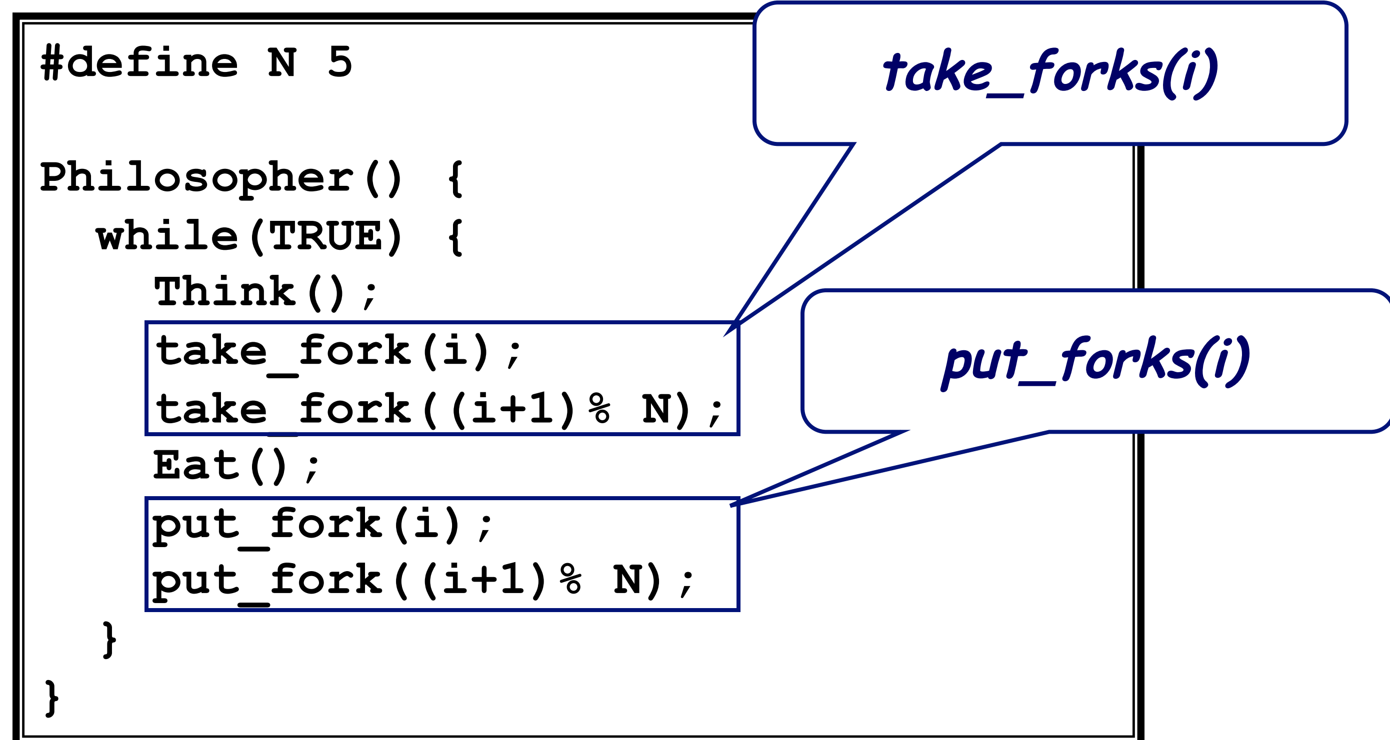
- *Why do they need to synchronize?*
- *How should they do it?*

Is this a valid solution?

```
#define N 5

Philosopher() {
  while(TRUE) {
    Think();
    take_fork(i);
    take_fork((i+1)% N);
    Eat();
    put_fork(i);
    put_fork((i+1)% N);
  }
}
```

Working towards a solution …



Working towards a solution …

```
#define N 5

Philosopher() {
  while(TRUE) {
    Think();
    take_forks(i);
    Eat();
    put_forks(i);
  }
}
```

Picking up forks

```
int state[N]
semaphore mutex = 1
semaphore sem[i]
```

```
take_forks(int i) {
  wait(mutex);
  state [i] = HUNGRY;
  test(i);
  signal(mutex);
  wait(sem[i]);
}
```

```
// only called with mutex set!

test(int i) {
 if (state[i] == HUNGRY &&
     state[LEFT] != EATING &&
     state[RIGHT] != EATING){
   state[i] = EATING;
   signal(sem[i]);
 }
}
```

Putting down forks

```
int state[N]
semaphore mutex = 1
semaphore sem[i]
```

```
put_forks(int i) {
  wait(mutex);
  state [i] = THINKING;
  test(LEFT);
  test(RIGHT);
  signal(mutex);
}
```

```
// only called with mutex set!

test(int i) {
 if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING){
   state[i] = EATING;
   signal(sem[i]);
 }
}
```

Dining philosophers

- Is the previous solution correct?
- What does it mean for it to be correct?
- Is there an easier way?