



UNIVERSITY OF OREGON

# **CIS 415:**

# **Operating Systems**

# **Synchronization**

Prof. Butler  
Spring 2012

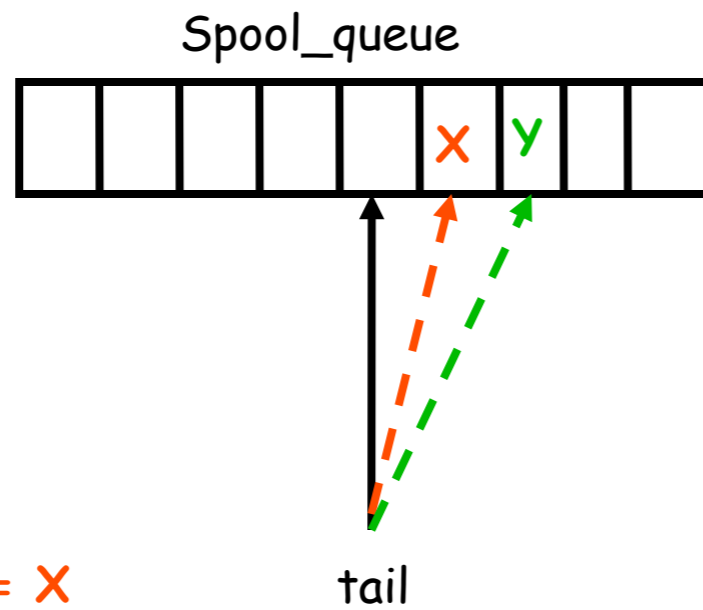
- There are different kinds of resources that are shared between processes:
  - Physical (terminal, disk, network, ...)
  - Logical (files, sockets, memory, ...)
- For the purposes of this discussion, let us focus on “memory” to be the shared resource
  - i.e. processes can all read and write into memory (variables) that are shared.

# Problems due to sharing



- Consider a shared printer queue, `spool_queue[N]`
- 2 processes want to enqueue an element each to this queue.
- `tail` points to the current end of the queue
- Each process needs to do
  - `tail = tail + 1;`
  - `spool_queue[tail] = "element";`

# What we are trying to do



Process 1

$tail = tail + 1;$   
 $Spool\_queue[tail] = X$

Process 2

$tail = tail + 1;$   
 $Spool\_queue[tail] = Y$

# What is the problem?



- $\text{tail} = \text{tail} + I$  is NOT 1 machine instruction
- It can translate as follows:
  - Load tail, R1**
  - Add R1, I, R2**
  - Store R2, tail**
- These 3 machine instructions may NOT be executed **atomically**.

- If each process is executing this set of 3 instructions, context switching can happen at any time.
- Let us say we get the following resultant sequence of instructions being executed:

P1: Load tail, R1

P1: Add R1, I, R2

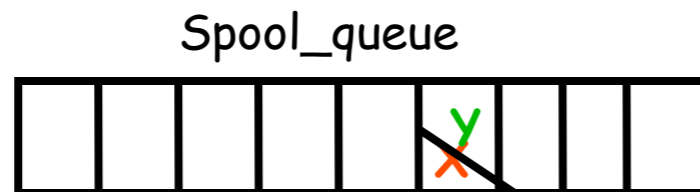
P2: Load tail, R1

P2: Add R1, I, R2

P1: Store R2, tail

P2: Store R2, tail

# Leading to ...



Process 1

$tail = tail + 1;$   
 $Spool\_queue[tail] = X$

Process 2

$tail = tail + 1;$   
 $Spool\_queue[tail] = Y$

- Situations like this that can lead to erroneous execution are called **race conditions**
  - The outcome of the execution depends on the particular interleaving of instructions
- Debugging race conditions can be fun!
  - since errors can be non-repeatable.



# Avoiding Race Conditions



UNIVERSITY  
OF OREGON

- If we had a way of making those (3) instructions atomic
  - i.e. while one process is executing those instructions, another process cannot execute the same instructions
  - then we could have avoided the race condition.
- These 3 instructions are said to constitute a **critical section**.

# Requirements for Solution



UNIVERSITY  
OF OREGON

1. **Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the  $N$  processes

# Implementing Critical Sections



UNIVERSITY  
OF OREGON

- Disable Interrupts
  - **Effectively stops scheduling other processes.**
- Busy-wait/spinlock Solutions
  - **Pure software solutions**
  - **Integrated hardware-software solutions**
- Blocking Solutions

# Disabling Interrupts




UNIVERSITY  
OF OREGON

- Advantages: Simple to implement
- Disadvantages:
  - **Do not want to give such power to user processes**
  - **Does not work on a multiprocessor**
  - **Disables multiprogramming even if another process is **NOT** interested in critical section**

# Implementing Critical Sections



UNIVERSITY  
OF OREGON

- Disable Interrupts
  - **Effectively stops scheduling other processes.**
-  • Busy-wait/spinlock Solutions
  - **Pure software solutions**
  - **Integrated hardware-software solutions**
- Blocking Solutions

- Overall philosophy: Keep checking some state (variables) until they indicate other process(es) are not in critical section.
- However, this is a non-trivial problem.

```
locked = FALSE;

P1 {
while (locked == TRUE)
;
locked = TRUE;

/*****
(critical section code)
*****/

locked = FALSE;
}

P2 {
while (locked == TRUE)
;
locked = TRUE;

/*****
(critical section code)
*****/

locked = FALSE;
}
```

We have a race condition again since there is a gap between detection  
locked is FALSE, and setting locked to TRUE.

# I. Strict Alternation



```
turn = 0;
```

```
P0 {  
  while (turn != 0);  
  /******/  
  critical section  
  /******/  
  turn = 1;  
}
```

```
P1 {  
  while (turn != 1);  
  /******/  
  critical section  
  /******/  
  turn = 0;  
}
```

It works!

Problems:

- requires processes to alternate getting into CS
- does NOT meet Progress requirement.



# Fixing “progress” requirement



```
bool flag[2]; // initialized to FALSE
```

```
P0 {  
    flag[0] = TRUE;  
    while (flag[1] == TRUE)  
        ;  
    /* critical section */  
    flag[0] = FALSE;  
}
```

```
P1 {  
    flag[1] = TRUE;  
    while (flag[0] == TRUE)  
        ;  
    /* critical section */  
    flag[1] = FALSE;  
}
```

**Problem:**

Both can set their  
flags to true and wait  
indefinitely for the other

# Peterson's Solution



- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int **turn**;
  - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process  $P_i$  is ready!

# Peterson's Algorithm



```
int turn;
int interested[N]; /* all set to FALSE initially */

enter_CS(int myid) { /* param. is 0 or 1 based on P0 or P1 */
    int other;

    otherid = 1 - myid; /* id of the other process */
    interested[myid] = TRUE;
    turn = otherid;
    while (turn == otherid && interested[otherid] == TRUE)
        ;
    /* proceed if turn == myid or interested[otherid] == FALSE */
}

leave_CS(int myid) {
    interested[myid] = FALSE;
}
```

- This works because a process can enter CS, either because
  - ▶ Other process is not even interested in critical section
  - ▶ Or even if the other process is interested, it did the “turn = otherid” first.

# Prove that



- It is correct (achieves **mutex**)
  - ▶ If both are interested, then  $I$  condition is false for one and true for the other.
  - ▶ This has to be the “ $\text{turn} == \text{otherid}$ ” which cannot be false for both processes.
  - ▶ Otherwise, only one is interested and gets in

- There is **progress**
  - ▶ If a process is waiting in the loop, the other person has to be interested.
  - ▶ One of the two will definitely get in during such scenarios.

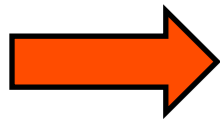
- There is **bounded waiting**
  - ▶ When there is only one process interested, it gets through
  - ▶ When there are two processes interested, the first one which did the “turn = otherid” statement goes through.
  - ▶ When the current process is done with CS, the next time it requests the CS, it will get it only after any other process waiting at the loop.

# Implementing Critical Sections



UNIVERSITY  
OF OREGON

- Disable Interrupts
  - **Effectively stops scheduling other processes.**
- Busy-wait/spinlock Solutions
  - **Pure software solutions**
  - **Integrated hardware-software solutions**
- Blocking Solutions





- Complications arose because we had atomicity only at the granularity of a machine instruction, and what a machine instruction could do was limited.
- Can we provide specialized instructions in hardware to provide additional functionality (with an instruction still being atomic)?

# Specialized Instructions



UNIVERSITY  
OF OREGON

- **Bool Test&Set(bool)**
- **Swap (bool, bool)**
- **Note that these are machine/assembly instructions, and are thus atomic.**

```
Atomic bool Test&Set(bool x) {  
    temp = x;  
    x = TRUE;  
    return (temp);  
}
```

- Note that “=x” and “x=” would have required at least 1 machine instruction each without this specialized instruction.

# Using Test&Set()



```
Bool lock;
```

```
Enter_CS() {  
    while  
    (Test&Set(lock))  
        ;  
}
```

```
Exit_CS() {  
    lock = FALSE;  
}
```

**NOTE:** This solution does not guarantee bounded Waiting.

# Swap()



```
Atomic Swap(bool a, bool b) {  
    temp = a;  
    a = b;  
    b = temp;  
}
```

- Again, all this is done atomically!

# Using swap()



```
Bool lock;
```

```
Enter_cs() {
```

```
    key = TRUE;          /* local var */
```

```
    while (key == TRUE) swap(key, lock);
```

```
}
```

```
Exit_cs() {
```

```
    lock = FALSE;
```

```
}
```

# Spinning vs. Blocking



- In the previous solns., we busy-waited for some condition to change.
- This change should be effected by some other process.
- We are “presuming” that this other process will eventually get the CPU (some kind of pre-emptive scheduler).
- This can be inefficient because:
  - You are wasting the rest of your time quantum in busy-waiting
  - Sometimes, your programs may not work! (if the OS scheduler is not pre-emptive).

- In blocking solutions, you relinquish the CPU at the time you cannot proceed, i.e. you are put in the blocked queue.
- It is the job of the process changing the condition to wake you up (i.e. move you from blocked back to ready queue).
- This way you do not unnecessarily occupy CPU cycles.



# More than just Exclusion



- Synchronization constructs required for more than exclusion.
  - E.g. If printer queue is full, I need to wait until there is at least 1 empty slot
  - Note that `mutex_lock()/mutex_unlock()` are not very suitable to implement such synchronization
  - We need constructs to enforce orderings (e.g. A should be done after B).

- You are given a data-type Semaphore\_t.
- On a variable of this type, you are allowed
  - P(Semaphore\_t) -- wait
  - V(Semaphore\_t) – signal
- Intuitive Functionality:
  - Logically one could visualize the semaphore as having a counter initially set to 0.
  - When you do a P(), you decrement the count, and need to block if the count becomes negative.
  - When you do a V(), you increment the count and you wake up 1 process from its blocked queue if not null.

# Semaphore Implementation



```
typedef struct {  
    int value;  
    struct process *L;  
} semaphore_t;
```

```
void P(semaphore_t S) {  
    S.value--;  
    if (S.value < 0) {  
        add this process to S.L and  
        remove from ready queue  
        context switch to another  
    }  
}
```

```
void V(semaphore_t S) {  
    S.value++;  
    if (S.value <= 0) {  
        remove a process from S.L  
        put it in ready queue  
    }  
}
```

**NOTE:** These are OS system calls, and there is no atomicity lost during the execution of these routines (interrupts are disabled).

# Binary vs. Counting Semaphores



UNIVERSITY  
OF OREGON

- What we just discussed is a counting semaphore.
- A binary semaphore restricts the “value” field to just 0 or 1.
- We will mainly restrict ourselves to counting semaphores.
- Exercise: Implement counting semaphores using binary semaphores.

# Semaphores can implement Mutex



```
Semaphore_t m;
```

```
Mutex_lock() {  
    P(m);  
}
```

```
Mutex_unlock() {  
    V(m);  
}
```

# Classic Synchronization Problems



UNIVERSITY  
OF OREGON

- Bounded-buffer problem
- Readers-writers problem
- Dining Philosophers problem
- ....
  
- We will compose solutions using semaphores

# Bounded Buffer problem



- A queue of finite size implemented as an array.
- You need mutual exclusion when adding/removing from the buffer to avoid race conditions
- Also, you need to wait when appending to buffer when it is full or when removing from buffer when it is empty.



# Bounded Buffer using Semaphores

```
int BB[N];
int count, head, tail = 0;
Semaphore_t m; // value initialized to 1
Semaphore_t empty; // value initialized to N
Semaphore_t full; // value initialized to 0

Append(int elem) {
    P(empty);
    P(m);

    BB[tail] = elem;
    tail = (tail + 1)%N;
    count = count + 1;

    V(m);
    V(full);
}

int Remove () {
    P(full);
    P(m);

    int temp = BB[head];
    head = (head + 1)%N;
    count = count - 1;

    V(m);
    V(empty);
    return(temp);
}
```



# Readers-Writers Problem



UNIVERSITY  
OF OREGON

- There is a database to which there are several readers and writers.
- The constraints to be enforced are:
  - **When there is a reader accessing the database, there could be other readers concurrently accessing it.**
  - **However, when there is a writer accessing it, there cannot be any other reader or writer.**



# Readers-writers using Semaphores

```
Database db;
int nreaders = 0;
Semaphore_t m; // value initialized to 1
Semaphore_t wrt; // value initialized to 1

Reader() {
    P(m);
    nreaders++;
    if (nreaders == 1) P(wrt);
    V(m);

    ... Read db here ...

    P(m);
    nreaders--;
    if (nreaders == 0) V(wrt);
    V(m);
}

Writer() {
    P(wrt);

    ... Write db here ...

    V(wrt);
}
```

# Dining Philosophers Problem



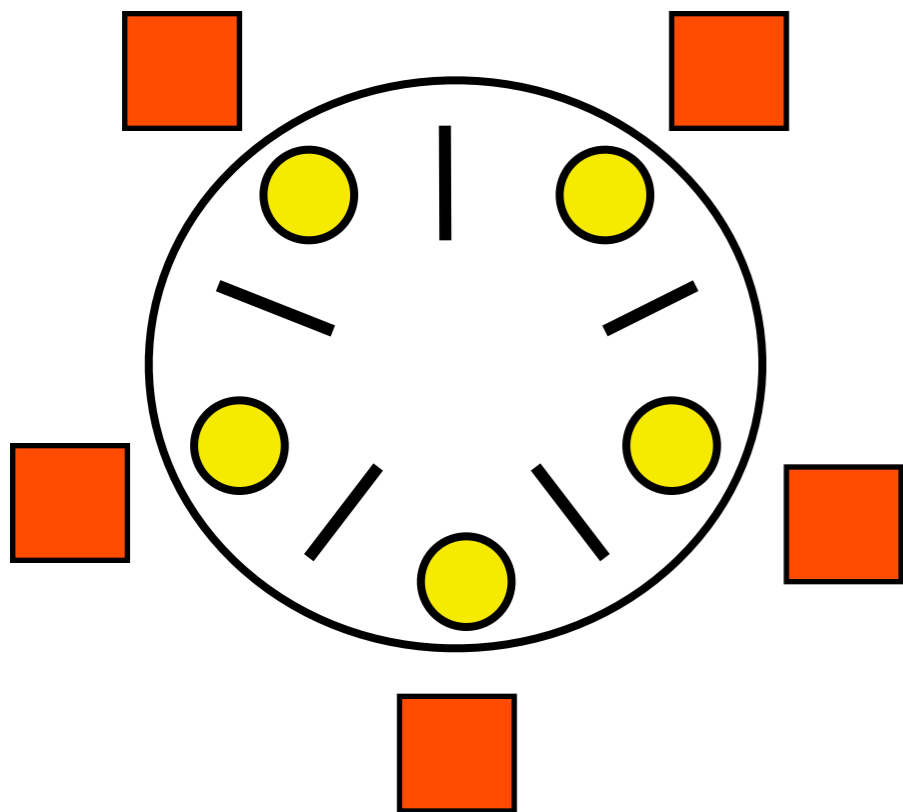
UNIVERSITY  
OF OREGON

Philosophers alternate between thinking and eating.

When eating, they need both (left and right) chopsticks.

A philosopher can pick up only 1 chopstick at a time.

After eating, the philosopher puts down both chopsticks.



```
Semaphore_t chopstick[5];

Philosopher(i) {
    while () {
        P(chopstick[i]);
        P(chopstick[(i+1)%5]);

        ... eat ...

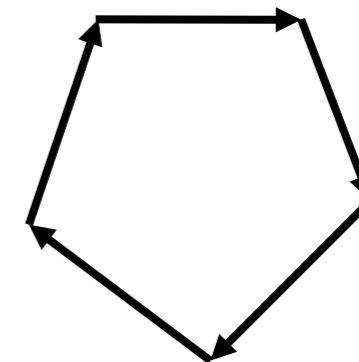
        V(chopstick[i]);
        V(chopstick[(i+1)%5]);

        ... think ...
    }
}
```

**This is NOT correct!**

**Though no 2  
philosophers  
use the same chopstick  
at any time, it can so  
happen that they all pick  
up 1 chopstick and wait  
indefinitely for another.**

**This is called a **deadlock****



- Note that putting  
     $P(\text{chopstick}[i]);$   
     $P(\text{chopstick}[(i+1)\%5];$   
within a critical section (using say  $P(\text{mutex})/V(\text{mutex})$ ) can avoid the deadlock.
- But then, only 1 philosopher can eat at any time!

```

int state[N];
Semaphore_t s[N]; // init. to 0
Semaphore_t mutex; // init. to 1

#define LEFT  (i-1)%N
#define RIGHT (i+1)%N

philosopher(i) {
    while () {
        take_forks(i);
        eat();
        put_forks(i);
        think();
    }
}

take_forks(i) {
    P(mutex);
    state[i] = HUNGRY;
    test(i);
    V(mutex);
    P(s[i]);
}

put_forks(i) {
    P(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    V(mutex);
}

test(i) { /* can phil i eat? if so, signal that philosopher */
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        V(s[i]);
    }
}
    
```

# Synchronization constructs



UNIVERSITY  
OF OREGON

- Mutual exclusion locks
- Semaphores
- Monitors
- Critical Regions
- Path Expressions
- Serializers
- ....

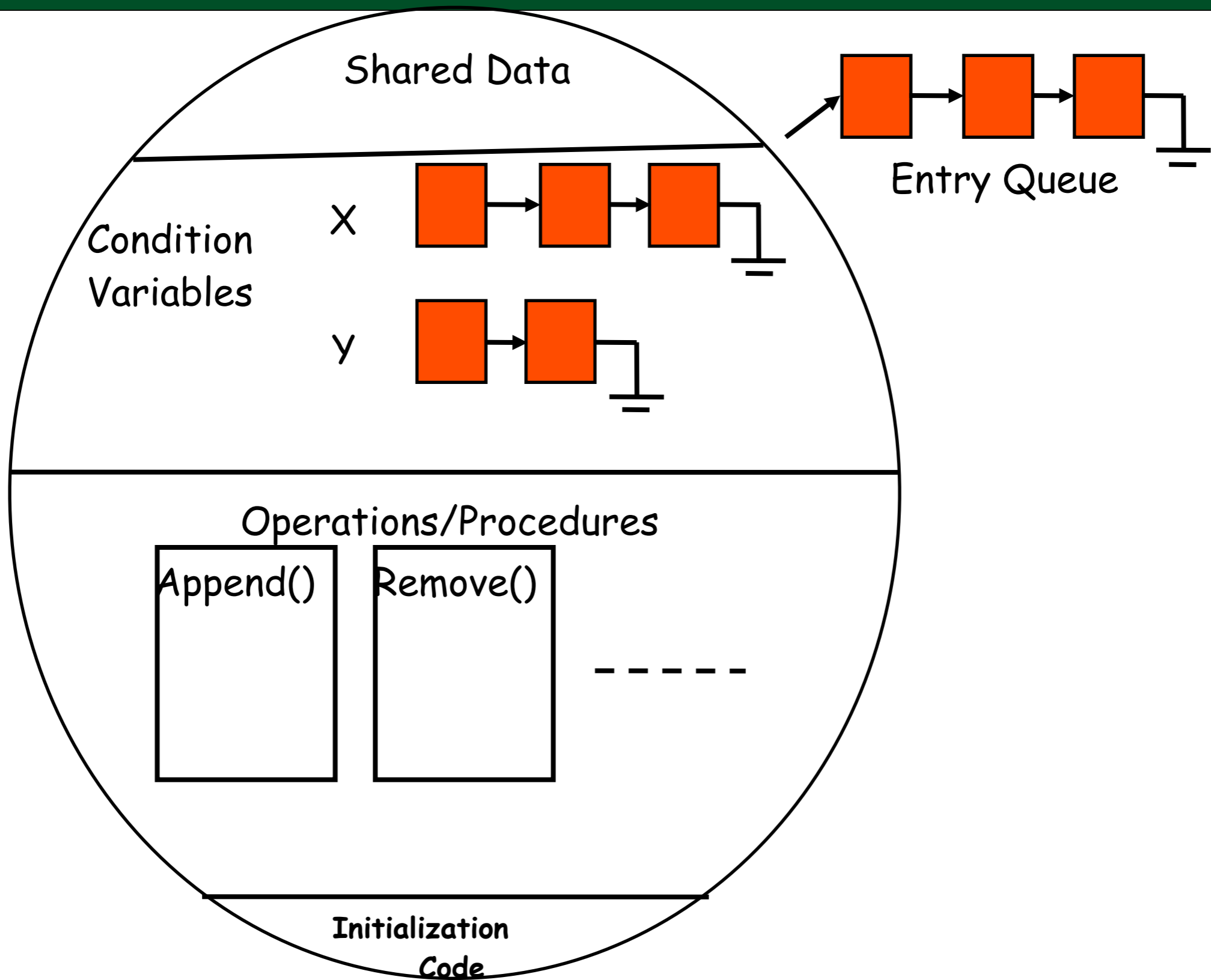
- An abstract data type consisting of
  - ▶ Shared data
  - ▶ Operations/procedures on this shared data
- External world only sees these operations (not the shared data or how the operations and sync. are implemented).
- Only 1 process can be “active” within the monitor at any time
  - ▶ i.e. of all the processes that are executing monitor code, there can be at most 1 process in ready queue (rest are either blocked or not in monitor!)



- In addition, you have a condition variable construct available within a monitor.
  - `Condition_t x, y;`
- You can perform the following operations on a condition variable:
  - `Wait(x)`: Process invoking this is blocked until someone does a signal.
  - `Signal(x)`; Resumes exactly one blocked process.
- **NOTE: If the signal comes before the wait, the signal gets lost!!!** – You need to be careful since signals are not stored unlike semaphores.

- When P1 signals to wake up P2, note that both cannot be simultaneously running as per monitor definition.
- There are these choices:
  - Signalling process (P1) executes, and P2 waits until the monitor becomes free.
  - P2 resumes execution in monitor, while P1 waits for monitor to become free.
  - Some other process (waiting for entry) gets the monitor, while both P1 and P2 wait for monitor to become free.
- In general, try to write solutions that do not depend on which choice is used when implementing the monitor.

# Structure of a Monitor



# Bounded Buffer using Monitors

```
Monitor Bounded_Buffer;

Buffer[0..N-1];
int count= 0, head=tail=0;
Cond_t not_full, not_empty;

Append(Data) {
    if count == N    wait(not_full);
    Buffer[head] = Data
    count++;
    head = (head+1)%N;
    if !empty(not_empty) signal(not_empty);
}

Remove() {
    if count == 0 wait(not_empty);
    Data = Buffer[tail];
    count--;
    tail = (tail+1)%N;
    if !empty(not_full) signal(not_full);
}
```

- Write monitor solutions for Readers-writers, and Dining Philosophers.

# Pthreads Synchronization



- Mutex Locks
  - Protection Critical Sections
  - `pthread_mutex_lock(&lock)`, `pthread_mutex_unlock(&lock)`
- Condition Variables
  - For Value-based Control
  - `pthread_cond_wait(&cond)`, `pthread_cond_signal(&cond)`

```
pthread_mutex_t lock;

big_lock() {

    pthread_mutex_init( &lock );

    /*
    ... initial code
    */
    pthread_mutex_lock( &lock );

    /*
    ... critical section
    */
    pthread_mutex_unlock( &lock );

    /*
    ... remainder
    */
}
```

**Put code like around every  
critical section, like big\_lock**

**What if **reading** and **writing**?**



# Readers-writers using Pthreads

```
thread_ongoing_t *ongoing;  
int nr = 0, nw = 0;  
pthread_cond_t OKR, OKW;
```

Reader Thread:

```
rw.req_read();  
read ongoing  
rw.rel_read();
```

Writer Thread:

```
rw.req_write();  
modify ongoing  
rw.rel_write();
```

```
                // Initialization done elsewhere  
  
void req_read(void) {  
    while (nw > 0) pthread_cond_wait(&OKR);  
    nr++;  
    pthread_cond_signal(&OKR);  
}  
void rel_read(void) {  
    nr--;  
    if (nr == 0) pthread_cond_signal(&OKW);  
}  
void req_write(void) {  
    while (nr > 0 || nw > 0) pthread_cond_wait(&OKW);  
    nw++;  
}  
void rel_write(void) {  
    nw--;  
    pthread_cond_signal(&OKW);  
    pthread_cond_signal(&OKR);  
}
```



- Semaphores
- Classical Synchronization Problems
- Monitors
- Implementation in Pthreads

- Next time: Deadlock