Comp 310
Computer Systems and Organization

Lecture #10

Process Management

(CPU Scheduling & Synchronization)

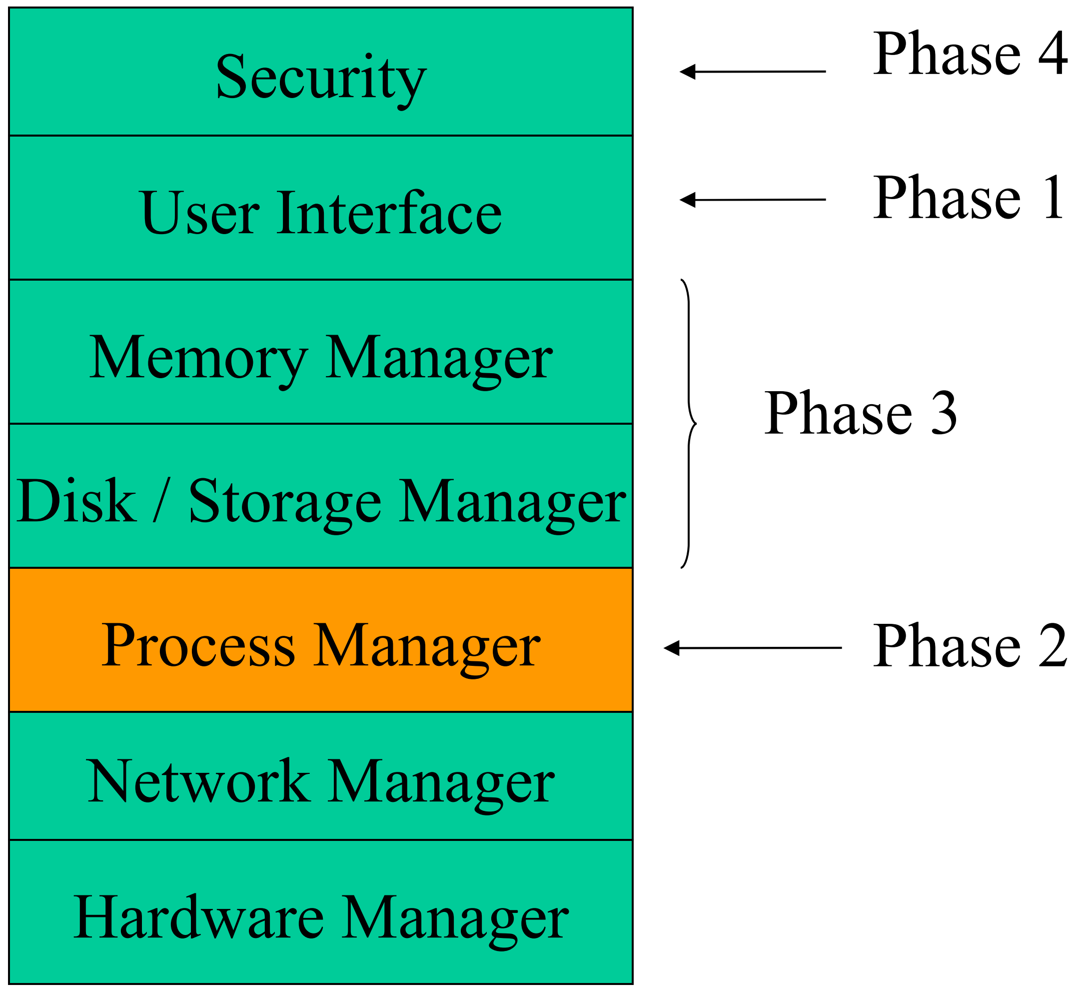
Prof. Joseph Vybihal

Announcements

- Oct 16 Midterm exam (in class)
  - In class review Oct 14 (½ class review)
  - Tutorials: TBA

Basic OS Architecture

(Course Table of Contents)

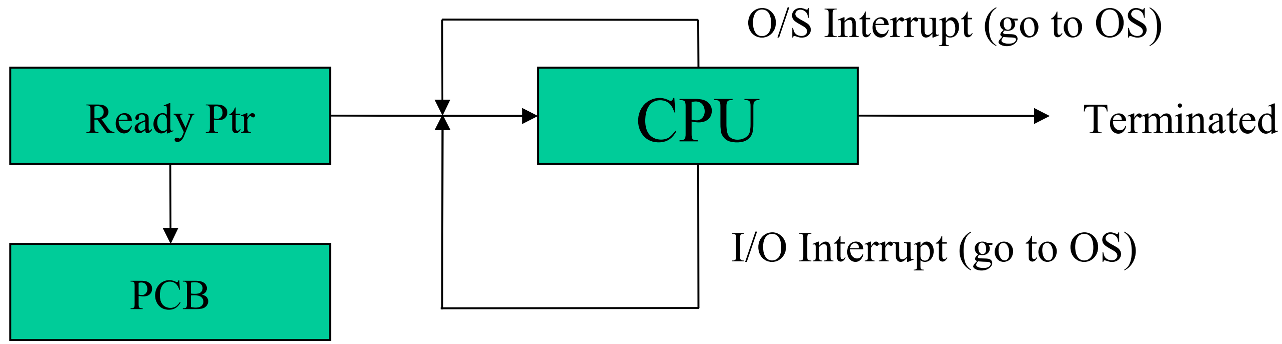


Part 1

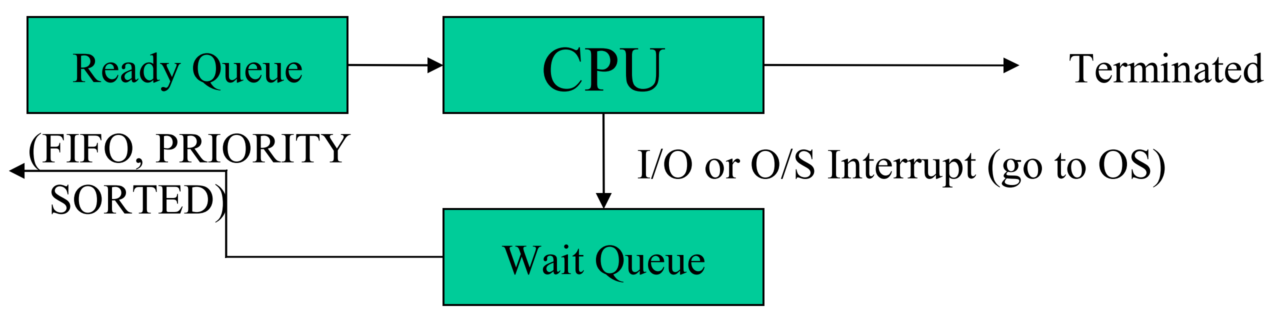
Types of CPU Scheduling

Sample Architectures

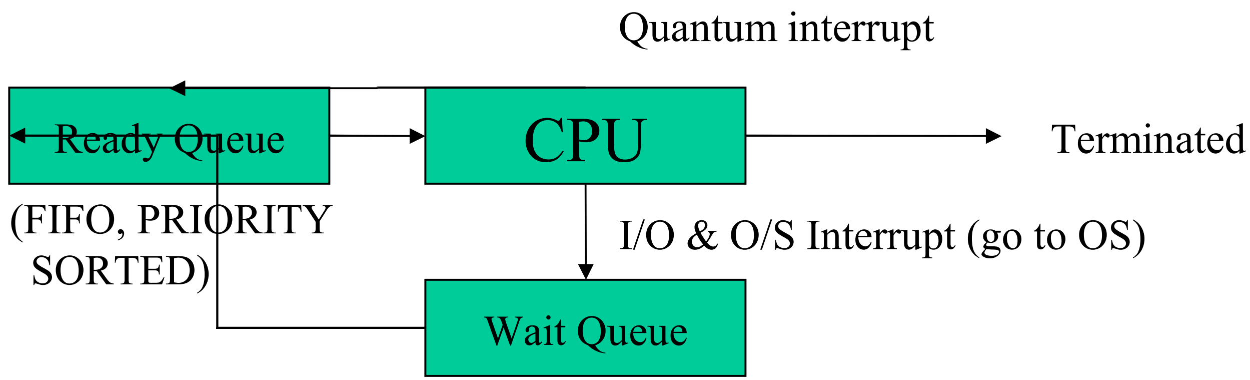
(a) Single-user Single-Process Execution



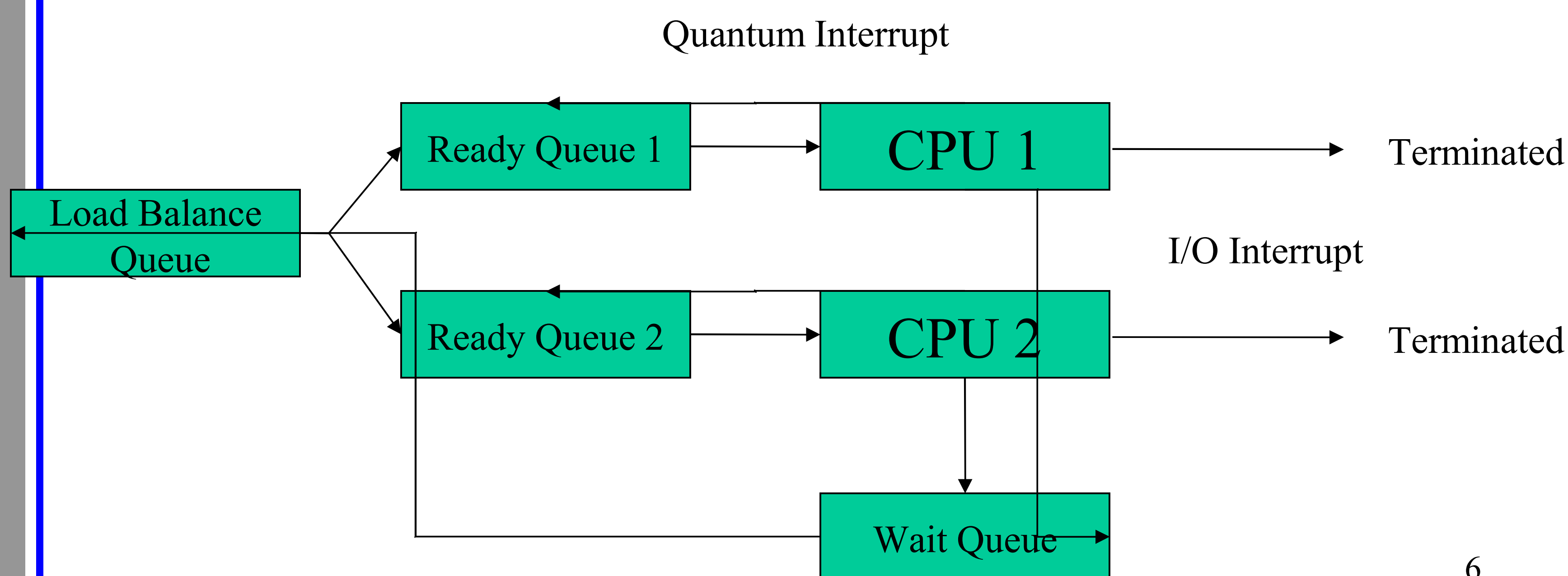
(b) Single-user Multi-process Execution



(c) Multi-tasking Execution & Multi-user (Ready queue = multilevel)



(d) Multi-user Multi-tasking Multi-Processor Execution (Multilevel ready queue)



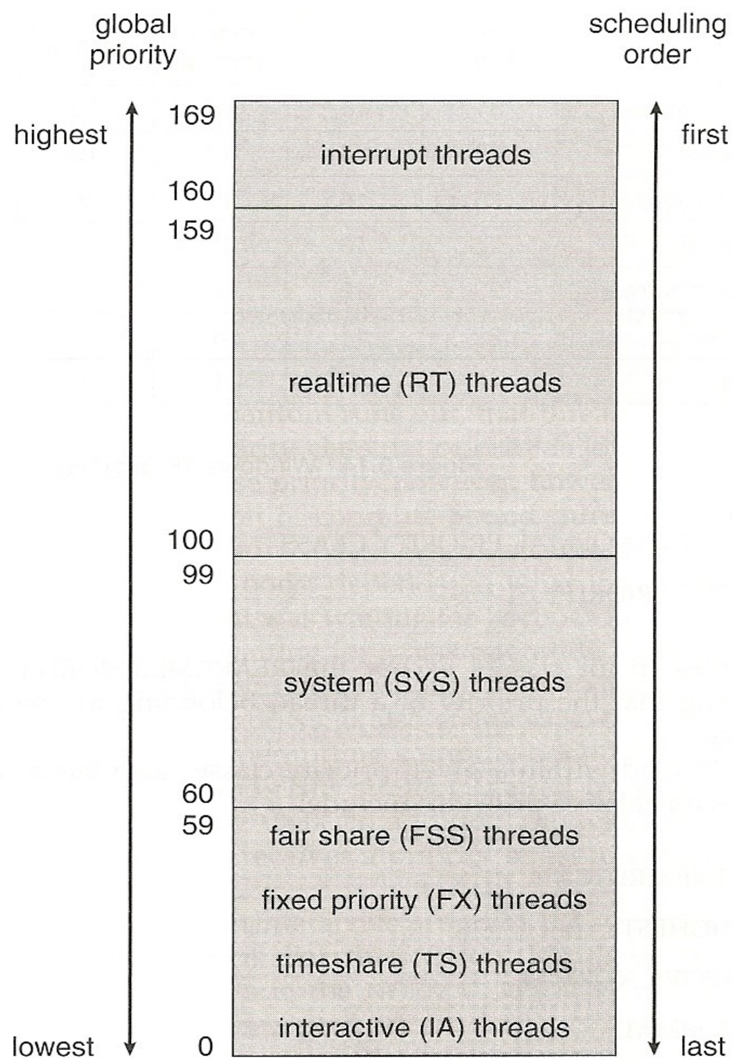
Solaris 2 Dispatch Table

High priority

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

New priorities

Solaris 2 Scheduling



Windows XP Scheduling

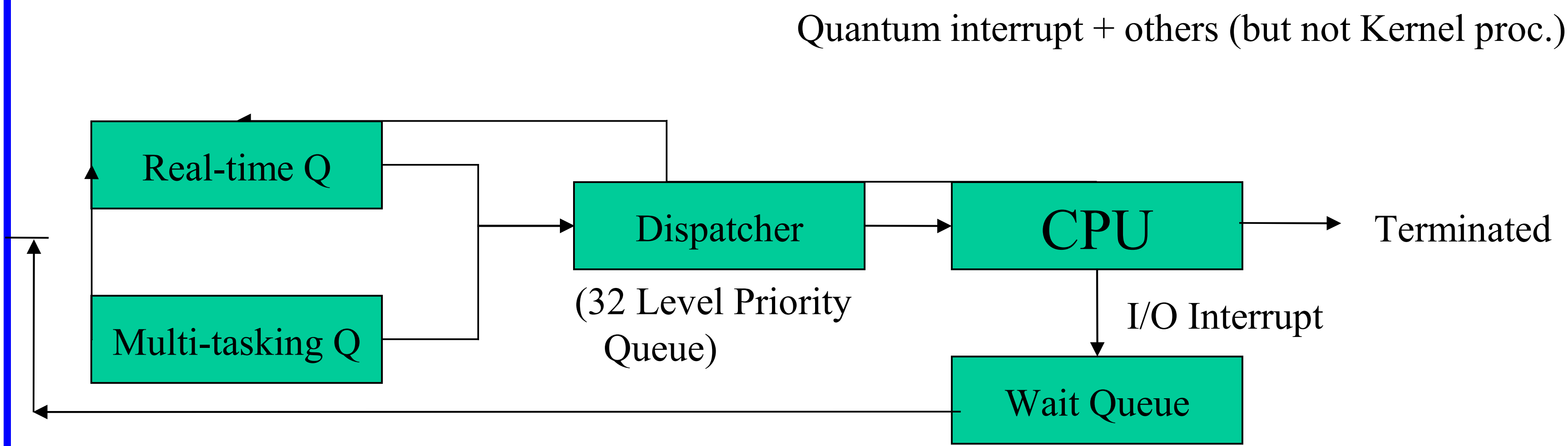
PRIORITY CLASS

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

IMPORTANCE

Standard priority sorted FIFO/RR queue

Linux (POSIX Standard)



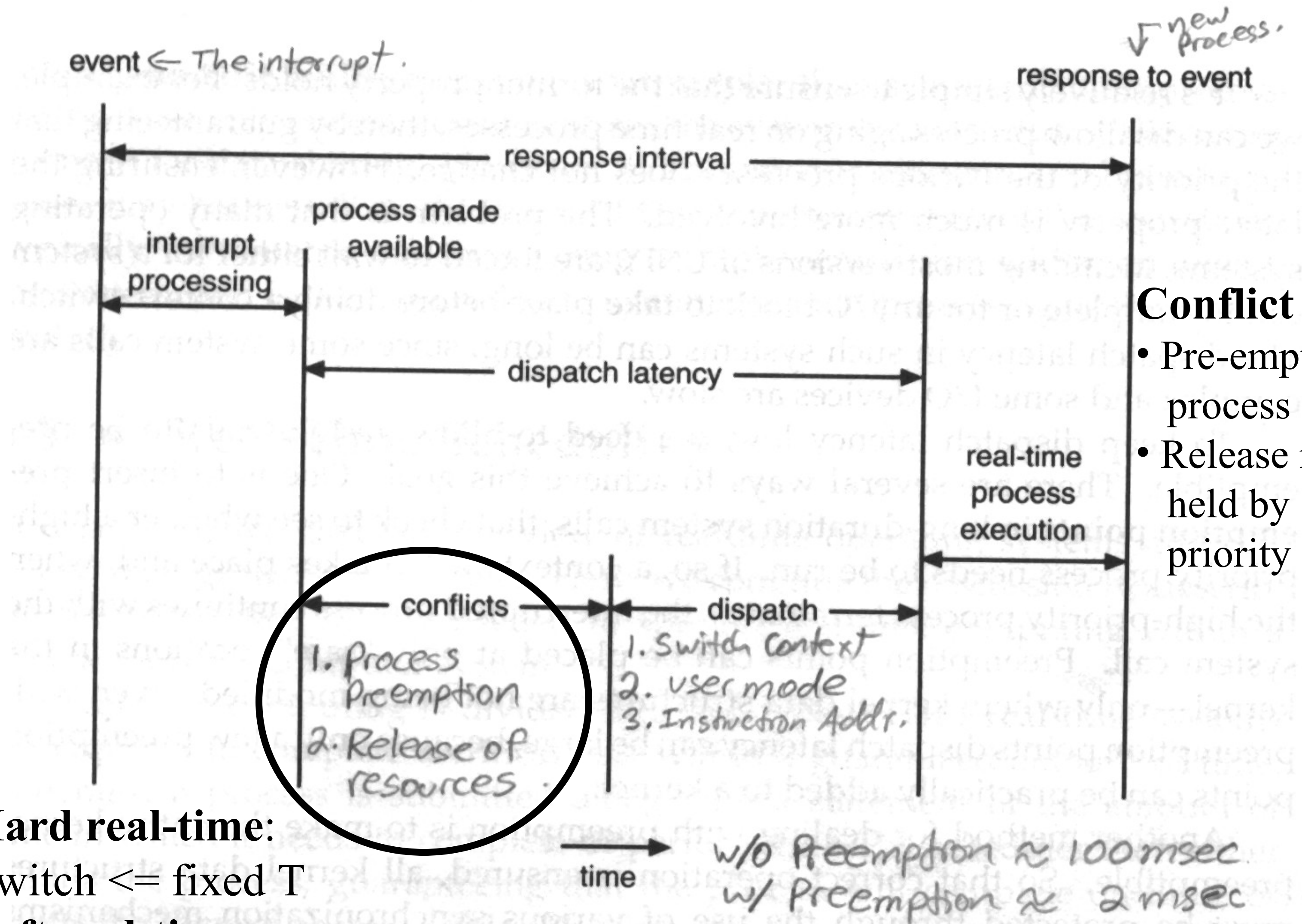
Real-time = Fixed priority queue (FIFO & RR)

Multi-tasking = Highest Credit System

Credit = (Credit / 2) + Priority

Quantum Interrupt = -1 to Credit per Q until 0 then get next process from Queue. When all Credits 0 then recalculate based on Credit formula.

The Dispatch Latency Issue



Conflict Phase:
- Pre-empt running process
- Release resources held by lower priority Pi

Hard real-time:
Switch <= fixed T
Soft real-time:
Switch to higher priority

Little's Formula

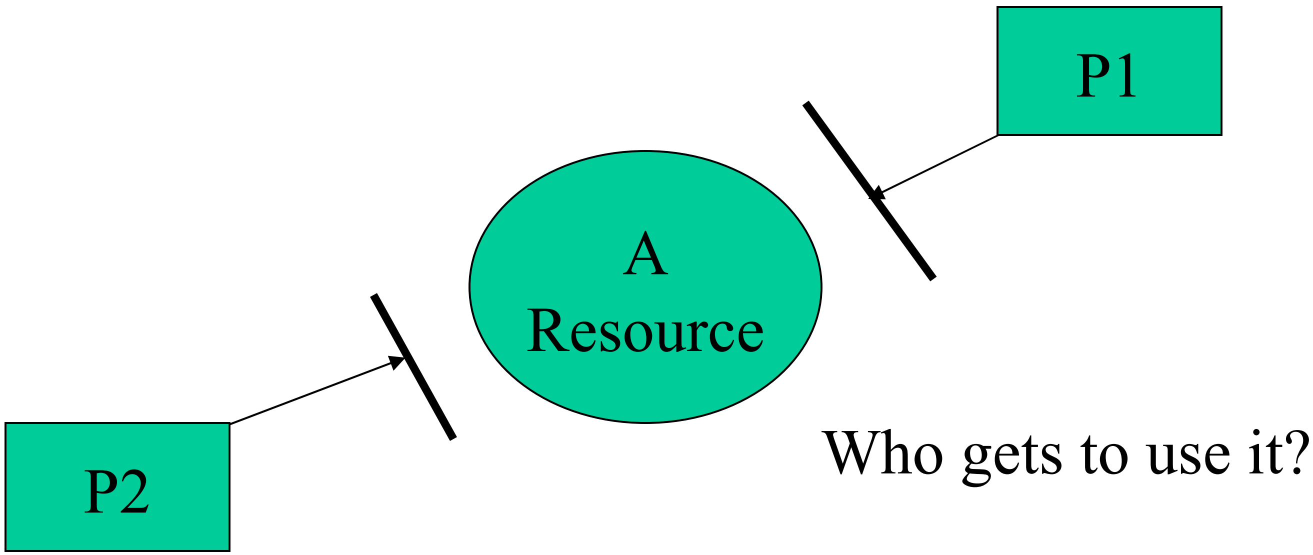
- $N = \lambda \times W$
- Where:
  - N is the length of the queue
  - λ is the average arrival rate of a new process
    - 3 processes per second
  - W is average queue waiting time
- Is the system in a **steady state**?

e.g. if $W = 5$ sec and $\lambda = 3$/sec
Then by the time Pi exists the queue 15 new processes have entered the queue

Part 2

Process Synchronization
(Accessing Resources)

The Issue



Problem: If P1 loses quanta while using resource, then what?

What resources?

- Algorithmic
  - Variables and data structures used to manage by OS
- Physical
  - Files, disk drives, printers, etc.

Example

Producer

```
while(1) {
   while (ctr = = BUF_SIZE);
   buf[in] = nextValue;
   in = (in + 1) % BUF_SIZE;
   ctr++;
}
```

Consumer

```
while (1) {
   while (ctr = = 0);
   nextValue = buf[out];
   out = (out + 1) %
BUF_SIZE;
   ctr--;
}
```

Concurrent modification of shared variable ctr!

↑

The resource

<u>NOTE</u>:

ctr++ in assembler:

```
move reg, ctr
incr reg
move ctr, reg
```

ctr-- in assembler:

```
move reg, ctr
decr reg
move ctr, reg
```

In Concurrent mode this is possible:

CTR starts at 5 and producer creates 1 while consumer uses 1, should stay as 5

T0: producer - move reg, ctr { reg = 5 }
T1: producer - incr reg { reg = 6 }
T2: Task switch
T3: consumer- move reg, ctr { reg = 5 }
T4: consumer- decr reg { reg = 4 }
T5: task switch
T6: producer - move ctr, reg { ctr = 6 }
T7: Task switch
T8: consumer- move ctr, reg { ctr = 4 }

How can we control this!

The Critical Section Problem

```
do {
    initial code;
        Entry section
    critical section;
        Exit section
    remaining code;
} while (1);
```

Does not use shared resources

A shared resource

Entry & Exit code guard the critical section:

- Mutual Exclusion: Only 1 Pi can be in the critical section (regardless of quanta)
- Progress: Entry queues requests to use critical section
- Bounded Waiting: Indefinite postponement is not permitted

The 2 Process Solution

```
do {
    initial section;
Shared vars {
    flag[i] = true;                      // indicates i wants to enter
    turn = j;                            // does j want to enter?
    while (flag[j] && turn == j);        // controls who enters
    critical section;
    flag[i] = false;                     // I'm done, says Pi
    remaining section;
} while(1);
```

THIS IS PROCESS Pi

Multi-process Solution

```
do {
    initial section;
    choosing[i] = true;        ← i wants a waiting number
    number[i] = max(num[0], num[1],…, num[n-1])+1;  ← bigger num
    choosing[i] = false;
    for(j=0; j < n; j++) {   ← FIFO ACCESS
      while (choosing[j]); ← Wait if someone getting a number
      while (num[j]!=0 && num[j]<= num[i] && j<i);
    }
    critical section;
      number[i] = 0;
    remainder section;
} while(1);
```

shared (the brace marks the first two lines of the boxed block: choosing[i] = true; and number[i] = max(...)+1;)

Don't want to go in (the arrows point to num[j]!=0 in the second while loop and to number[i] = 0;)

Hardware Solution: Atomic Instructions

ONLY one program can execute this instruction at any "clock tick". It executes in one CPU operation.

```
boolean TestAndSet(boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```

```
void Swap(boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Mutual-Exclusion Examples

```
do {
    initial section;
        while (TestAndSet(lock));
    critical section;
        lock = false;
    remainder section;
} while(1);
```

Common Structures:

- lock

```
do {
    initial section;
        key = true;
        while (key == true)
            Swap(lock, key);

    critical section;
        lock = false;
    remainder section;
} while(1);
```

Common Structures:

- waiting[n]
- lock

Bounded-waiting with TestAndSet

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = TestAndSet(lock);
    waiting[i] = false;

        critical section

    j = (i+1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;

        remainder section

} while (1);
```

- waiting[i] = true; key = true; ← i wants to enter
- key = TestAndSet(lock); → {key = lock, lock = true} ∴ if lock = true? if lock = false?
- TestAndSet ← ATOMIC; lock ← shared
- I'm in! → waiting[i] = false;
- get adjacent → j = (i+1) % n;
- scan all adjacent → while ((j != i) && !waiting[j]) ← Find the next process who wants to enter (FIFO).
- lock = false; ← no one wants it
- waiting[j] = false; ← j can get it.

Part 3

Semaphores

Basic Definition

```
wait(S) {
        while (S < 0); // spinlock
        S--;
}

signal(S) {
        S++;
}
```

Controls # who can get past (refers to S in while (S < 0))

S is a shared integer variable initialized to 1.

```
do {
    initial code;
        wait(mutex);
    critical section;
        signal(mutex);
    remaining code;
} while (1);
```

Problems to avoid

- Deadlock
  - Pi has resource Q and wants resource R
  - Pj has resource R and wants resource Q
- Indefinite Postponement (starvation)
  - Deadlock forever

Practical Uses

- Memory Buffers (Bounding Buffer problem)
- Shared Files / Vars (Readers & Writers Problem)
- Limited Resources Many Processes (Dining Philosophers Problem

(Next class)

Part 4

Monitors

Semaphore Queue Implementation

```
typedef struct {
        int val; // val=1 to start
        struct PROCESS *q;
} semaphore;
```

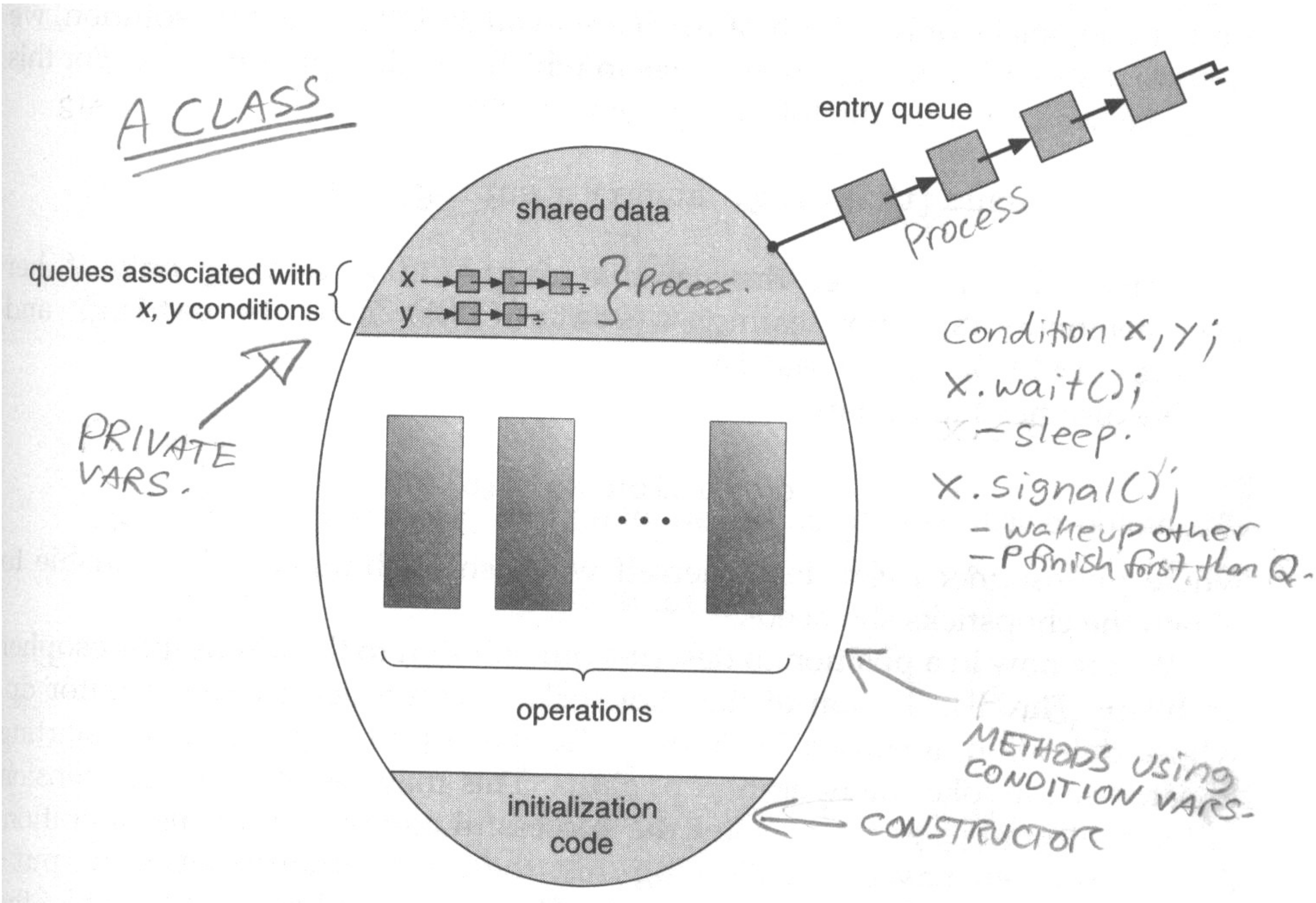
```
void wait(semaphore S) {
        S.val--;

        if (S.val < 0) // must wait
        {
            tail(process,q);
            block(); // sleep
        }
}
```

```
void signal(semaphore S) {
    S.val++;

    if (S.val <= 0)
    { // give access
          p = head(q);
          wakeup(p);
    }
```

Abstract View



One process it reduces to a standard semaphore.

Question

- How could we implement a monitor using object?

Part 5

At Home

Things to try out

- Try to implement a two process synchronization problem using C.

2. Web Resources (Monitors & Threads):
    1. http://msdn2.microsoft.com/en-us/library/aa645740(vs.71).aspx