Operating Systems

Operating Systems

Winter 2018

Sina Meraji

U of T

More Special Instructions

- *Swap* (or *Exchange*) instruction
  - Operates on two words atomically
  - Can also be used to solve critical section problem
- Machine instructions have three problems:
  - Busy waiting

Higher-level Abstractions for CS's

- Locks
  - Very primitive, minimal semantics
  - Operations: acquire(lock), release(lock)
- Semaphores
  - Basic, easy to understand, hard to program with
- Monitors
  - High-level, ideally has language support (Java)
- Messages
  - Simple model for communication & synchronization
  - Direct application to distributed systems

Producer and Consumer

- Two processes share a bounded buffer
- The producer puts info in buffer
- The consumer takes info out

- Solution
  - Sleep: Cause caller to block
  - Wakeup: Awaken a process

The Producer-Consumer Problem

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count − 1;                    /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

The producer-consumer

What happens if Cons. wakes up the Prod. before it really sleeps

Semaphores

- Semaphores are abstract data types that provide synchronization. They include:
  - An integer variable, accessed only through 2 atomic operations
  - The atomic operation *wait* (also called *P* or *decrement*) - decrement the variable and block until semaphore is free
  - The atomic operation *signal* (also called *V* or *increment*) - increment the variable, unblock a waiting a thread if there are any
  - A queue of waiting threads

Types of Semaphores

- Mutex (or Binary) Semaphore
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore
  - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
  - Multiple threads can pass the semaphore
  - Max number of threads is determined by semaphore’s initial value, *count*
    - Mutex has *count = 1,* counting has *count = N*

Semaphores

- Integer variable `count` with two **atomic** operations
  - Operation *wait* (also called *P* or *decrement*)
    - block until `count` > 0 then decrement variable

```
wait(semaphore *s) {
        while (s->count == 0) ;
        s->count -= 1;
}
```

  - Operation *signal* (also called *V* or *increment*)
    - increment `count`, unblock a waiting thread if any

```
signal(semaphore *s) {
        s->count += 1;
        ......               //unblock one waiter
}
```

  - A queue of waiting threads

Using Binary Semaphores

- Use is similar to locks, but semantics are different

Have semaphore, S, associated with acct

```
typedef struct account {
    double balance;
    semaphore S;
} account_t;

Withdraw(account_t *acct, amt){
    double bal;
    wait(acct->S);
    bal = acct->balance;
    bal = bal - amt;
    acct->balance = bal;
    signal(acct->S);
    return bal;
}
```

Three threads execute Withdraw()

```
wait(S);
bal = acct->balance;
bal = bal - amt;
```

```
wait(acct->S);
```

```
wait(acct->S);
```

```
acct->balance = bal;
signal(acct->S);
```

```
...
signal(acct->S);
```

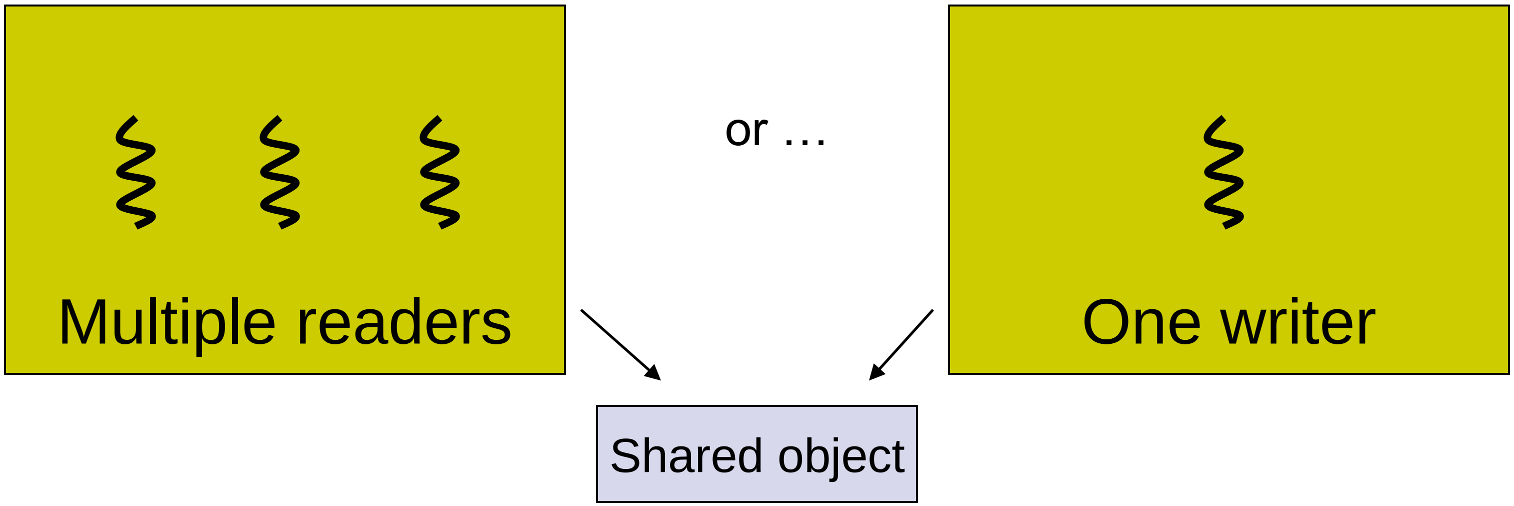
```
...
signal(acct->S);
```

It is **undefined** which thread runs after a **signal**

Atomicity of wait() and signal()

- We must ensure that two threads cannot execute *wait* and *signal* at the same time
- This is another critical section problem!
  - Use lower-level primitives
  - Uniprocessor: disable interrupts
  - Multiprocessor: use hardware instructions

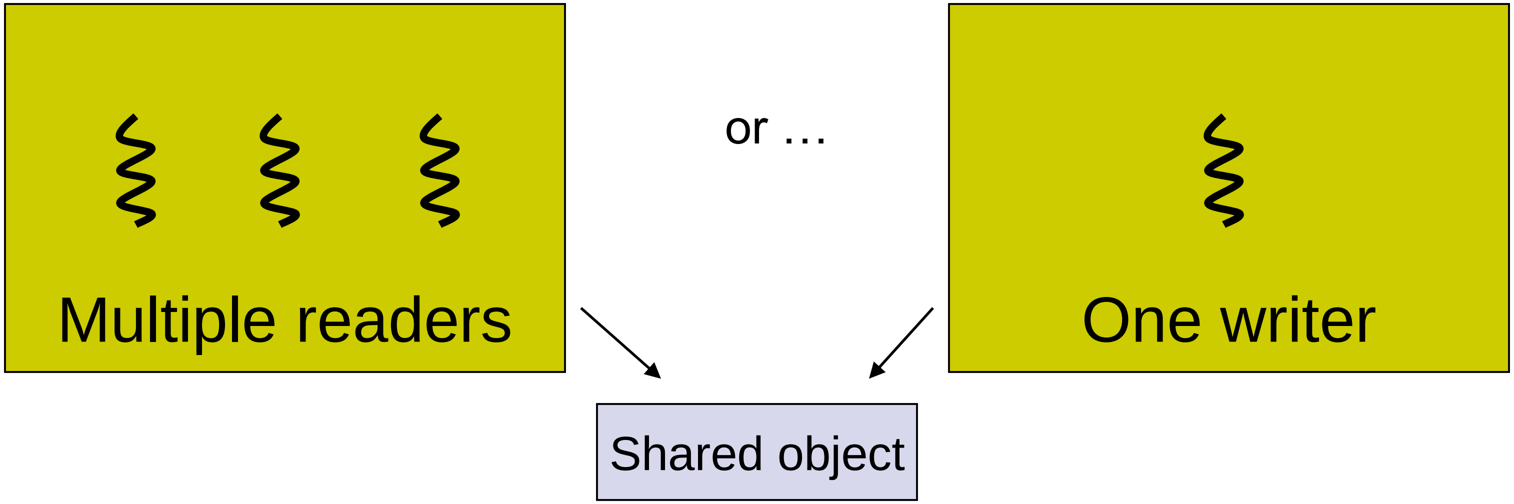
The readers/writers problem



- An object is shared among several threads
- Some only read the object, others only write it
- We can allow multiple concurrent *readers*
- But only one *writer*

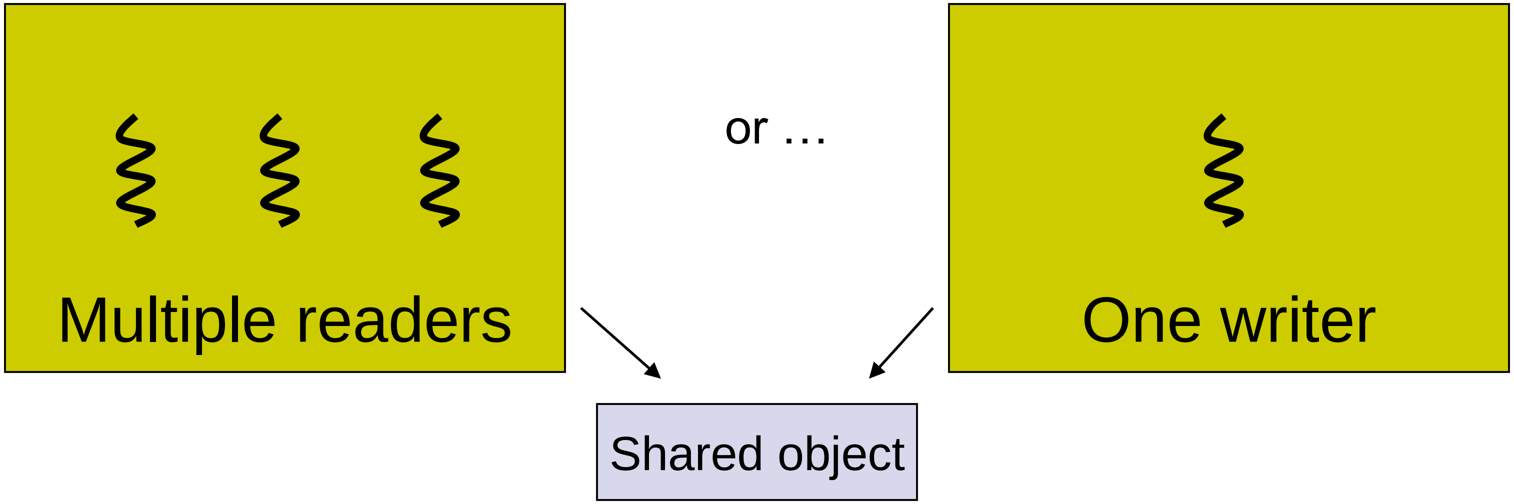
•*How can we implement this with semaphores?*

The readers/writers problem



- Use three variables
  - Semaphore **w_or_r** - exclusive writing or reading
    - Think of it as a token that can be held either by the group of readers or by one individual writer.
    - Which thread in the group of readers is in charge of getting and returning the token?
    - *“Last to leave the room turns off the light”*

The readers/writers problem



- Use three variables
  - Semaphore **w_or_r** - exclusive writing or reading
  - int **readcount** - number of threads reading object
    - Needed to detect when a reader is the first or last of a group.
  - Semaphore **mutex** - control access to readcount

Writer's operation:

```
//number of readers
int readcount = 0;
//mutual exclusion to readcount
Semaphore mutex = 1;
//exclusive writer or reading
Semaphore w_or_r = 1;

Writer {
   wait(w_or_r);  //lock out others
   Write;
   signal(w_or_r); //up for grabs
}
```

Reader’s operation:

```
Reader {
   wait(mutex); //lock readcount
   // one more reader
   readcount += 1;
```

• Update read_count

Reader's operation:

```
Reader {
   wait(mutex); //lock readcount
   // one more reader
   readcount += 1;
   // is this the first reader?
   if(readcount == 1)
        //synch w/ writers
        wait(w_or_r);
   //unlock readcount
   signal(mutex);
   Read;
```

- Update read_count
- Am I the first reader? => decrement w_or_r

Reader's operation:

```
Reader {
   wait(mutex); //lock readcount
   // one more reader
   readcount += 1;
   // is this the first reader?
   if(readcount == 1)
        //synch w/ writers
        wait(w_or_r);
   //unlock readcount
   signal(mutex);
   Read;
   wait(mutex); //lock readcount
   readcount -= 1;
   if(readcount == 0)
        signal(w_or_r);
   signal(mutex);
}
```

- Update read_count
- Am I the first reader? => decrement w_or_r

- Update read_count
- Am I the last reader? => increment w_or_r

Reader's and writers operation:

```
//number of readers
int readcount = 0;
//mutual exclusion to readcount
Semaphore mutex = 1;
//exclusive writer or reading
Semaphore w_or_r = 1;

Writer {
   wait(w_or_r);  //lock out others
   Write;
   signal(w_or_r); //up for grabs
}
```

```
Reader {
   wait(mutex); //lock readcount
   // one more reader
   readcount += 1;
   // is this the first reader?
   if(readcount == 1)
        //synch w/ writers
        wait(w_or_r);
   //unlock readcount
   signal(mutex);
   Read;
   wait(mutex); //lock readcount
   readcount -= 1;
   if(readcount == 0)
        signal(w_or_r);
   signal(mutex);
}
```

Suppose I'm the first reader arriving while writer is active. What happens?

Reader's and writers operation:

```
//number of readers
int readcount = 0;
//mutual exclusion to readcount
Semaphore mutex = 1;
//exclusive writer or reading
Semaphore w_or_r = 1;

Writer {
   wait(w_or_r);  //lock out others
   Write;
   signal(w_or_r); //up for grabs
}
```

```
Reader {
   wait(mutex); //lock readcount
   // one more reader
   readcount += 1;
   // is this the first reader?
   if(readcount == 1)
        //synch w/ writers
        wait(w_or_r);
   //unlock readcount
   signal(mutex);
   Read;
   wait(mutex); //lock readcount
   readcount -= 1;
   if(readcount == 0)
        signal(w_or_r);
   signal(mutex);
}
```

Suppose I'm the second reader arriving while writer is active. What happens?

Reader's and writers operation:

```
//number of readers
int readcount = 0;
//mutual exclusion to readcount
Semaphore mutex = 1;
//exclusive writer or reading
Semaphore w_or_r = 1;

Writer {
   wait(w_or_r);  //lock out others
   Write;
   signal(w_or_r); //up for grabs
}
```

```
Reader {
   wait(mutex); //lock readcount
   // one more reader
   readcount += 1;
   // is this the first reader?
   if(readcount == 1)
        //synch w/ writers
        wait(w_or_r);
   //unlock readcount
   signal(mutex);
   Read;
   wait(mutex); //lock readcount
   readcount -= 1;
   if(readcount == 0)
        signal(w_or_r);
   signal(mutex);
}
```

Once the writer exits, which reader gets to go first?

Reader's and writers operation:

```
//number of readers
int readcount = 0;
//mutual exclusion to readcount
Semaphore mutex = 1;
//exclusive writer or reading
Semaphore w_or_r = 1;

Writer {
   wait(w_or_r);  //lock out others
   Write;
   signal(w_or_r); //up for grabs
}
```

```
Reader {
   wait(mutex); //lock readcount
   // one more reader
   readcount += 1;
   // is this the first reader?
   if(readcount == 1)
        //synch w/ writers
        wait(w_or_r);
   //unlock readcount
   signal(mutex);
   Read;
   wait(mutex); //lock readcount
   readcount -= 1;
   if(readcount == 0)
        signal(w_or_r);
   signal(mutex);
}
```

If both readers and writers are waiting, once the writer exits, who goes first?

Notes on Readers/Writers

- If there is a writer
  - First reader blocks on w_or_r
  - All other readers block on mutex
- Once a writer exits, all readers can proceed
  - Which reader gets to go first?
- The last reader to exit signals a waiting writer
  - If no writer, then readers can continue
- If readers and writers are waiting on w_or_r, and a writer exits, who goes first?
  - Depends on the scheduler

Higher-level Abstractions for CS's

- Locks
  - Very primitive, minimal semantics
  - Operations: acquire(lock), release(lock)
- Semaphores
  - Basic, easy to understand, hard to program with
- Monitors
  - High-level, ideally has language support (Java)
- Messages
  - Simple model for communication & synchronization
  - Direct application to distributed systems

Motivation for monitors

- It's easy to make mistakes with semaphores

```
Writer {
   wait(w_or_r);
   Write;
   wait(w_or_r);
}
```

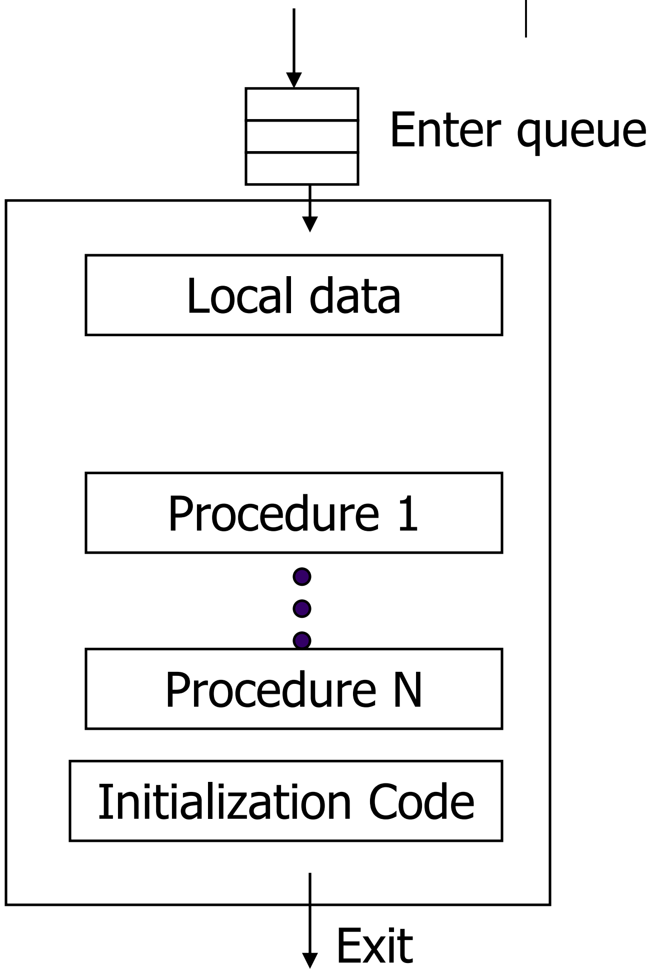
```
Writer {
   signal(w_or_r);
   Write;
   signal(w_or_r);
}
```

Monitors

- Similar in a sense to an *abstract data type* (data and operations on the data) with the restriction that only one process at a time can be active within the monitor
  - Local data accessed only by the monitor's procedures (not by any external procedure)
  - A process *enters* the monitor by invoking 1 of its procdures
  - Other processes that attempt to enter monitor are blocked
- A process in the monitor may need to wait for something to happen
  - May need to allow another process to use the monitor
  - provide a *condition* type for variables with operations
    - *wait* (suspend the invoking process)
    - *signal* (resume exactly one suspended process)

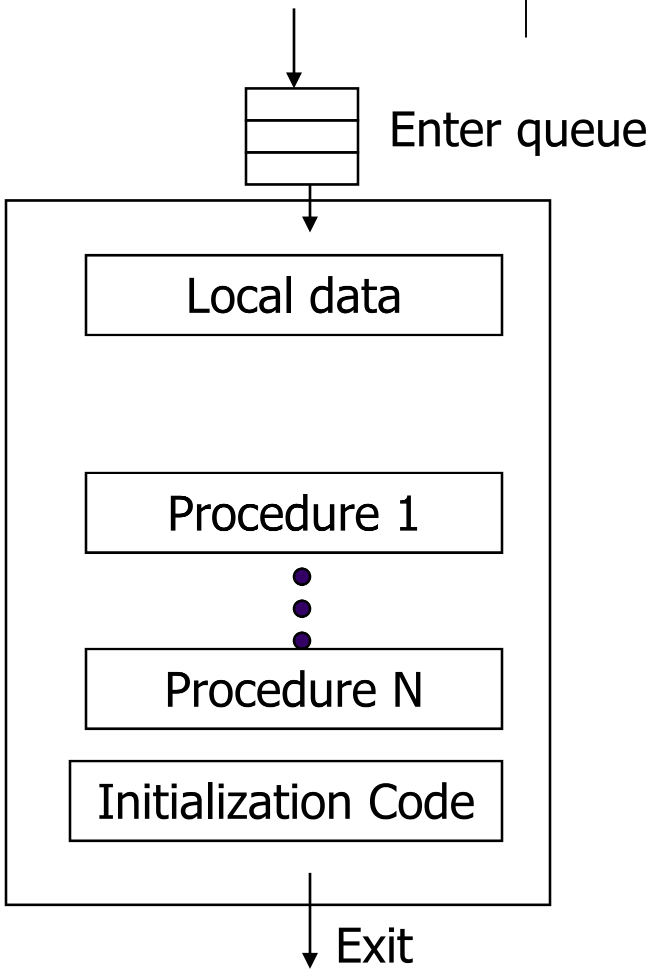
Monitor Diagram

- An *abstract data type:* with restriction that **only one process** at a time can be **active** within the monitor
  - Local data accessed only by monitor’s procedures
  - Process *enters* monitor by invoking 1 of its procedures
  - Other processes that attempt to enter monitor are blocked



Bank example with monitors

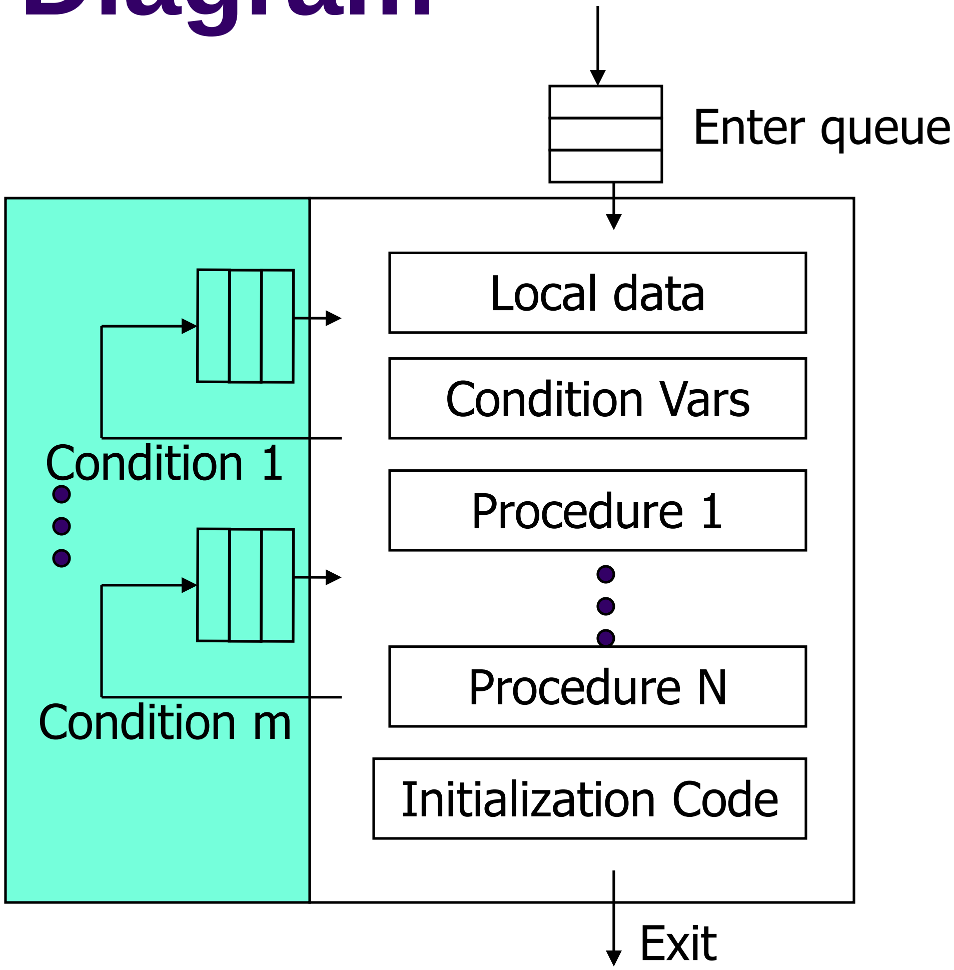
```
Monitor Account {
      int balance;

      void withdraw(int amount){
            balance -= amount;
      }
      void deposit (int amount){
            balance += amount;
      }
      ….
}
```



Enforcing single access

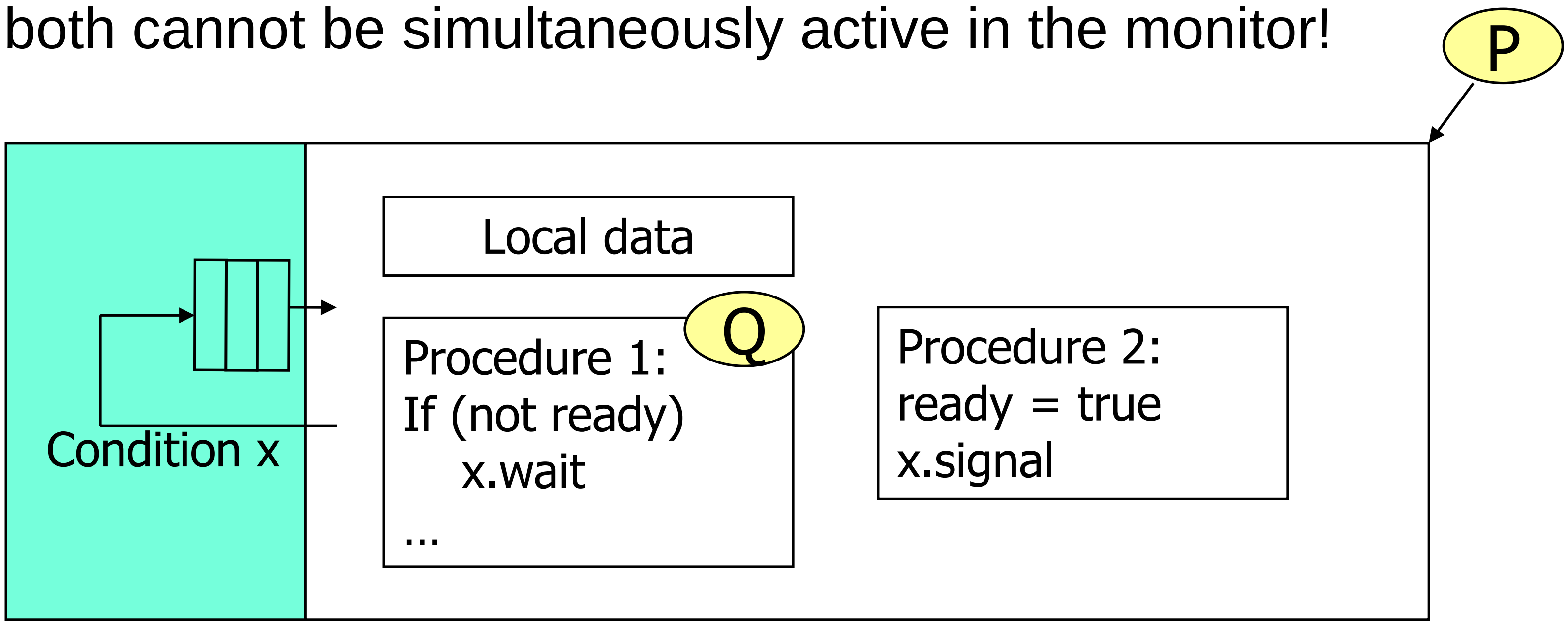
- A process in the monitor may need to wait for something to happen
  - May need to let other process use the monitor
  - Provide a special type of variable called a *condition*
  - Operations on a *condition* variable are:
    - *wait* (suspend the invoking process)
    - *signal* (resume exactly one suspended process)
      - if no process is suspended, a *signal* has no effect
    - How does that differ from Semaphore’s wait & signal?

Monitor Diagram



More on Monitors

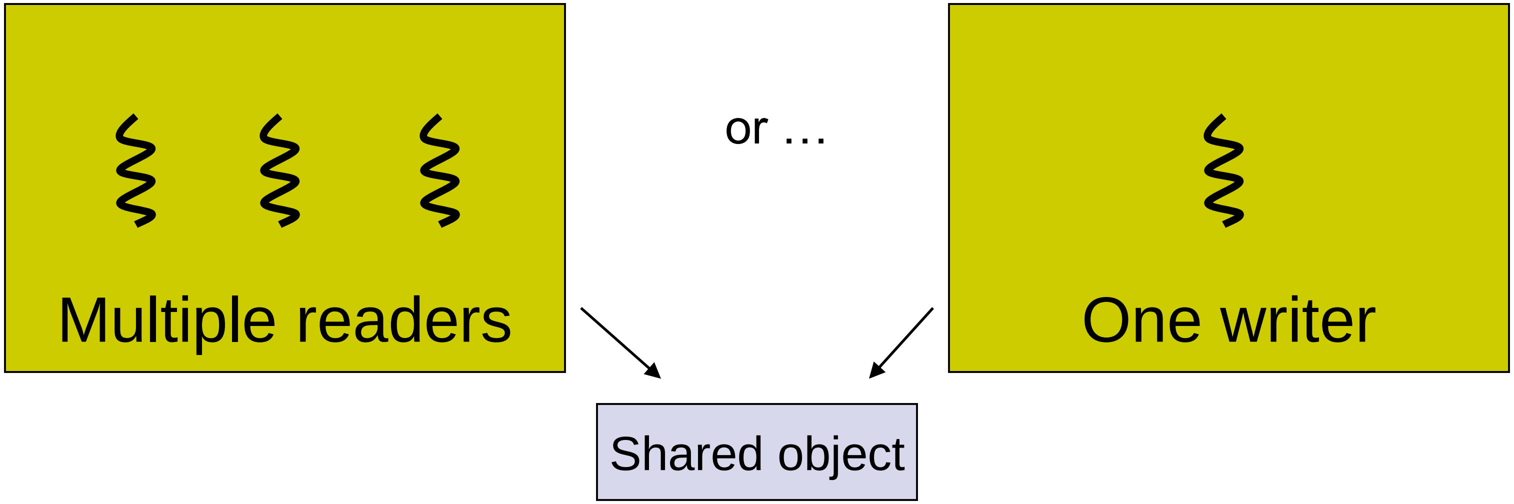
- If process *P* executes an *x.signal* operation and ∃ a process *Q* waiting on condition *x*, we have a problem:
  - P is already “in the monitor”, does not need to block
  - Q becomes unblocked by the signal, and wants to resume execution in the monitor
  - But both cannot be simultaneously active in the monitor!



Monitor Semantics for Signal

- **Hoare monitors**
  - Signal() immediately switches from the caller to a waiting thread
  - Need another queue for the signaler, if signaler was not done using the monitor
- **Brinch Hansen**
  - Signaler must exit monitor immediately
    - i.e. signal() is always the last statement in monitor procedure
- **Mesa monitors**
  - Signal() places a waiter on the ready queue, but signaler continues inside monitor

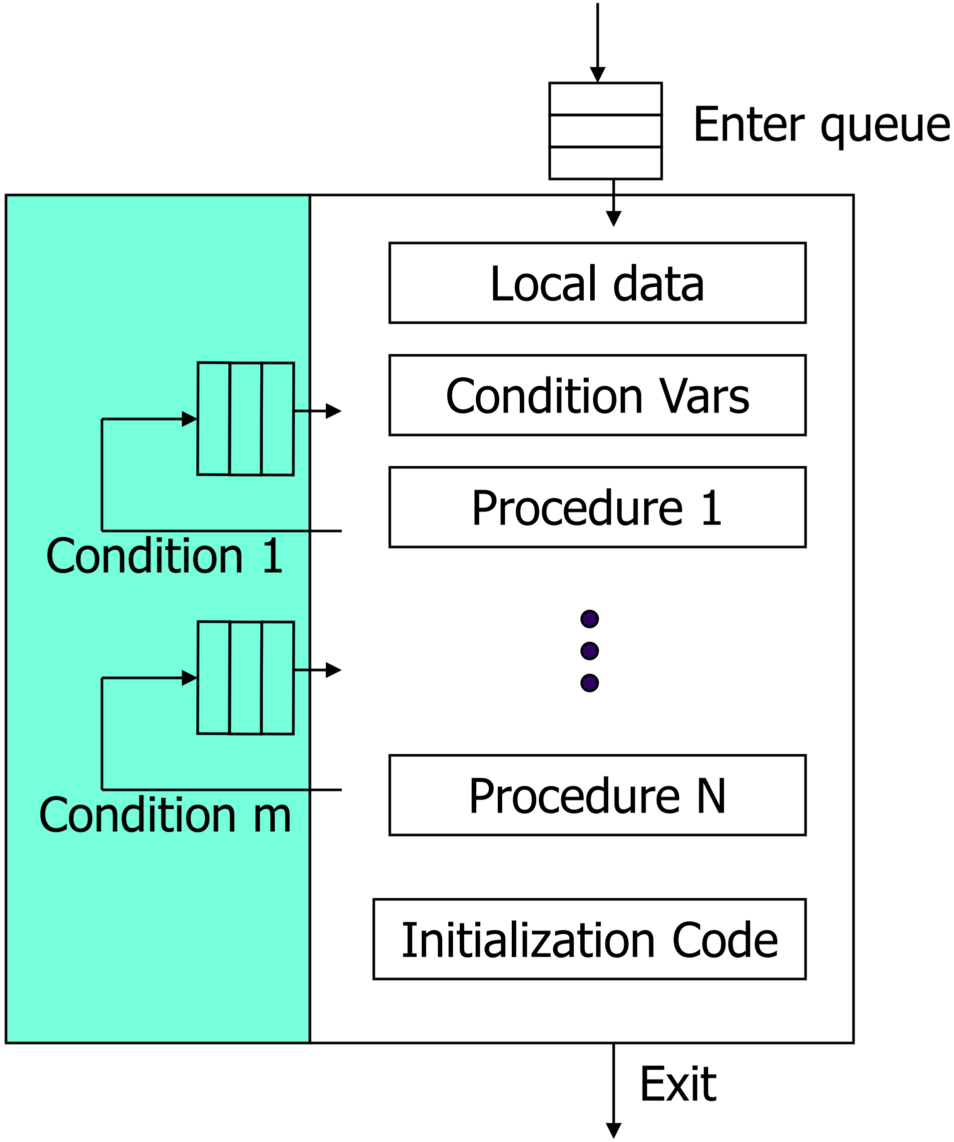
The readers/writers problem



- An object is shared among several threads
- Some only read the object, others only write it
- We can allow multiple concurrent *readers*
- But only one *writer*

•*How can we implement this with* ***monitors****?*

Monitor for readers/writers



Using Monitors in C

- Not integrated with the language (as in Java)
- Bounded buffer: Want a monitor to control access to a buffer of limited size, N
  - Producers add to the buffer if it is not full
  - Consumers remove from the buffer if it is not empty
- Need two functions – add_to_buffer() and remove_from_buffer()
- Need one lock – only lock holder is allowed to be active in one of the monitor's functions
- Need two conditions – one to make producers wait, one to make consumers wait

Bounded Buffer Monitor – Variables

```
#define N 100
typedef struct buf_s {
   int data[N];
   int inpos; /* producer inserts here */
   int outpos; /* consumer removes from here */
   int numelements; /* # items in buffer */
   } buf_t;

buf_t buf;    //Do proper initialization
void add_to_buff(int value);
int remove_from_buff();
```

Bounded Buffer: The Producer thread (no synchronization)

```
void add_to_buf(int value) {

    while (buf.nelements == N) {
      /* buffer is full, wait */
      /* implement wait here  */
    }
    buf.data[buf.inpos] = value;
    buf.inpos = (buf.inpos + 1) % N;
    buf.nelements++;

    /* Make sure that potentially      */
    /* waiting consumers are notified */
}
```

Bounded Buffer: The Consumer thread (no synchronization)

```
int remove_from_buf() {
  int val;

    while (buf.nelements == 0) {
      /* buffer is empty, wait */
      /* implement wait here   */
    }
    val = buf.data[buf.outpos];
    buf.outpos = (buf.outpos + 1) % N;
    buf.nelements--;

    /* Make sure that potentially */
    /* waiting producers are notified */

   return val;
}
```

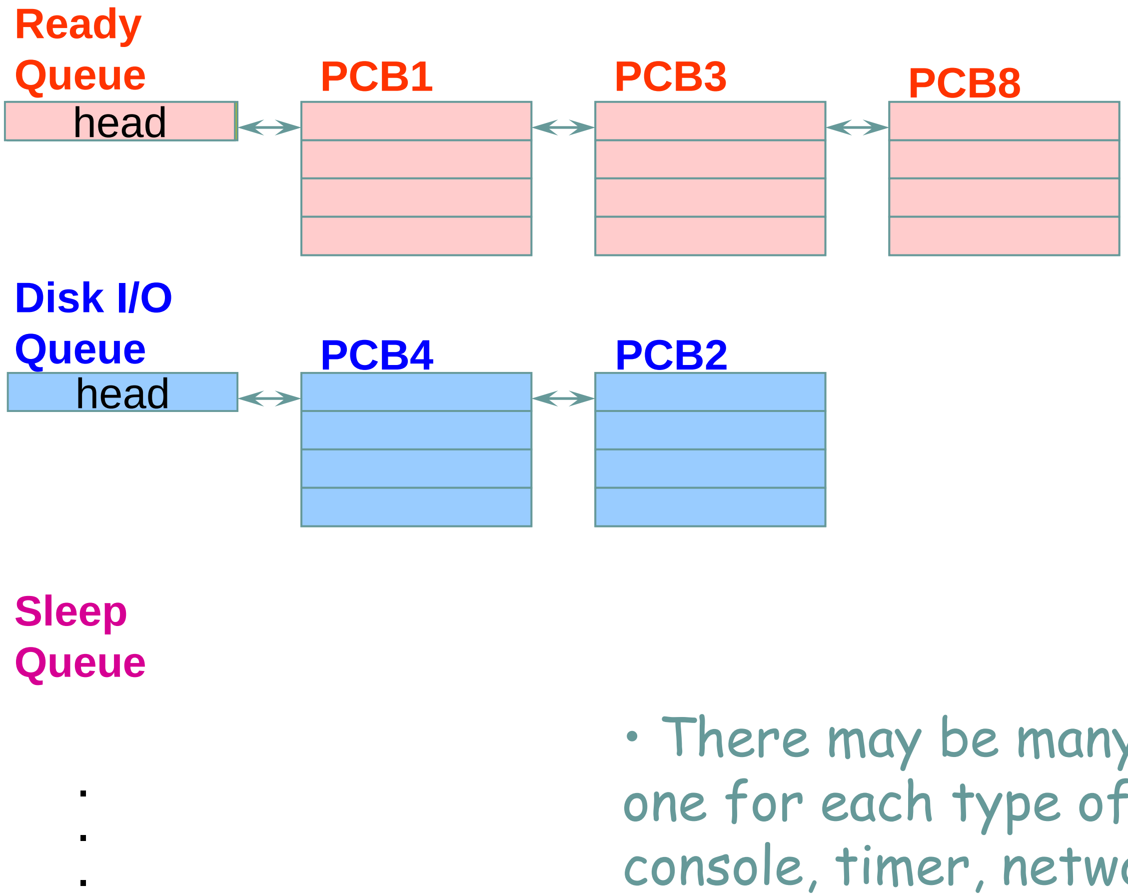
Solution in pthreads….

```
void add_to_buf(int value) {
 pthread_mutex_lock(buf.mylock);
    while (buf.nelements == N) {
      /* buffer is full, wait */
    pthread_cond_wait(
   buf.notFull, buf.mylock);
    }
    buf.data[buf.inpos] = value;
    buf.inpos = (buf.inpos + 1)%N;
    buf.nelements++;
    pthread_cond_signal(
    buf.notEmpty);
    pthread_mutex_release(
        buf.mylock);
}
```

```
int remove_from_buf() {
  int val;
  pthread_mutex_lock(buf.mylock);
    while (buf.nelements == 0) {
      /* buffer is empty, wait */
      pthread_cond_wait(buf.notEmpty,
buf.mylock);
    }
    val = buf.data[buf.outpos];
    buf.outpos = (buf.outpos + 1)%N;
    buf.nelements—;
    pthread_cond_signal(buf.notFull);

pthread_mutex_release(buf.mylock);
   return val;
}
```

Next: Process Scheduling

State Queues



- There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)

Process Scheduling

- Only one process can run at a time on a CPU
- Scheduler decides which process to run
- Goal of CPU scheduling:
  - Give illusion that processes are running concurrently
  - Maximize CPU utilization

- Will talk about CPU scheduling in more detail …

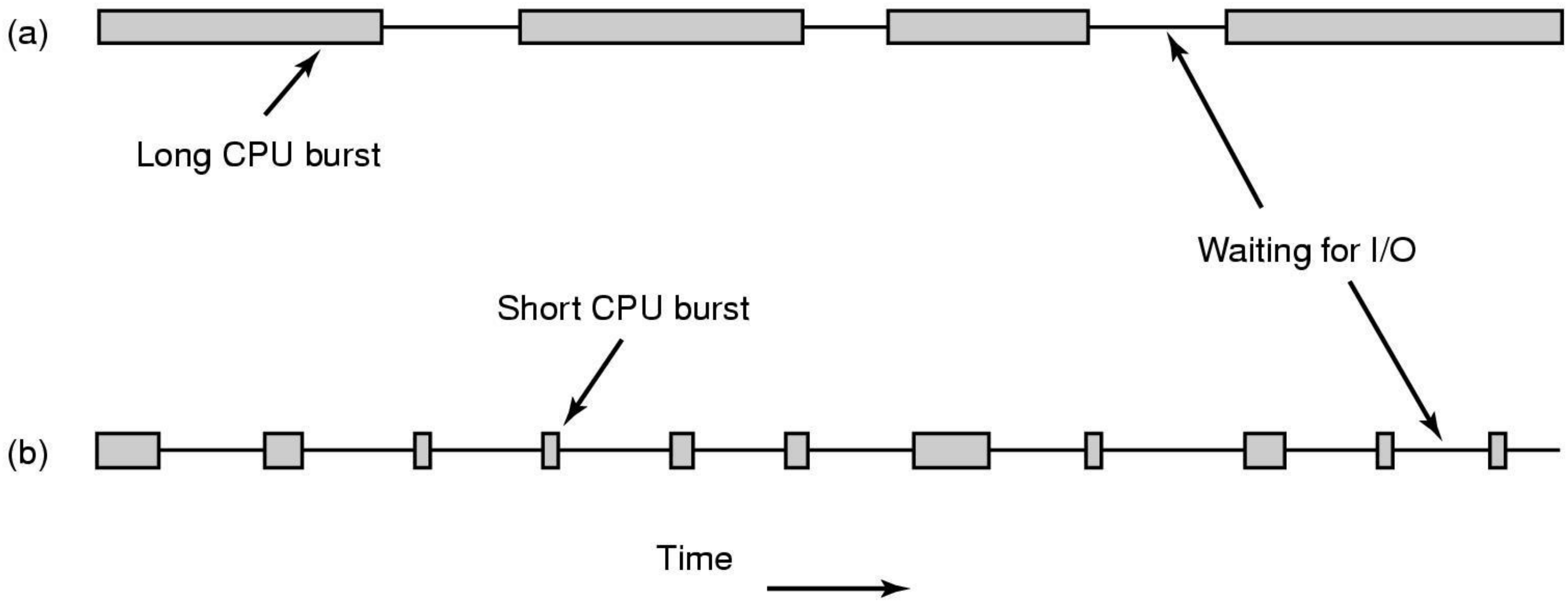
What happens on dispatch/context switch?

- Switch the CPU to another process
  - Save currently running process state
    - Unless the current process is exiting
  - Select next process from ready queue
  - Restore state of next process
    - Restore registers
    - Switch to user mode
    - Set PC to next instruction in this process

Process Life Cycle

- Processes repeatedly alternate between computation and I/O
  - Called CPU bursts and I/O bursts
  - Last CPU burst ends with a call to terminate the process (_exit() or equivalent)
  - CPU-bound: very long CPU bursts, infrequent I/O bursts
  - I/O-bound: short CPU bursts, frequent (long) I/O bursts
- During I/O bursts, CPU is not needed
  - Opportunity to execute another process!

Scheduling – Process Behavior



Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

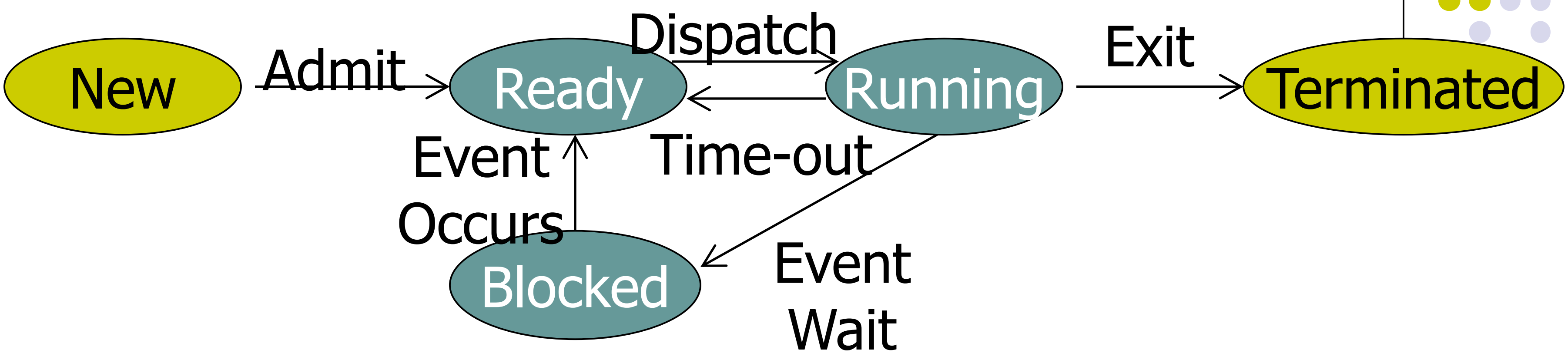
What is processor scheduling?

- The allocation of processors to processes over time
- This is the key to *multiprogramming*
  - We want to increase CPU utilization and job throughput by overlapping I/O and computation
  - *Mechanisms*:
    - process states, process queues

What is processor scheduling?

- The allocation of processors to processes over time
- This is the key to *multiprogramming*
  - We want to increase CPU utilization and job throughput by overlapping I/O and computation
  - *Mechanisms*:
    - Process states, Process queues
  - *Policies:*
    - Given more than one runnable process, how do we choose which to run next?
    - When do we make this decision?

When to schedule?



- When the running process blocks (or exits)
  - Operating system calls (e.g., I/O)
- At fixed intervals
  - Clock interrupts
- When a process enters *Ready* state
  - I/O interrupts, signals, process creation

Scheduling Goals

- All systems
  - **Fairness** - each process receives fair share of CPU
  - Avoid starvation
  - Policy enforcement - usage policies should be met
  - Balance - all parts of the system should be busy
- Batch systems
  - **Throughput** - maximize jobs completed per hour
  - **Turnaround time** - minimize time between submission and completion
  - CPU utilization - keep the CPU busy all the time

More Goals

- Interactive Systems
  - **Response time** - minimize time between receiving request and *starting* to produce output
  - Proportionality - “simple” tasks complete quickly
- Real-time systems
  - Meet **deadlines**
  - Predictability

- Goals sometimes conflict with each other!

Types of Scheduling

- Non-preemptive scheduling
  - once the CPU has been allocated to a process, it keeps the CPU until it terminates
  - Suitable for batch scheduling
- Preemptive scheduling
  - CPU can be taken from a running process and allocated to another
  - Needed in interactive or real-time systems

Next week

- More on Scheduling