Operating Systems

Operating Systems

Summer 2017

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 with signal

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? */
     }
}
```

this may be a problem when starting producer first waking up a process that is never slept… (so should keep track of if the process is sleeping/awake, and then decide

```
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 semaphore is a software implementation

Semaphores

a data type with
+ an int variable
+ wait/decrement/sleep
+ signal/increment/wakeup

- 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 enforces mutex
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore does not enforce mutex
  - 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

for counting semaphore, say count=10, first 10 processes calling wait will pass the 11th process will be blocked.

  - Operation *wait* (also called *P* or *decrement*)
    - block until `count` > 0 then decrement variable

wait / signal are atomic -> have to disable CPU interrups (i.e. context switch) before calling function for short period of tiem (to test for availability of count)

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

if count is 0, then blocked since semaphore is taken

while execute forever?

  - 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

hence able to have many semaphore for different structs in memory

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);
```

wait wait

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

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

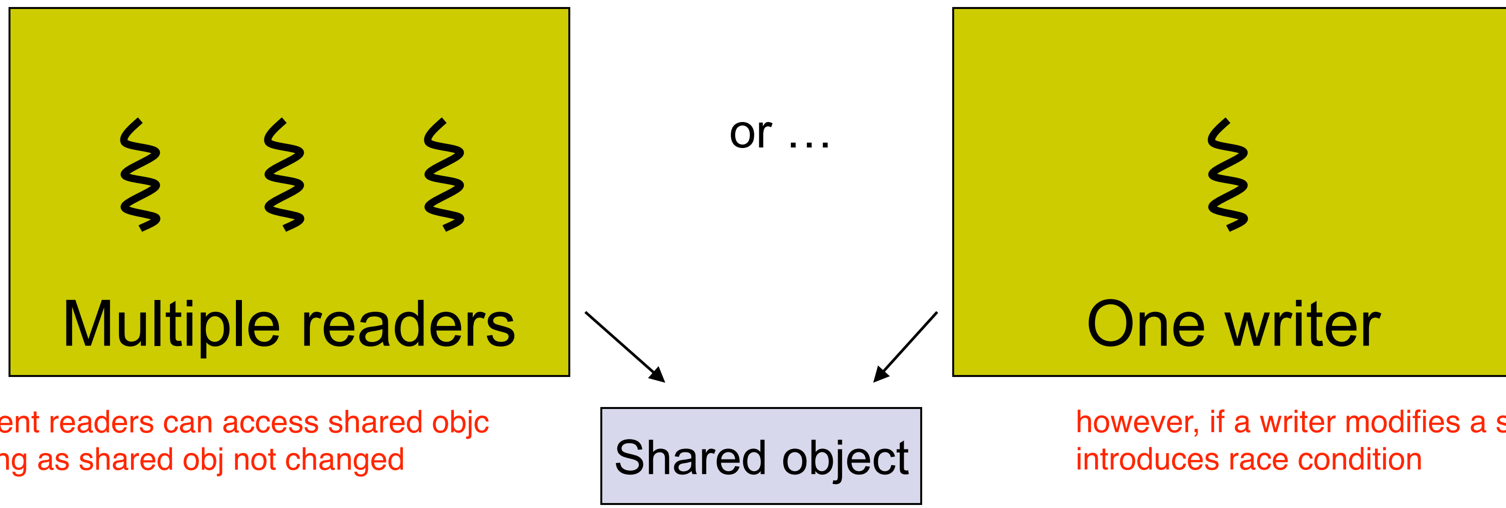
a scheduling problem.. either threads can get the semaphore

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



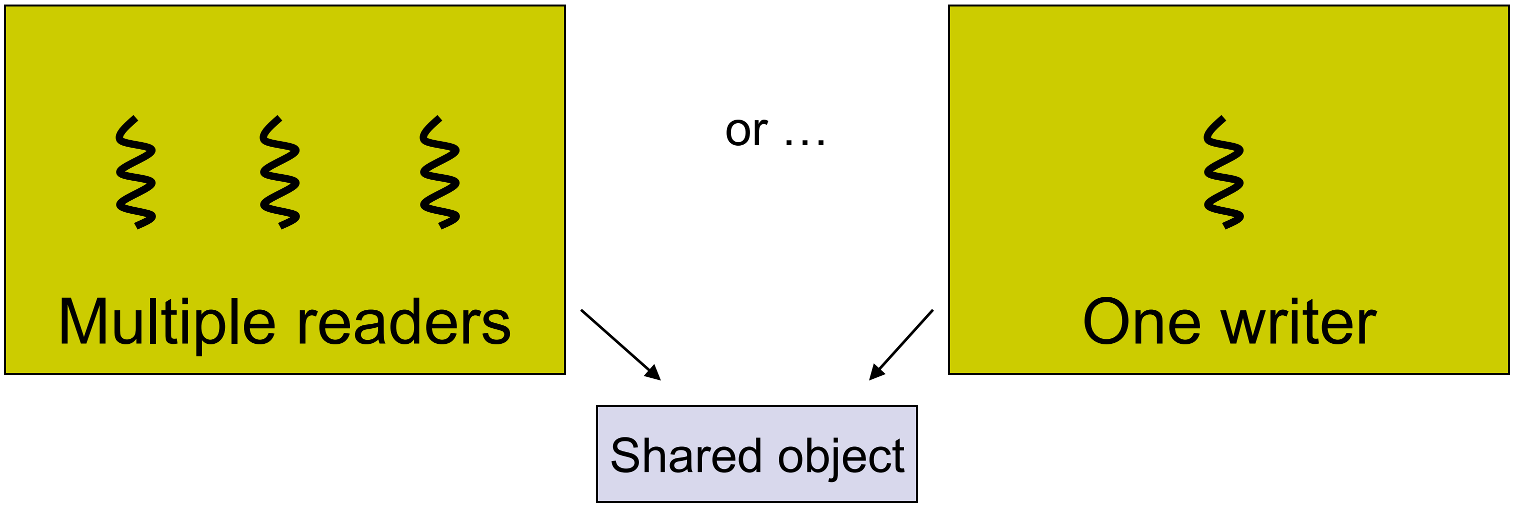
different readers can access shared objc
as long as shared obj not changed

however, if a writer modifies a shared obj
introduces race condition

- 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?*

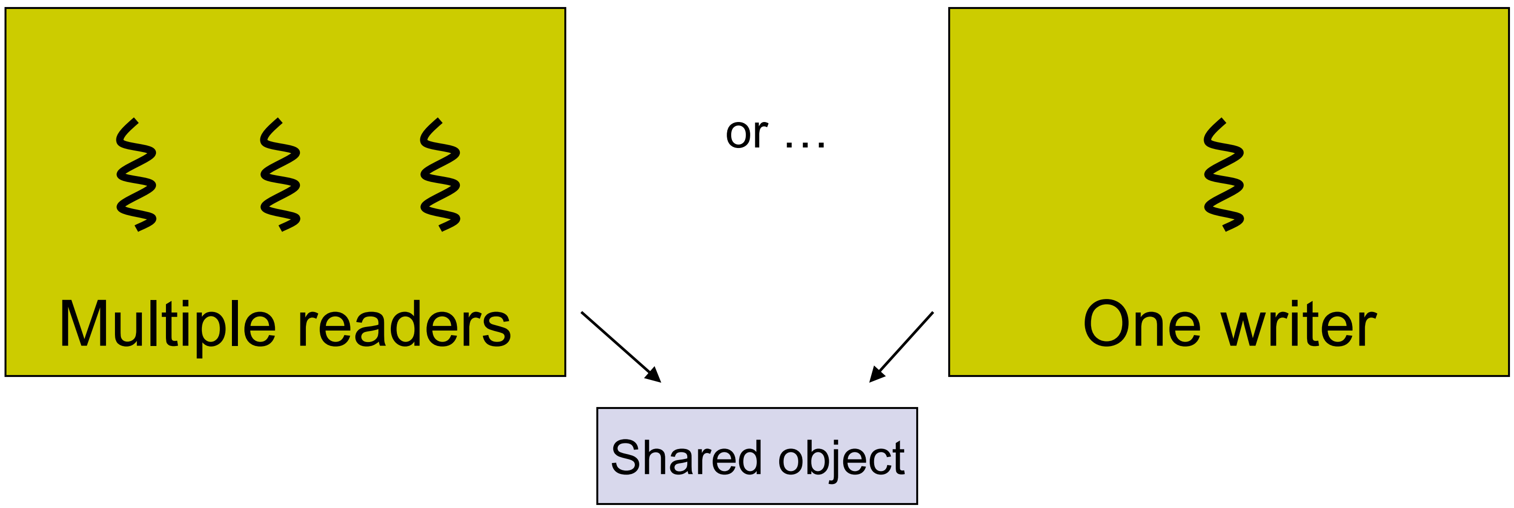
The readers/writers problem



- 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"*

first reader gets semaphore and any reader can get semaphore; last reader releases semaphore (either reader or writer can take next)

The readers/writers problem



- Use three variables
  - Semaphore **w_or_r** - exclusive writing or reading
  - int **readcount** - number of threads reading object
    i.e. when readcount -> 0, current reader is first reader, should get w_or_r
    - Needed to detect when a reader is the first or last of a group.
    i.e. when readcount -> 1, current reader is the last reader, hence should release w_or_r
  - Semaphore **mutex** - control access to readcount

readcount is shared variable, hence needs protection for preventing concurrent hcanges

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
}
```

w_or_r = 1
wait -> see w_or_r available
w_or_r -= 1 -> 0

(any read process getting CPU time here will be blocked on wait)

Now does writing

then signal -> w_or_r += 1 -> 1

(any read process getting blocked earlier can now finish wait)

Reader’s operation:

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

if Reader gets mutex
+ other reader have to wait
+ so readcount reflect atomic changes

- 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;
```

if Reader gets mutex
+ other reader have to wait
+ so changes to readcount is synchronized

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

wait until writer finishes writing
gets w_or_r semaphore if first reader
otherwise, already have w_or_r,

mutex released, since no more subsequent readcount modification; let other reads to read concurrently

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

sync on mutex again since have to modify read_count
release w_or_r if last reader

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?

blocked here

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);
}
```

implies first reader waiting at wait(w_or_r),
hence first reader holds mutex, second
reader hence wait for mutex to be released

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?

first reader will finish read first, since it holds
mutex, the second reader blocks at first lone until
first reader finishes by signal(mutex)

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?

undefined, scheduling problem, either can get w_or_r 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)

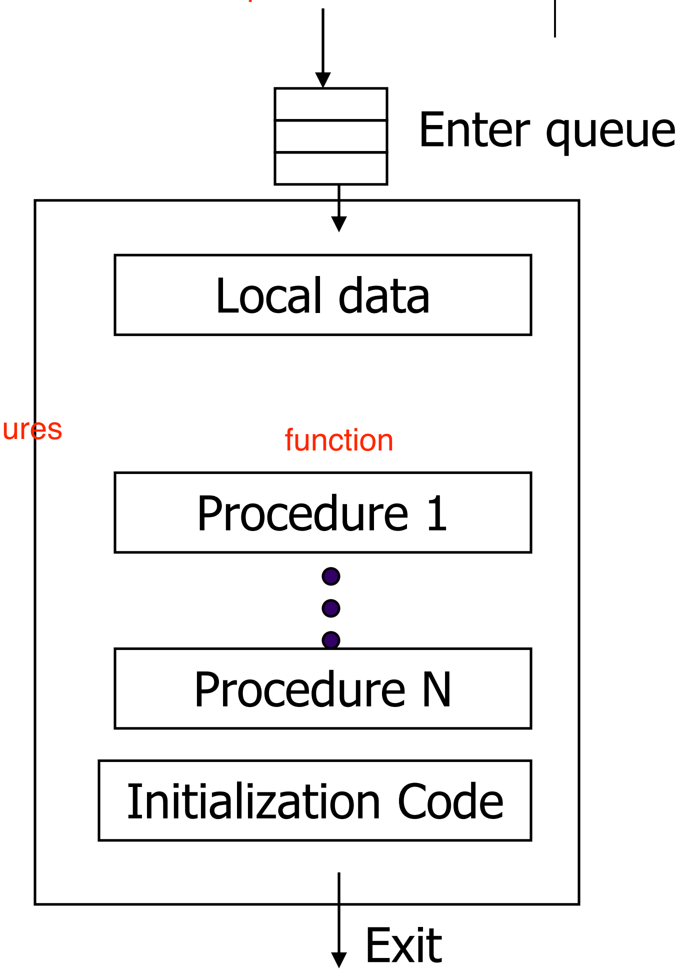
Monitor Diagram

advantage:
+ provided in library, easy to implement
disadvantage
+ the critical region change from a section of a procedure to the entire procedure

- 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

only one process in monitor,
but several available procedures

processes



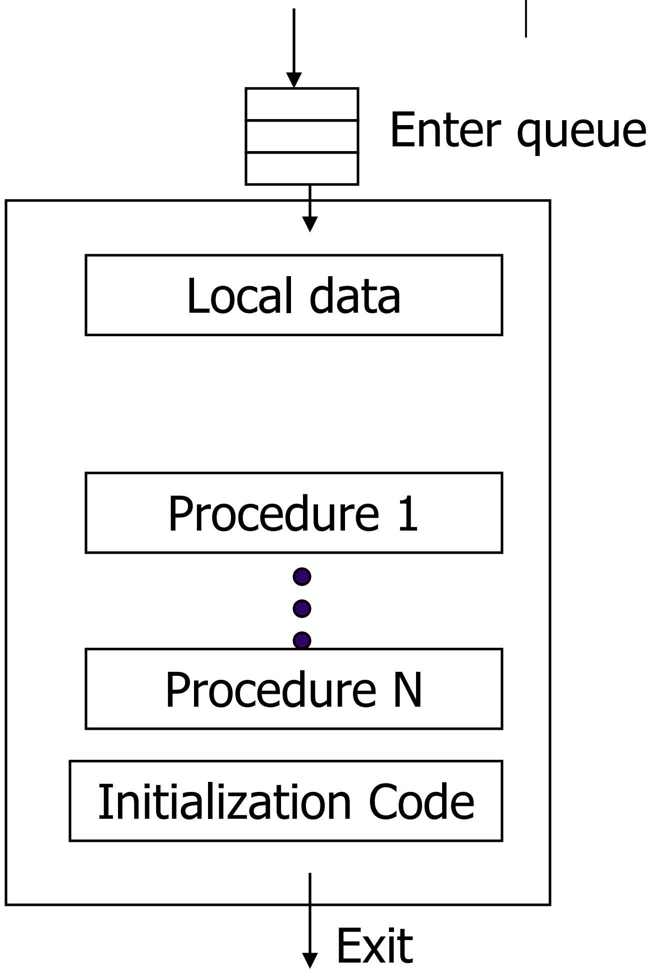
Bank example with monitors

```
Monitor Account {
      int balance;

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

shared

procedure for modifying shared variable



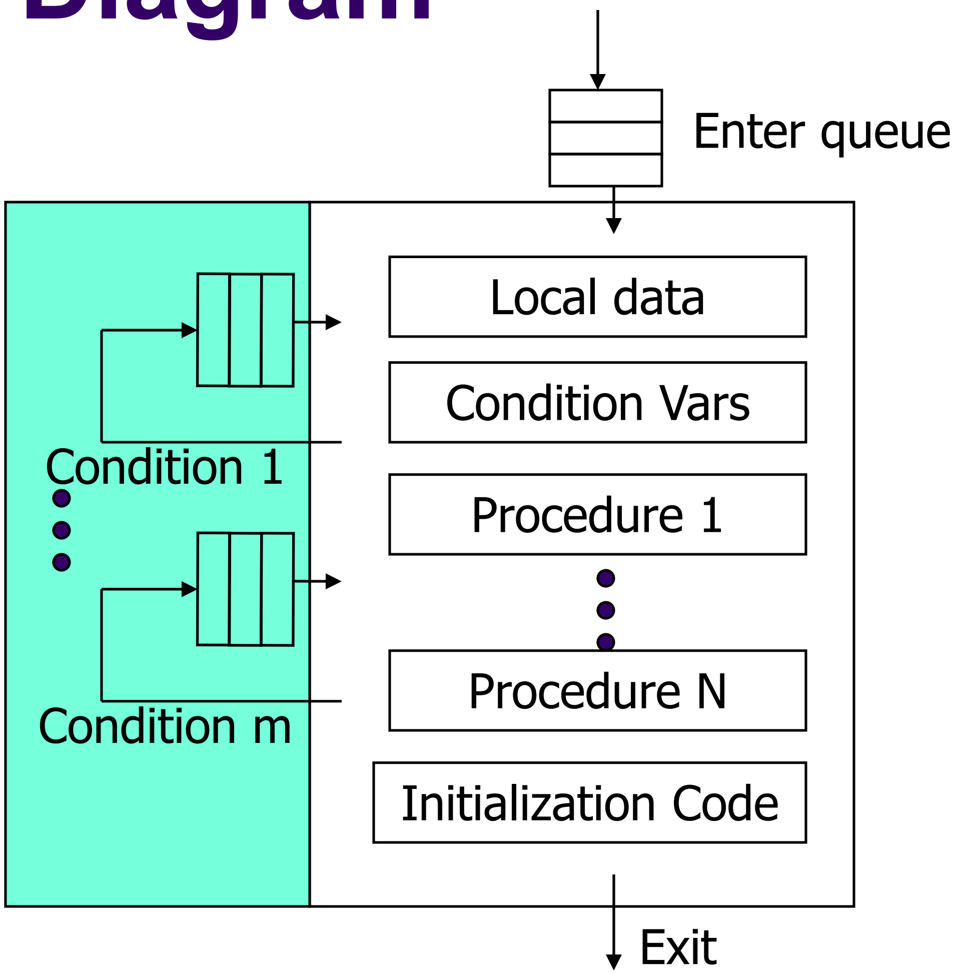
Enforcing single access

- 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?

any blocked process exits and reenters the queue

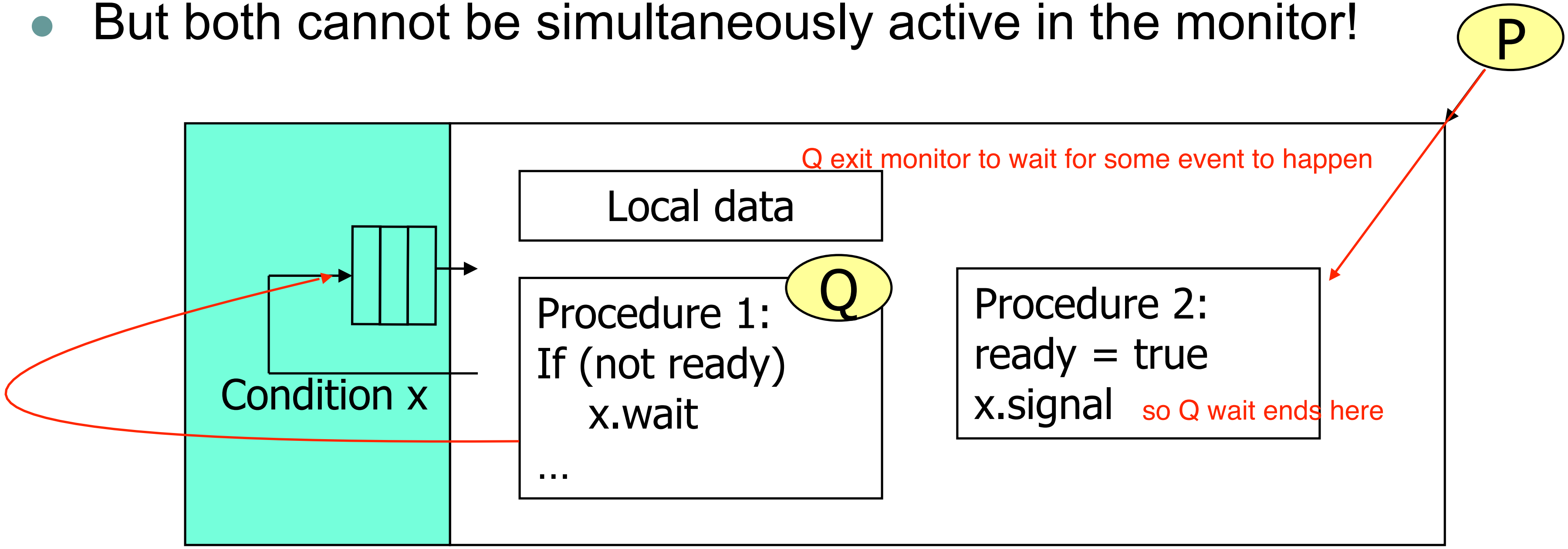
Monitor Diagram

condition let calling
process exits monitor



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!

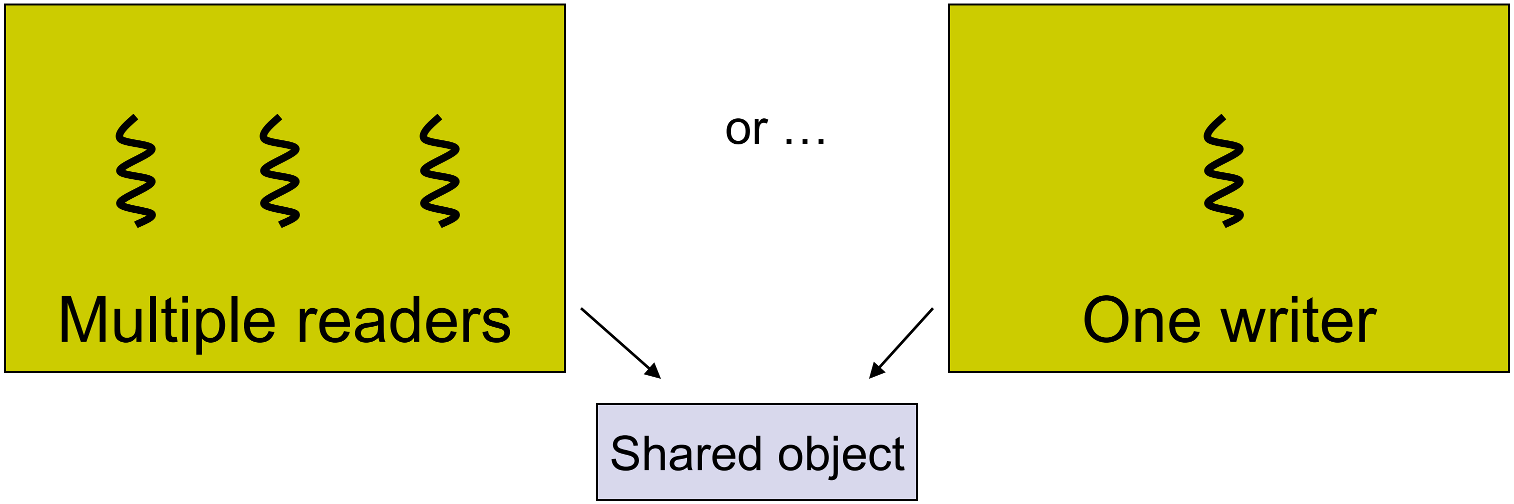


Monitor Semantics for Signal

- **Hoare monitors**
  - Signal() immediately switches from the caller to a waiting thread i.e. P exits and then Q enters monitor this adds complication
  - Need another queue for the signaler, if signaler was not done using the monitor i.e. if P has some additional code, have not finished procedure 2. need a queue for signalers like P
- **Brinch Hansen**
  - Signaler must exit monitor immediately
    - i.e. signal() is always the last statement in monitor procedure so guarantees P finish procedure, and exits naturally
- **Mesa monitors**
  - Signal() places a waiter on the ready queue, but signaler continues inside monitor

different from previous 2 types of monitors in that the wait is not guaranteed to be executed next

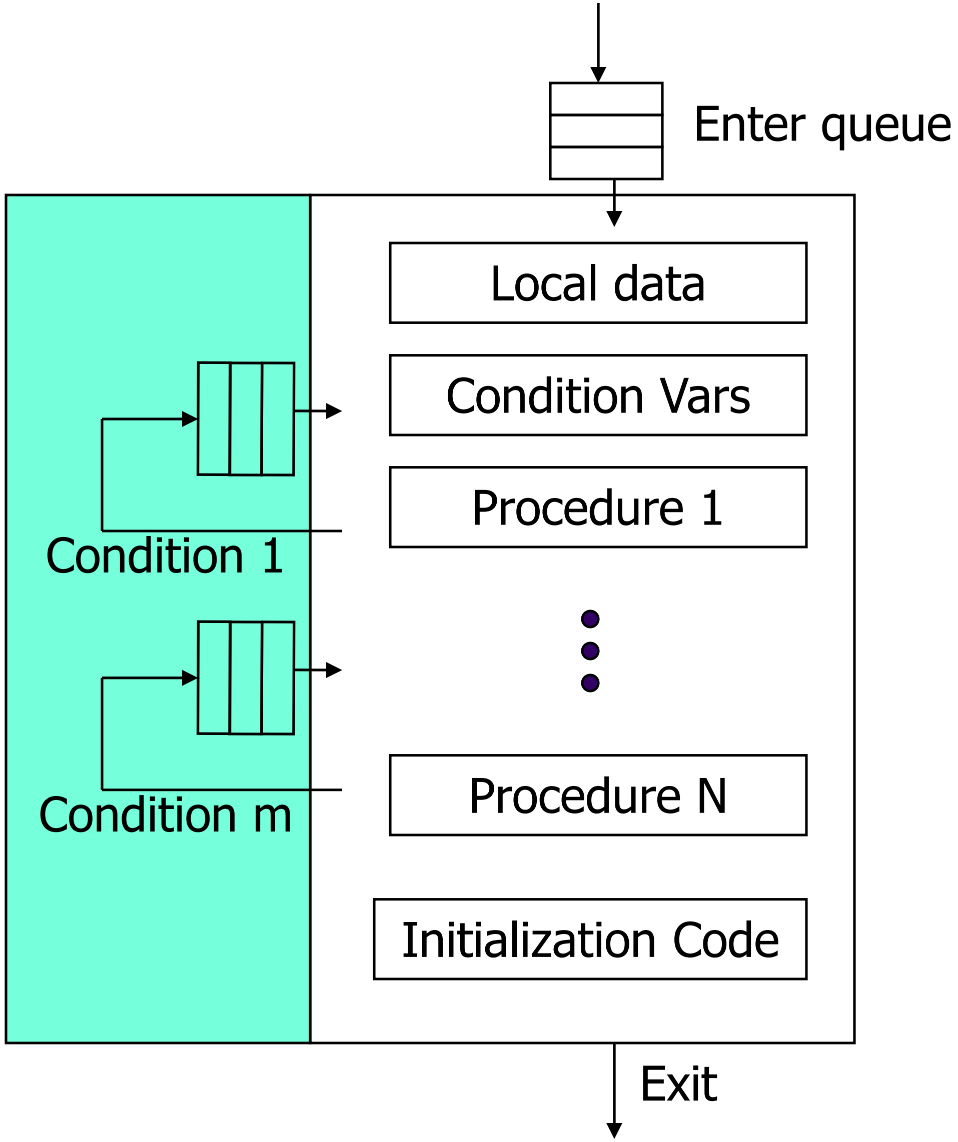
The readers/writers problem



- 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

a circular buffer

```
#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 */
}
```

circular.. so loop over on N

Bounded Buffer: The Consumer thread (no synchronization)

consumer

```
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….

get a lock at start of procedure, so that only one procedure is executing at one time in monitor

```
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);
}
```

pthread_cond_wait, releases lock, exits monitor, wait for condition to be satisfied

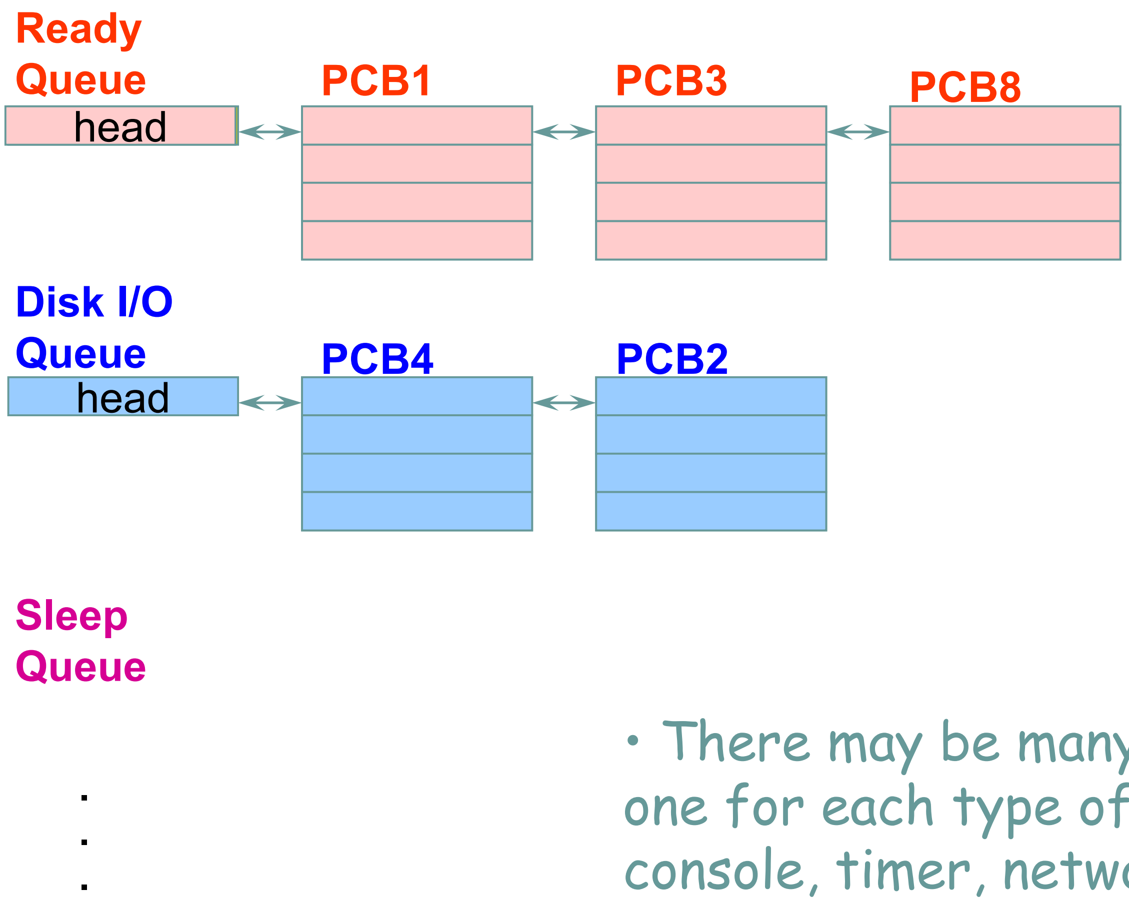
release lock at end to exit monitor

```
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 …

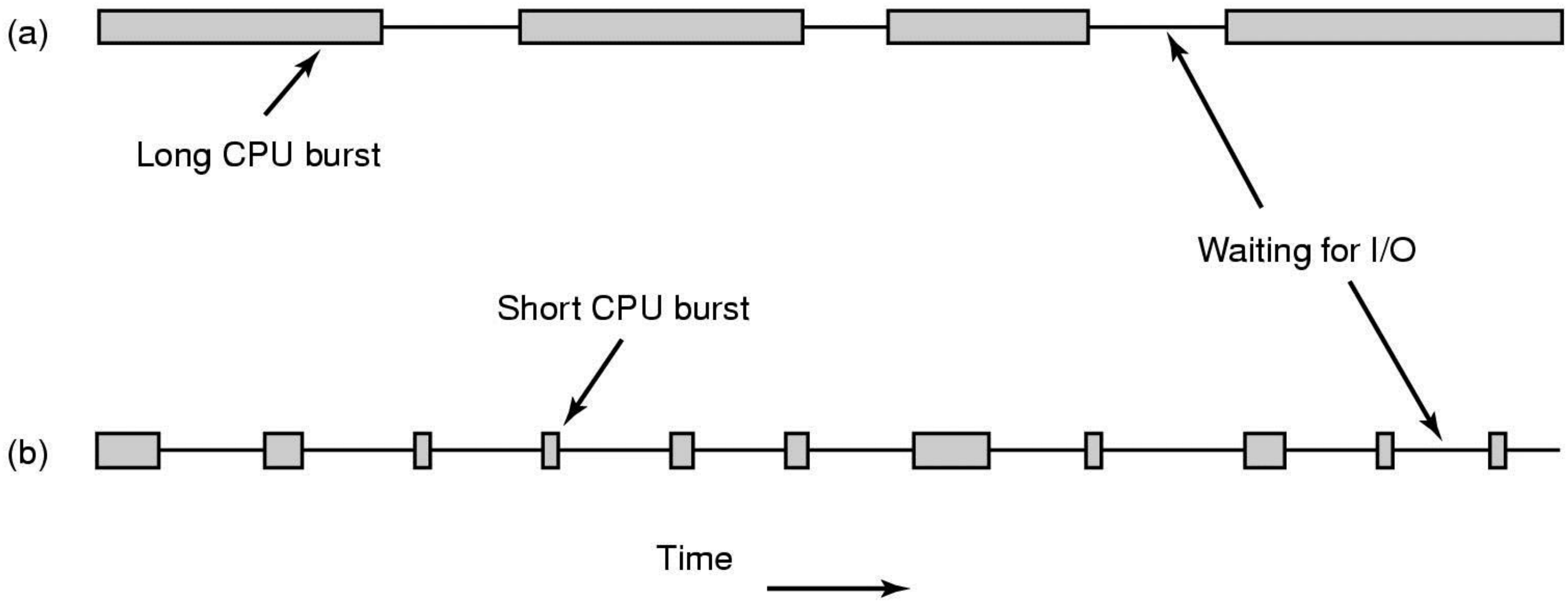
What happens on dispatch/context switch?

- 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.

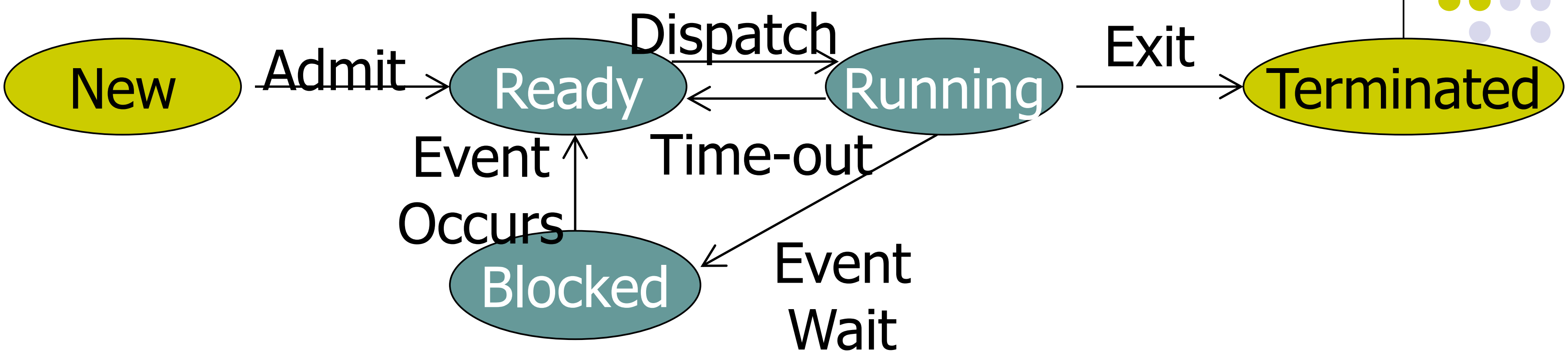
What is processor scheduling?

- 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