Recap from last class

- Operating System basics
  - Logical OS structure
  - Key OS issues
- Processes
  - Basic unit of program execution
  - Process states: running, ready and blocked
  - Metadata maintained Process Control Block (PCB)

ECE 1175
Embedded Systems Design

Operating Systems - II

Wei Gao

What’s in a process?

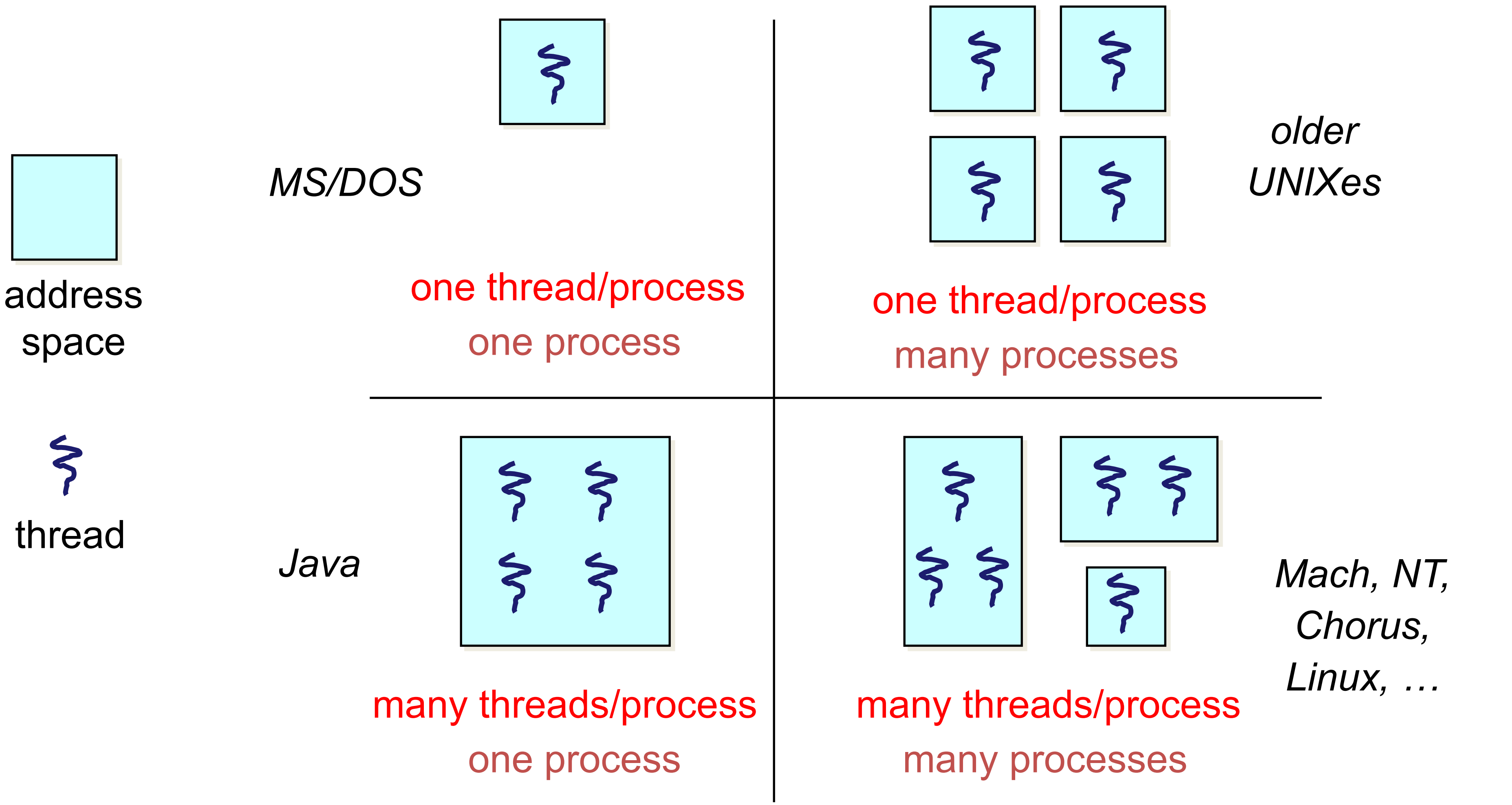
- A process consists of (at least):
  - an address space
  - the code for the running program
  - the data for the running program
  - an execution stack and stack pointer (SP)
    - traces state of procedure calls made
  - the program counter (PC), indicating the next instruction
  - a set of general-purpose processor registers and their values
  - a set of OS resources
    - open files, network connections, sound channels, …
- That’s a lot!!
- Can we decompose a process?

Thread

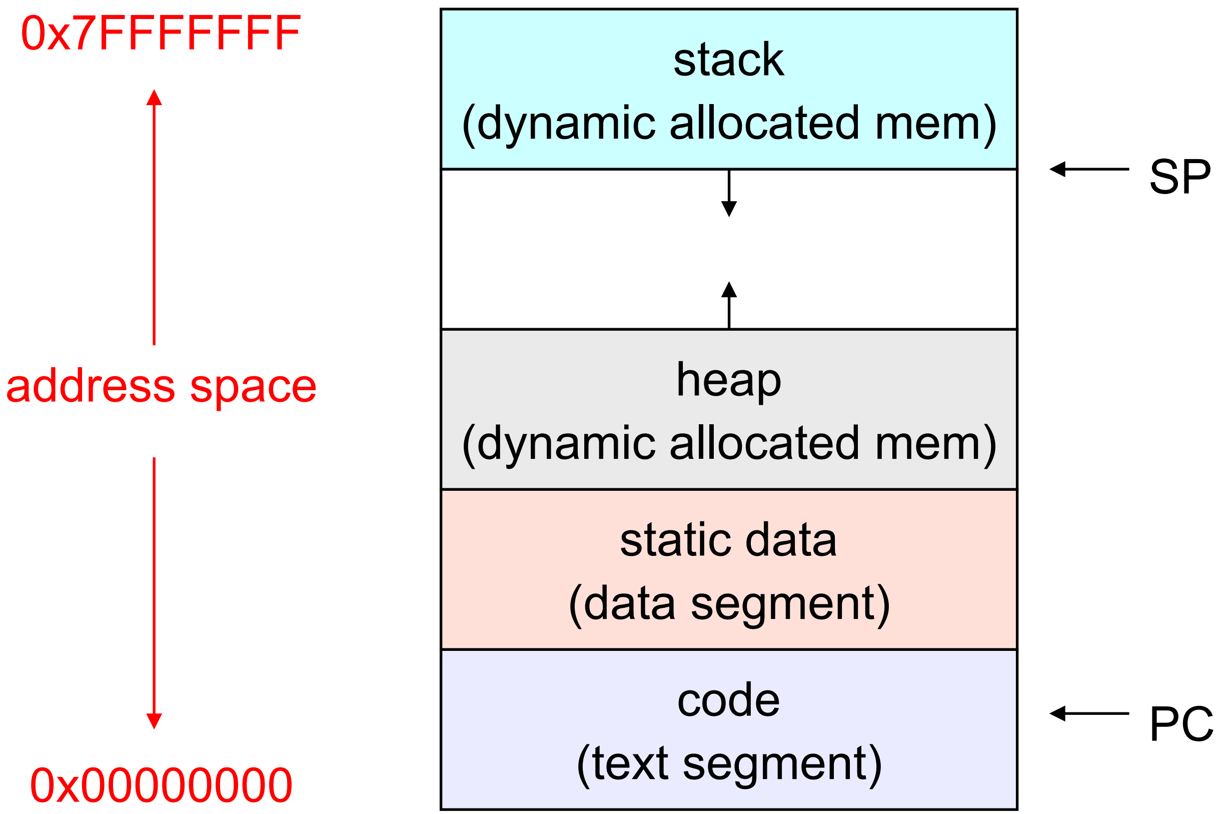
- A lightweight process
  - Separating the process's memory space
  - Better concurrency!

- Multithreading is useful even on a uniprocessor
  - even though only one thread can run at a time
  - creating concurrency does not require creating new processes
  - "faster / better / cheaper"

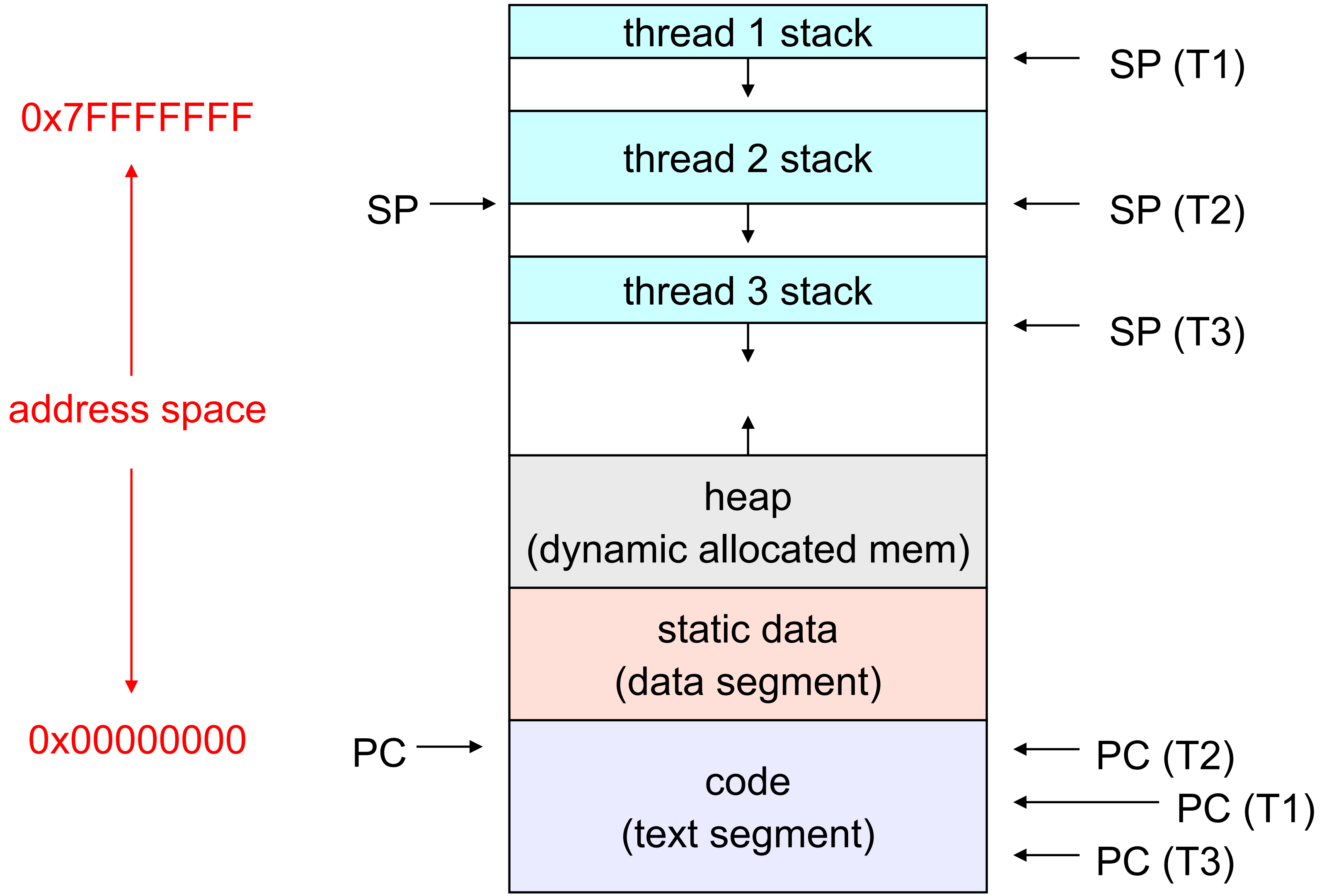
Thread


address space
thread
MS/DOS
one thread/process
one process
older UNIXes
one thread/process
many processes
Java
many threads/process
one process
Mach, NT, Chorus, Linux, …
many threads/process
many processes

Process Memory Space

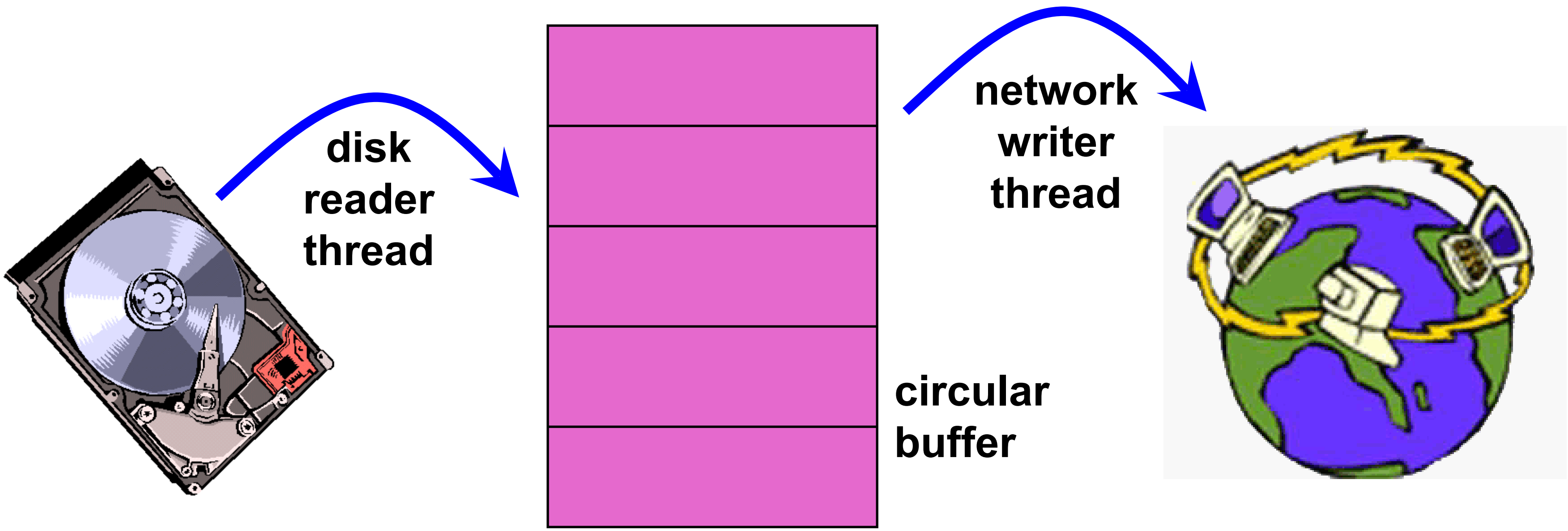


Memory Space with Threads



Synchronization

- Threads cooperate in multithreaded programs
  - to share resources, access shared data structures
    - e.g., threads accessing a memory cache in a web server
  - also, to coordinate their execution
    - e.g., a disk reader thread hands off blocks to a network writer thread through a circular buffer



Shared Resources

- Major focus of synchronization across processes/threads
- Basic problem:
  - two concurrent threads are accessing a shared variable
  - if the variable is read/modified/written by both threads, then access to the variable must be controlled
  - otherwise, unexpected results may occur

The Classic Example

- Suppose we have to implement a function to withdraw money from a bank account:

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

- Now suppose that you and your partner share a bank account with a balance of $100.00
  - what happens if you both go to separate ATM machines, and simultaneously withdraw $10.00 from the account?

The Classic Example

- Represent the situation by creating a separate thread for each person to do the withdrawals
  - have both threads run on the same bank mainframe:

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Interleaved Schedule

Execution sequence as seen by CPU

```
balance = get_balance(account);
balance -= amount;
```

context switch

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
```

context switch

```
put_balance(account, balance);
```

- Who comes first??

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

And this?

```
i++;
```

```
i++;
```

The Real Issue Here

- The problem is that two concurrent threads (or processes) access a shared resource (account) without any synchronization
  - creates a **race condition**
    - output is non-deterministic, depends on timing
- Synchronization is necessary for any shared data structure
  - buffers, queues, lists, hash tables, scalars, …

Mutual Exclusion

- Code that uses mutual exclusion to synchronize its execution is called a critical section
  - only one thread at a time can execute in the critical section
  - all other threads are forced to wait on entry
  - when a thread leaves a critical section, another can enter
- Mechanics for building critical sections
  - Messages
  - Locks
  - Semaphores

Locks

- A lock is a object (in memory) that provides the following two operations:
  - `acquire()`: a thread calls this before entering a critical section
  - `release()`: a thread calls this after leaving a critical section

- Two basic types of locks
  - Spinlock: Test-and-Set
  - Blocking

Semaphores

- A semaphore is:
  - a variable that is manipulated through two operations, P and V (Dutch for “test” and “increment”)
    - **P(sem)** (wait/down)
      - block until sem > 0, then subtract 1 from sem and proceed
    - **V(sem)** (signal/up)
      - add 1 to sem
- Do these operations *atomically*

How Semaphores work?

- Each semaphore has an associated queue of threads
  - when P(sem) is called by a thread,
    - if sem was “available” (>0), decrement sem and let thread continue
    - if sem was “unavailable” (<=0), place thread on associated queue; dispatch some other runnable thread
  - when V(sem) is called by a thread
    - if thread(s) are waiting on the associated queue, unblock one
      - place it on the ready queue
      - might as well let the “V-ing” thread continue execution
      - or not, depending on priority
    - otherwise (when no threads are waiting on the sem), increment sem
      - the signal is “remembered” for next time P(sem) is called

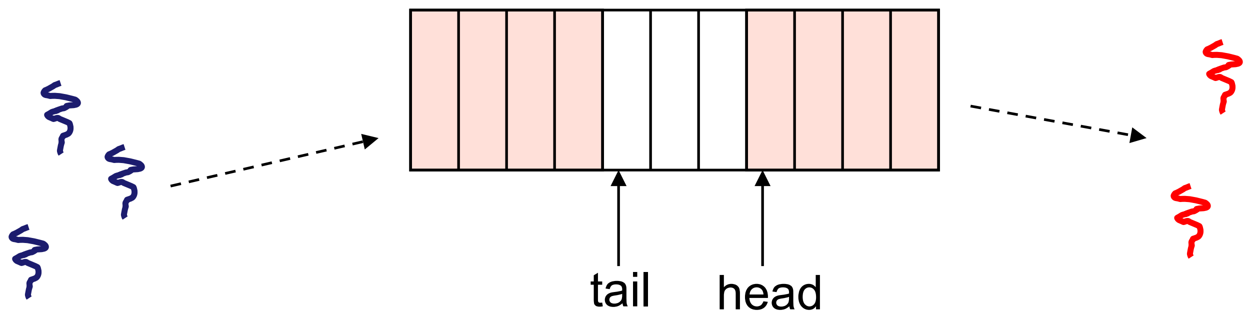
Two Types of Semaphores

- Binary semaphore (aka mutex semaphore)
  - sem is initialized to 1
  - guarantees mutually exclusive access to resource (e.g., a critical section of code)
  - only one thread/process allowed entry at a time

- Counting semaphore
  - sem is initialized to N
    - N = number of units available
  - represents resources with many (identical) units available
  - allows threads to enter as long as more units are available

Classic Problems

- Reader/writer problem
  - there is a buffer in memory with N entries
  - producer threads insert entries into it (one at a time)
  - consumer threads remove entries from it (one at a time)
- Threads are concurrent
  - so, we must use synchronization constructs to control access to shared variables describing buffer state



Reader/Writer Using Semaphores

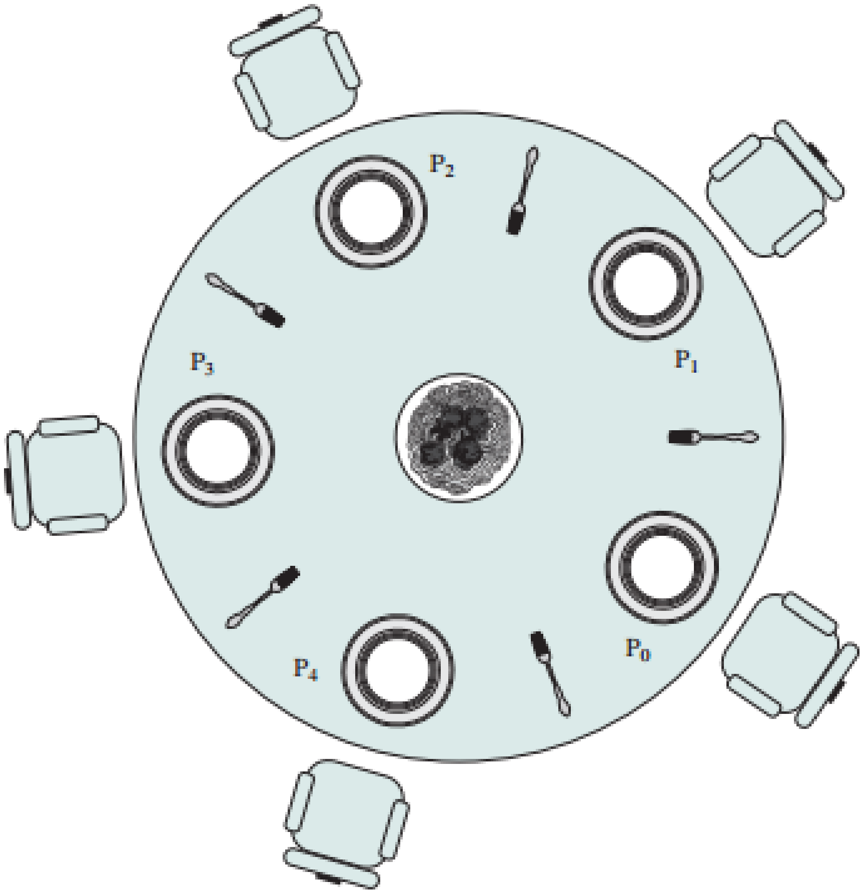
```
var mutex: semaphore = 1        ; controls access to readcount
    wrt: semaphore = 1          ; control entry for a writer or first reader
    readcount: integer = 0      ; number of active readers
```

```
writer:
        P(wrt)                  ; any writers or readers?
                <perform write operation>
        V(wrt)                  ; allow others
```

```
reader:
        P(mutex)                            ; ensure exclusion
          readcount++                       ; one more reader
          if readcount == 1 then P(wrt)     ; if we're the first, synch with writers
        V(mutex)
                <perform read operation>
        P(mutex)                            ; ensure exclusion
          readcount--                       ; one fewer reader
          if readcount == 0 then V(wrt)     ; no more readers, allow a writer
        V(mutex)
```

Classic Problems

- Dining philosopher problem



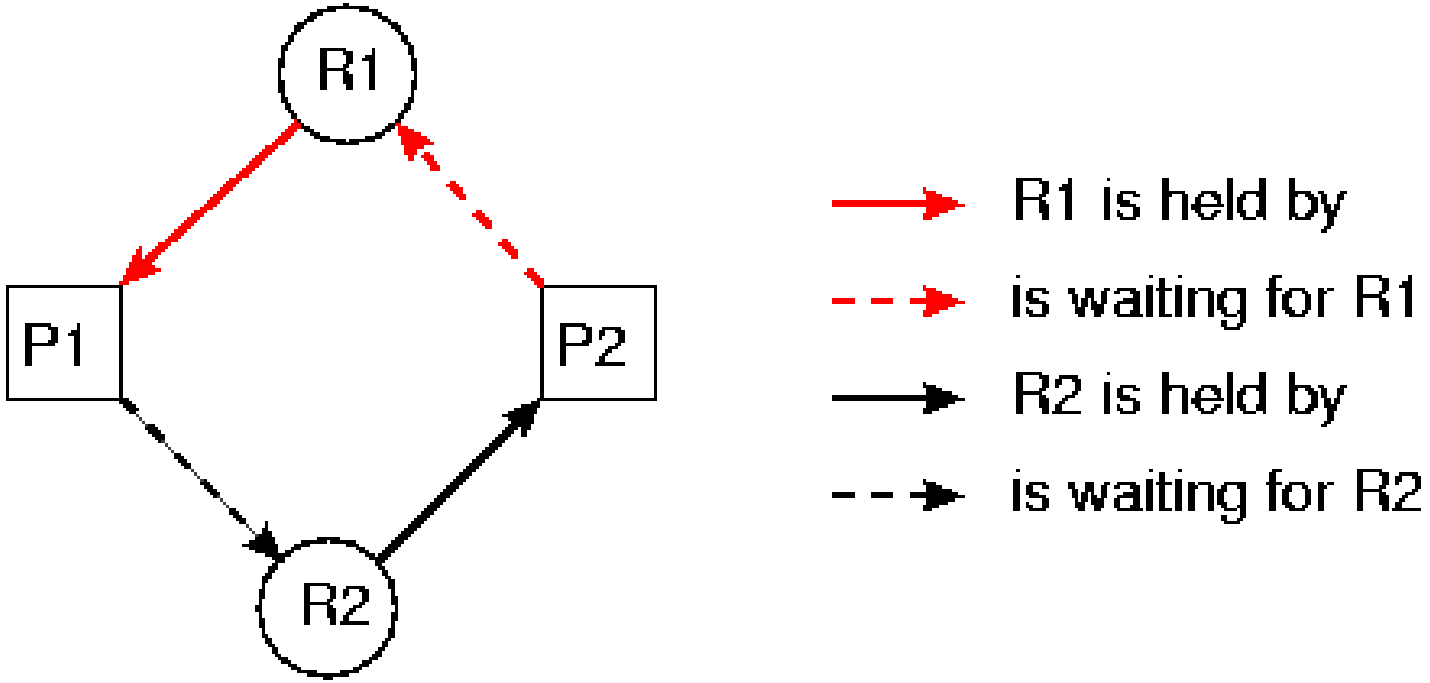
Deadlock

Deadlock

- Definition: A thread is deadlocked when it's waiting for an event that can never occur
  - I'm waiting for you to clear the intersection, so I can proceed
    - but you can't move until he moves, and he can't move until she moves, and she can't move until I move

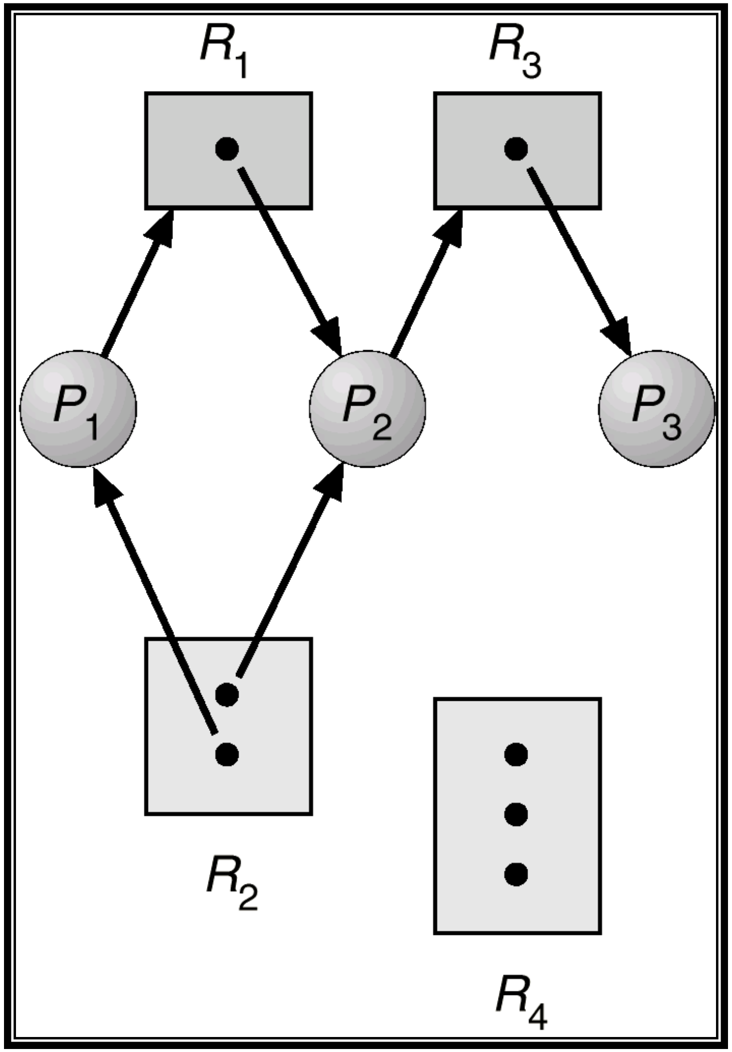
Deadlock

- Resource graph
  - A deadlock exists if there is an *irreducible cycle* in the resource graph



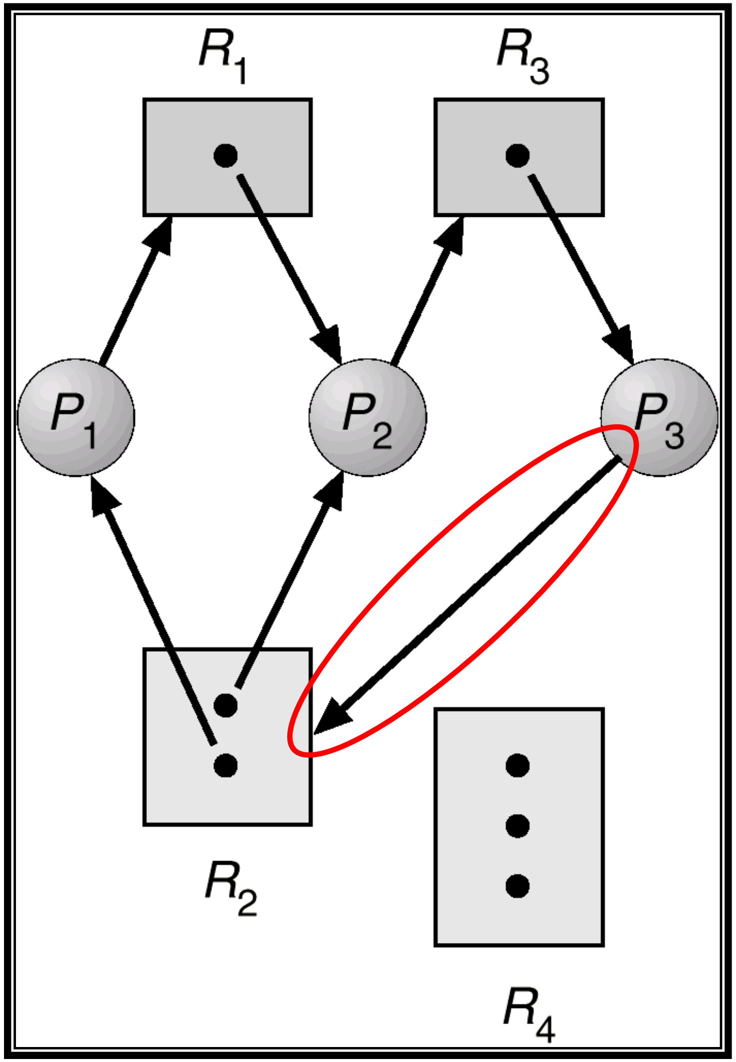
Resource Graph with No Cycle

- No deadlock



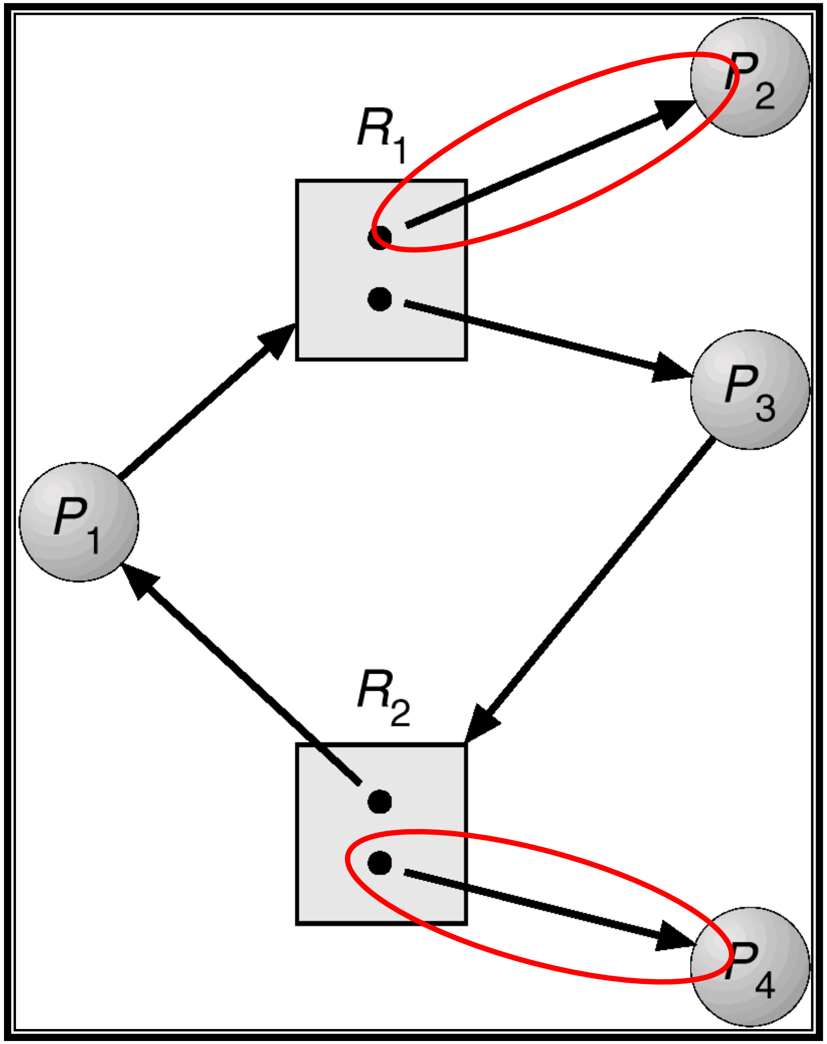
Resource Graph with A Deadlock

- An irreducible cycle



What is a reducible cycle?

- Resource graph with a cycle but no deadlock
- How to reduce?



Approaches to Deadlock

- Break one of the four required conditions
  - Mutual Exclusion?
  - Hold and Wait?
  - No Preemption?
  - Circular Wait?

- Broadly classified as:
  - Prevention (static), or
  - Avoidance (dynamic)

Prevention (static)

- Hold and Wait
  - each thread obtains all resources at the beginning; blocks until all are available
    - drawback?
- Circular Wait
  - resources are ordered; each thread obtains them in sequence (which means acquiring some before they are actually needed)
    - why does this work?
    - pros and cons?

Avoidance (dynamic)

- Circular Wait
  - each thread states its maximum claim for every resource type
  - system runs the Banker's Algorithm at each allocation request
    - Banker $\Rightarrow$ incredibly conservative
    - if I were to allocate you that resource, and then everyone were to request their maximum claim for every resource, could I find a way to allocate remaining resources so that everyone finished?

Summary

- Managing concurrency – the core task of OS
- Synchronization between process/threads
  - Race condition
- Mutual exclusion
  - Locks
  - Semaphores
  - Deadlocks