# Part I: Unix Signals

# Stopping a Program

What if you run this program?

```
int main () {
  while (1);
  printf("bye\n");
  return 0;
}
```

What happens if you hit Ctl-C?

Could you make Ctl-C print “bye” before exiting?

# Signals

A shell handles Ctl-C by sending the **`SIGINT`** signal to a process

The **`sigaction()`** function can be used to install a ***signal handler***

See **`bye.c`** and **`bye2.c`**

# Some Other Signals

| | |
|---|---|
| **SIGHUP** | terminal is gone |
| **SIGQUIT** | please quit |
| **SIGKILL** | force quit (cannot handle) |
| **SIGSEGV** | seg fault |
| **SIGALRM** | timer expired |
| **SIGPIPE** | write to pipe with closed read end |
| **SIGCHLD** | child completed |

# Timers

Use **`setitimer()`** to start a timer

See **`timer.c`** and **`timer2.c`**...

# Signal Handlers and Races

**Beware!** — a signal handler is practically a thread

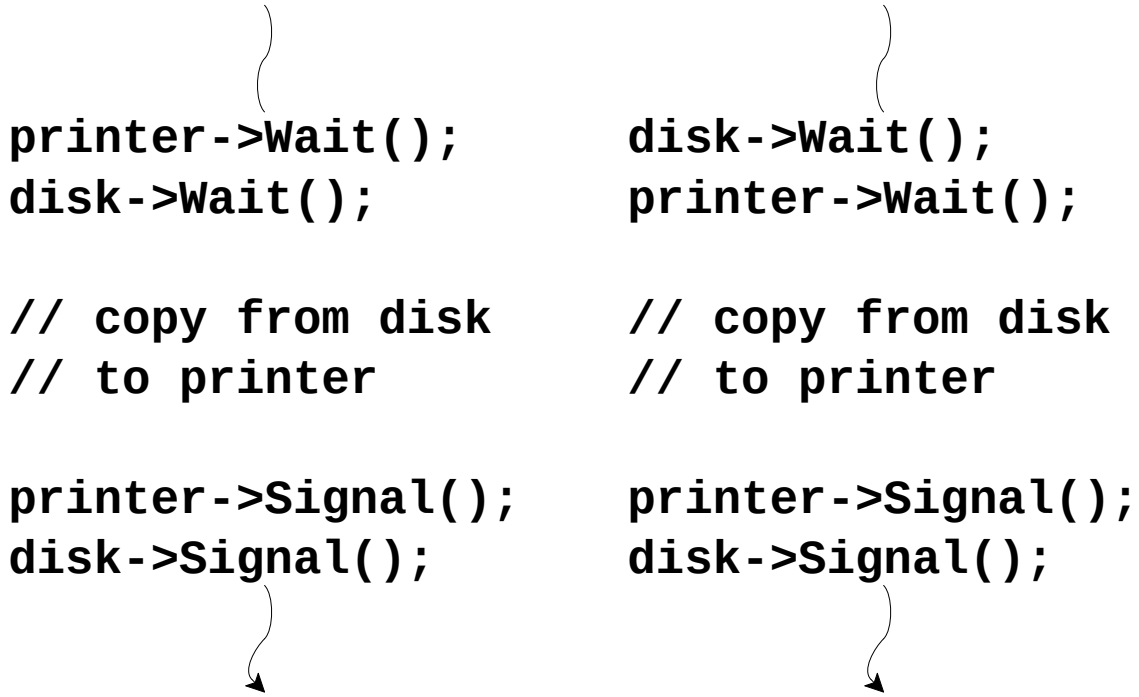
Use **`sigprocmask()`** to (un)block signals

See **`timer3.c`**

# Part II: Deadlock

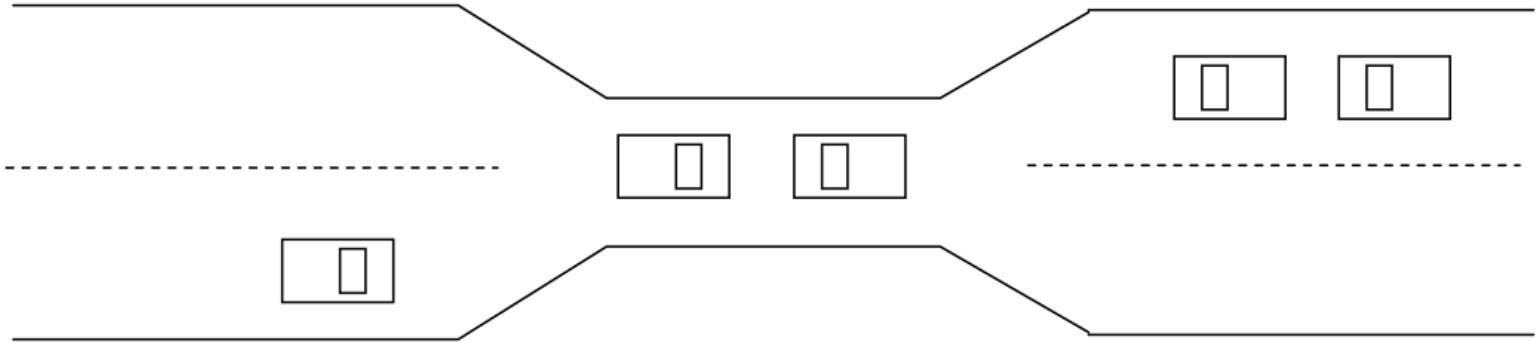
- Conditions
- Prevention
- Detection

***Deadlock*** is when two or more threads are waiting for an event that can only be generated by these same threads

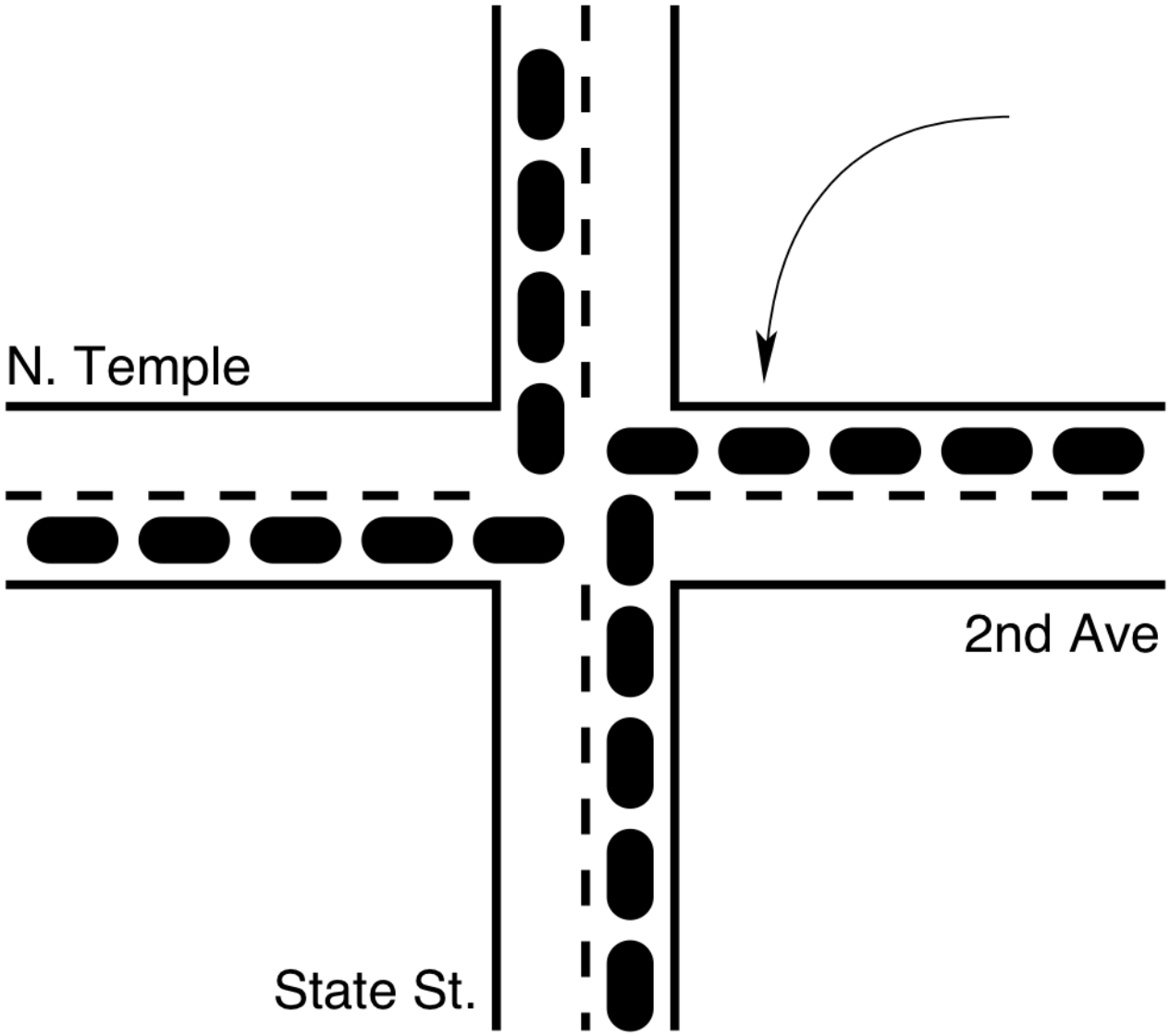



Deadlock can occur anytime threads acquire multiple resources (printers, disks, etc.), perform work, and then release their resources

# Deadlock



# Deadlock



# Deadlock Examples

- **Linux:** In-kernel memory allocator runs out of pages, causing an "out of memory handler" to run, which calls a function that tries to allocate a page.
- **Windows 2000:** The OS keeps a pool of "worker threads" waiting to do work; one worker thread submits a job to another worker thread, but there are no free worker-thread slots.
- **Win32:** A graphical application sends a piece of work to a worker thread and then waits for the worker to complete; the worker, as part of its job, queries the state of the user interface.
- **Any OS:** You are writing a threaded program and you attempt to acquire a lock that you already hold. (This is easier than it sounds when you are using library code.)

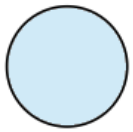
# Necessary Conditions for Deadlock

Requires all of the following:

1. **Mutual Exclusion** — at least one thread must hold a resource that cannot be shared (e.g., a lock or a printer)
2. **Hold and Wait** — at least one thread holds a resource and is waiting for other resource(s) to become available; a different thread holds the resource(s)
3. **No Preemption** — a thread can release a resource only voluntarily; another thread or the OS cannot force the thread to release the resource
4. **Circular Wait** — a set of waiting threads are waiting on each other

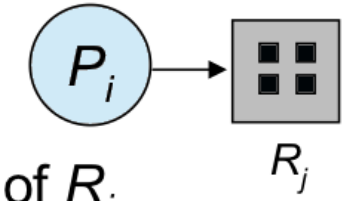
# Resource-Allocation Graphs

Process

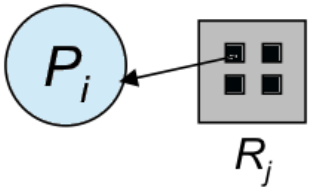


Resource Type with 4 instances

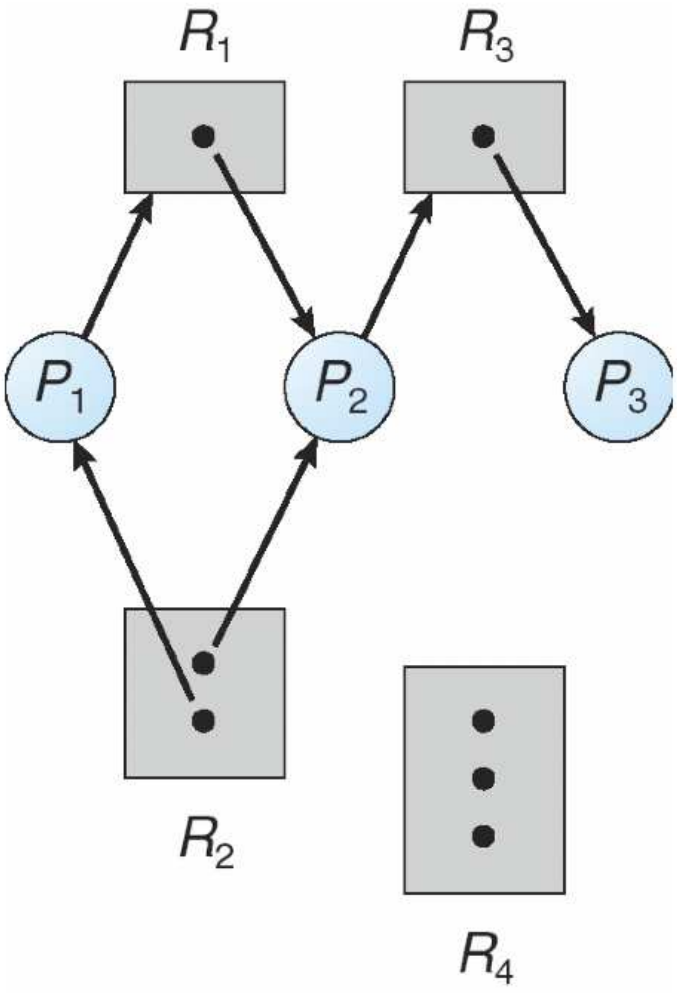
$P_i$ requests instance of $R_j$



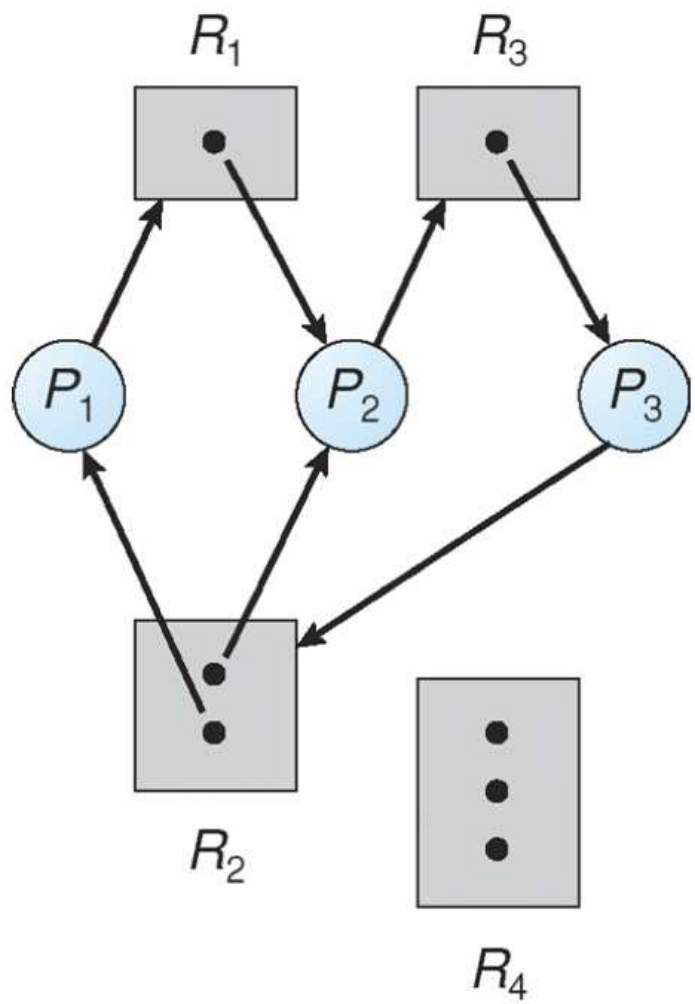

$P_i$ is holding an instance of $R_j$



# Resource-Allocation Graphs

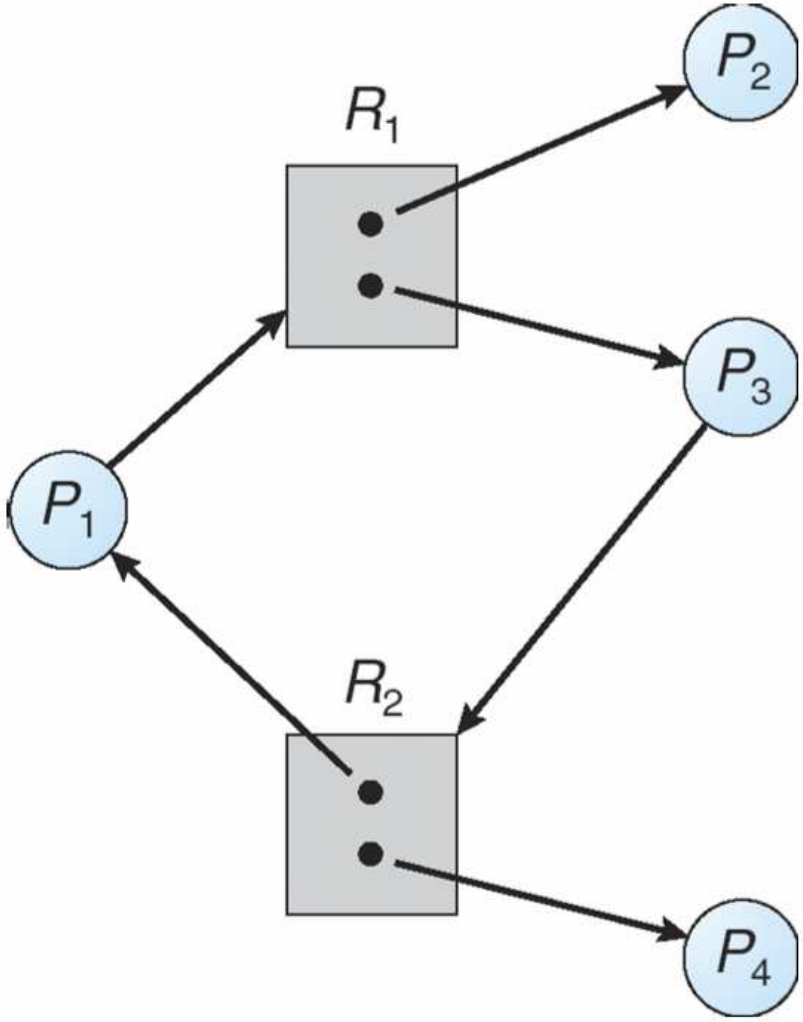


# Resource-Allocation Graphs




deadlock

# Resource-Allocation Graphs




probably not deadlock

# Resource-Allocation Graphs

- No cycles in graph $\Rightarrow$ no deadlock
- Cycles in graph and...
  - ... one instance/resource $\Rightarrow$ deadlock
  - ... multiple instances $\Rightarrow$ maybe deadlock

# Handling Deadlock

- ***Deadlock prevention***: design the system or application so that deadlock is impossible
- ***Deadlock avoidance***: exploit knowledge about worst-case resource requirements; the OS artificially delays some requests to aviod deadlock
- ***Deadlock detection***: find instances of deadlock and recover somehow

# Deadlock Prevention

Ensure that at least one of the necessary conditions doesn't hold:

1. **Mutual Exclusion —** make resources sharable (but some resources cannot be shared)
2. **Hold and Wait —** disallow a thread holding one resource to request another, or make threads request all the resources they need at once
3. **No Preemption** — if a thread requests an unavailable resource, preempt (releases) all the resources that the thread is currently holding; only when all resources are available restart the thread (but not all resources can be easily preempted, like printers)
4. **Circular Wait —** impose an ordering (numbering) on the resources and request them in order

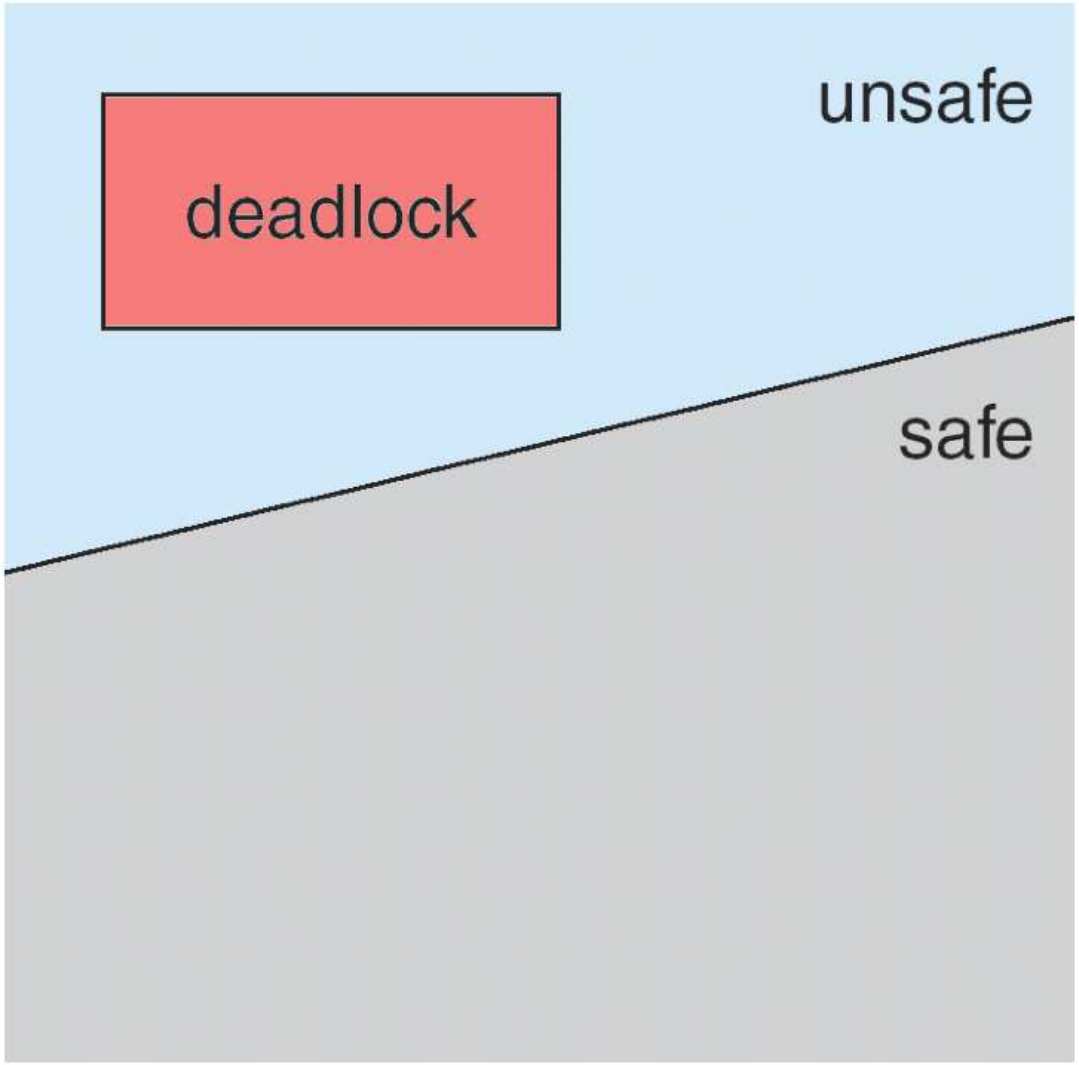
# Deadlock Avoidance

Requires information about future resource requests

Given that knowledge, for any configuration of resource callocations:

- **Safe** — from the current state, processes can be ordered to get resources they may want
- **Unsafe** — from the current state, couldn’t necessarily satisfy all requests

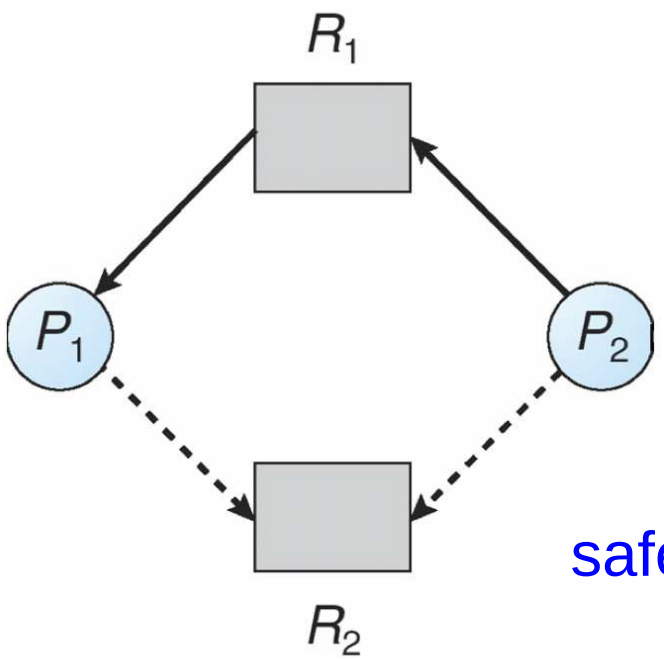
# Deadlock Avoidance



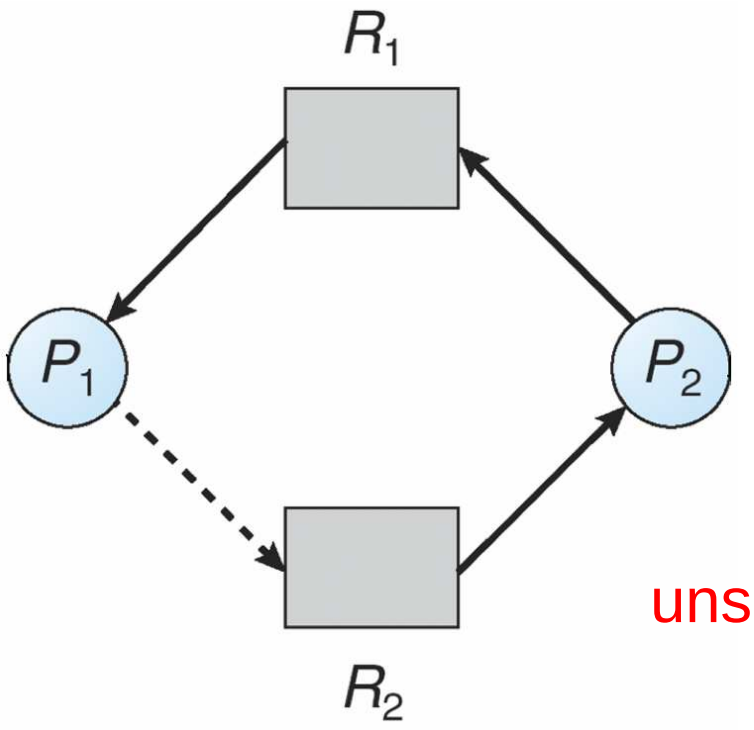
# Deadlock Avoidance

Avoidance algorithms:

- Single instance per resource: extended resource-allocation graph




safe




unsafe

- Multiple instances per resource: Banker's algorithm

# Deadlock Detection

Problem 1: detection

- Detecting cycles takes O($n^2$) where $n$ is $T + R$

  Check every allocation? On every failed allocation? On a regular schedule?

Problem 2: correction

- Terminate a thread?
- Revoke a resource and notify thread?
- Rollback system?

# Deadlock Handling in Practice

Deadlock avoidance not that useful in practice: expensive to compute, difficult to predict resource usage

Real OSs like UNIX and Windows tend to use a combination of deadlock prevention and deadlock detection

- Deadlock detection might be automatic (Windows 2000 worker threads) or might involve a human
- Deadlock prevention can be made practical by dividing resources into categories

# Part III: Synchronization Summary

# Low-level vs. High-level Synchronization

Low-level synchronization:
- Disabling interrupts
- Test-and-set or compare-and-swap
- Spinlock

High-level synchronization:
- Locks (mutexes)
- Semaphores
- Reader/writer locks
- Monitors

Questions you should now be able to answer:
- What can the OS do with these low-level primitives? The user?
- What are the advantages and disadvantages of each?
- How would you decide when to use each of these?

# Locks

- Inially always free
- Acquire makes threads wait to get the lock
- Release allows other threads to acquire
- Acquiring thread normally has to release

# Semaphores

- Initial value depends on the problem
- “Wait” decrements count; thread must actually wait if the count is 0
- “Signal” increments count; a wait thread may proceed
- Canceling wait and signal need not be from the same thread

# Reader/writer Locks

- Initially allows either reader or writer
- Acquire for read $\Rightarrow$ other readers allowed, writer must wait
- Acquired for write $\Rightarrow$ all other threads must wait
- Release allows waiting threads to proceed
- Acquiring thread normally has to release

# Monitors

- Typically provided by a language
- Acquires a lock before accessing monitor-protected data
- Releases lock after accessing monitor-protected data
- Often uses a re-entrant lock

# Condition Variables

- Cooperate with a lock; sometimes built into a monitor
- “Wait” releases a lock and waits for a signal
- “Signal” wakes up (roughly) one waiting thread (if any)
- “Broadcast” wakes up all waiting threads

Condition variables and semaphores both have “wait” and “signal,” but

- A semaphore can act as a lock
- A semaphore “remembers” a signal until it is used

# Layered Synchronization

Usually, more sophisticated synchronization abstractions are built on simpler ones:

- Reader/writer locks can be implemented with semaphores
- Semaphores can be implemented with spinlocks
- Spinlocks can be implemented with test-and-set
- Test-and-set can be implemented
  - Easily by CPU for a uniprocessor
  - Less easily by CPU for a multiprocessor

Why have so many layers? Why not just implement reader/writer locks using test-and-set?

# Waiting

All synchronization involves waiting

- What are we waiting for?
- Who waits?
- How does the waiting happen?

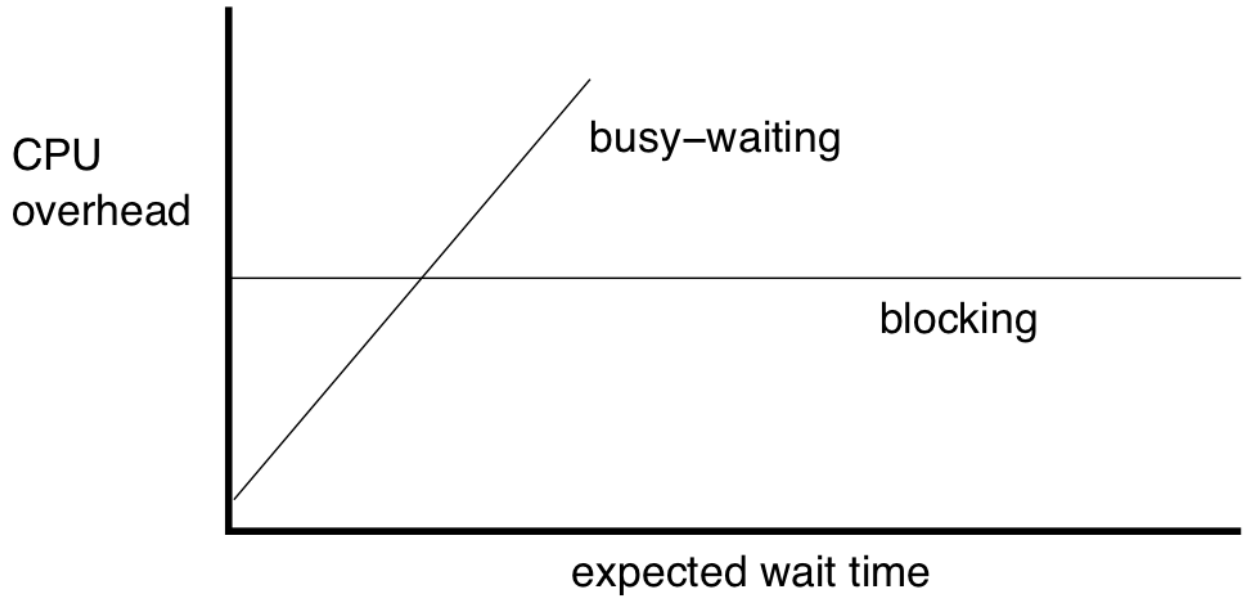
Two ways to wait:

- Busy wait
- Block

Low-level sync tends to involve busy-waiting while high-level sync is blocking — why?

What are the advantages and disadvantages of each?

# Busy Waiting vs. Blocking




An ***adaptive lock*** spins for a while, then blocks

# Safety vs. Liveness

***Safety*** properties ensure that the system will never get into some bad state

- Deadlock
- Races
- Abnormal termination

***Liveness*** properties ensure that the system eventually accomplishes work

# Part IV: Information on HW4

# Implementing a Threads Package

To implement a **user threads** package, you need

- A new stack for each thread
- A way to capture the registers of the current thread and swap in previously saved registers

**`getcontext()`** saves the current registers

**`setcontext()`** restores saved registers

**`makecontext()`** initializes registers given a starting function

# Homework 4

- Due next Friday (midnight)
- Work in groups of 2
  - Pick your own partner
  - Handin by one for both
- Basic user-thread implementation provided
- You implement
  - semaphores
  - sleeping
  - proportional-share scheduling