The deadlock problem

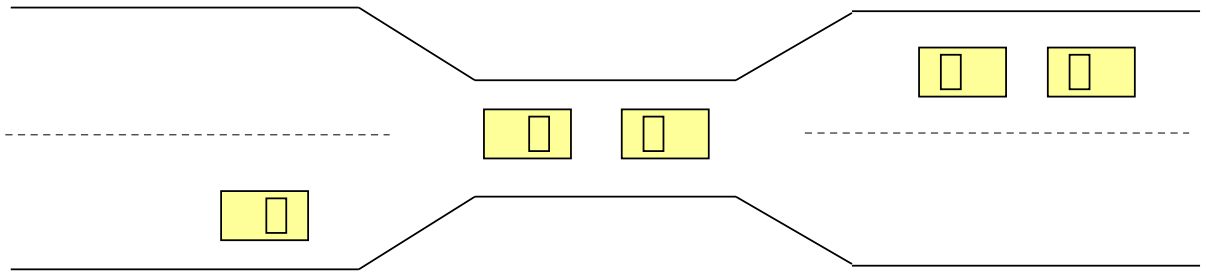
```
mutex_t m1, m2;

void p1 (void *ignored) {
  lock (m1);
  lock (m2);
  /* critical section */
  unlock (m2);
  unlock (m1);
}

void p2 (void *ignored) {
  lock (m2);
  lock (m1);
  /* critical section */
  unlock (m1);
  unlock (m2);
}
```

- **This program can cease to make progress – how?**
- **Can you have deadlock w/o mutexes?**

More deadlocks

- **Same problem with condition variables**
  - Suppose resource 1 managed by c_1, resource 2 by c_2
  - A has 1, waits on $c2$, B has 2, waits on $c1$
- **Or have combined mutex/condition variable deadlock:**
  - `lock (a); lock (b); while (!ready) wait (b, c); unlock (b); unlock (a);`
  - `lock (a); lock (b); ready = true; signal (c); unlock (b); unlock (a);`
- **One lesson: Dangerous to hold locks when crossing abstraction barriers!**
  - I.e., `lock (a)` then call function that uses condition variable

Deadlocks w/o computers



- **Real issue is *resources* & how required**
- **E.g., bridge only allows traffic in one direction**
  - Each section of a bridge can be viewed as a resource.
  - If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
  - Several cars may have to be backed up if a deadlock occurs.
  - Starvation is possible.

Deadlock conditions

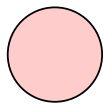
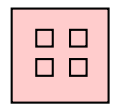
1. **Limited access (mutual exclusion):**
   - Resource can only be shared with finite users.
2. **No preemption:**
   - once resource granted, cannot be taken away.
3. **Multiple independent requests (hold and wait):**
   - don't ask all at once (wait for next resource while holding current one)
4. **Circularity in graph of requests**

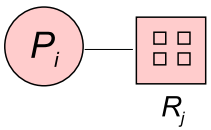
- **All of 1–4 necessary for deadlock to occur**
- **Two approaches to dealing with deadlock:**
  - pro-active: prevention
  - reactive: detection + corrective action

Prevent by eliminating one condition

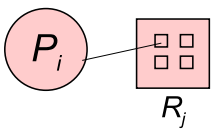
1. **Limited access (mutual exclusion):**
   - Buy more resources, split into pieces, or virtualize to make "infinite" copies
2. **No preemption:**
   - Threads: threads have copy of registers = no lock
   - Physical memory: virtualized with VM, can take physical page away and give to another process!
3. **Multiple independent requests (hold and wait):**
   - Wait on all resources at once (must know in advance)
4. **Circularity in graph of requests**
   - Single lock for entire system: (problems?)
   - Partial ordering of resources (next)

Resource-allocation graph

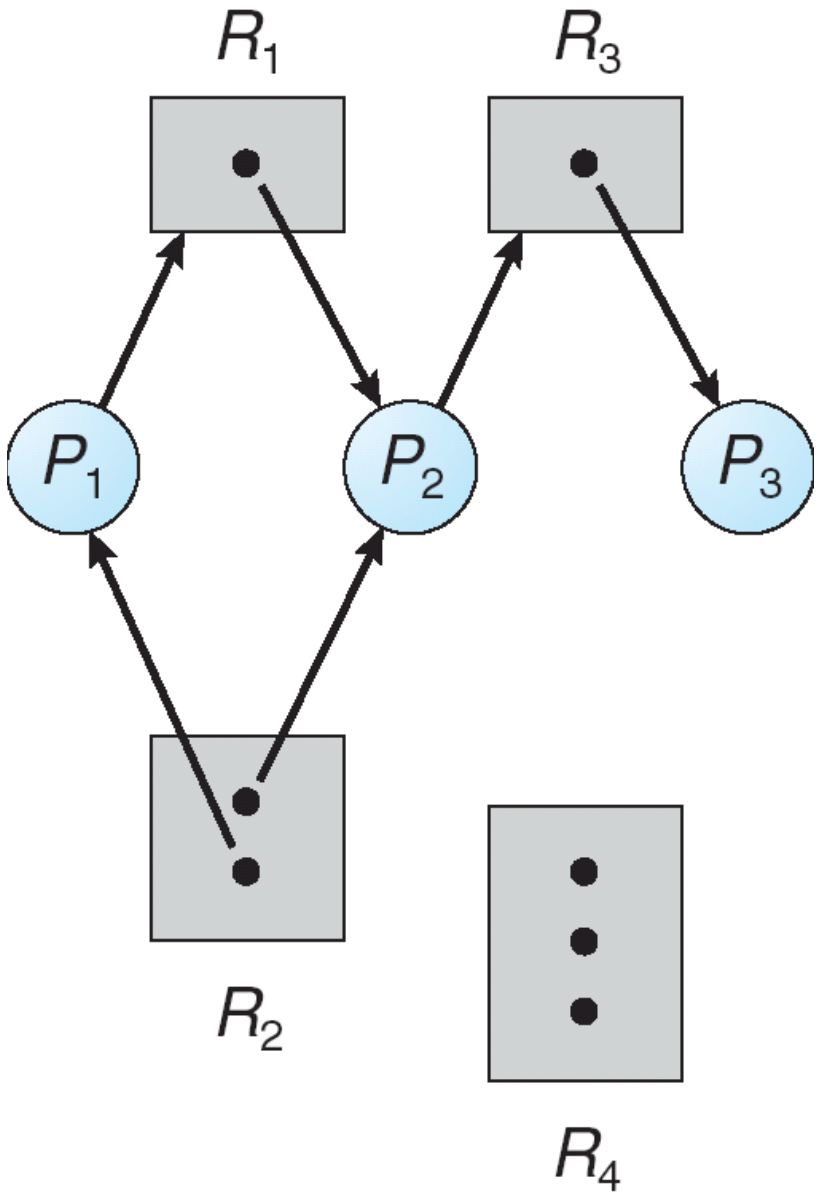
- **View system as graph**
  - Processes and Resources are nodes
  - Resource Requests and Assignments are edges
- **Process:** 
- **Resource w. 4 instances:** 
- P_i **requesting** R_j**:**



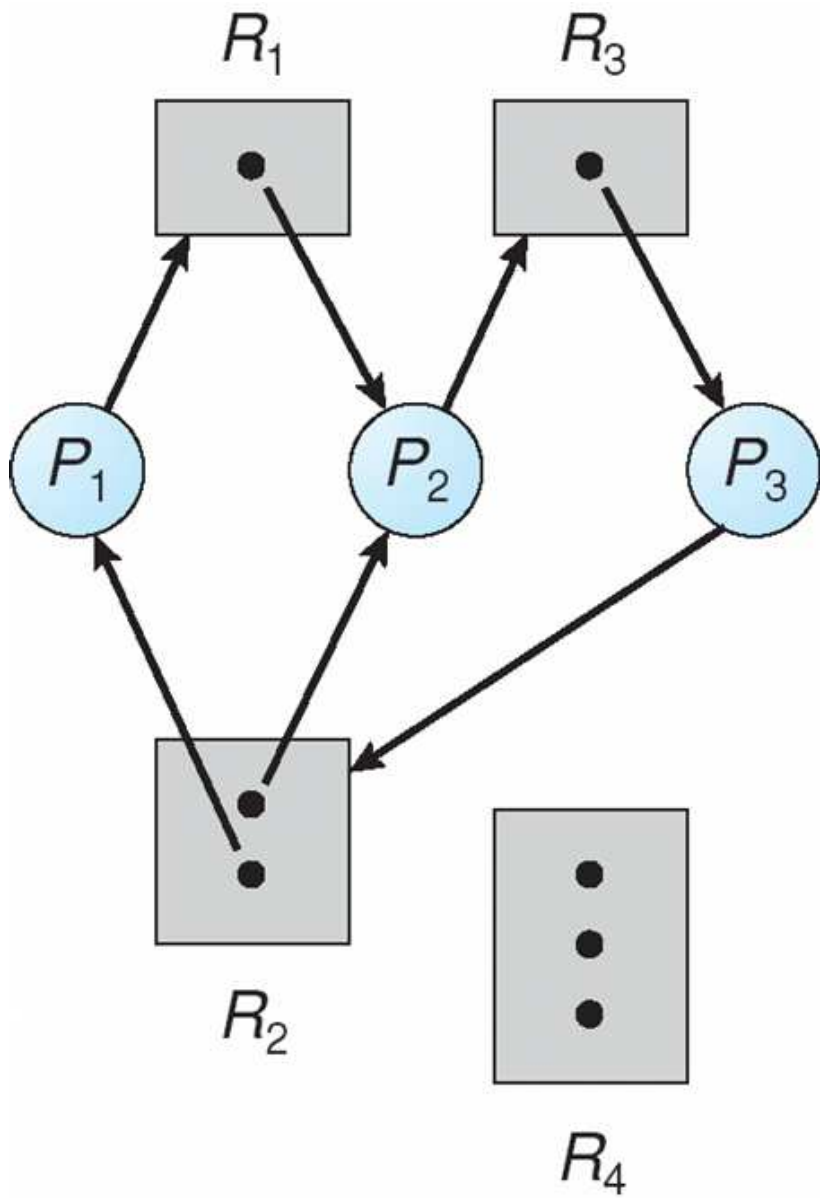
- P_i **holding instance of** R_j**:**



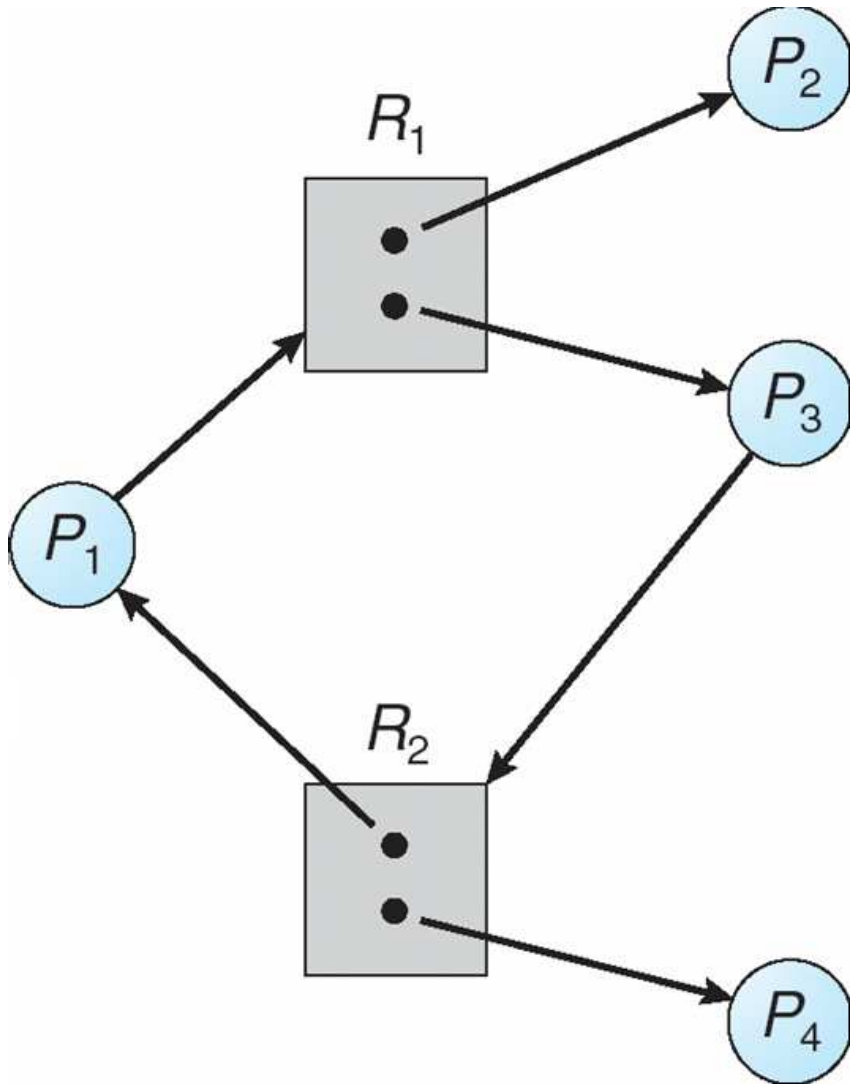
Example resource allocation graph



Graph with deadlock



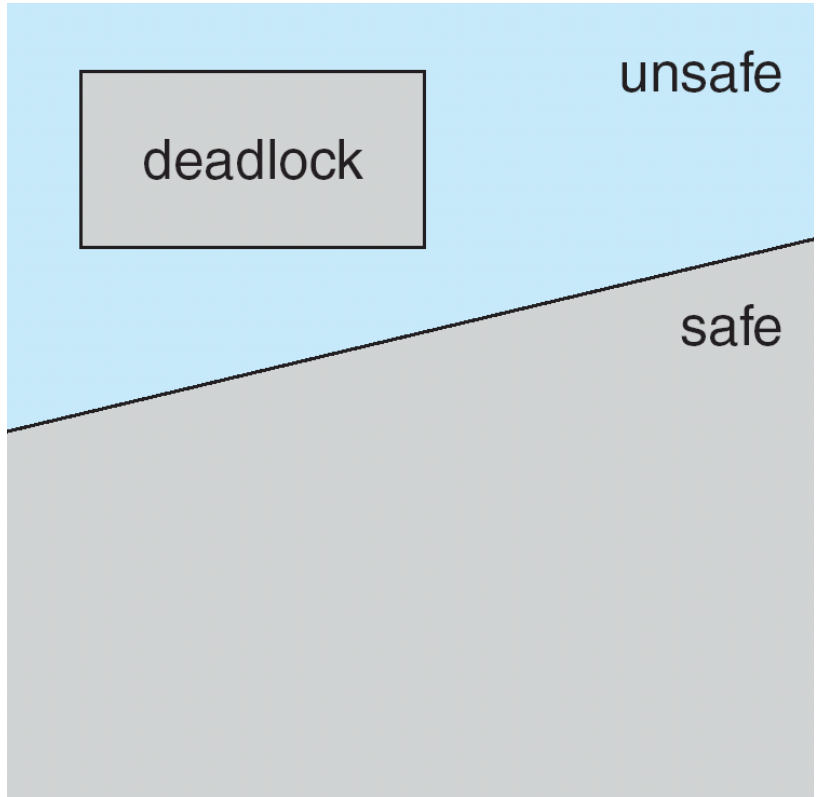
Is this deadlock?



Cycles and deadlock

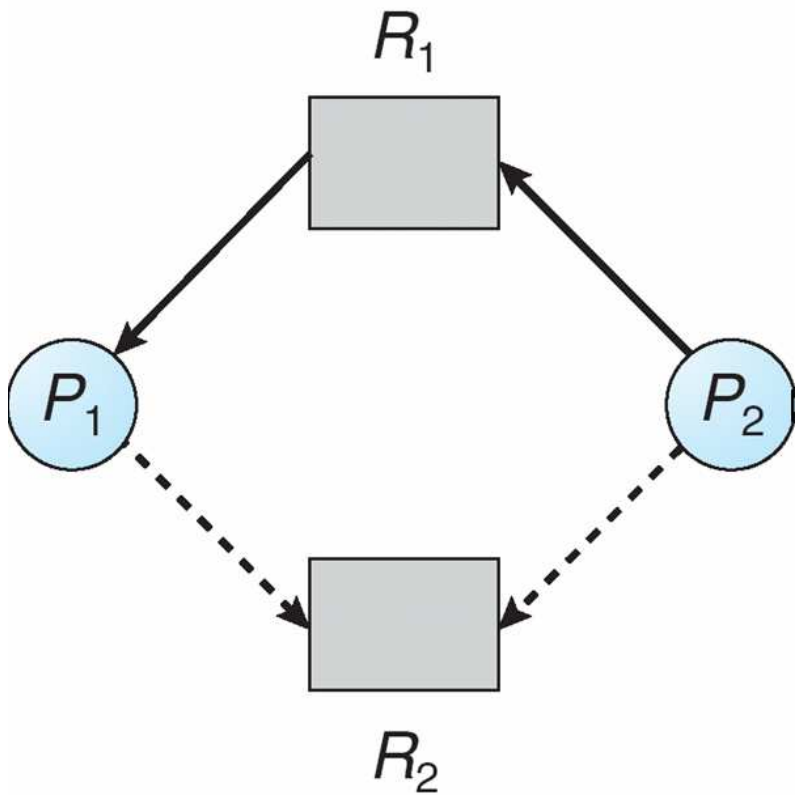
- **If graph has no cycles $\Longrightarrow$ no deadlock**
- **If graph contains a cycle**
  - Definitely deadlock if only one instance per resource
  - Otherwise, maybe deadlock, maybe not
- **Prevent deadlock w. partial order on resources**
  - E.g., always acquire mutex m_1 before m_2
  - Usually design locking discipline for application this way

Prevention



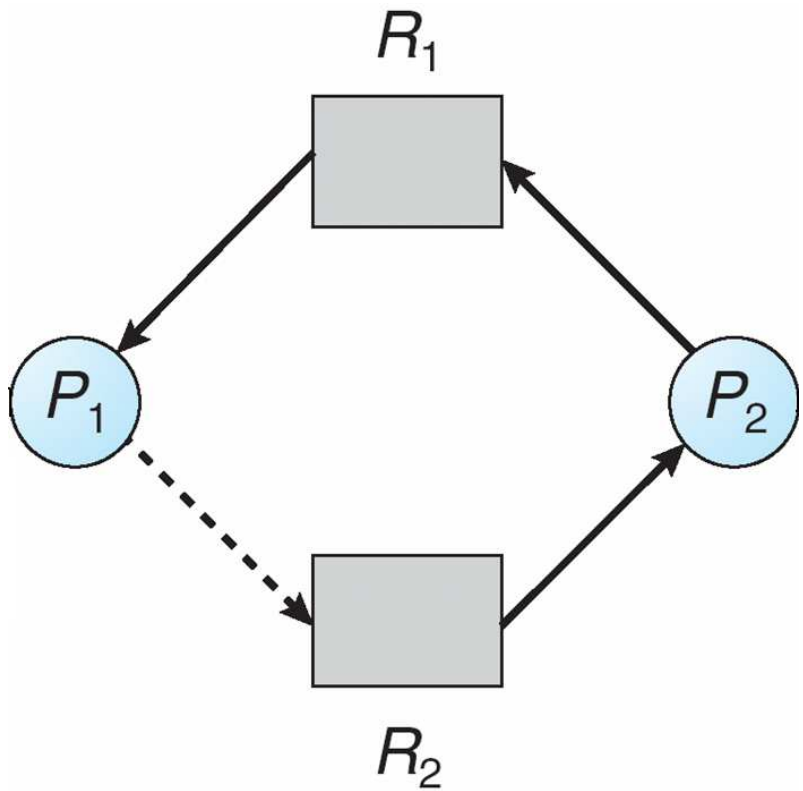
- **Determine safe states based on *possible* resource allocation**
- **Conservatively prohibits non-deadlocked states**

Claim edges



- **Dotted line is *claim edge***
  - Signifies process *may* request resource

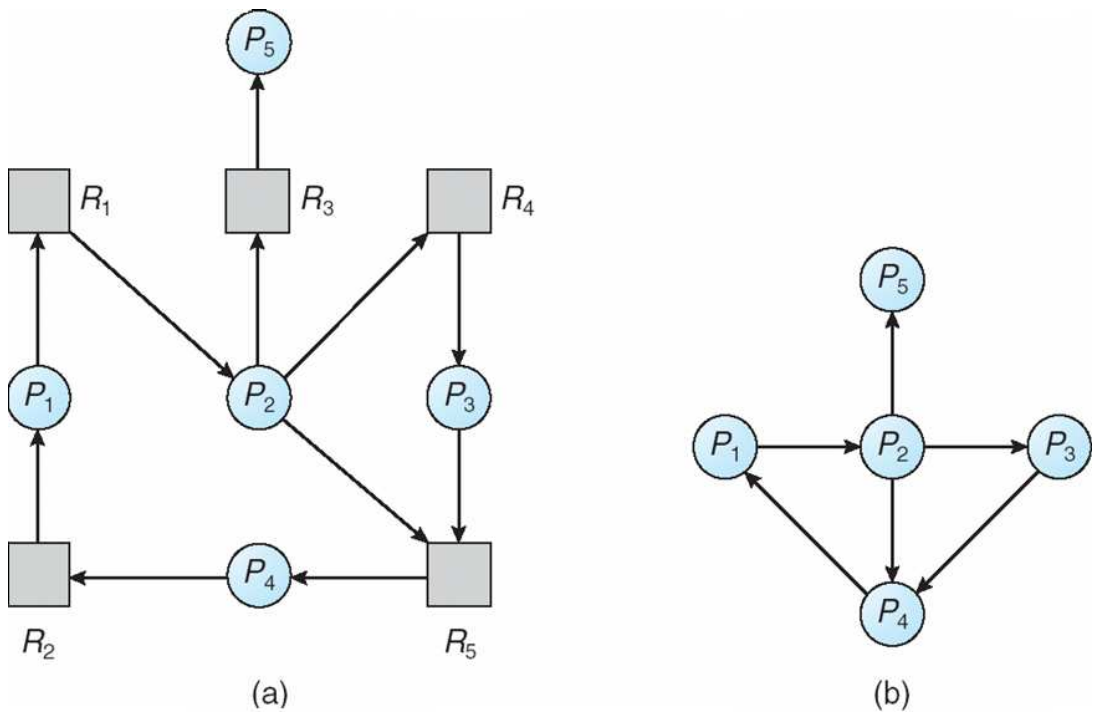
Example: unsafe state



- **Note cycle in graph**
  - P_1 might request R_2 before relinquishing R_1
  - Would cause deadlock

Detecting deadlock

- **Static approaches (hard)**
- **Program grinds to a halt**
- **Threads package can keep track of locks held:**



Resource-Allocation Graph Corresponding wait-for graph

Fixing & debugging deadlocks

- **Reboot system (windows approach)**
- **Examine hung process with debugger**
- **Threads package can deduce partial order**
  - For each lock acquired, order with other locks held
  - If cycle occurs, abort with error
  - Detects *potential* deadlocks even if they do not occur
- **Or use *transactions*...**
  - Another paradigm for handling concurrency
  - Often provided by databases, but some OSes use them
  - *Vino* OS used transactions to abort after failures [Seltzer]
  - OS support for transactional memory now hot research topic