UNIVERSITY OF OREGON

CIS 415: Operating Systems

Deadlock

Spring 2014

Prof. Kevin Butler

- Last class:
  - Synchronization
- Today:
  - Deadlocks

Pthreads Synchronization

UNIVERSITY OF OREGON

- Mutex Locks
  - Protection Critical Sections
  - **pthread_mutex_lock(&lock), pthread_mutex_unlock(&lock)**
- Condition Variables
  - For Value-based Control
  - **pthread_cond_wait(&cond), pthread_cond_signal(&cond)**

```
pthread_mutex_t lock;

big_lock() {

   pthread_mutex_init( &lock );

   /*
   … initial code
   */
   pthread_mutex_lock( &lock );

   /*
   … critical section
   */
   pthread_mutex_unlock( &lock );

   /*
   … remainder
    */
}
```

Put code like around every critical section, like big_lock

What if reading and writing?

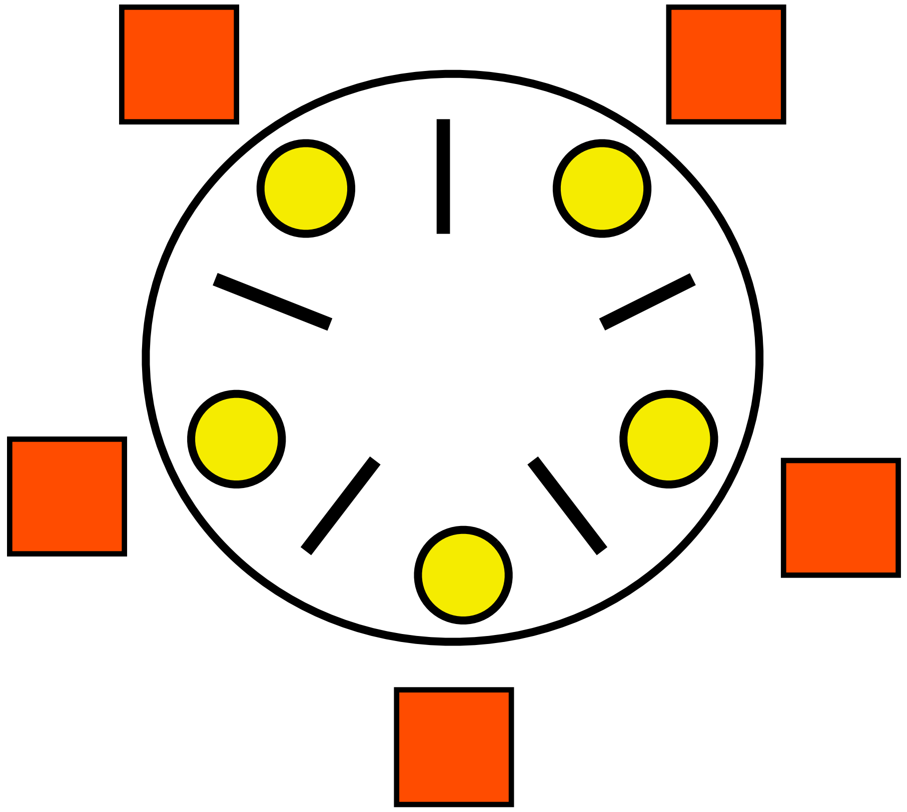
Readers-writers using Pthreads

```
thread_ongoing_t *ongoing;
int nr = 0, nw = 0;
pthread_cond_t OKR, OKW;
```

```
Reader Thread:
    rw.req_read();
    read ongoing
    rw.rel_read();
Writer Thread:
    rw.req_write();
    modify ongoing
    rw.rel_write();
```

```
    // Initialization done elsewhere

void req_read(void) {
    while (nw > 0) pthread_cond_wait(&OKR);
    nr++;
    pthread_cond_signal(&OKR);
}
void rel_read(void) {
    nr--;
    if (nr == 0) pthread_cond_signal(&OKW);
}
void req_write(void) {
    while (nr > 0 || nw > 0) pthread_cond_wait(&OKW);
    nw++;
}
void rel_write(void) {
    nw--;
    pthread_cond_signal(&OKW);
    pthread_cond_signal(&OKR);
}
```

Dining Philosophers Problem



Philosophers alternate between thinking and eating.

When eating, they need both (left and right) chopsticks.

A philosopher can pick up only 1 chopstick at a time.

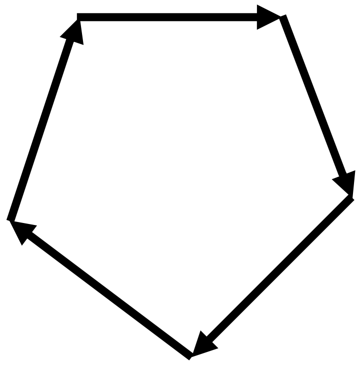
After eating, the philosopher puts down both chopsticks.

```
Semaphore_t chopstick[5];

Philosopher(i) {
   while () {
         P(chopstick[i]);
         P(chopstick[(i+1)%5];

         … eat …

         V(chopstick[i]);
         V(chopstick[(i+1)%5];

         … think …
    }
}
```

This is NOT correct!

Though no 2 philosophers use the same chopstick at any time, it can so happen that they all pick up 1 chopstick and wait indefinitely for another.

This is called a deadlock



Definition

UNIVERSITY OF OREGON

- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

- An event could be:
  - Waiting for a critical section
  - Waiting for a condition to change
  - Waiting for a physical resource

Conditions for Deadlock

- Mutual exclusion: The requesting process is delayed until the resource held by another is released.
- Hold and wait: A process must be holding at least 1 resource and must be waiting for 1 or more resources held by others.
- No preemption: Resources cannot be preempted from one and given to another.
- Circular wait: A set (P0,P1,…Pn) of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for …. by P2, … Pn is waiting for … held by P0.

Resource Allocation Graph

UNIVERSITY OF OREGON

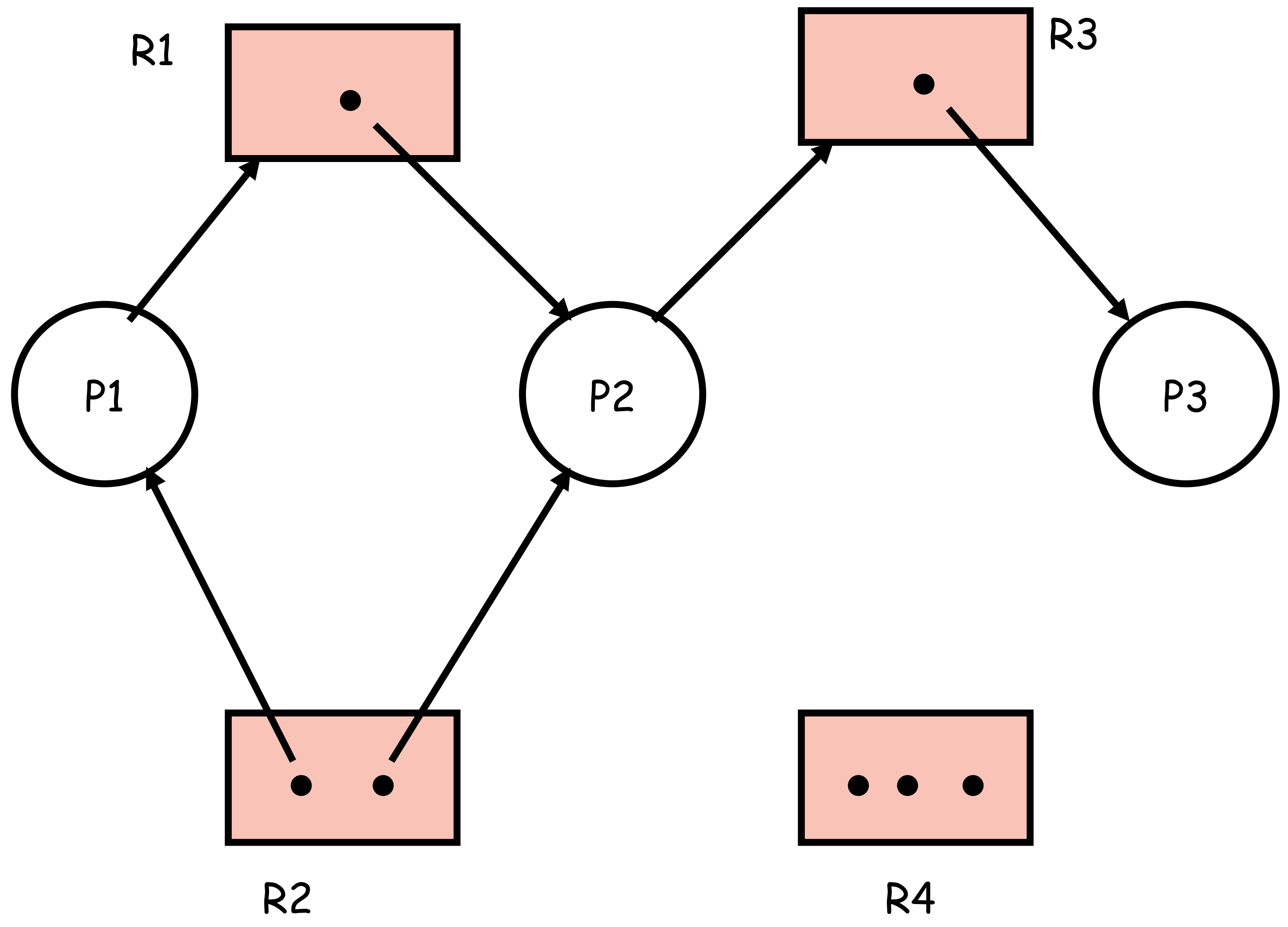
- Vertices (V) = Processes (Pi) and Resources (Rj)

- Edges (E) = Assignments (Rj->Pi, Rj is allocated to Pi) and Request (Pi->Rj, Pi is waiting for Rj).

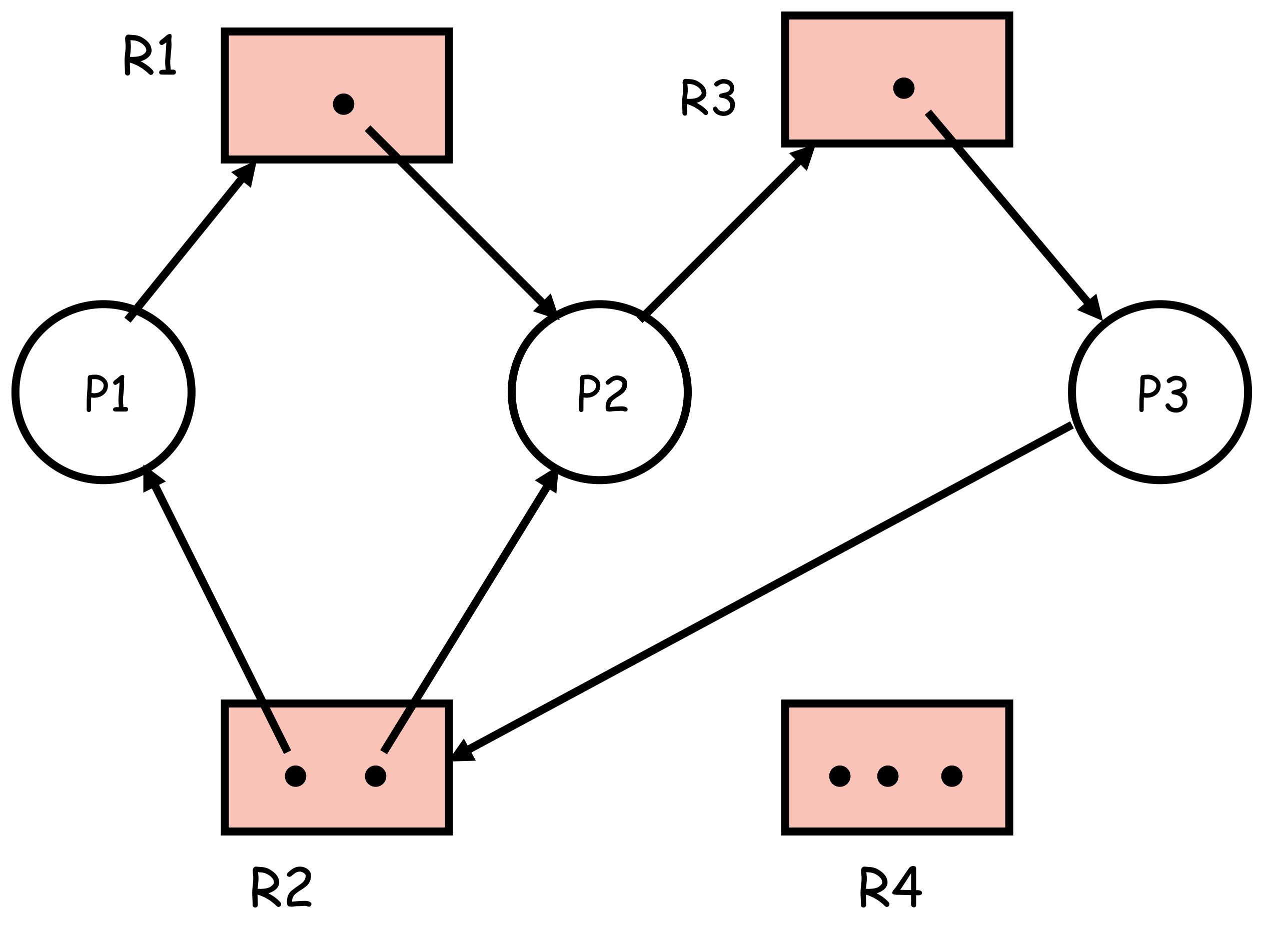
- For each Resource Rj, there could be multiple instances.

- A requesting process can be granted any one of those instances if available.

An example

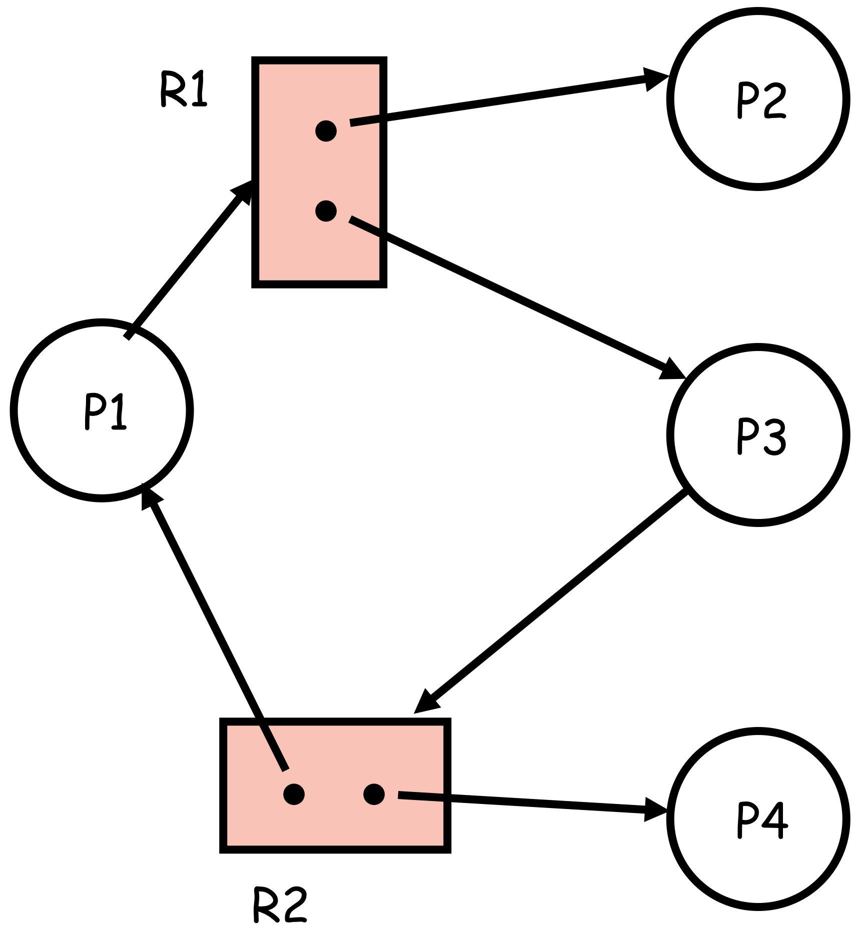


A deadlock



If there is a deadlock, there will be a cycle (Necessary Condition).

Cycle is NOT sufficient



Strategies for Handling Deadlocks

- Ignore the problem altogether (ostrich algorithm) since it may occur very infrequently, cost of detection/ prevention may not be worth it.
- Detect and recover after its occurrence.
- Avoidance by careful resource allocation
- Prevention by structurally negating one of the four necessary conditions

Deadlock Prevention

UNIVERSITY OF OREGON

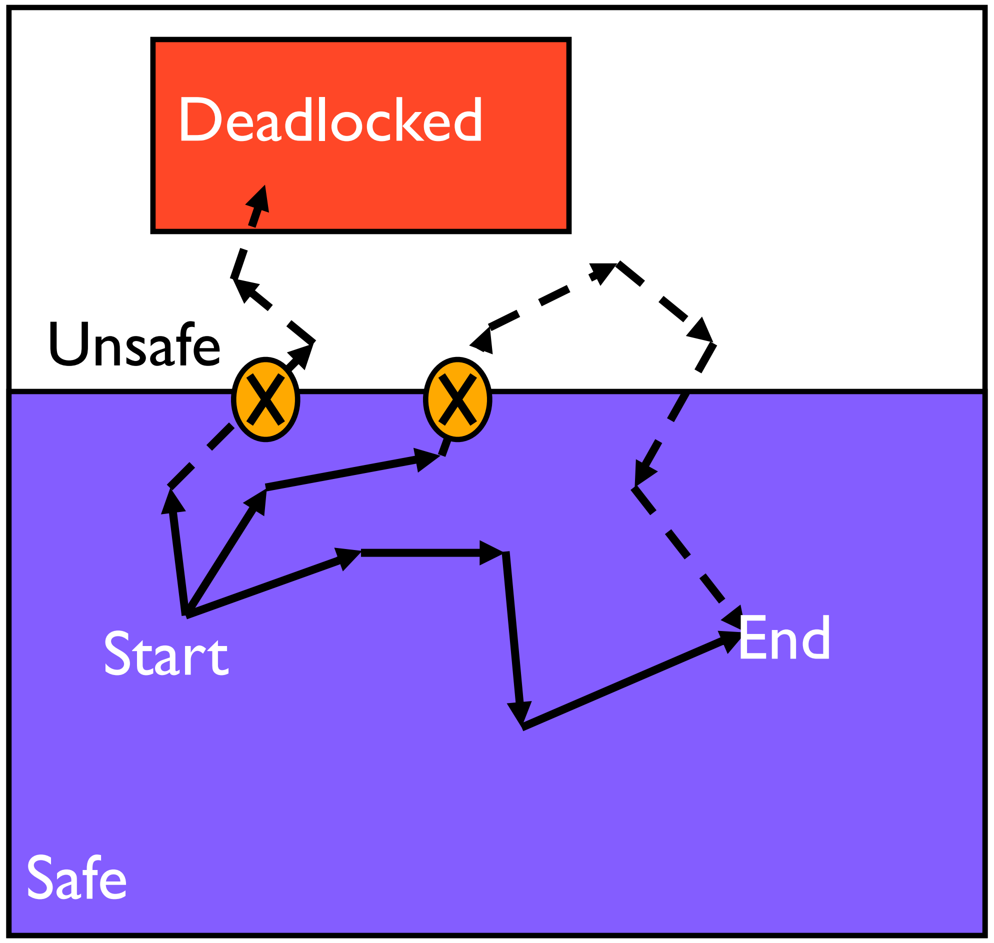
- Note that all 4 necessary conditions need to hold for deadlock to occur.
- We can try to disallow one of them from happening:
  - Mutual exclusion: This is usually not possible to avoid with many resources.
  - No preemption: This is again not easy to address with many resources. Possible for some resources (e.g. CPU)
  - Hold and Wait:
    - Allow at most 1 resource to be held/requested at any time
    - Make sure all requests are made at the same time.
    - …

- Circular Wait
  - Number the resources, and make sure requests are always made in increasing/decreasing order.
  - Or make sure you are never holding a lower numbered resource when requesting a higher numbered resource.

Deadlock Avoidance

- Avoid actions that may lead to a deadlock.
- Visualize the system as a state machine moving from 1 state to another as each instruction is executed.
- A state can be: *safe*, *unsafe* or *deadlocked*.
- Safe state is one where
  - it is not a deadlocked state
  - there is some sequence by which all requests can be satisfied.
- To avoid deadlocks, we try to make only those transitions that will take you from one safe state to another.
- This may be a little conservative, but it avoids deadlocks

State Transitions



Safe State

UNIVERSITY OF OREGON

1 resource with 12 units of that resource available.

Current State: Free = (12 – (5 + 2 + 2)) = 3

	Max. Needs	Currently Allocated	Still Needs
P0	10	5	5
P1	4	2	2
P2	9	2	7

Free = 3

After reducing P1,
Free = 5

After reducing P0,
Free = 10
Then reduce P2.

This state is safe because, there is a sequence (P1 followed by P0 followed by P2) by which max needs of each process can be satisfied. This is called the reduction sequence.

Unsafe State

UNIVERSITY OF OREGON

What if P2 requests 1 more and is allocated 1 more?

New State:

	Max. Needs	Currently Allocated	Still Needs
P0	10	5	5
P1	4	2	2
P2	9	3	6

Free = 2

This is unsafe.

Only P1 can be reduced. If P0 and P2 then come and ask for their full needs, the system can become deadlocked. Hence, by granting P2's request for 1 more, we have moved from a safe to unsafe state. Deadlock avoidance algorithm will NOT allow such a transition, and will not grant P2's request immediately.

Deadlock Avoidance

- Deadlock avoidance essentially allows requests to be satisfied only when the allocation of that request would lead to a safe state.

- Else do not grant that request immediately.

Banker's algorithm

- When a request is made, check to see if after the request is satisfied, there is (at least one!) sequence of moves that can satisfy all possible requests. ie. the new state is safe.

- If so, satisfy the request, else make the request wait.

Checking for Safe State

```
N processes and M resources

Data Structures:
  MaxNeeds[N][M];
  Allocated[N][M];
  StillNeeds[N][M];
  Free[M];
  Temp[M];
  Done[N];
```

```
while () {
   Temp[j]=Free[j] for all j
   Find an i such that
       a) Done[i] = False
       b) StillNeeds[i,j] <= Temp[j]
   if so {
       Temp[j] += Allocated[i,j] for all j
       Done[i] = TRUE   /* release Allocated[i] */
   }
   else if Done[i] = TRUE for all i then state is safe
   else state is unsafe
}
```

M*N^2 steps to detect if a state is safe!

An example

5 processes, 3 resource types A (10 instances), B (5 instances), C (7 instances)

MaxNeeds

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Allocated

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

StillNeeds

	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Free

A	B	C
3	3	2

This state is safe, because there is a reduction sequence <P1, P3, P4, P2, P0> that can satisfy all the requests.
Exercise: Formally go through each of the steps that update these matrices for the reduction sequence.

An example

If P1 requests 1 more instance of A and 2 more instances of C
can we safely allocate these? – Note these are all allocated together!
and we denote this set of requests as (1,0,2)

If allocated the resulting state would be:

MaxNeeds

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Allocated

	A	B	C
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2

StillNeeds

	A	B	C
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

Free

A	B	C
2	3	0

This is still safe since there is a reduction sequence <P1,P3,P4,P0,P2> to satisfy all the requests. (work this out!) Hence the requested allocations can be made.

An example

After this allocation, P0 then makes a request for (0,2,0).
If granted the resulting state would be:

MaxNeeds

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Allocated

	A	B	C
P0	0	3	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2

StillNeeds

	A	B	C
P0	7	2	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

Free

A	B	C
2	1	0

This is an UNSAFE state.

So this request should NOT be granted.

Handling Deadlocks

UNIVERSITY OF OREGON

- Ignore the problem altogether (ostrich algorithm) since it may occur very infrequently, cost of detection/prevention may not be worth it.
- ➡ Detect and recover after its occurrence.
- Avoidance by careful resource allocation
- Prevention by structurally negating one of the four necessary conditions

Detection & Recovery

UNIVERSITY OF OREGON

- If there is only 1 instance of each resource, then a cycle in the resource-allocation graph is a "sufficient" condition for a deadlock, i.e. you can run a cycle-detection algorithm to detect a deadlock.
- With multiple instances of each resource, ???

Detection Algorithm

UNIVERSITY OF OREGON

```
N processes, M resources

Data structures:
  Free[M];
  Allocated[N][M];
  Request[N][M];
  Temp[M];
  Done[N];
```

```
1. Temp[i] = Free[i] for all i
     Done[i] = FALSE unless there is
     no resources allocated to it.

2. Find an index i such that both
    (a) Done[i] == FALSE
    (b) Request[i] <= Temp (vector comp.)
    If no such i, go to step 4.

3. Temp = Temp + Allocated[i] (vector add)
    Done[i]= TRUE;  /* release Allocated[i] */
    Go to step 2.

4. If Done[i]=FALSE for some i, then
     there is a deadlock.
```

Basic idea is that there is at least 1 execution that will unblock all processes.

M*N^2 algorithm!

Example

UNIVERSITY OF OREGON

5 processes, 3 resource types A (7 instances), B (2 instances), C (6 instances)

Allocated

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2

Request

	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	0	0	2

Free

A	B	C
0	0	0

This state is **NOT** deadlocked.

By applying algorithm, the sequence <P0, P2, P3, P1, P4> will result in Done[i] being TRUE for all processes.

Recovery

UNIVERSITY OF OREGON

- Once deadlock is detected what should we do?
  - Preempt resources (whenever possible)
  - Kill the processes (and forcibly remove resources)
  - Checkpoint processes periodically, and roll them back to last checkpoint (relinquishing any resources they may have acquired since then).

Ordering

- To date: we've thought about how to order events in order to provide synchronization and prevent deadlock
- What have we been relying on in order to get ordering?
  - A consistent clock amongst processes
- What happens when processes aren't sharing the same clock?
  - Distributed system

Happened-Before

- Without sharing a clock, it's not possible to get a total ordering over events
  - Instead, we get partial ordering
- Within a sequential process, all events are executed in a totally ordered fashion
- Message can only be received after it's sent
- *Happened-before* relation → reflects partial ordering

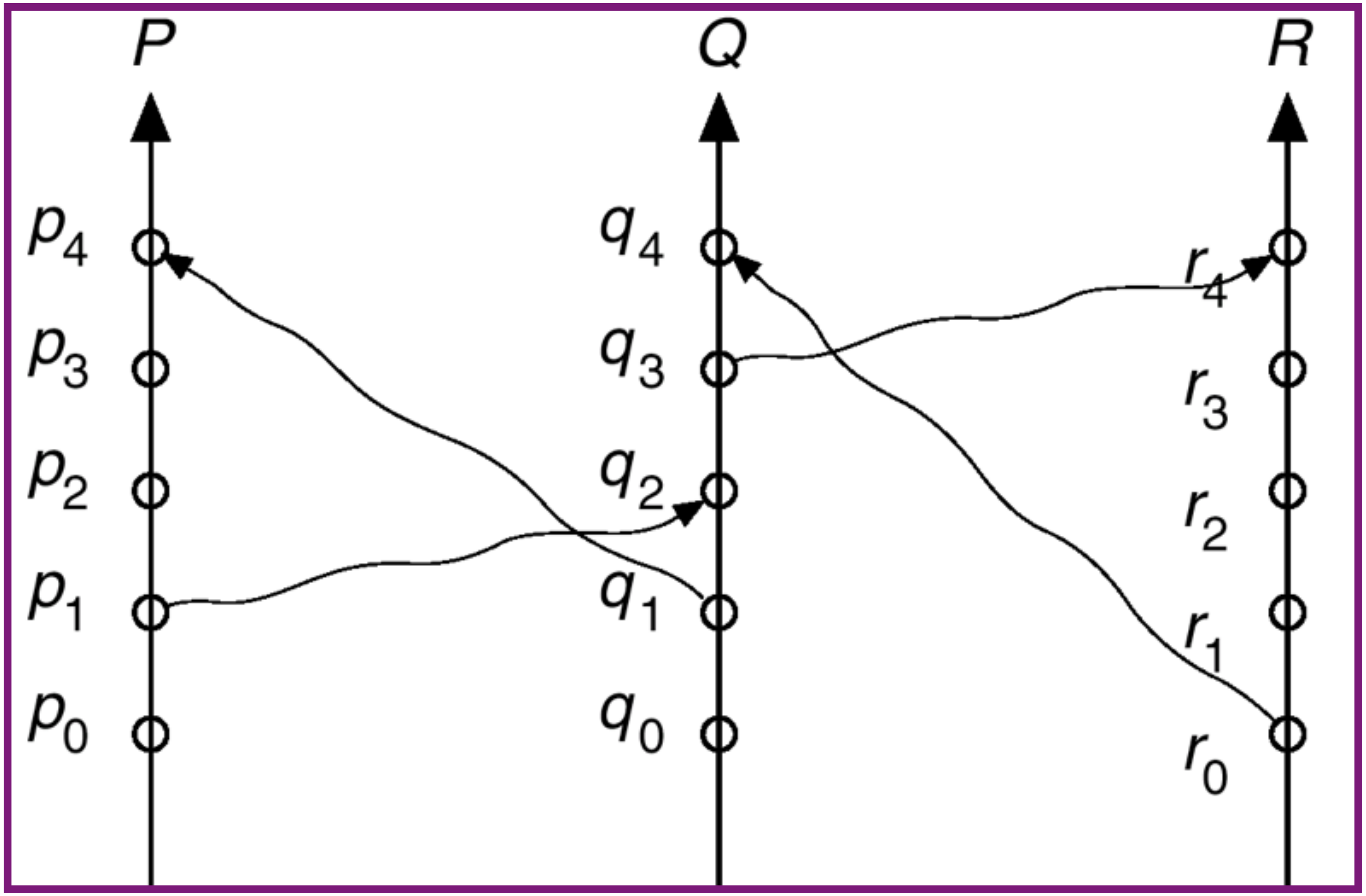
Happened-Before

- Properties of the happened-before relation
  - If A, B are events in the same process and A executes before B then A $\rightarrow$ B
  - If A is a send message event from a process and B is a receive message event from another process then A $\rightarrow$ B
  - If A $\rightarrow$ B and B $\rightarrow$ C then A $\rightarrow$ C *(what property is this?)*

- If events A and B are not related by $\rightarrow$ then they can execute *concurrently* (no effect of A on B)

Concurrent vs →

UNIVERSITY OF OREGON



Implementing →

- Associate a timestamp with each system event
  - Require that for every pair of events A and B, if A → B, then the timestamp of A is less than the timestamp of B
- Associate *logical (Lamport) clock* LC_i with process P_i
  - Implement as a simple counter incremented between any two successive events executed within a process.
- Process advances logical clock when receiving message with timestamp > current value of LC
- If $TS_A == TS_B$, events are concurrent (use process ID to break ties and create a total ordering)

Summary

- Deadlocks
  - Necessary and sufficient conditions
    - Resource allocation graph
  - Strategies
    - Ignore
    - Prevention
      - Safe States
    - Avoidance
    - Detection and recovery
    - Distributed ordering