Synchronization – Monitors and CV

CS 416: Operating Systems Design, Spring 2011

Department of Computer Science
Rutgers University

Java Condition Variables

Wait(Lock lock)

- Release the lock
- Put thread object on wait queue of this CondVar object
- Yield the CPU to another thread
- When waken by the system, reacquire the lock and return

Notify()

- If at least 1 thread is sleeping on cond_var, wake 1 up. Otherwise, no effect
- Waking up a thread means changing its state to Ready and moving the thread object to the run queue

NotifyAll()

- If 1 or more threads are sleeping on cond_var, wakeup everyone

Implementing Wait and Notify

```
Wait(lock){
    schedLock->acquire();
    lock->numWaiting++;
    lock→release();
     Put TCB on the waiting queue for the CV;
     schedLock->release()
     switch();
     lock→acquire();  -> The lock has to be re-acquired
}
```

Why do we need schedLock?

```
Notify(lock){
     schedLock->acquire();
     if (lock->numWaiting > 0) {
               Move a TCB from waiting queue to ready queue;
               lock->numWaiting--;
     }
     schedLock->release();
}
```

Re-Writing Producer/Consumer with CV

```
Class MyBuffer{
        Buffer[BUFFER_SIZE]
        Lock lock;
        int count = 0;
        Condition notFull, notEmpty;
}
```

Checking a CV should always be done inside a lock
Why ?

Functions defined within the class

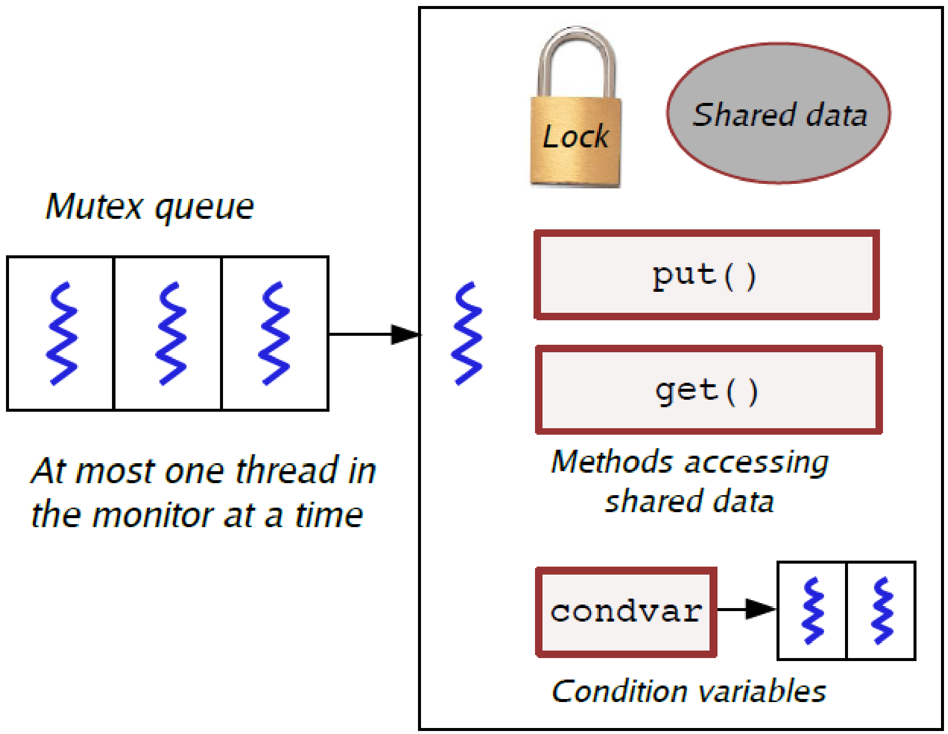
```
put(){
    lock→acquire();
    while (count == n) {
            notFull.wait(&lock); }
    Add items to buffer;
    count++;
    notEmpty.notify();
    lock→release();
}
```

```
get(){
    lock→acquire();
    while (count == 0) {
            notEmpty.wait(&lock); }
    Remove items from buffer
    count--;
    notFull.notify();
    lock→release();
}
```

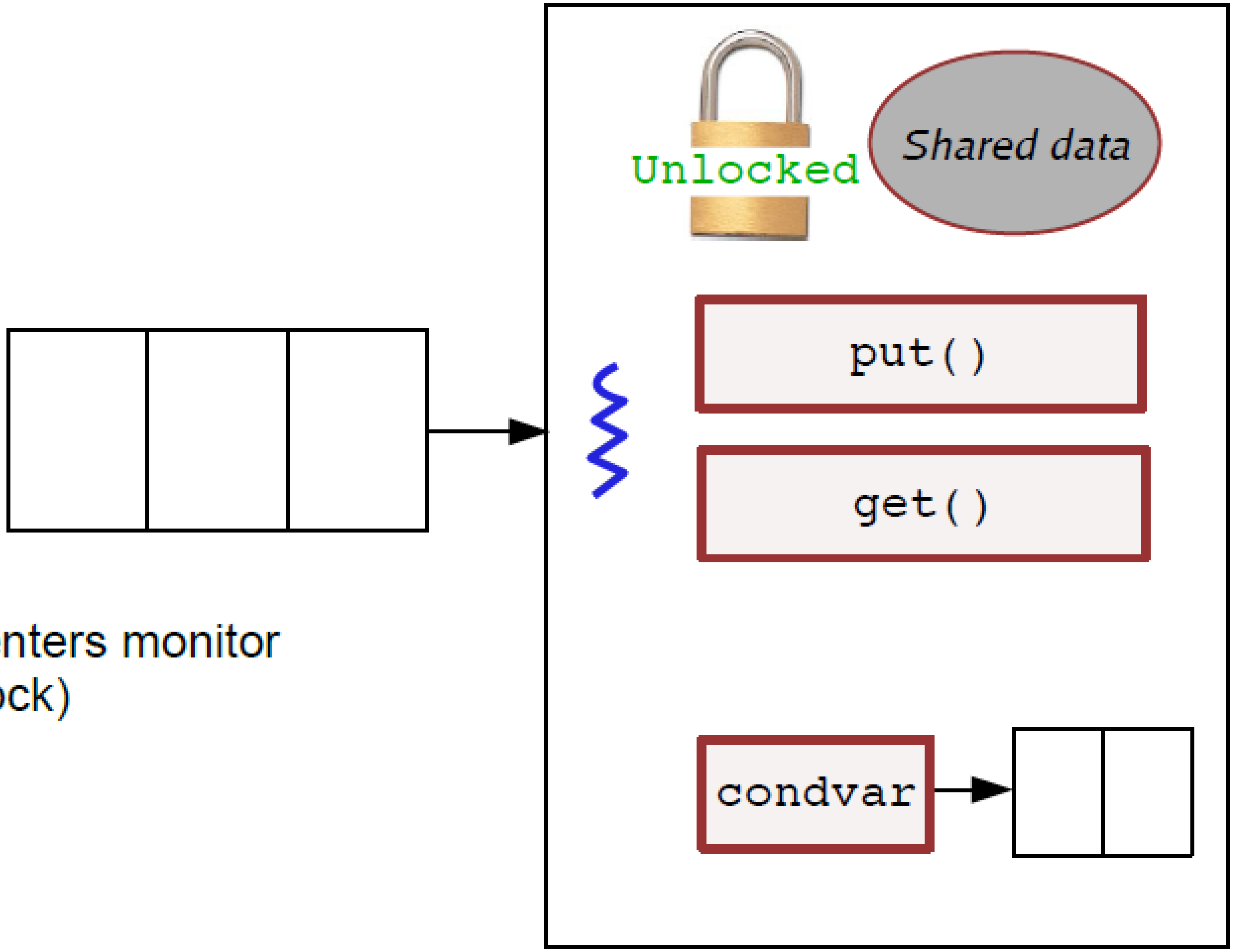
Monitors

This style of using locks and CVs to protect access to a shared object is called a monitor

- Monitor is like a lock protecting an object, its methods and the associated condition variables

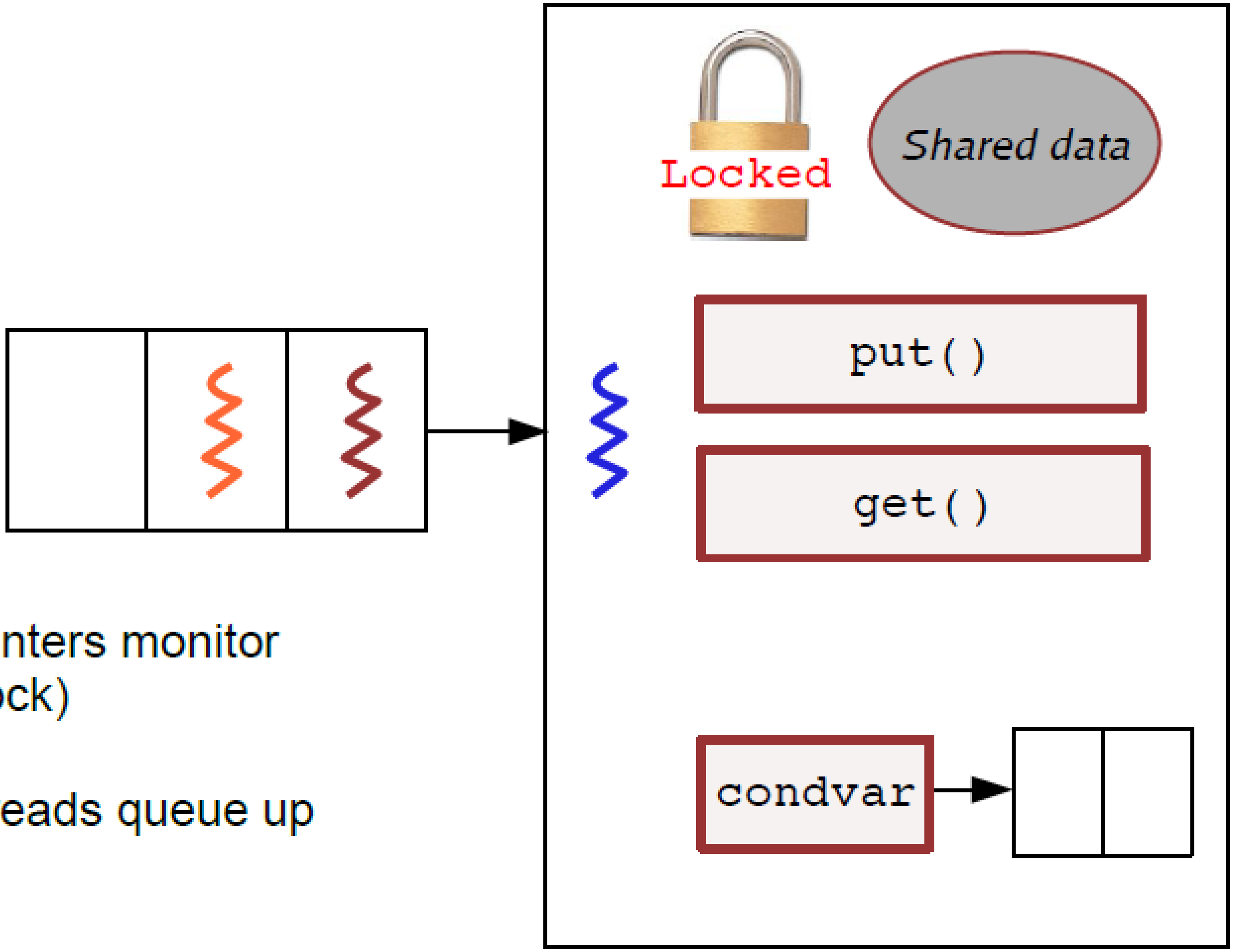


Monitors



1) Thread enters monitor
(grabs lock)

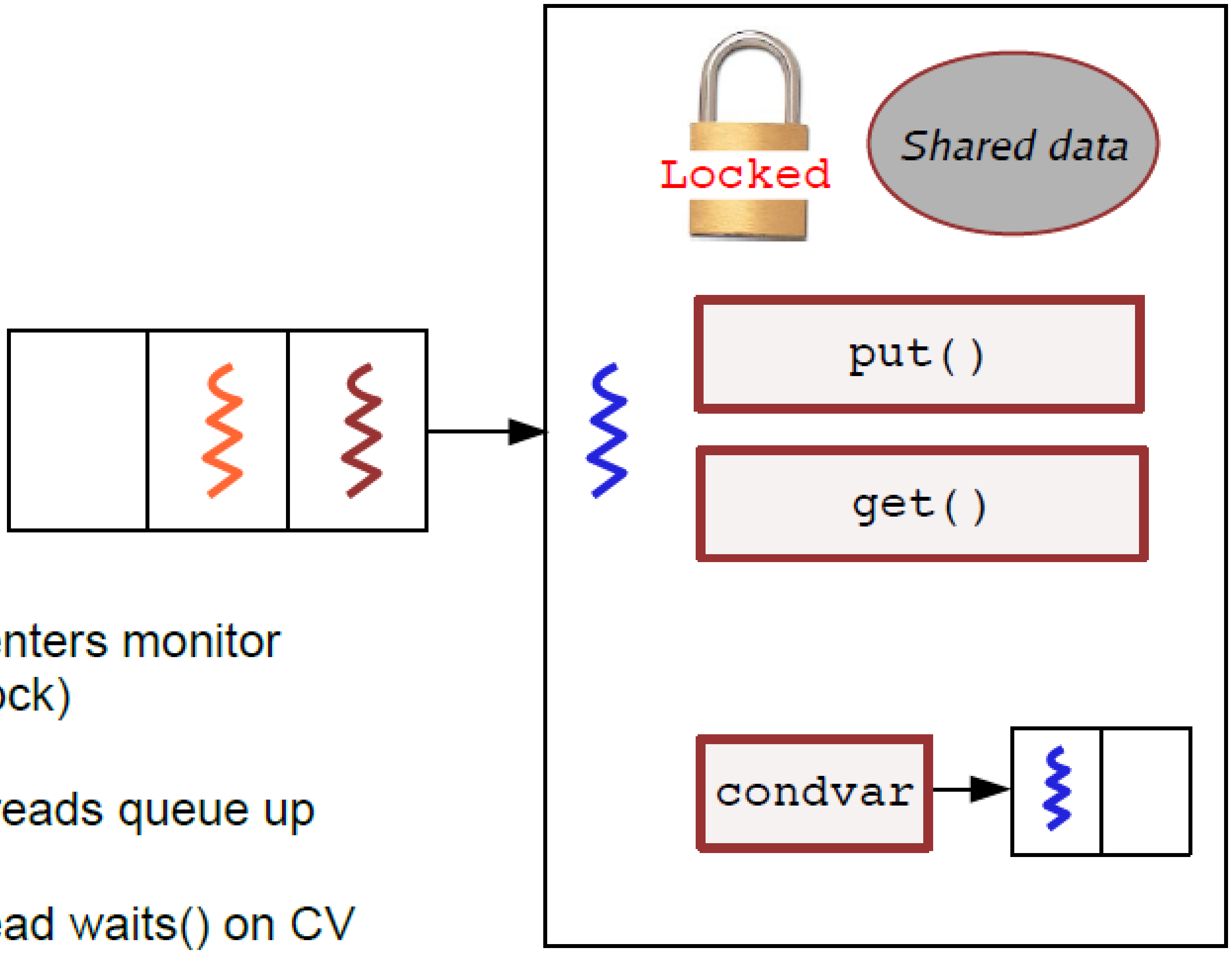
Monitors



1) Thread enters monitor
(grabs lock)

2) Other threads queue up

Monitors

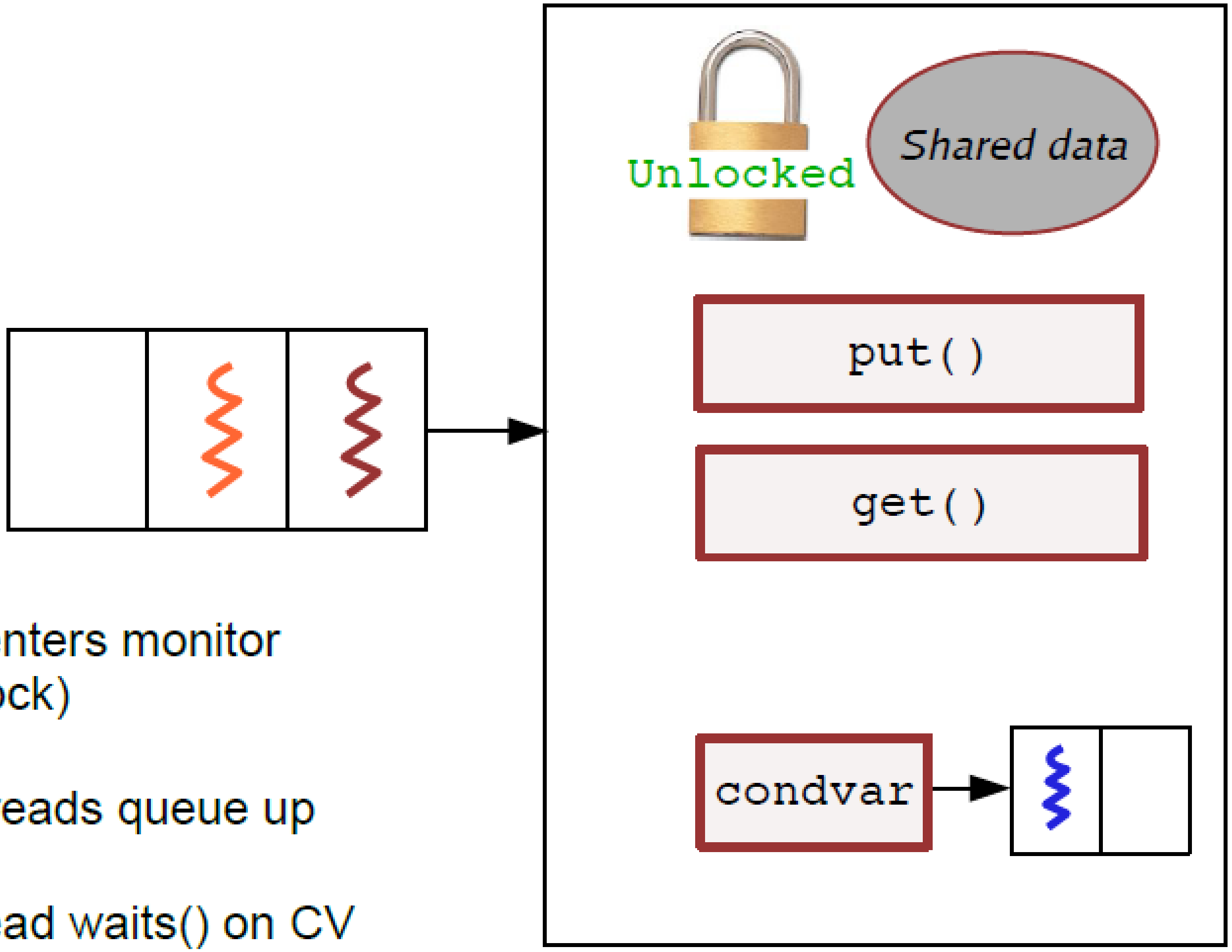


1) Thread enters monitor (grabs lock)

2) Other threads queue up

3) Blue thread waits() on CV

Monitors



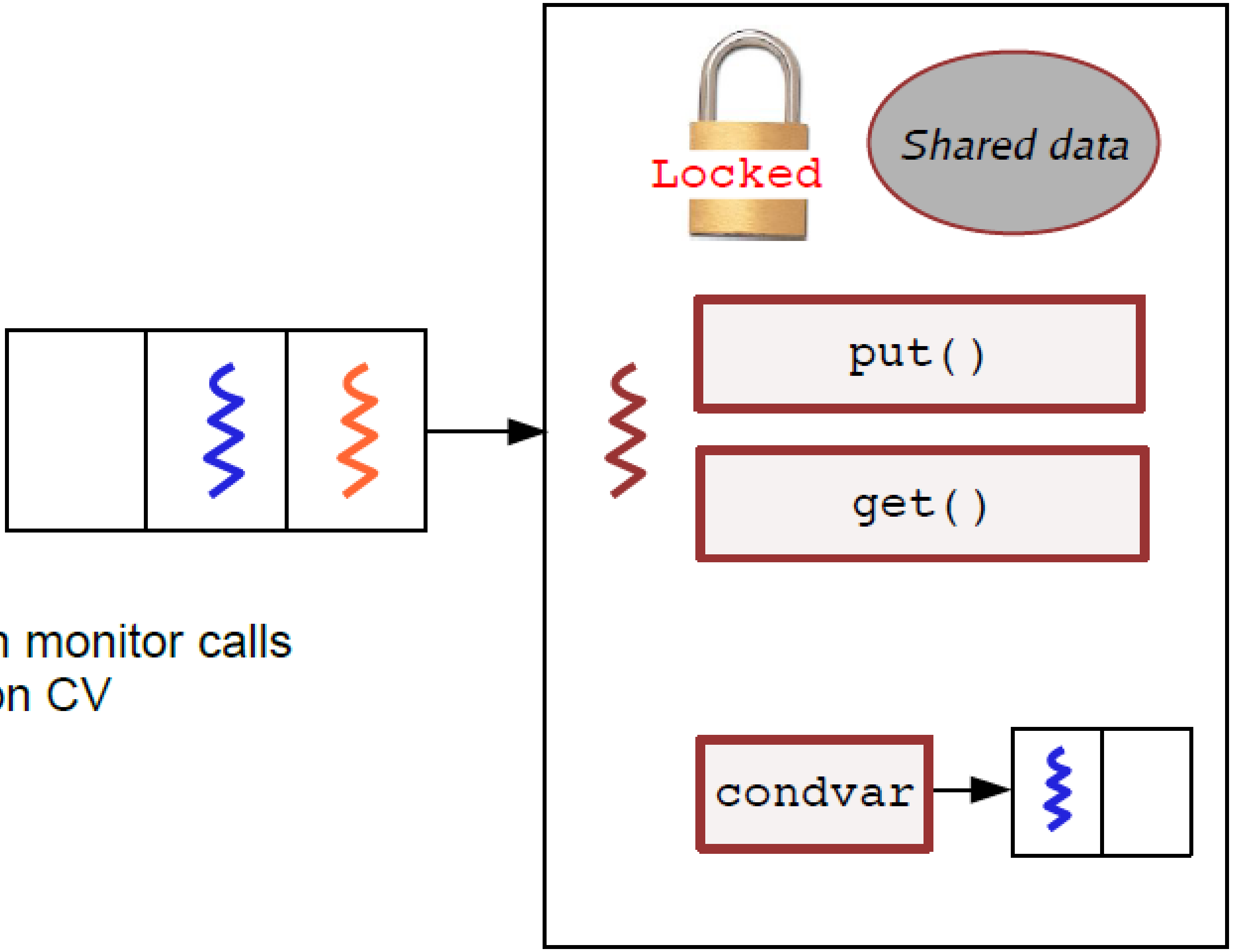
1) Thread enters monitor (grabs lock)

2) Other threads queue up

3) Blue thread waits() on CV

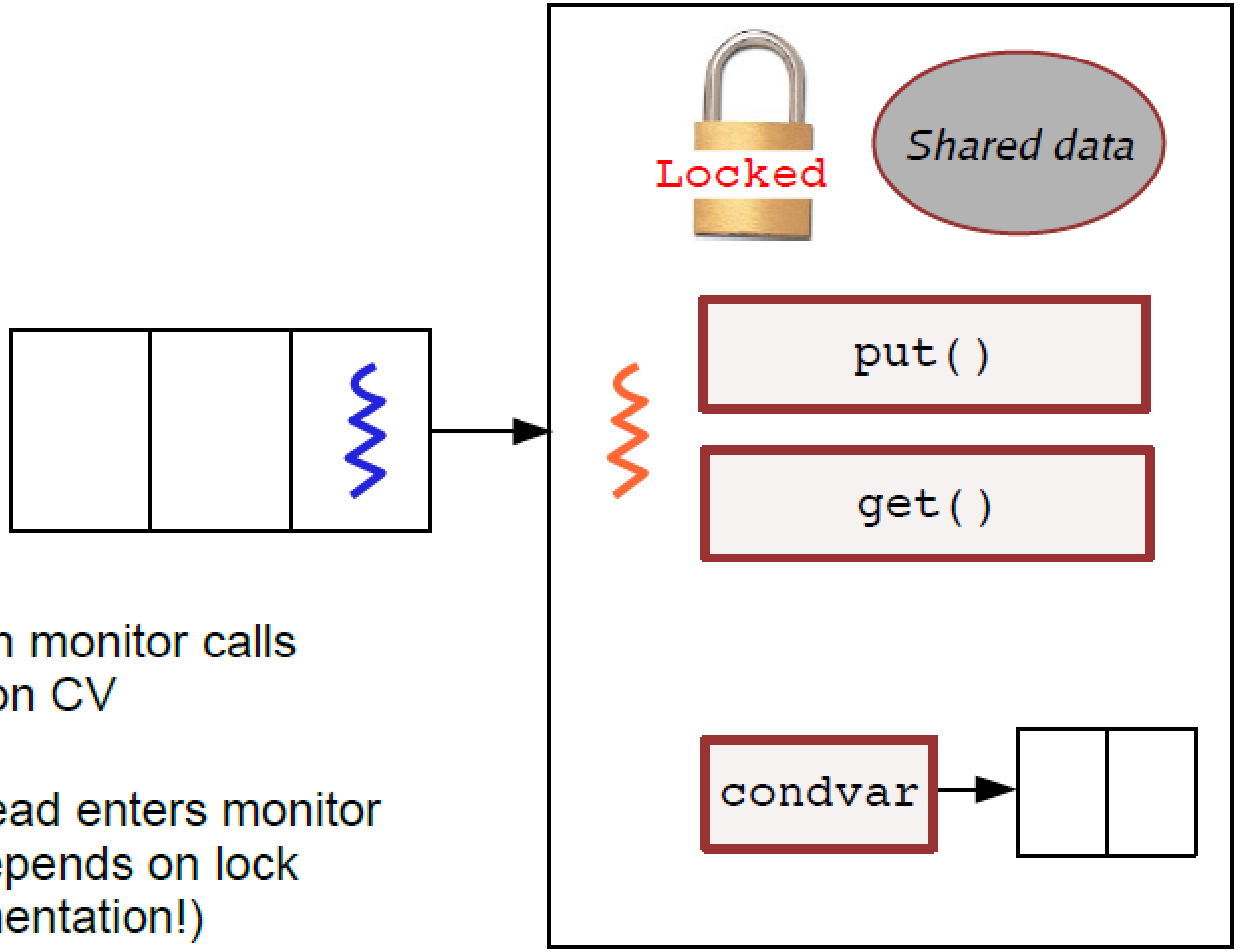
4) Next thread enters monitor

Monitors



5) Thread in monitor calls
notify() on CV

Monitors



5) Thread in monitor calls notify() on CV

6) Next thread enters monitor (order depends on lock implementation!)

Condition Vars != Semaphores

- Condition Variables != Semaphores
  - Although their operations have the same names, they have entirely different semantics.
  - However, they each can be used to implement the other. How ?
- Access to the monitor is controlled by a lock
  - wait() blocks the calling thread, and gives up the lock
    - To call wait, the thread has to be in the monitor (hence has lock)
    - Semaphore::wait just blocks the thread on the queue
  - signal() causes a waiting thread to wake up
    - » If there is no waiting thread, the signal is lost
    - » Semaphore::signal increases the semaphore count, allowing future entry even if no thread is waiting
    - » Condition variables have no history

Monitors: Syntax

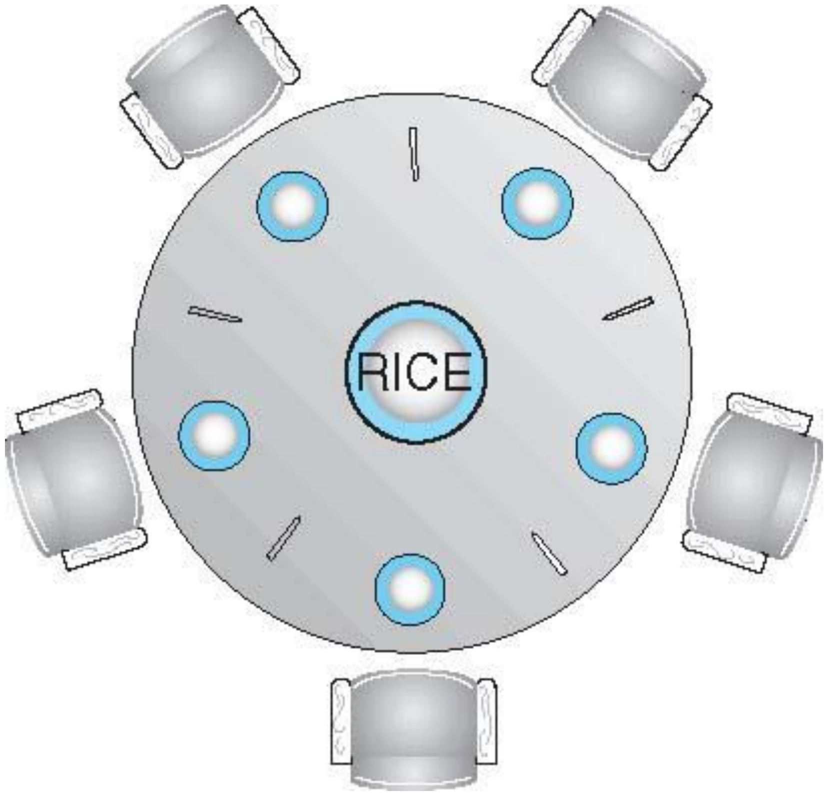
Only one process may be active within the monitor at a time

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
                …

        procedure Pn (…) {……}

        Initialization code ( ….) { … }
                …
}
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem

The structure of Philosopher *i*:

```
do  {
          get_fork( chopstick[i] );
          get_fork( chopStick[ (i + 1) % 5] );
                  //  eat
          put_fork( chopstick[i] );
          put_fork(chopstick[ (i + 1) % 5] );
                    //  think
} while (TRUE);
```

What is the problem with the above code ?

- What schemes can you use to avoid the above problem ?

Solution to Dining Philosopher's Problem using Semaphores

```
#define N 5 /* Number of philosphers */
#define LEFT(i) ((i+1) %N)
#define RIGHT(i) (i+N-1 % N)

enum {THINKING,HUNGRY,EATING} phil_state;
phil_state state[N];
semaphore mutex =1;
semaphore s[N];
/* one per philosopher, all 0 */

/*Testing the state adjacent Phil */
void test(int i) {
    if ( state[i] == HUNGRY &&
    state[LEFT(i)] != EATING &&
    state[RIGHT(i)] != EATING )
    {
        state[i] = EATING;
        V(s[i]);
    }
}
void get_forks(int i) {
    P(mutex);
    state[i] = HUNGRY;
    test(i);
    V(mutex);
    P(s[i]);
}
```

We are **explicitly preventing** multiple processes from entering the functions using mutex

```
void put_forks(int i) {
    P(mutex);
    state[i]= THINKING;
    test(LEFT(i));
    test(RIGHT(i));
    V(mutex);
}
void philosopher(int process) {
    while(1) {
    think();
    get_forks(process);
    eat();
    put_forks(process);
   }
}
```

Solution to Dining Philosopher's problem using Monitors and CV

```
Monitor DP{

    enum {THINKING,HUNGRY,EATING} phil_state;
    Condition s[N]
    void test(int i) {
        if ( state[i] == HUNGRY &&
        state[LEFT(i)] != EATING &&
        state[RIGHT(i)] != EATING )
        {
            state[i] = EATING;
            s[i].signal;
        }
    }
    void get_forks(int i) {
        state[i] = HUNGRY;
        test(i);
        If(state[i] !=EATING)
            s[i].wait();
    }

    void put_forks(int i) {
        state[i]= THINKING;
        test(LEFT(i));
        test(RIGHT(i));
    }
}
```

Only ***one process*** can be active inside Monitor

Therefore, we do not need to explicitly add mutex around Critical Sections

```
void philosopher(int process) {
    while(1) {
    think();
    get_forks(process);
    eat();
    put_forks(process);
   }
}
```

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

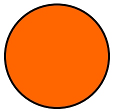
Resource Allocation Graph

A set of vertices *V* and a set of edges *E*

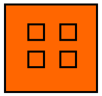
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

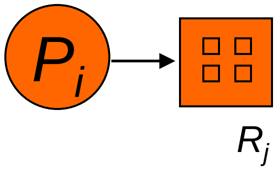
- Process



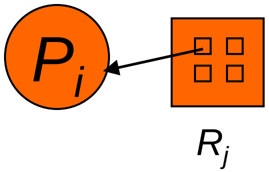
- Resource Type with 4 instances



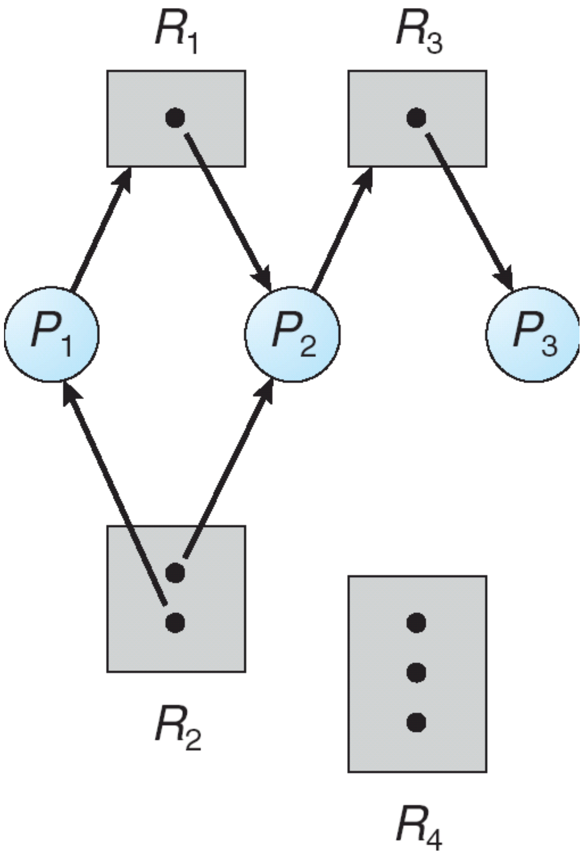
- ***Request Edge:*** P_i requests instance of R_j



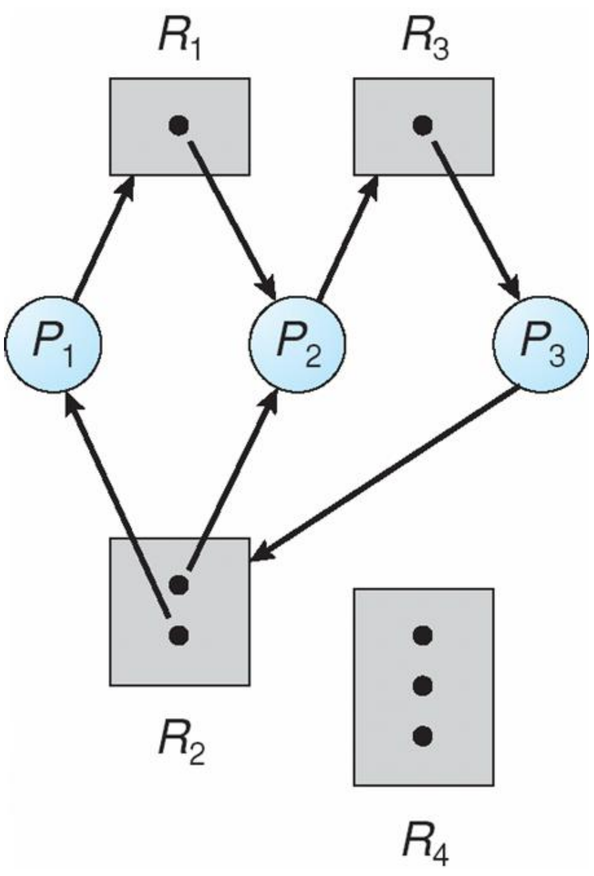
- ***Assignment Edge:*** P_i is holding an instance of R_j



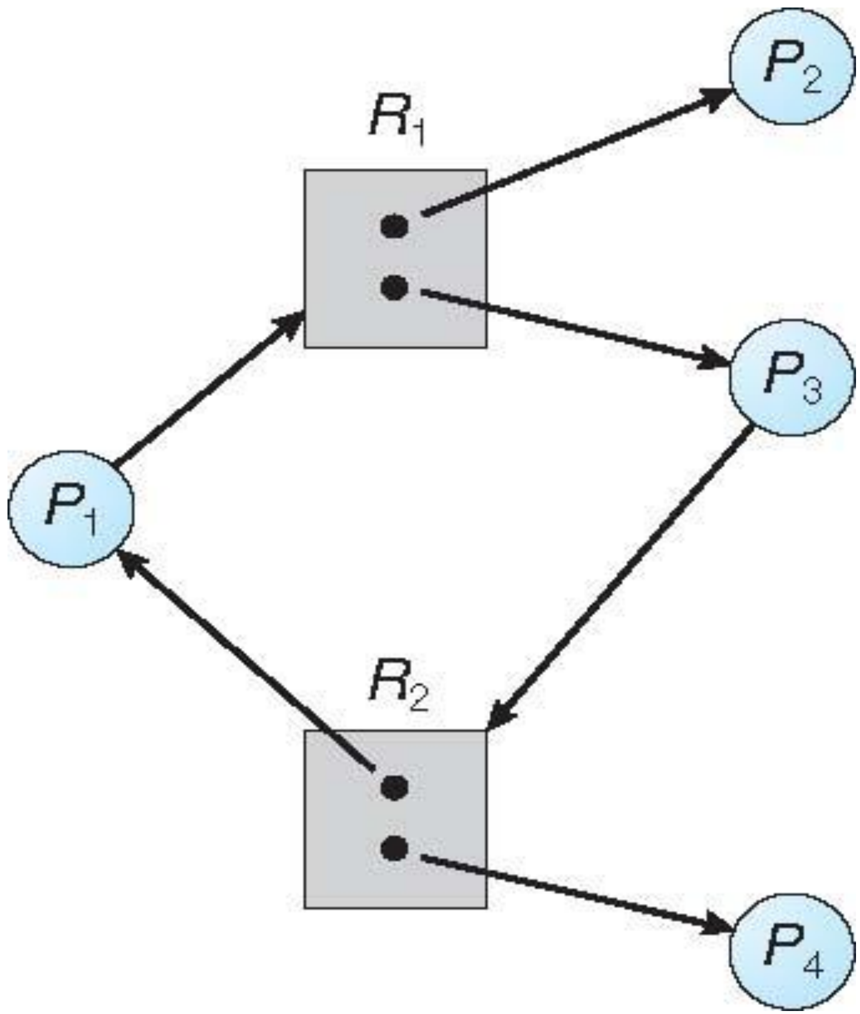
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Fact

- If graph contains no cycles $\Rightarrow$ no deadlock

- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Reactions to Deadlock

An OS can react to deadlock in one of the 4 ways

1. Ignore it : General purpose OS like UNIX does this !
2. Detect and Recover from it : Once in a while, check if the system is in deadlock state
3. Avoid it (Invest effort at runtime to avoid deadlock): Whenever resources are requested, verify if that would lead to deadlock
4. Prevent it (Disallow one of the 4 conditions for deadlock)

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources. ***What's the point ?***

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
  - Low resource utilization; starvation possible

1.

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

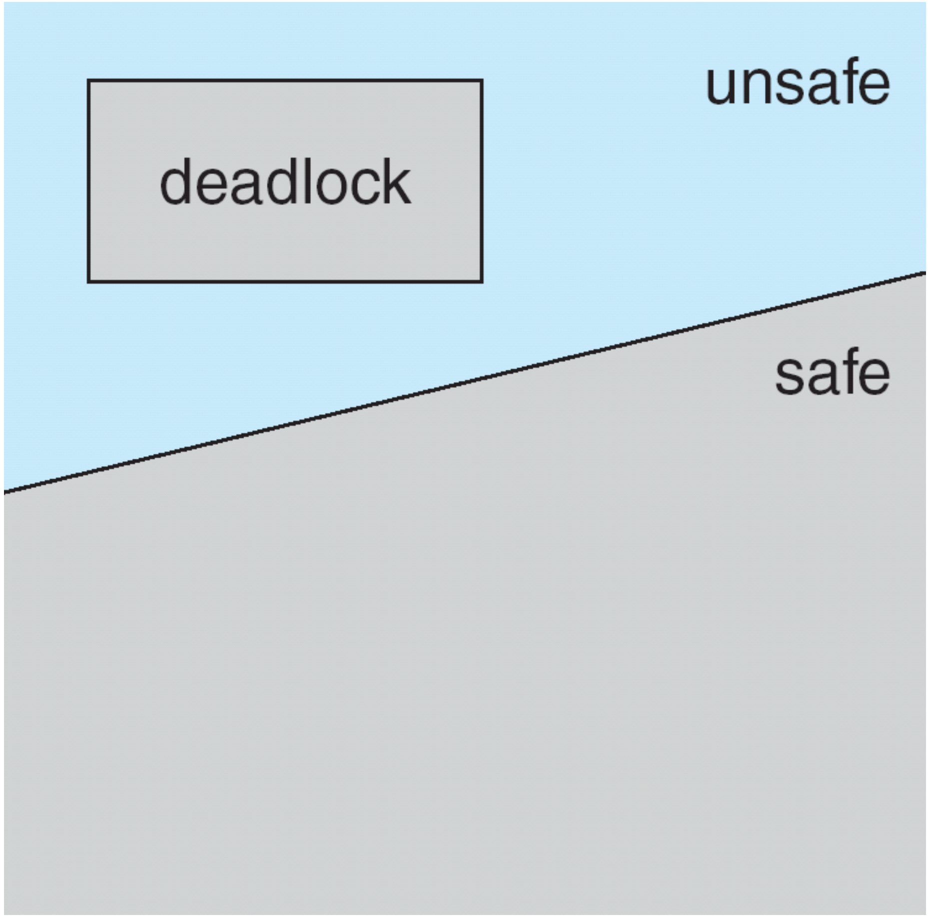
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$

Safe State

That is:

- If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Safe, Unsafe, Deadlock State



Avoidance algorithms

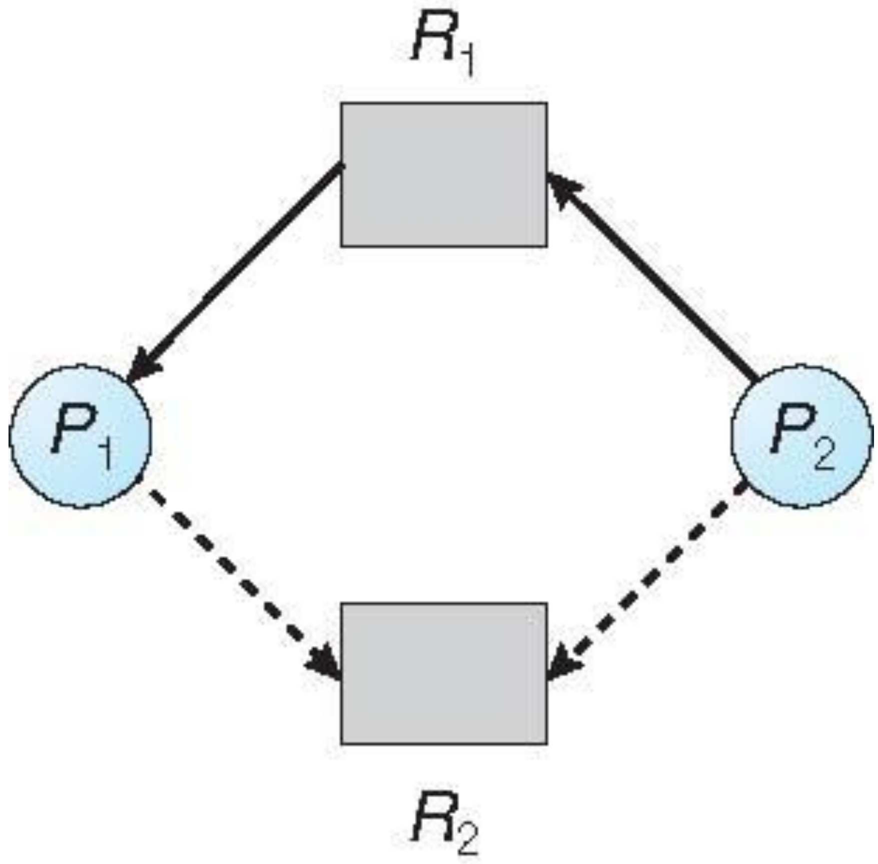
- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the banker's algorithm

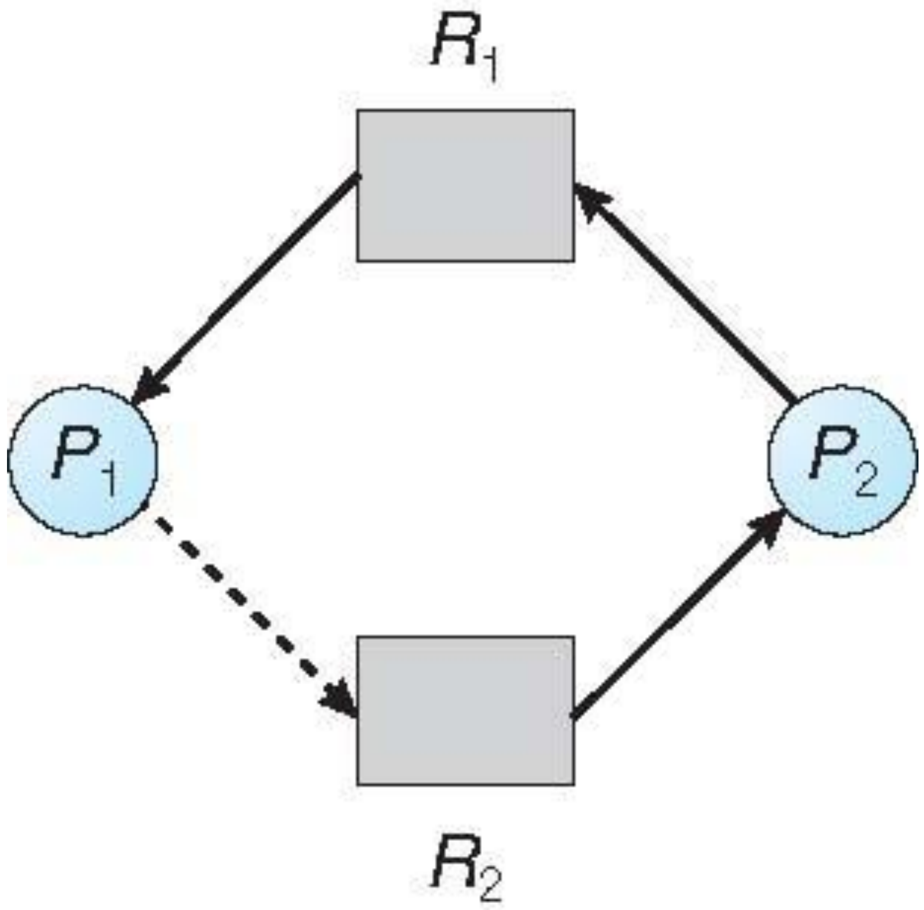
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- Idea: reject resource allocation requests that might leave the system in an "unsafe state".
- A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. Note that not all unsafe states are deadlock states.
- ***Like most bankers***, this algorithm is conservative and simply avoids unsafe states altogether.

Banker’s Algorithm

Details:

- A new process must declare its maximum resource requirements (this number should not exceed the total number of resources in the system, of course)
- When a process requests a set of resources, the system must check whether the allocation of these resources would leave the system in an unsafe state
- If so, the process must wait until some other process releases enough resources

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available
- **Max**: n x m matrix. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If *Need*$[i,j] = k$, then P_i may need k more instances of R_j to complete its task

  Need $[i,j]$ = *Max*$[i,j]$ – *Allocation* $[i,j]$

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize:

   Work = *Available*

   Finish [*i*] = *false* for $i = 0, 1, \ldots, n-1$

2. Find an i such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such i exists, go to step 4

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2

4. If *Finish* [*i*] == true for all i, then the system is in a safe state

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available

3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$
$$Allocation_i = Allocation_i + Request_i$$
$$Need_i = Need_i - Request_i$$

- *If safe $\Rightarrow$ the resources are allocated to Pi*
- *If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored*

Banker's Algorithm (Cont.)

Example: System has 12 tape drives

Processes	Maximum needs	Current allocation
P0	10	5
P1	4	2
P2	9	2

Is system in a safe state?

What if we allocated another tape drive to P2?

Banker's Algorithm (Cont.)

Example: System has 12 tape drives

Processes	Maximum needs	Current allocation
P0	10	5
P1	4	2
P2	9	2

- Is system in a safe state? Yes. 3 tape drives are available and <P1, P0, P2> is a safe sequence.
- What if we allocated another tape drive to P2? No. Only P1 could be allocated all its required resources. P2 would still require 6 drives and P0 would require 5, but only 4 drives would be available => potential for deadlock.