Synchronization

CS 416: Operating Systems Design

Department of Computer Science
Rutgers University

http://www.cs.rutgers.edu/~vinodg/teaching/416

Synchronization

Problem

Threads may share data

Data consistency must be maintained

Example

Suppose a thread wants to withdraw $5 from a bank account and another thread wants to deposit $10 to the same account

What should the balance be after the two transactions have been completed?

What might happen instead if the two transactions were executed concurrently?

Synchronization (cont)

The balance might be CB – 5

- Thread 1 reads CB (current balance)
- Thread 2 reads CB
- Thread 2 computes CB + 10 and saves new balance
- Thread 1 computes CB – 5 and saves new balance

The balance might be CB + 10

- How?

Ensure the orderly execution of cooperating threads/processes

Terminology

Critical section: a section of code which reads or writes shared data

Race condition: potential for interleaved execution of a critical section by multiple threads

- Results are non-deterministic

Mutual exclusion: synchronization mechanism to avoid race conditions by ensuring exclusive execution of critical sections

Deadlock: permanent blocking of threads

Livelock: execution but no progress

Starvation: one or more threads denied resources

Peterson’s algorithm

Goal: synchronize access by two threads to critical section.

Two shared vars: int turn; bool flag[2].

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    // Critical section;
    flag[i] = FALSE;
} while (TRUE);
```

Conceptual solution only!

Need not work on modern architectures!

Requirements for ME

- No assumptions on hardware: speed, # of processors
- Mutual exclusion is maintained – that is, only one thread at a time can be executing inside a CS
- Execution of CS takes a finite time
- A thread/process not in CS cannot prevent other threads/processes to enter the CS
- Entering CS cannot be delayed indefinitely: no deadlock or starvation

Synchronization Primitives

Most common primitives

- Locks (mutual exclusion)
- Condition variables
- Semaphores
- Monitors
- Barriers

Locks

Mutual exclusion ≡ want to be the only thread modifying a set of data items

Can look at it as exclusive access to data items or to a piece of code

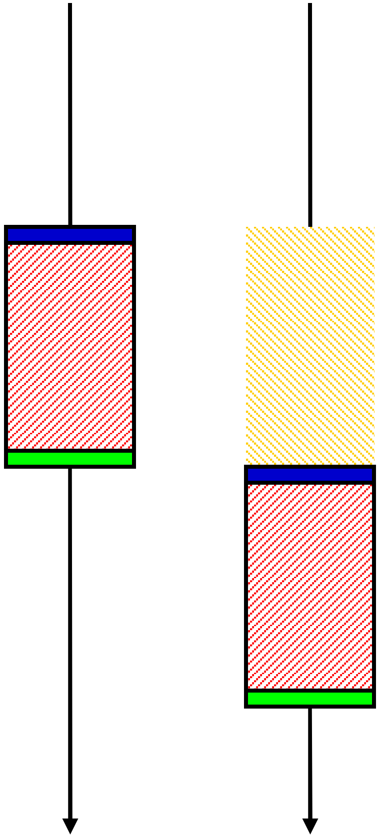
Have three components:

Acquire, Release, Waiting

Examples:

```
Acquire(A)
Acquire(B)
A ← A + 10
B ← B - 10
Release(B)
Release(A)
```

```
Function Transfer (Amount, A, B)
    Acquire(Transfer_Lock)
    A ← A + 10
    B ← B - 10
    Release(Transfer_Lock)
```



Example

```
public class BankAccount
{
    Lock aLock = new Lock;
    int balance = 0;

    ...

    public void deposit(int amount)
    {
              aLock.acquire();
              balance = balance + amount;
              aLock.release();
    }

    public void withdrawal(int amount)
    {
              aLock.acquire();
              balance = balance - amount;
              aLock.release();
    }
```

Implementing (Blocking) Locks Inside OS Kernel

From Nachos (with some simplifications)

```
public class Lock {
    private KThread lockHolder = null;
    private ThreadQueue waitQueue =
            ThreadedKernel.scheduler.newThreadQueue(true);

    public void acquire() {
          KThread thread = KThread.currentThread();  // Get thread object (TCB)
          if (lockHolder != null) {                  // Gotta wait
              waitQueue.waitForAccess(thread);       // Put thread on wait queue
              KThread.sleep();                       // Context switch
          }
          else
              lockHolder = thread;                   // Got the lock
    }
```

Implementing (Blocking) Locks Inside OS Kernel

```
    public void release() {
            if ((lockHolder = waitQueue.nextThread()) != null)
               lockHolder.ready();        // Wake up a waiting thread
    }
}
```

This implementation is not quite right … what's missing?

Implementing (Blocking) Locks Inside OS Kernel

```
    public void release() {
            if ((lockHolder = waitQueue.nextThread()) != null)
               lockHolder.ready();        // Wake up a waiting thread
    }
 }
```

This implementation is not quite right … what’s missing?
Mutual exclusion when accessing lockHolder and waitQueue.
An approach is to protect against interrupts.

Implementing (Blocking) Locks Inside OS Kernel

```
public void release() {
        boolean intStatus = Machine.interrupt().disable();

        if ((lockHolder = waitQueue.nextThread()) != null)
            lockHolder.ready();

        Machine.interrupt().restore(intStatus);
}
```

acquire() also needs to block interrupts

Implementing Locks At User-Level

Why?

- Expensive to enter the kernel

What's the problem?

- Can't disable interrupts …

Many software algorithms for mutual exclusion

- See any OS book
- Disadvantages: very difficult to get correct

So what do we do?

Implementing Locks At User-Level

- Simple with a “little bit” of help from the hardware
- Atomic read-modify-write instructions
  - Test-and-set
    - Atomically read a variable and, if the value of the variable is currently 0, set it to 1
  - Fetch-and-increment
    - Atomically return the current value of a memory location and increment the value in memory by 1
  - Compare-and-swap
    - Atomically compare the value of a memory location with an old value, and if the same, replace with a new value
- Modern architectures perform atomic operations in the cache

Implementing **Spin** Locks Using Test-and-Set

```
#define UNLOCKED 0
#define LOCKED 1

Spin_acquire(lock)
{
        while (test-and-set(lock) == LOCKED);
}

Spin_release(lock)
{
        lock = UNLOCKED;
}
```

Problems?

Implementing **Spin** Locks Using Test-and-Set

```
#define UNLOCKED 0
#define LOCKED 1

Spin_acquire(lock)
{
        while (test-and-set(lock) == LOCKED);
}

Spin_release(lock)
{
        lock = UNLOCKED;
}
```

Problems? Lots of memory traffic if TAS always sets; lots of traffic when lock is released; no ordering guarantees. Solutions?

Spin Locks Using Test and Test-and-Set

```
Spin_acquire(lock)
{
            while (1)  {
                  while (lock == LOCKED);
                  if (test-and-set(lock) == UNLOCKED)
break;
            }
}

Spin_release(lock)
{
            lock = UNLOCKED;
}
```

Better, since TAS is guaranteed not to generate traffic unnecessarily. But there is still lots of traffic after a release. Still no ordering guarantees.

Announcements

Please send info on your groups for homework 2 to Ana Paula (anapaula@cs.rutgers.edu) by midnight today.

Groups can contain up to three students.

We will forward your NetIDs to the lab and get you accounts on osconsole and VOSLab. You will not be able to do homework 2 without an account on these machines.

If we don’t receive this information by tonight, you will have to contact the lab yourself.

Implementing (Blocking) Locks Inside OS Kernel

From Nachos (with some simplifications)

```
public class Lock {
    private KThread lockHolder = null;
    private ThreadQueue waitQueue =
            ThreadedKernel.scheduler.newThreadQueue(true);

    public void acquire() {
        KThread thread = KThread.currentThread();   // Get thread object (TCB)
        if (lockHolder != null) {                    // Gotta wait
            waitQueue.waitForAccess(thread);         // Put thread on wait queue
            KThread.sleep();                         // Context switch
        }
        else
            lockHolder = thread;                     // Got the lock
    }
```

Implementing (Blocking) Locks Inside OS Kernel

```
    public void release() {
            if ((lockHolder = waitQueue.nextThread()) != null)
               lockHolder.ready();       // Wake up a waiting thread
    }
}
```

This implementation is not quite right … what’s missing?

Implementing (Blocking) Locks Inside OS Kernel

```
public void release() {
        boolean intStatus = Machine.interrupt().disable();

        if ((lockHolder = waitQueue.nextThread()) != null)
            lockHolder.ready();

        Machine.interrupt().restore(intStatus);
}
```

acquire() also needs to block interrupts

Implementing Locks At User-Level

- Simple with a “little bit” of help from the hardware
- Atomic read-modify-write instructions
  - Test-and-set
    - Atomically read a variable and, if the value of the variable is currently 0, set it to 1
  - Fetch-and-increment
    - Atomically return the current value of a memory location and increment the value in memory by 1
  - Compare-and-swap
    - Atomically compare the value of a memory location with an old value, and if the same, replace with a new value
- Modern architectures perform atomic operations in the cache

Implementing **Spin** Locks Using Test-and-Set

```
#define UNLOCKED 0
#define LOCKED 1

Spin_acquire(lock)
{
            while (test-and-set(lock) == LOCKED);
}

Spin_release(lock)
{
            lock = UNLOCKED;
}
```

Problems?

Exercise

Write acquire and release using compare_and_swap

```
bool compare_and_swap (mem_addr, old_val, new_val) {
    if (*mem_addr == old_val)
    then *mem_addr = new_val; return true;
    else return false;
}
```

Spin Locks Using Fetch-and-Increment

- Ticket lock using fetch-and-increment:
  - Each thread gets a ticket from variable *next-ticket*
  - *Now-serving* variable holds ticket of current lock holder
- Think about how to implement acquire and release!
- Many other spin lock implementations exist

Ticket locks

```
ticket = 0; now_serving = 0;
Acquire (int *my_ticket) {
    my_ticket = fetch_and_increment(ticket);
    while (my_ticket != now_serving) { };
}
Release(){
    fetch_and_increment(now_serving);
}
```

Implementing (Spin) Barriers

- Centralized barrier:
  - Each thread increments a shared counter upon arriving
  - Each thread polls the shared counter until all have arrived

```
Barrier (num_threads)
{
        if (fetch-and-increment (counter) == num_threads)
                counter = 0;
        else
                while (counter != 0);
}
```

Problems?

Implementing (Spin) Barriers

- Centralized barrier:
  - Each thread increments a shared counter upon arriving
  - Each thread polls the shared counter until all have arrived

```
Barrier (num_threads)
{
                if (fetch-and-increment (counter) ==
num_threads)
                        counter = 0;
                else
                        while (counter != 0);
}
```

Problems? Consecutive barriers may mess shared counter. Contention for shared counter. Solutions?

Implementing (Spin) Barriers

- Centralized barrier with sense reversal:
  - Odd barriers wait for sense flag to go from true to false
  - Even barriers wait for sense flag to go from false to true

```
Barrier (num_threads)
{
        local_sense = ! local_sense;
        if (fetch-and-increment (counter) == num_threads) {
                counter = 0;
                sense = local_sense;
        }   else
                while (sense != local_sense);
}
```

Implementing (Spin) Barriers

- Previous implementation still suffers from contention
- Possible solution: combining tree barrier with sense reversal
  - Writes done in a tree; only *tree-degree* threads write to same counter
  - Last arrival at each level goes further up
  - Thread that arrives at the root wakes up the others by changing the sense variables on which they are spinning
- Think about how to implement this combining tree barrier!

- Many other spin barrier implementations exist

Implementing Blocking Locks Using Test-and-Set

```
public class Lock
{
    private int val = UNLOCKED;
    private ThreadQueue waitQueue = new ThreadQueue();

    public void acquire() {
            Thread me = Thread.currentThread();
            while (TestAndSet(val) == LOCKED) {
                waitQueue.waitForAccess(me);  // Put self in queue
                Thread.sleep();               // Put self to sleep
            }
            // Got the lock
    }
```

Implementing Blocking Locks Using Test-and-Set

```
        public void release() {
                Thread next = waitQueue.nextThread();
                val = UNLOCKED;
                if (next != null)
                    next.ready();    // Wake up a waiting thread
        }
    }
```

Does this implementation work as is? No, for two reasons: (1) we need mutual exclusion in the access to the wait queue; and (2) we need an extra check of the lock in the acquire (the releaser may release the lock after a thread finds the lock busy, but before it enqueues itself).

Blocking Locks Using Test-and-Set: Correct

```
public void acquire() {
    Thread me = Thread.currentThread();
        while (TestAndSet(val) == LOCKED) {
              Machine.interrupt().disable();
              if (val == LOCKED) {
                     waitQueue.waitForAccess(me);  // Put self in queue
                     Machine.interrupt().enable();
                     Thread.sleep();                         // Put self to sleep
               }
              else {
                     Machine.interrupt().enable();
              }
        }
        // Got the lock
}
```

Blocking Locks Using Test-and-Set: Correct

```
public void release() {
        Machine.interrupt().disable();
        Thread next = waitQueue.nextThread();
        val = UNLOCKED;
        Machine.interrupt().enable();
        if (next != null)
            next.ready();    // Wake up a waiting thread
}
```

Disabling interrupts doesn't do the job in multiprocessors and is generally undesirable. So, what should be do instead?

Blocking Locks Using Test-and-Set: Correct

```
public void acquire() {
    Thread me = Thread.currentThread();
        while (TestAndSet(val) == LOCKED) {
              spin_lock(another_lock);
              if (val == LOCKED) {
                     waitQueue.waitForAccess(me);  // Put self in queue
                     spin_unlock(another_lock);
                     Thread.sleep();                               // Put self to sleep
               }
              else {
                      spin_unlock(another_lock);
              }
         }
         // Got the lock
}
```

Blocking Locks Using Test-and-Set: Correct

```
public void release() {
        spin_lock(another_lock);
        Thread next = waitQueue.nextThread();
        val = UNLOCKED;
        spin_unlock(another_lock);
        if (next != null)
            next.ready();    // Wake up a waiting thread
}
```

What happens if a thread is killed when holding a lock?

What happens when a thread is woken up as “next” above but there are more threads waiting for the lock?

What To Do While Waiting?

We have considered two types of primitives:

Blocking

- OS or RT system de-schedules waiting threads

Spinning

- Waiting threads keep testing location until it changes value
- Hmm … doesn’t quite work in single-threaded uniprocessor, does it?

Spinning vs. blocking becomes an issue in multithreaded processors and multiprocessors

What is the main tradeoff?

What To Do While Waiting?

We have considered two types of primitives:

Blocking

- OS or RT system de-schedules waiting threads

Spinning

- Waiting threads keep testing location until it changes value
- Hmm … doesn’t quite work in single-threaded uniprocessor, does it?

Spinning vs. blocking becomes an issue in multithreaded processors and multiprocessors

What is the main tradeoff? Overhead of blocking vs. expected waiting time

Condition Variables

A condition variable is always associated with:

- A condition
- A lock

Typically used to wait for the condition to take on a given value

Three operations:

```
public class CondVar
{
   public Wait(Lock lock);
   public Signal();
   public Broadcast();
   // ... other stuff
}
```

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

Signal()

- 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

Broadcast()

- If 1 or more threads are sleeping on cond_var, wake everyone up
- Otherwise, no effect

Producer/Consumer Example

Imagine a web server with the following architecture:

- One "producer" thread listens for client http requests
- When a request is received, the producer enqueues it on a circular request queue with finite capacity (if there is room)
- A number of "consumer" threads service the queue as follows
  - Remove the 1st request from the queue (if there is a request)
  - Read data from disk to service the request
- How can the producer and consumers synchronize?

Producer/Consumer (cont)

```
public class SyncQueue
{
    public boolean IsEmpty();
    public boolean IsFull();
    public boolean Enqueue(Request r);
    public Request Dequeue();

    public LockVar lock = new Lock;
    public CondVar waitForNotEmpty = new CondVar(LockVar lock);
    public CondVar waitForNotFull = new CondVar(LockVar lock);

    ...
}
```

Producer

```
public class Producer extends Thread
{
   private SyncQueue requestQ;
   public Producer(SyncQueue q) {requestQ = q;}
   public void run()
   {
         // Accept a request from some client
         // The request is stored in the object newRequest
         requestQ.lock.Acquire();
         while (requestQ.IsFull()) {
            waitForNotFull.Wait(requestQ.lock);
         }
         requestQ.Enqueue(newRequest);
         waitForNotEmpty.Signal();
         requestQ.lock.Release();
   }
}
```

Consumer

```
public class Consumer extends Thread
{
    private SyncQueue requestQ;
    public Consumer(SyncQueue q) {requestQ = q;}
    public void run()
    {
            requestQ.lock.Acquire();
            while (requestQ.IsEmpty()) {
                waitForNotEmpty.Wait(requestQ.lock);
            }
            Request r = requestQ.Dequeue();
            waitForNotFull.Signal()
            requestQ.lock.Release();

            // Process the request
    }
}
```

Implementing Condition Variables

Condition variables are implemented using locks

Implementation is tricky because it involves multiple locks and scheduling queue

Implemented in the OS or run-time thread systems because they involve scheduling operations

- Sleep/Wakeup

Can you see how to do this from our discussion of how to implement locks?

You may be asked to implement condition variables!

Semaphores

Synchronized counting variables

Formally, a semaphore sem is comprised of:

- An integer value: sem->counter
- Two operations: P(sem) [or wait(sem)] and V(sem) [or signal(sem)]

P(sem)

- While sem->counter == 0, sleep
  Decrement sem->counter and return

V(sem)

- Increment sem->counter
- If there are any threads sleeping waiting for sem to become non-zero, wakeup at least 1 thread

Semaphores

```
P(sem) {
        sem->counter --;
        if (sem->counter < 0) {
                nextproc = remove elem from sem->queue;
                unblock(nextproc)
        }
}
```

Semaphores

```
V(sem) {
        sem->counter ++;
        if (sem->counter <= 0) {
                add process to sem->queue;
                block();
        }
}
```

Can initialize value of sem->counter to any positive value ‘n’.

Called a counting semaphore.

Bounded Buffers problem

Have a pool of n buffers. Each buffer can hold one item.

<u>Producer</u>

```
while (true) {
    while (counter == BUF_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1) % BUF_SIZE;
    counter++;
```

<u>Consumer</u>

```
While (true) {
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1) % BUF_SIZE;
    counter--;
```

Bounded Buffers problem – first try

Have a pool of n buffers. Each buffer can hold one item.

Producer

```
while (true) {
    P(mutex)
    // CS
    V(mutex)
}
```

Consumer

```
While (true) {
    P(mutex)
    // CS
    V(mutex)
}
```

Bounded Buffers problem – correct solution

Init values of sem counters: empty -> n, full -> 0, mutex -> 1.

Producer

```
while (true) {
     P(empty);
     P(mutex);
    // CS;
    V(mutex);
    V(full);
}
```

Consumer

```
while (true) {
    P(full);
    V(mutex);
    // CS
    V(mutex);
    P(empty);
}
```

Readers-Writers problem

You have a shared database, and two kinds of processes that can access this database: readers and writers.

Reader processes do not modify the database.

Writer processes can read/modify the database.

How do you synchronize access to the database?

One solution: No reader should be kept waiting unless a writer already has acquired a lock on the database.

Exercise: Implement this solution using semaphores.

Hint: Use a counting semaphore to count the number of readers.

Readers/Writers problem – first try

```
P (write)
// Perform the write
V (write)
```

```
P(write);
 // Perform the read.
 V(write);
```

Readers/Writers problem – second try

```
P (write)
// Perform the write
V (write)
```

```
readcount ++;
if (readcount == 1) P(write);
// Perform the read.
readcount--;
if (readcount == 0) V(write);
```

Readers/Writers problem – correct solution

```
wait (write)
// Perform the write
signal(write)
```

```
wait (mutex);
readcount ++;
if (readcount == 1)
wait(write);
signal(mutex);
// Perform the read.
wait(mutex);
readcount--;
if (readcount == 0)
signal(write);
```

Monitors

Semaphores have a few limitations: unstructured, difficult to program correctly. Monitors eliminate these limitations and are as powerful as semaphores

A monitor consists of a software module with one or more procedures, an initialization sequence, and local data (can only be accessed by procedures)

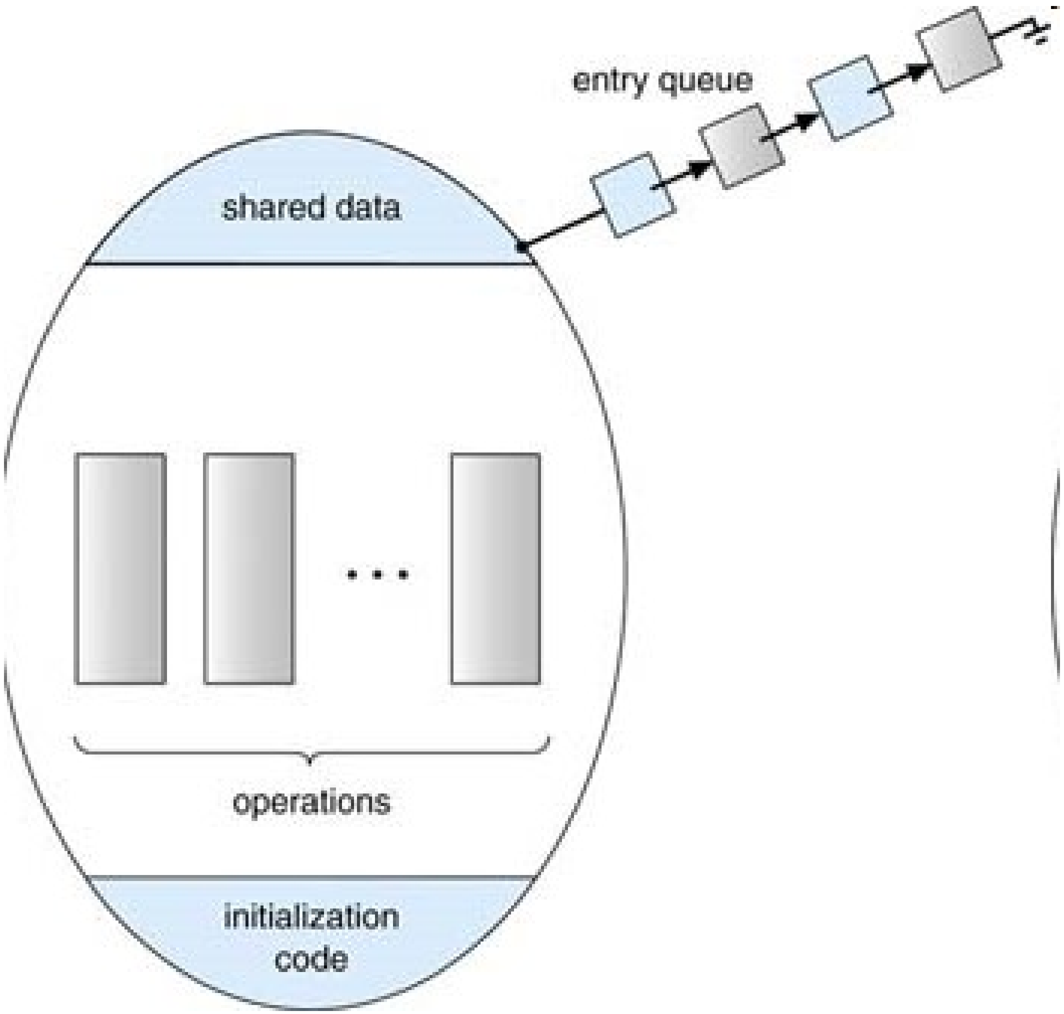
Only one process can execute within the monitor at any one time (mutual exclusion) => entry queue

Synchronization within the monitor implemented with condition variables (wait/signal) => one queue per condition variable

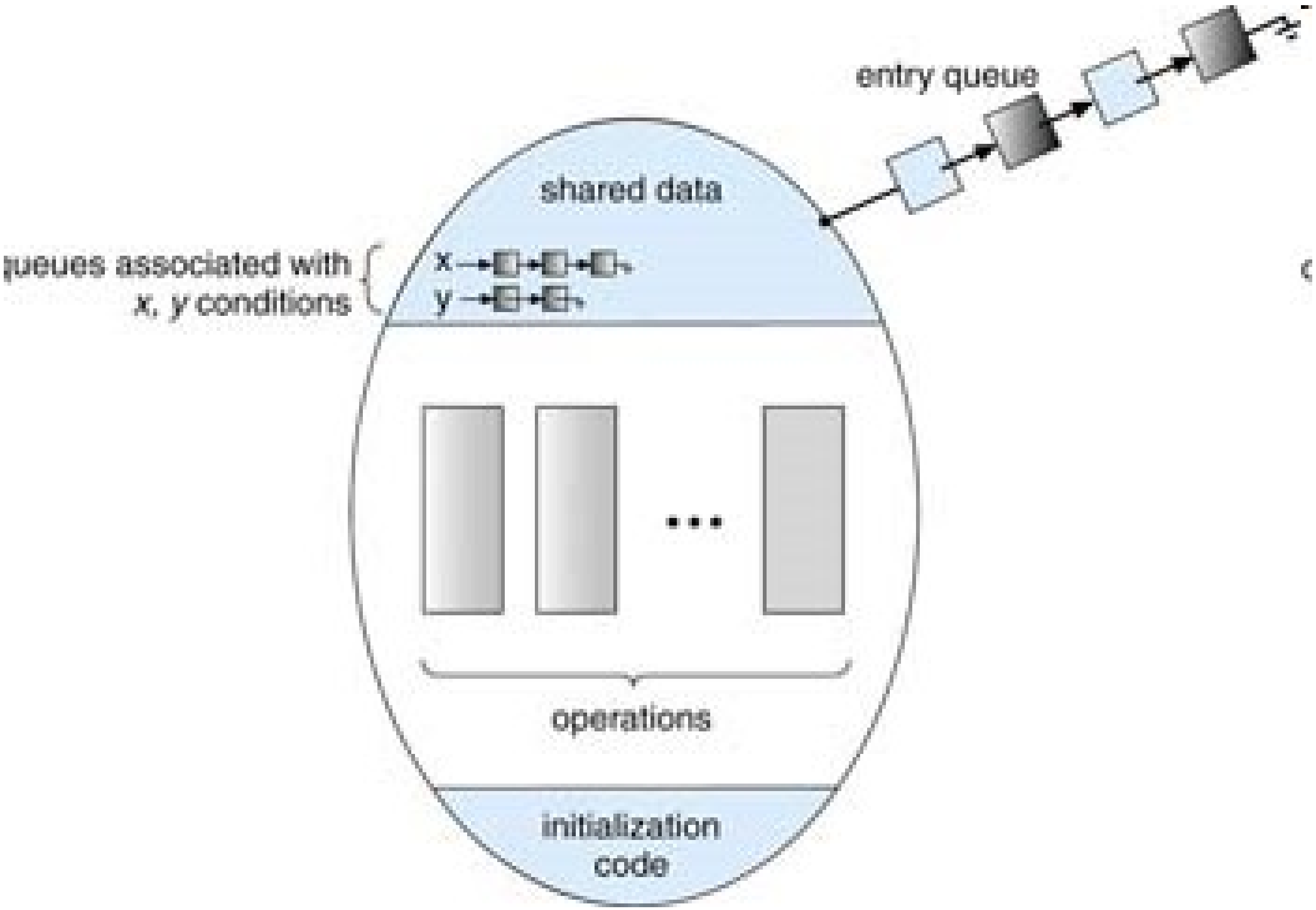
Monitors: Syntax

```
Monitor monitor-name
{
        shared variable declarations

        procedure body P1 (…) {
                …
        }
        procedure body Pn (…) {
                …
        }

        {
                initialization code
        }
}
```

Monitors



Monitors with condition variables



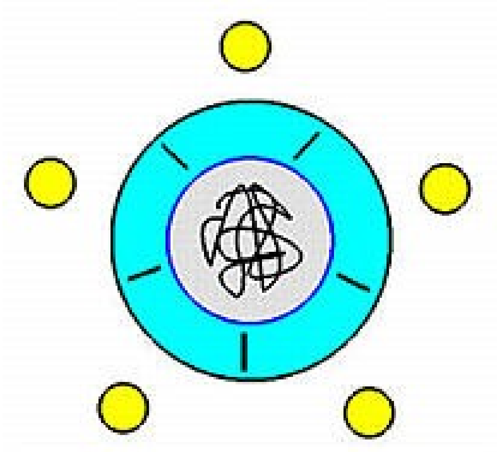
The Dining-Philosophers Problem

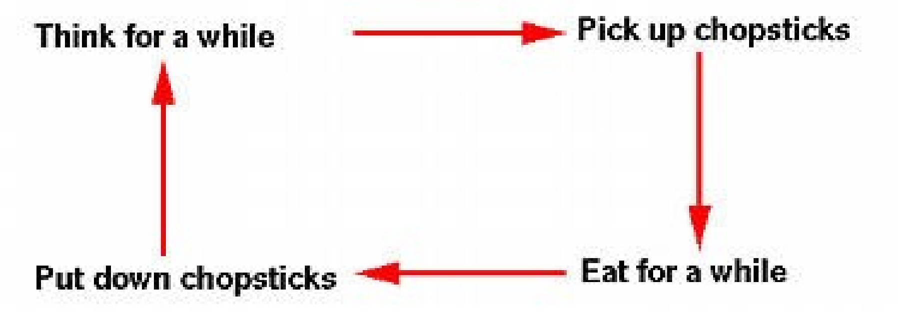
Let’s solve the following problem together:

Consider 5 philosophers who spend their lives thinking and eating. The philosophers share a common circular table. Each philosopher has a bowl of rice (that magically replenishes itself). There are 5 chopsticks, each between a pair of philosophers. Occasionally, a philosopher gets hungry. He needs to get both the right and left chopsticks before he can eat. He eats for a while until he’s full, at which point he puts the chopsticks back down on the table and resumes thinking.

How can we help the philosophers to synchronize their use of the chopsticks?

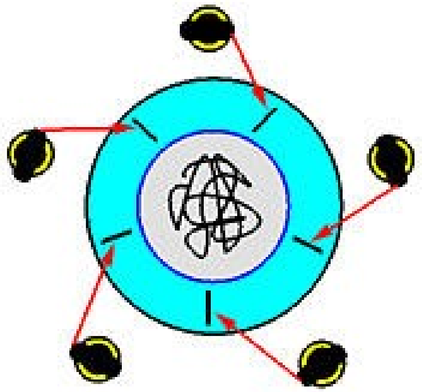
The Dining-Philosophers Problem

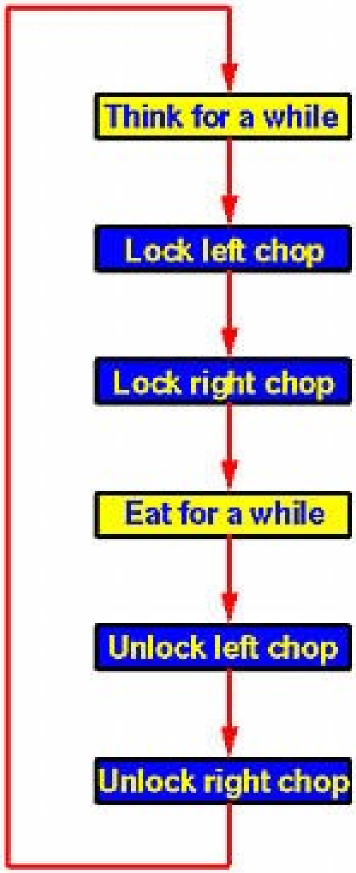




The Dining-Philosophers Problem

Any problems with alg?





Think for a while

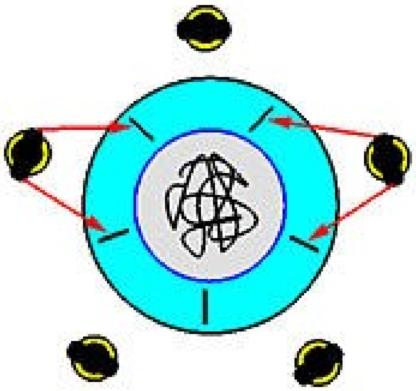
Lock left chop

Lock right chop

Eat for a while

Unlock left chop

Unlock right chop



Philosopher with semaphores

```
Do {
    P (chopstick[i]);
    P (chopstick[i+1]%5);
    // gobble, gobble, gobble.
    V (chopstick[i]);
    V (chopstick[i+1]%5);
    // think;
} while (true);
```

Dining Philosopher's Problem with monitors

enum {think, hungry, eat} state[5];

condition self [5]

Create a monitor dp, and do the following for each philosopher

dp.pickup(i);

//eat;

dp.putdown(i);

Dining Philosopher Monitor

```
Void pickup(i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING) self[i].wait();
}
```

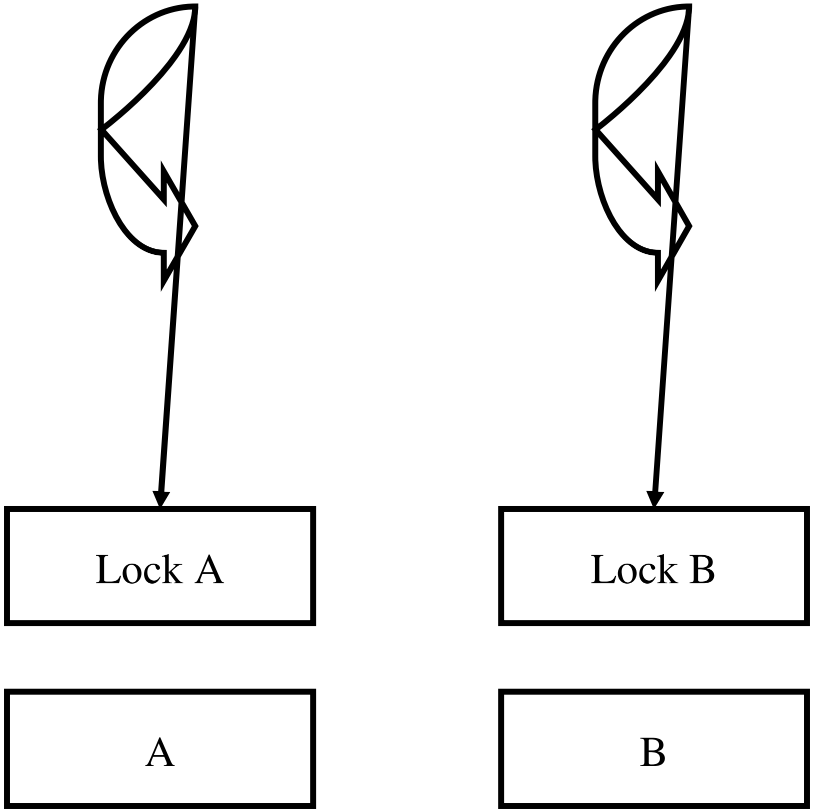
DP Monitor

```
Test (i) {
    if (state(nbr) != EATING && state[i] == HUNGRY) {
        state[i] = EATING;
        self[i].signal();
    }
}
```

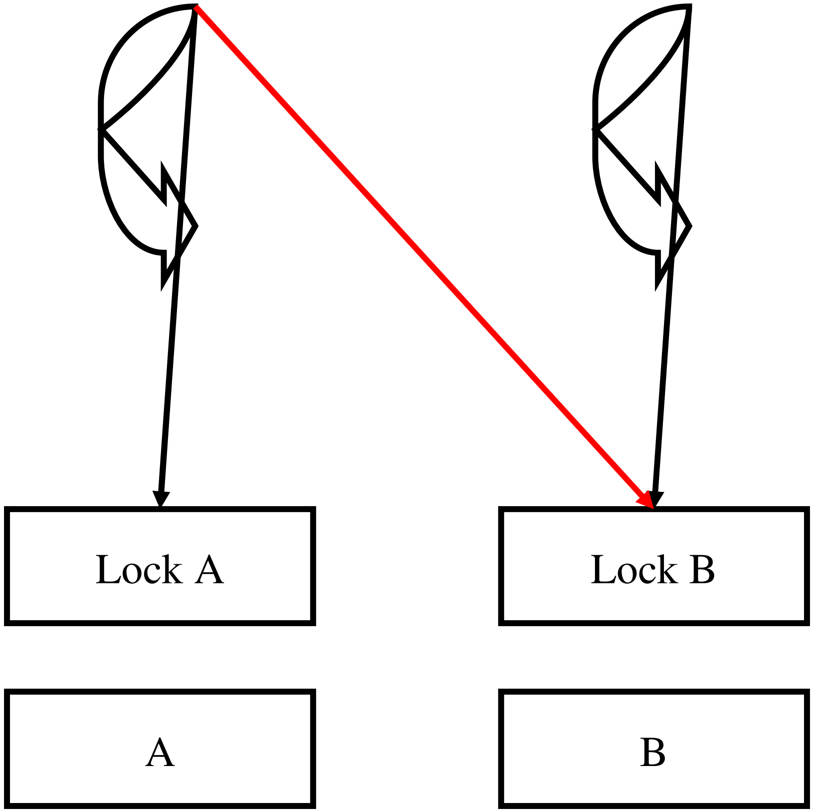
DP Monitor

```
void putdown(i) {
    state[i] = THINKING;
    test(i+4%5)
    test(i+1%5)
}
```

Deadlock



Deadlock



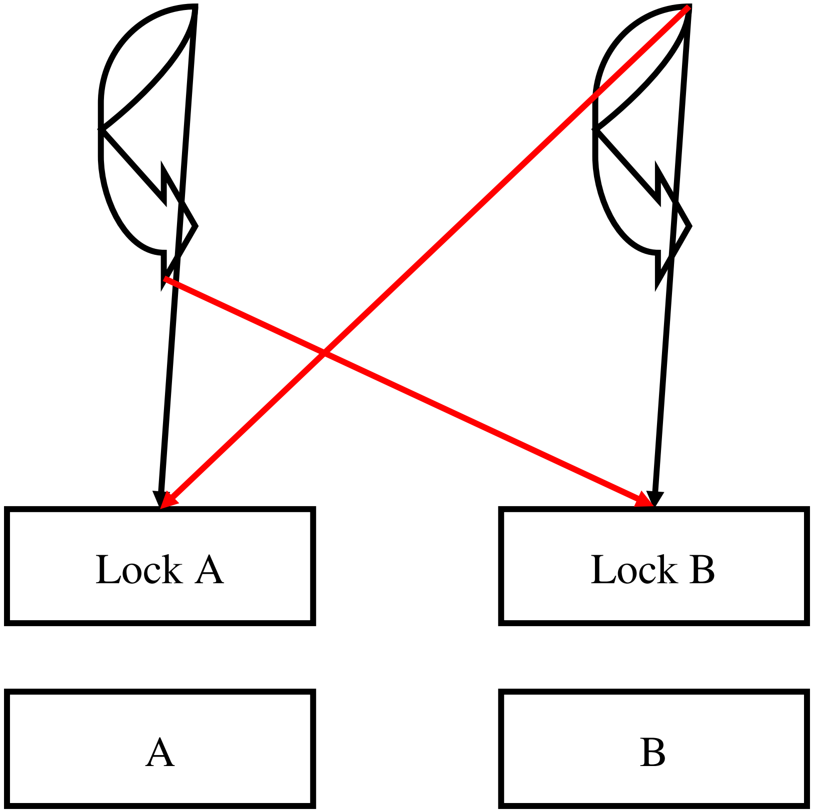
Lock A

Lock B

A

B

Deadlock



Deadlock (Cont'd)

Deadlock can occur whenever multiple parties are competing for exclusive access to multiple resources

How can we avoid deadlocks? Prevention, Avoidance, and Detection + Recovery

Necessary conditions for deadlock: mutual exclusion, hold and wait, no preemption, circular wait

- Hold and wait – a thread/process that needs a resource that is currently taken holds on to the resources it already has and waits for the resource
- No preemption – a thread/process never has a resource that it is holding taken away

Deadlock Characterization

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 $\{P_0, P_1, \ldots, P_0\}$ 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_0 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

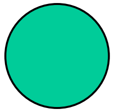
$R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

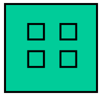
request edge – directed edge $P_1 \rightarrow R_j$

assignment edge – directed edge $R_j \rightarrow P_i$

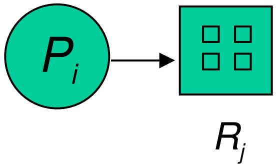
Resource-Allocation Graph (Cont.)

Process

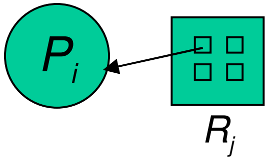




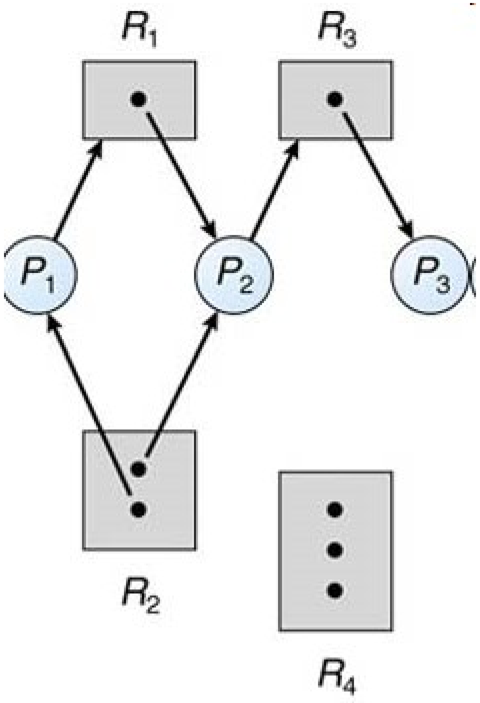
Resource Type with 4 instances



P_i requests instance of R_j



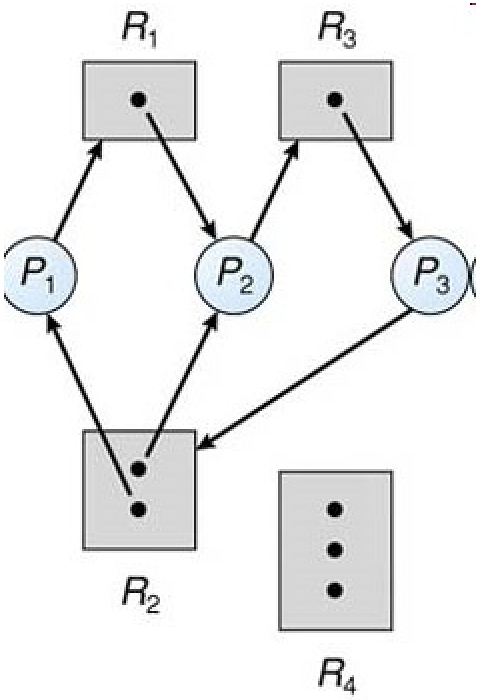
Resource Allocation Graphs



Resource Allocation Graphs.

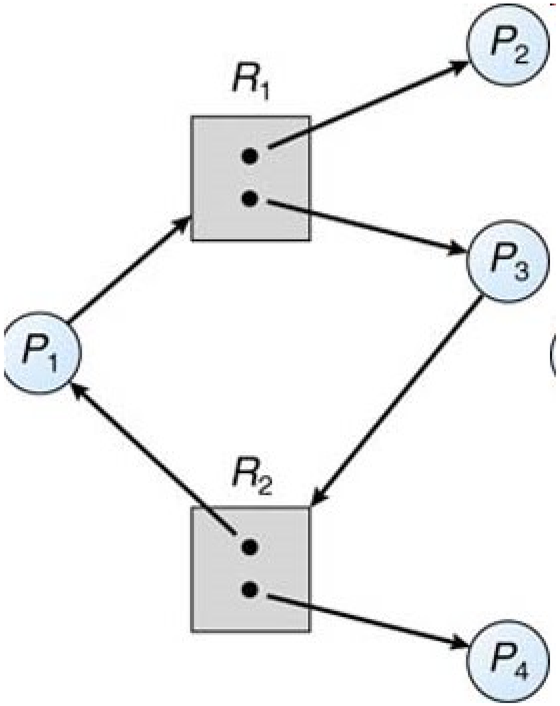
If there are no cycles in this graph, no deadlocks exist.

If there are cycles, a deadlock may exist



Detecting deadlocks

If there are cycles, a deadlock may exist – necessary but not sufficient



What We Can Do About Deadlocks

- Deadlock prevention
  - Design a system without one of mutual exclusion, hold and wait, no preemption or circular wait (four necessary conditions)
  - To prevent circular wait, impose a strict ordering on resources. For instance, if need to lock A and B, always lock A first, then lock B
- Deadlock avoidance
  - Deny requests that may lead to unsafe states (Banker's algorithm)
  - Running the algorithm on all resource requests is expensive
- Deadlock detection and recovery
  - Check for circular wait periodically. If circular wait is found, abort all deadlocked processes (extreme solution but very common)
  - Checking for circular wait is expensive

Methods for Handling Deadlocks

Ensure that the system will *never* enter a deadlock state

Allow the system to enter a deadlock state and then recover

Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources

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

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

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, …, 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

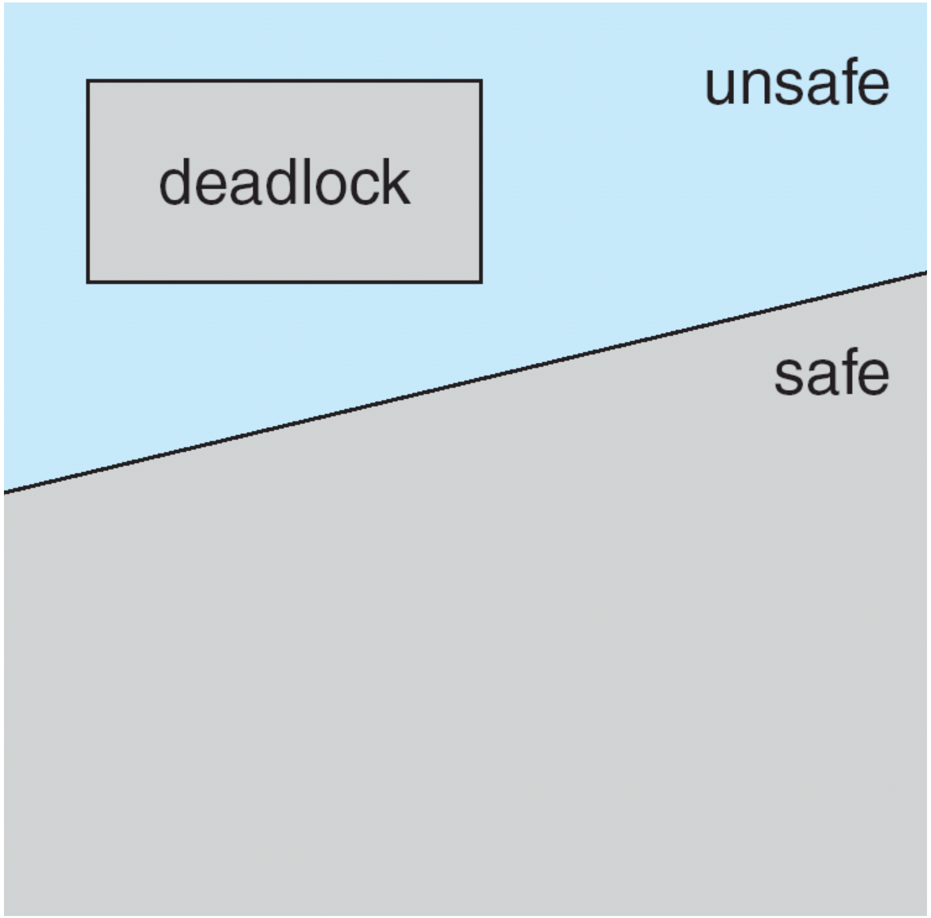
Basic Facts

If a system is in safe state ⇒ no deadlocks

If a system is in unsafe state ⇒ possibility of deadlock

Avoidance ⇒ ensure that a system will never enter an unsafe state.

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

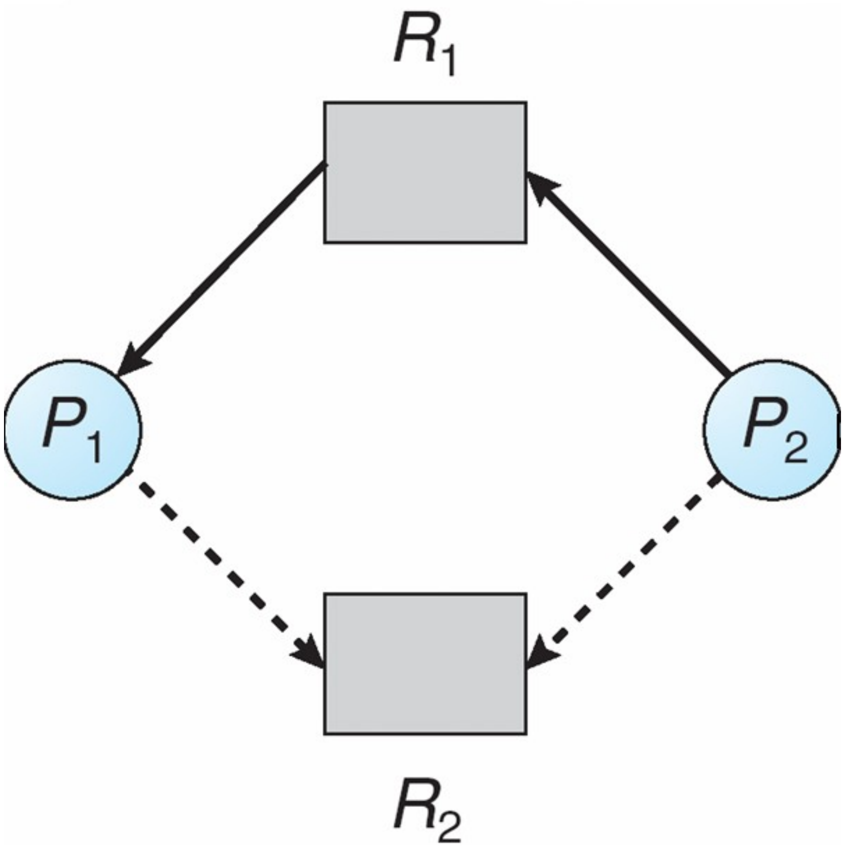
Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line

Claim edge converts to request edge when a process requests a resource

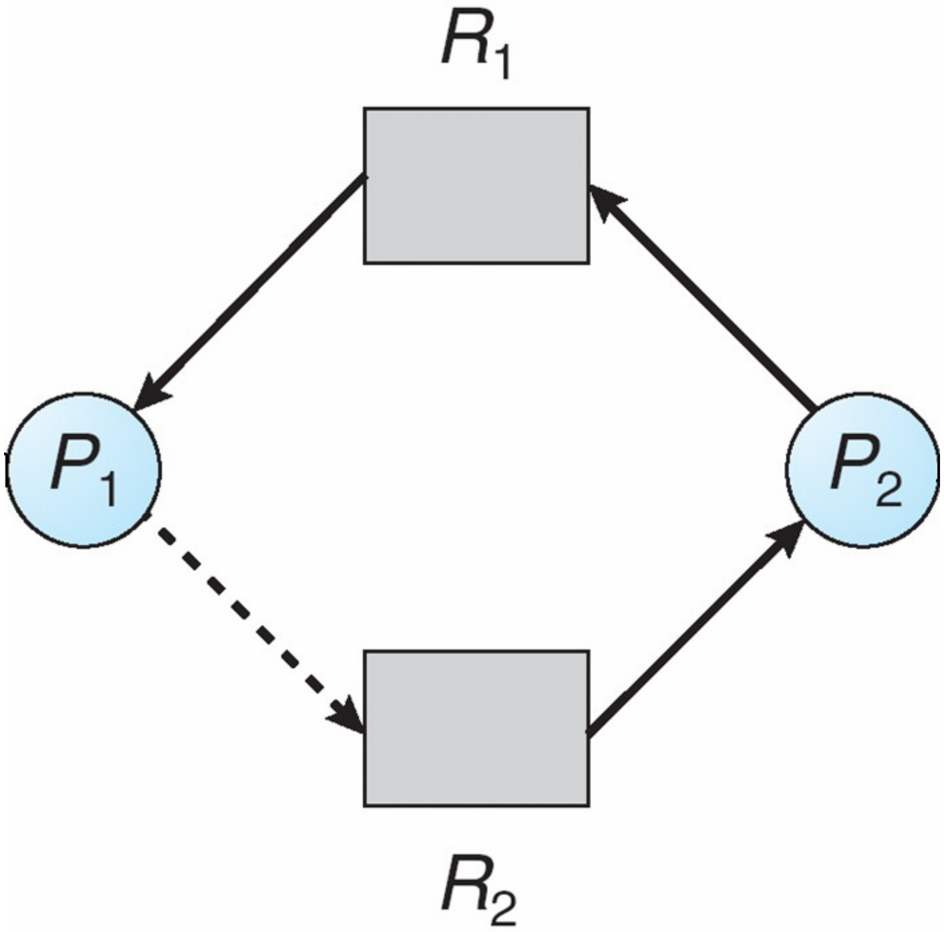
Request edge converted to an assignment edge when the resource is allocated to the process

When a resource is released by a process, assignment edge reconverts to a claim edge

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

Suppose that process P_i requests a resource R_j

The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the 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.

Hmm…

Banker’s Algorithm (Cont’d)

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: $n \times m$ 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, …, *n*- 1

2. Find and *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 $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \le Available$, 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 ⇒ the resources are allocated to Pi

If unsafe ⇒ Pi 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.