Lecture 4: Inter-process Communication and Synchronization, Deadlocks

Contents

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Producer-Consumer Problem
  - Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
    - *unbounded-buffer* places no practical limit on the size of the buffer
    - ***bounded-buffer*** assumes that there is a fixed buffer size

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

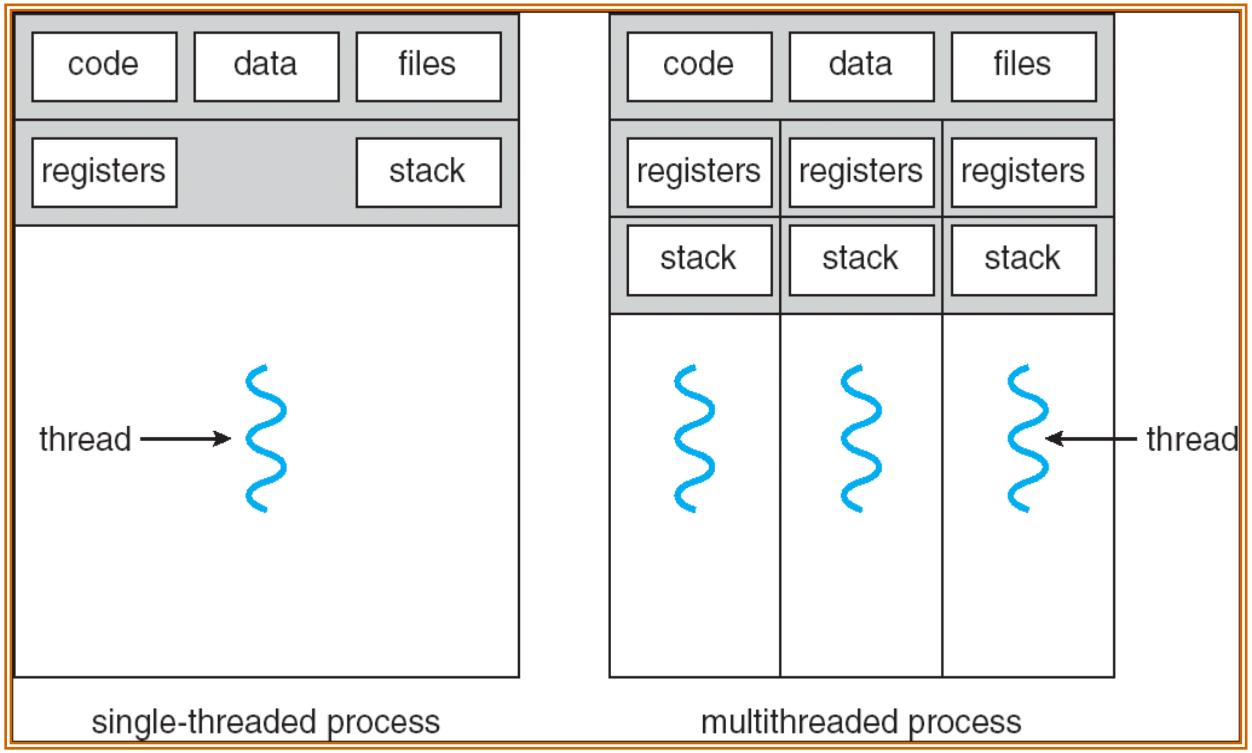
Direct & Indirect Communication

- Direct Communication
  - Processes must name each other explicitly:
    - **send** (*P, message*) – send a message to process P
    - **receive**(*Q, message*) – receive a message from process Q
  - Properties of communication link
    - Links are established automatically
    - A link is associated with exactly one pair of communicating processes
    - Between each pair there exists exactly one link
    - The link may be unidirectional, but is usually bi-directional
- Indirect Communication
  - Messages are directed and received from *mailboxes* (also referred to as *ports*)
    - Each mailbox has a unique *id* and is created by the kernel on request
    - Processes can communicate only if they share a mailbox
  - Properties of communication link
    - Link established only if processes share a common mailbox
    - A link may be associated with many processes
    - Each pair of processes may share several communication links
    - Link may be unidirectional or bi-directional

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send:** the sender blocks until the message is received by the other party
  - **Blocking receive:** the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send:** the sender sends the message and continues executing
  - **Non-blocking receive:** the receiver gets either a valid message or a null message (when nothing has been sent to the receiver)
- Often a combination:
  - Non-blocking send and blocking receive

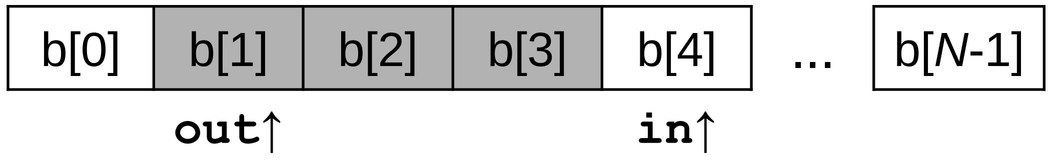
Single and Multithreaded Processes

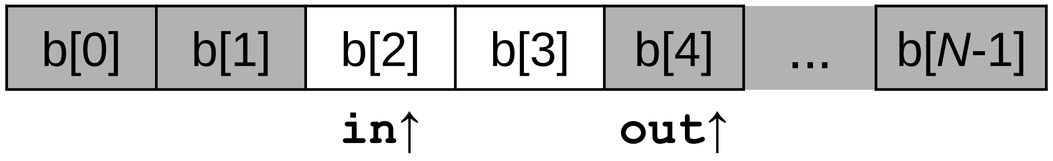


- Benefits of Multithreading
  - Responsiveness
  - Easy Resource Sharing
  - Economy
  - Utilization of Multi-processor Architectures

Example

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the producer-consumer problem:
  - We have a limited size buffer (*N* items). The *producer* puts data into the buffer and the *consumer* takes data from the buffer
  - We can have an integer count that keeps track of the number of occupied buffer entries. Initially, count is set to 0.
  - It is incremented by the producer after it inserts a new item in the buffer and is decremented by the consumer after it consumes a buffer item





Producer & Consumer Problem

Shared data:

```
#define BUF_SZ = 20
typedef struct { … } item;
item buffer[BUF_SZ];
int count = 0;
```

Producer:

```
void producer() {
    int in = 0;
    item nextProduced;
    while (1) {
        /* Generate new item */
        while (count == BUF_SZ) ;
            /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUF_SZ;
        count++ ;
    }
}
```

Consumer:

```
void consumer() {
    int out = 0;
    item nextConsumed;
    while (1) {
        while (count == 0) ;
            /* do nothing */
        nextConsumed =  buffer[out];
        out = (out + 1) % BUF_SZ;
        count-- ;
        /* Process nextConsumed */
    }
}
```

- This is a naive solution that does not work

Race Condition

- count++ could be implemented as

```
reg1 = count
reg1 = reg1 + 1
count = reg1
```

- count-- could be implemented as

```
reg2 = count
reg2 = reg2 - 1
count = reg2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer executes	reg1 = count	{reg1 = 5}
S1: producer executes	reg1 = reg1 + 1	{reg1 = 6}
S2: consumer executes	reg2 = count	{reg2 = 5}
S3: consumer executes	reg2 = reg2 – 1	{reg2 = 4}
S4: consumer executes	count = reg2	{count = 4}
S5: producer executes	count = reg1	{count = 6}

- Variable `count` represents a **shared resource**

Critical-Section Problem

What is a CRITICAL SECTION?

Part of the code when one process tries to access a **particular** resource shared with another process. We speak about a **critical section related to** that **resource**.

1. Mutual Exclusion – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections related to that resource
2. Progress – If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then one of the processes that wants to enter the critical section should be allowed as soon as possible
3. Bounded Waiting – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Critical Section Solution

Critical section has two basic operation: enter_CS and leave_CS

Possible implementation of this operation:

- Only SW at application layer
- Hardware support for operations
- SW solution with supprot of OS

SW solution for 2 processes

- Have a variable *turn* whose value indicates which process may enter the critical section. If *turn* == 0 then P_0 can enter, if *turn* == 1 then P_1 can.

***P**₀*

```
P0
while(TRUE) {
        while(turn!=0);   /* wait */
        critical_section();
        turn = 1;
        noncritical_section();
}
```

```
P1
while(TRUE) {
        while(turn!=1);   /* wait */
        critical_section();
        turn = 0;
        noncritical_section();
}
```

- However:
  - Suppose that P_0 finishes its critical section quickly and sets *turn* = 1; both processes are in their non-critical parts. P_0 is quick also in its non-critical part and wants to enter the critical section. As *turn* == 1, it will have to wait even though the critical section is free.
    - The requirement #2 (Progression) is violated
    - Moreover, the behaviour inadmissibly depends on the relative speed of the processes

Peterson’s Solution

- Two processes solution from 1981
- Assume that the LOAD and STORE **instructions are atomic**; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready (i = 0,1)

```
j = 1-i;
flag[i] = TRUE;
turn = j;
while ( flag[j] && turn == j);
     // CRITICAL SECTION
flag[i] = FALSE;
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Dangerous to disable interrupts at application level
    - Disabling interrupts is usually unavailable in CPU user mode
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this are not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptible
  - Test memory word and set value
  - Swap contents of two memory words

TestAndSet Instruction

- Semantics:

```
boolean TestAndSet (boolean *target)
    {
        boolean rv = *target;
        *target = TRUE;
        return rv:
    }
```

- Shared boolean variable lock, initialized to false.
- Solution:

```
    while (TestAndSet (&lock )) ;   // active waiting
        //    critical section
    lock = FALSE;
        //      remainder section
```

Swap Instruction

- Semantics:

```
void Swap (boolean *a, boolean *b)
   {
        boolean temp = *a;
        *a = *b;
        *b = temp:
   }
```

- Shared Boolean variable lock initialized to FALSE; each process has a local Boolean variable key.
- Solution:

```
key = TRUE;
while (key == TRUE) {   // waiting
      Swap (&lock, &key );
}
      //    critical section
lock = FALSE;
      //    remainder section
```

Synchronization without active waiting

- Active waiting waste CPU
  - Can lead to failure if process with high priority is actively waiting for process with low priority
- Solution: blocking by system functions
  - sleep() — the process is inactive
  - wakeup(process) — wake up process after leaving critical section

```
void producer() {
      while (1) {
            if (count == BUFFER_SIZE)  sleep();                    // if there is no space wait - sleep
            buffer[in] = nextProduced;  in = (in + 1) % BUFFER_SIZE;
            count++ ;
            if  (count == 1) wakeup(consumer);                     // if there is something to consume
      }
}
void consumer() {
      while (1) {
            if (count == 0) sleep();                               // cannot do anything – wait - sleep
            nextConsumed =  buffer[out];  out = (out + 1) % BUFFER_SIZE;
            count-- ;
            if (count == BUFFER_SIZE-1) wakeup(producer);          // now there is space for new product
      }
}
```

Synchronization without active waiting (2)

- Presented code is not good solution:
  - Critical section for shared variable count and function sleep() is not solved
    - Consumer read count == 0 and then Producer is switch before it call sleep() function
    - Producer insert new product into buffer and try to wake up Consumer because count == 1. But Consumer is not sleeping!
    - Producer is switched to Consumer that continues in program by calling sleep() function
    - When producer fill the buffer it call function sleep() – both processes are sleeping!
- Better solution: Semaphores

Semaphore

- Synchronization tool that does not require busy waiting
  - Busy waiting waists CPU time
- Semaphore *S* – <u>system</u> object
  - With each semaphore there is an associated waiting queue. Each entry in waiting queue has two data items:
    - `value` (of type integer)
    - pointer to next record in the list
  - Two standard operations modify S: wait() and signal()

```
wait(S)    {
              value--;
              if (value < 0) {
                 add caller to waiting queue
                 block(P);     }
           }
signal(S)  {
              value++;
              if (value <= 0) {
                 remove caller from the waiting queue
                 wakeup(P);     }
              }
```

Semaphore as General Synchronization Tool

- Counting semaphore – the integer value can range over an unrestricted domain
- Binary semaphore – the integer value can be only 0 or 1
  - Also known as mutex lock
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion (mutex)

```
Semaphore S;    //  initialized to 1
wait (S);
            Critical Section
signal (S);
```

Spin-lock

- Spin-lock is a general (counting) semaphore using busy waiting instead of blocking
  - Blocking and switching between threads and/or processes may be much more time demanding than the time waste caused by short-time busy waiting
  - One CPU does busy waiting and another CPU executes to clear away the reason for waiting
- Used in multiprocessors to implement short critical sections
  - Typically inside the OS kernel
- Used in many multiprocessor operating systems
  - Windows 2k/XP, Linuxes, ...

Deadlock and Starvation

- **Overlapping critical sections related to different resources**
- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

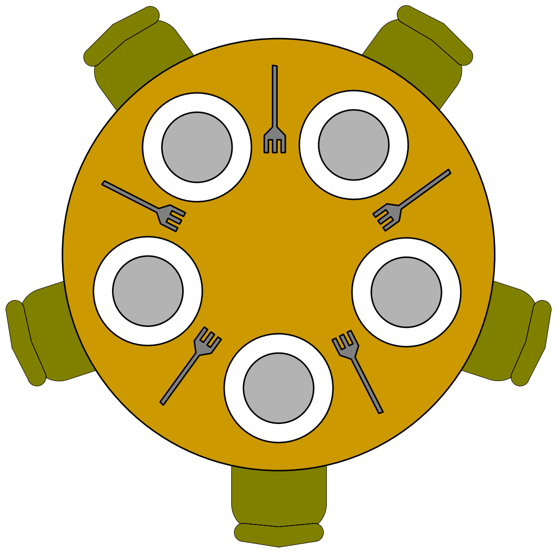
P_0 preempted

```
        P0                          P1
    wait (S);          --->     wait (Q);
    wait (Q);                   wait (S);
        .                           .
        .                           .
        .                           .
    signal  (S);                signal (Q);
    signal (Q);                 signal (S);
```

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
  - Passing data between 2 processes
- Readers and Writers Problem
  - Concurrent reading and writing data (in databases, ...)
- Dining-Philosophers Problem from 1965
  - An interesting illustrative problem to solve deadlocks
    - Five philosophers sit around a table; they either think or eat
    - They eat slippery spaghetti and each needs two sticks (forks)
    - What happens if all five philosophers pick-up their right-hand side stick?
      “They will die of hunger”



Bounded-Buffer Problem using Semaphores

- Three semaphores
  - mutex – for mutually exclusive access to the buffer – initialized to 1
  - used – counting semaphore indicating item count in buffer – initialized to 0
  - free – number of free items – initialized to `BUF_SZ`

```
void producer() {
   while (1) { /* Generate new item into nextProduced */
               wait(free);
               wait(mutex);
               buffer[in] =  nextProduced; in = (in + 1) % BUF_SZ;
               signal(mutex);
               signal(used);
   }
}

void consumer() {
   while (1) { wait(used);
               wait(mutex);
               nextConsumed = buffer[out];  out = (out + 1) % BUF_SZ;
               signal(mutex);
               signal(free);
               /* Process the item from nextConsumed */
   }
}
```

Readers and Writers

- The task: Several processes access shared data
    - Some processes read the data – readers
    - Other processes need to write (modify) the data – writers
  - Concurrent reads are allowed
    - An arbitrary number of readers can access the data with no limitation
  - Writing must be mutually exclusive to any other action (reading and writing)
    - At a moment, only one writer may access the data
    - Whenever a writer modifies the data, no reader may read it
- Two possible approaches
  - Priority for readers
    - No reader will wait unless the shared data are locked by a writer. In other words: Any reader waits only for leaving the critical section by a writer
    - Consequence: Writers may starve
  - Priority for writers
    - Any ready writer waits for freeing the critical section (by reader of writer). In other words: Any ready writer overtakes all ready readers.
    - Consequence: Readers may starve

Readers and Writers with Readers' Priority

Shared data

- semaphore wrt, readcountmutex;
- int readcount

Initialization

- wrt = 1; readcountmutex = 1; readcount = 0;

Implementation

Writer:

```
wait(wrt);
....
    writer modifies data
....
signal(wrt);
```

Reader:

```
wait(readcountmutex);
readcount++;
if (readcount==1) wait(wrt);
signal(readcountmutex);

    ... read shared data ...

wait(readcountmutex);
readcount--;
if (readcount==0) signal(wrt);
signal(readcountmutex);
```

Readers and Writers with Writers’ Priority

Shared data

- semaphore wrt, rdr, readcountmutex, writecountmutex;
  int readcount, writecount;

Initialization

- wrt = 1; rdr = 1; readcountmutex = 1; writecountmutex = 1;
  readcount = 0; writecount = 0;

Implementation

Reader:

```
wait(rdr);
wait(readcountmutex);
readcount++;
if (readcount == 1) wait(wrt);
signal(readcountmutex);
signal(rdr);

    ... read shared data ...

wait(readcountmutex);
readcount--;
if (readcount == 0) signal(wrt);
signal(readcountmutex);
```

Writer:

```
wait(writecountmutex);
writecount++;
if (writecount==1) wait(rdr);
signal(writecountmutex);
wait(wrt);

            ... modify shared data ...

signal(wrt);
wait(writecountmutex);
writecount--;
if (writecount==0) release(rdr);
signal(writecountmutex);
```

Dining Philosophers Problém

Shared data

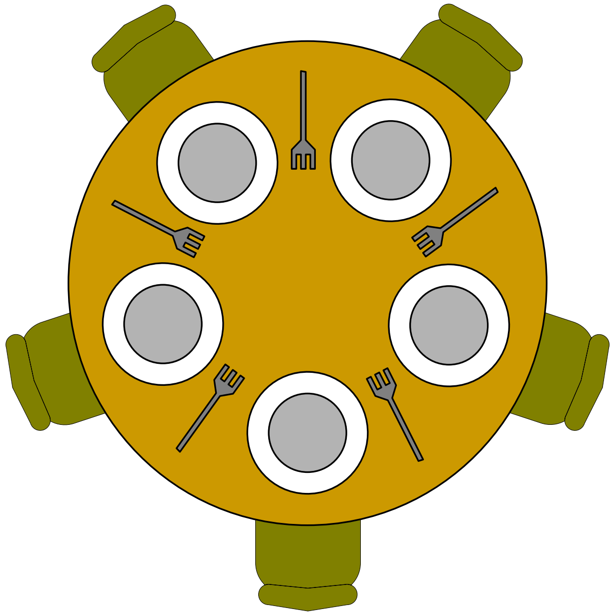
- semaphore chopStick[ ] = new Semaphore[5];

Initialization

- for(i=0; i<5; i++) chopStick[i] = 1;

Implementation of philosopher *i*:

```
do  {
      chopStick[i].wait;
      chopStick[(i+1) % 5].wait;
            eating();               //  Now eating
      chopStick[i].signal;
      chopStick[(i+1) % 5].signal;
            thinking();             //  Now thinking
 } while (TRUE) ;
```



- This solution contains NO deadlock prevention
  - A rigorous avoidance of deadlock for this task is very complicated

Monitors

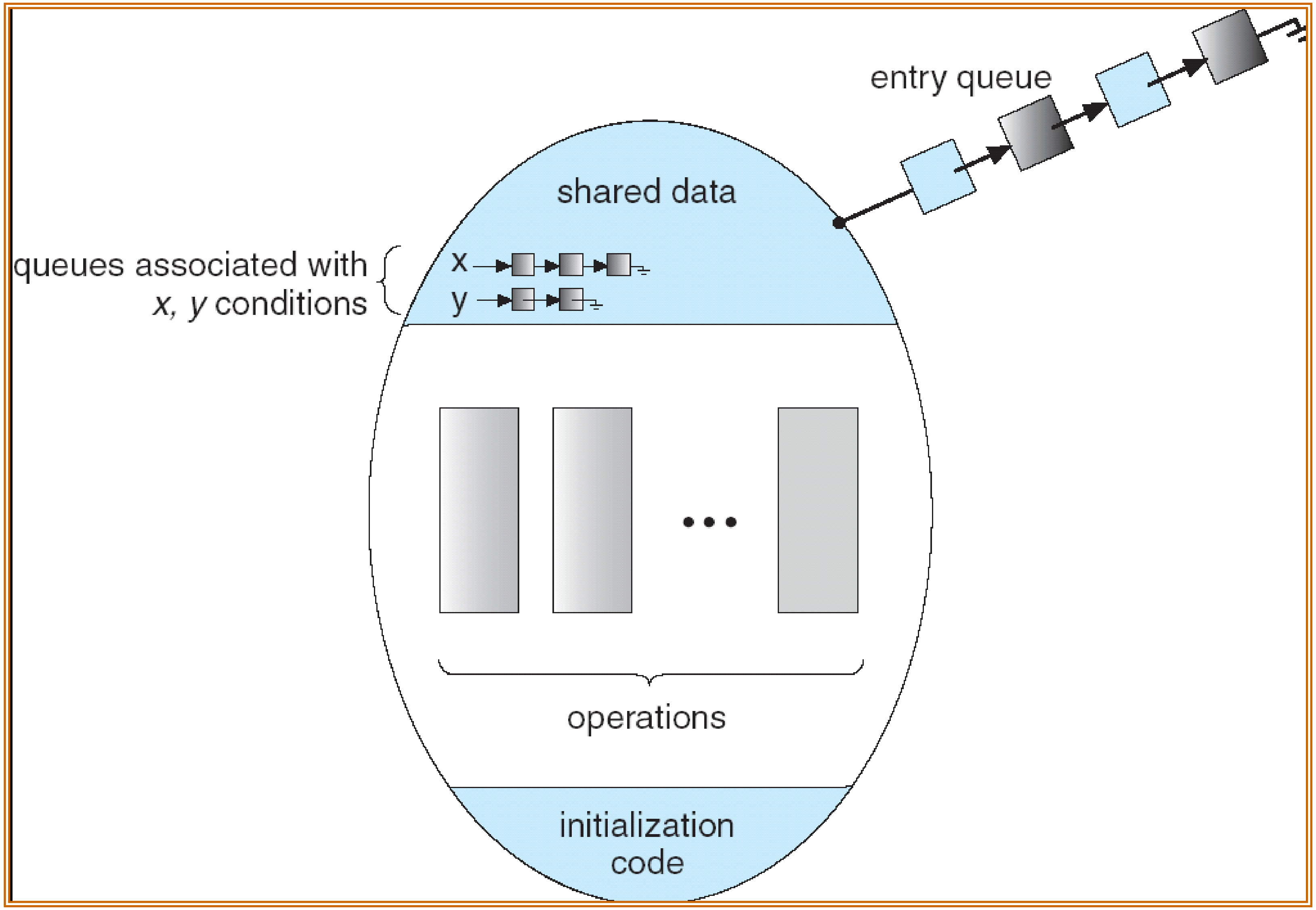
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor_name
{
    // shared variable declarations
   condition x, y;  // condition variables declarations
   procedure P1 (…) { …. }
              …
    procedure Pn (…) {……}

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

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Dining Philosophers with Monitors

```
monitor DP
{
       enum {THINKING; HUNGRY, EATING} state [5] ;
       condition self [5];

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

      void putdown (int i) {
              state[i] = THINKING;
                   // test left and right neighbors
               test((i + 4) % 5);
               test((i + 1) % 5);
        }

        void test (int i) {
               if ( (state[(i + 4) % 5] != EATING) &&
               (state[i] == HUNGRY) &&
               (state[(i + 1) % 5] != EATING) ) {
                      state[i] = EATING ;
                      self[i].signal () ;
                }
        }

      initialization_code() {
              for (int i = 0; i < 5; i++)
              state[i] = THINKING;
       }
}
```

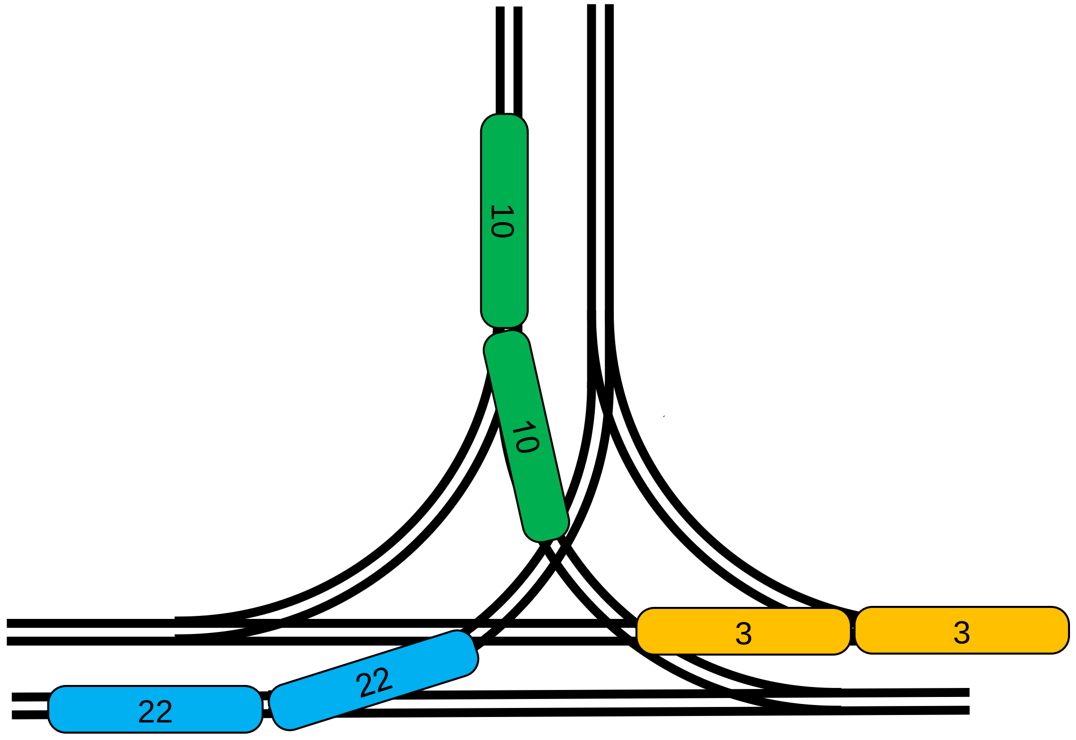
Synchronization Examples

- Windows XP Synchronization
  - Uses interrupt masks to protect access to global resources on uniprocessor systems
  - Uses spinlocks on multiprocessor systems
  - Also provides dispatcher objects which may act as either mutexes and semaphores
  - Dispatcher objects may also provide events
    - An event acts much like a condition variable
- Linux Synchronization
  - Disables interrupts to implement short critical sections
  - Provides semaphores and spin locks
- Pthreads Synchronization
  - Pthreads API is OS-independent and the detailed implementation depends on the particular OS
  - By POSIX, it provides
    - mutex locks
    - condition variables (monitors)
    - read-write locks (for long critical sections)
    - spin locks

Deadlock

Deadlock in real-life

Charles square – no tram can continue, one need to move back
Resources are tracks – for crossing you need to allocate other tracks for your motion

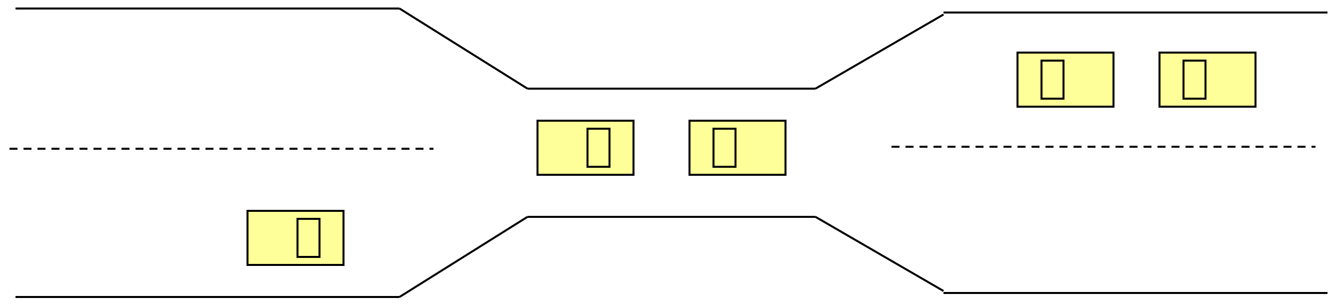


The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 files.
  - P_1 and P_2 each holds one file for writing and needs another one.
- Example
  - Semaphores *A* and *B*, initialized to 1 (mutexes)

```
        P0                  P1
    wait (A);           wait(B);
    wait (B);           wait(A);
      :                   :
```

Bridge Crossing Example



- Traffic only 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.

System Model

- Resource types R_1, R_2, . . ., R_m

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

Deadlock **can** occur if all **four conditions hold simultaneously**.

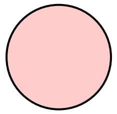
- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that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

NECESSARY condition! (not sufficient)

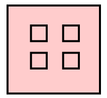
Coffman's conditions [E. G. Coffman, 1971]

Resource-Allocation Graph

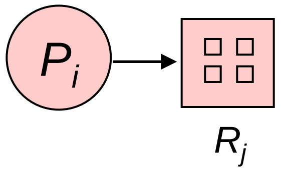
- A set of vertices *V* and a set of edges *E*
- V is partitioned into two types (bipartite graph):
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- Request edge – directed edge $P_1 \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$
- Process



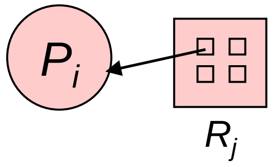
- Resource Type with 4 instances



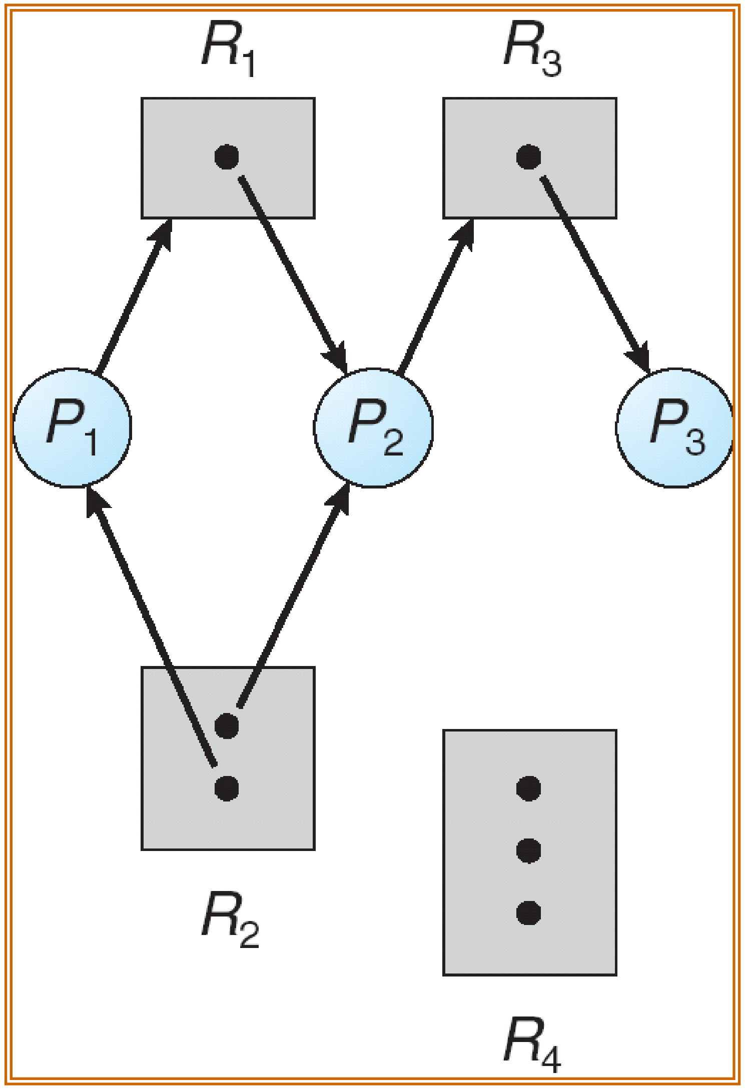
- P_i requests an instance of R_j



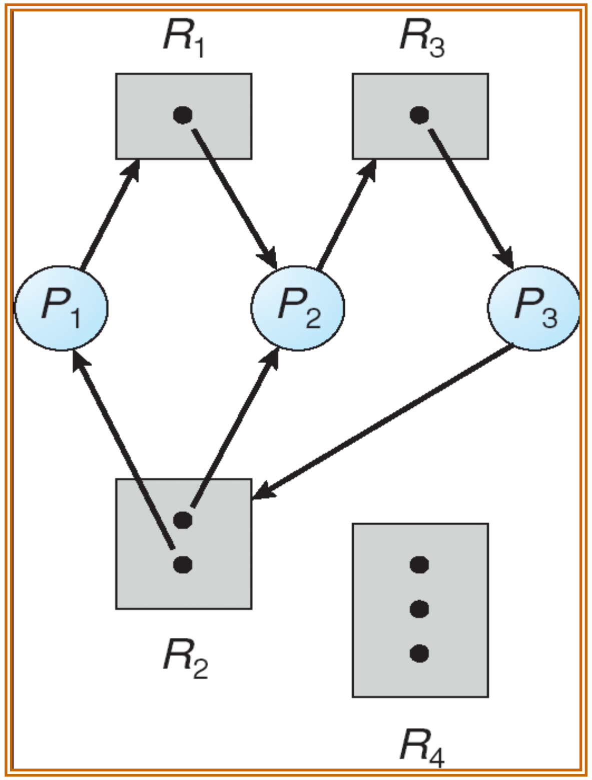
- P_i is holding an instance of R_j



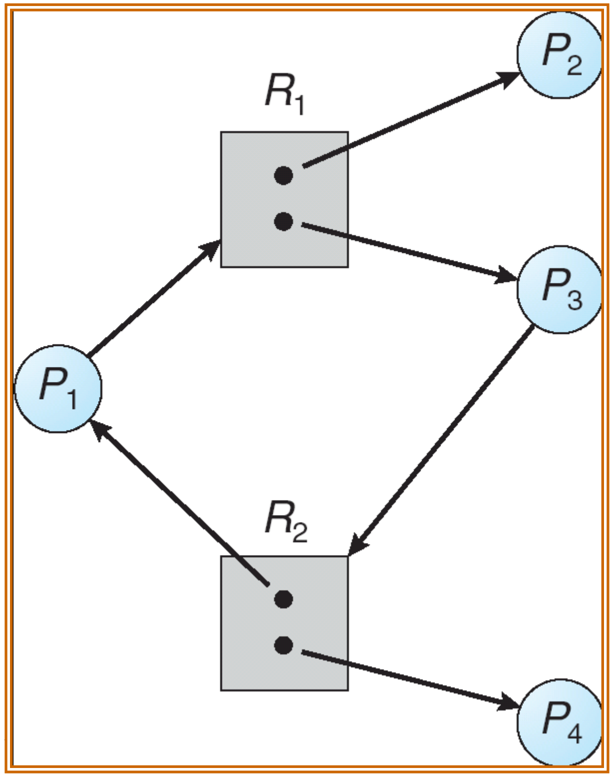
Example of a Resource Allocation Graph



Resource Allocation Graph With A Cycle



Deadlock



No Deadlock

- Conclusions:
  - If graph contains no cycles ⇒ no deadlock.
  - If graph contains a cycle ⇒
    - if only one instance per resource type, then deadlock.
    - if several instances per resource type, possibility of deadlock.

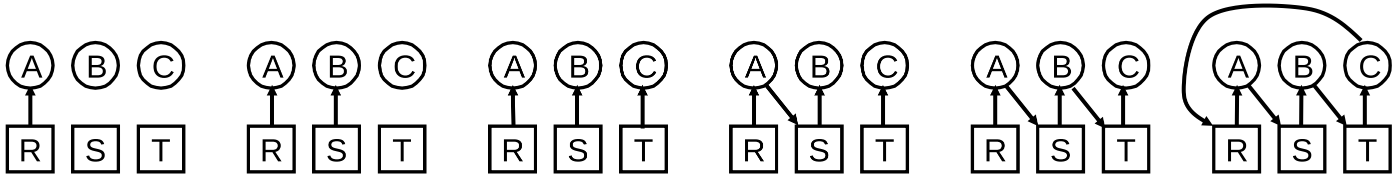
Can Scheduling Avoid Deadlocks?

- Consider an example:
  - Processes **A**, **B**, **C** compete for 3 single-instance resources R, S, T

A	**B**	**C**
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

1. *A* requests *R*
2. *B* requests *S*
3. *C* requests *T*
4. *A* requests *S*
5. *B* requests *T*
6. *C* requests *R*

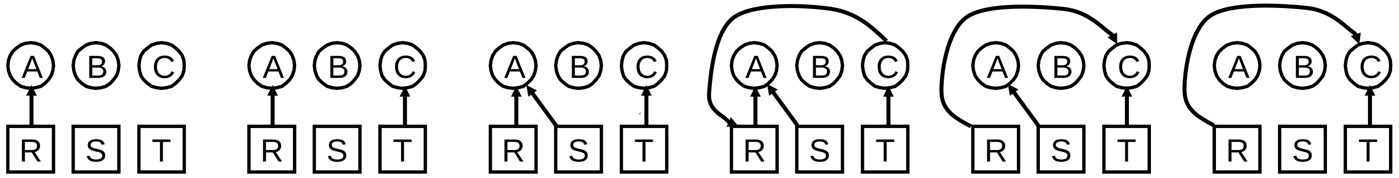
deadlock



1. *A* requests *R*
2. *C* requests *T*
3. *A* requests *S*
4. *C* requests *R*
5. *A* releases *R*
6. *A* releases *S*

no deadlock results

No more problems with B



- Can a careful scheduling avoid deadlocks?
  - What are the conditions?
  - What algorithm to use?

Approaches to Handling Deadlocks

- **Ostrich approach**: Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.
- **Deadlock Prevention**: Take such precautions that deadlock state is unlikely.
- **Deadlock Avoidance**: Ensure that the system will *never* enter a deadlock state.
- **Detect & Recover:** Allow the system to enter a deadlock state and then recover.

Coping with Deadlocks

Try to break at least one of the Coffman conditions

Restrain the ways request can be made

- **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.

Coping with Deadlocks (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.

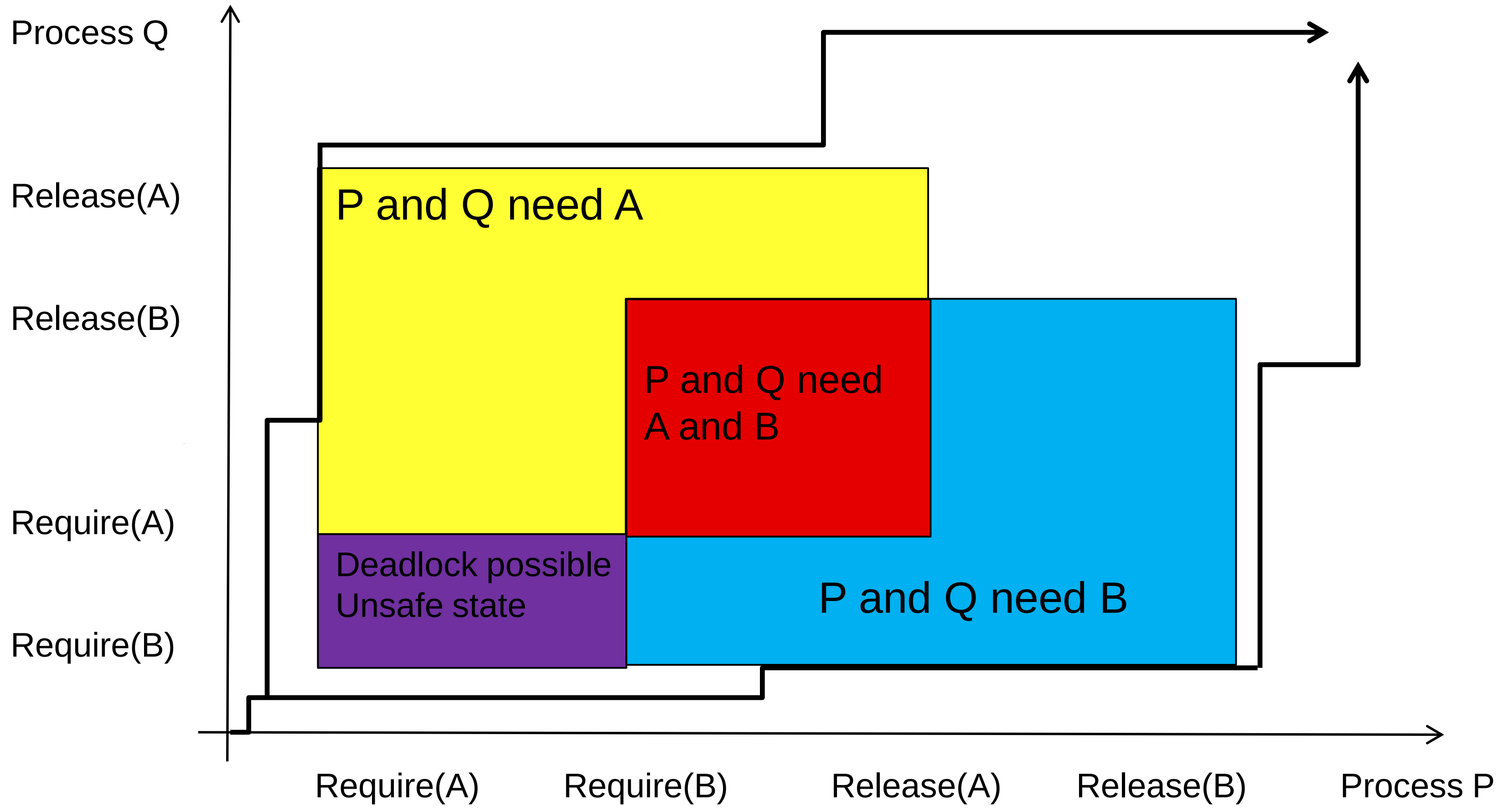
- Simplest and most useful model requires that each process declares the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *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 safe sequence of all processes.
- **Sequence** of processes <P_1, P_2, …, P_n> is **safe** if 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_k, with $k<i$.
  - If P_i resource needs are not immediately available, then P_i can wait until all P_k have finished.
  - When P_k 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.

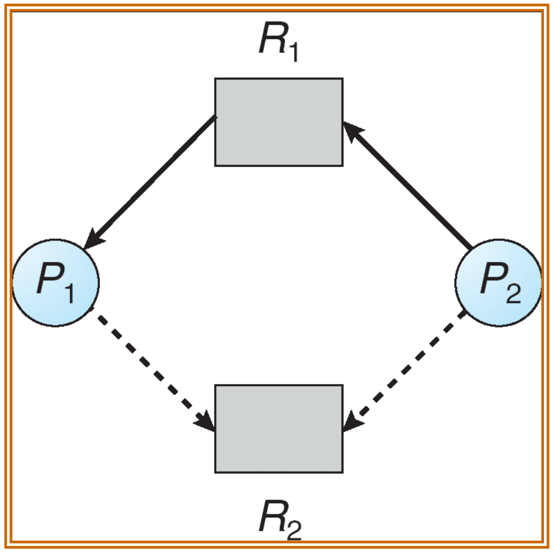
Basic Facts – System states

- 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.

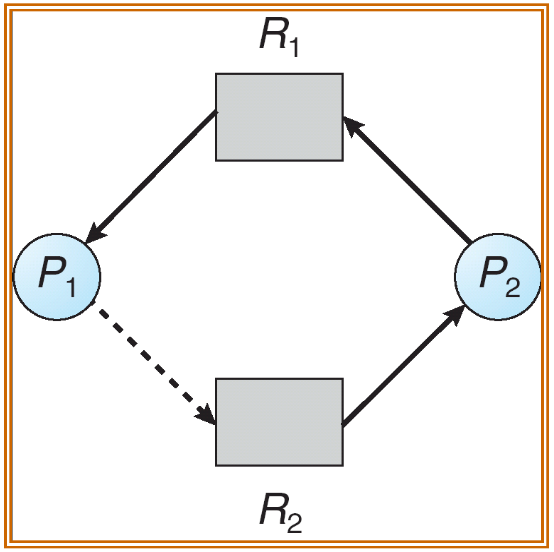


Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j
  - represented by a dashed line.
- Claim edge changes to a request edge when the process actually requests the resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed in the system *a priori*.



Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph

End of Lecture 4

Questions?