Chapter 6

Process Synchronization

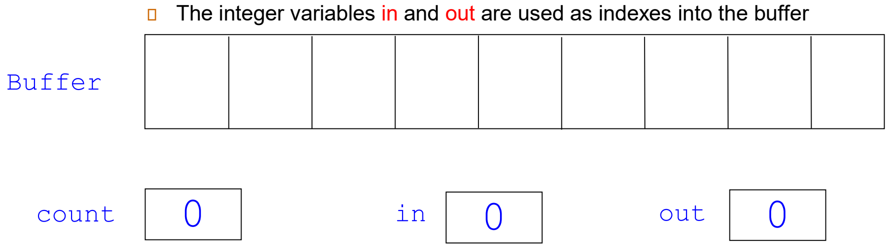
Module 6: Process Synchronization

- 6.1 Background
- 6.2 The Critical-Section Problem
- 6.3 Peterson's Solution
- 6.4 Synchronization Hardware
- 6.5 Semaphores
- 6.6 Classic Problems of Synchronization
- 6.7 Monitors
- 6.8 Synchronization Examples
- 6.9 Atomic Transactions (skip)

6.1 Background

Background

- Concurrent access of two or more processes 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 consumer-producer problem that can potentially fill all of the locations in a buffer
  - An integer variable count is used to keep track of the number of filled locations. Initially, count is set to 0. It is incremented by the producer when it produces a new item and places it in a buffer location and is decremented by the consumer when it consumes an item from a buffer location
  - The integer variables in and out are used as indexes into the buffer



Producer and Consumer

```
// Producer
item nextProduced;
while (true)
   {
   /*  produce an item and put in nextProduced  */
   . . .
   while (count == BUFFER_SIZE); // do nothing
   buffer [in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   count++;
   } // End while
```

```
#define BUFFER_SIZE 10

typedef struct
{. . .} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count = 0;
```

```
// Consumer
item nextConsumed;
while (true)
   {
   while (count == 0); // do nothing
   nextConsumed =  buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;
   /*  consume the item in nextConsumed
   . . .
   } // End while
```

Race Condition

- Although both the producer and consumer code segments are correct when each runs **sequentially**, they may not function correctly when executed **concurrently**
- `count++` could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1
- `count--` could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2
- One execution may interleave the above statements in an arbitrary order, where count initially is 5:

```
T0: producer executes register1 = count            {register1 = 5}
T1: producer executes register1 = register1 + 1    {register1 = 6}
T2: consumer executes register2 = count            {register2 = 5}
T3: consumer executes register2 = register2 - 1    {register2 = 4}
T4: producer executes count = register1            {count = 6 }
T5: consumer executes count = register2            {count = 4}
```

- This results in the incorrect state of “counter == 4”, indicating that 4 locations are full in the buffer, when, in fact, 5 locations are full

Race Condition (continued)

- A race condition has occurred
  - Several processes are able to access and manipulate the same data concurrently
  - The outcome of the execution depends on the particular order in which the data access and manipulation takes place
- To guard against the race condition shown in the previous example, we need to guarantee that only one process at a time can manipulate the counter variable

6.2 The Critical Section Problem

The Critical Section Problem

- Consider a system consisting of **n** processes in which each process has a segment of code called a **critical section**
  - In the critical section, the process may change a common variable, update a table, write to a file, etc.
  - When one process is updating in its critical section, no other process can be allowed to execute in its critical section (i.e., no two processes may execute in their critical sections at the same time)
- The **critical section problem** is to design a protocol that the processes can use to cooperate in manipulating common data
- A solution is to use three code sections
  - **Entry section**: Contains the code where each process requests permission to enter its critical section; this code can be run concurrently
  - **Critical section**: Contains the code for data manipulation; this code is only allowed to run exclusively for one process at a time
  - **Remainder section**: Contains any remaining code that can execute concurrently

General structure of a process with a critical section

```
while (TRUE)
   {
   entry section

   critical section

   remainder section

   } // End while
```

Solution to Critical-Section Problem

A solution to the critical section problem must satisfy the following three requirements:

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are **not** executing in their remainder sections can participate in the decision on who will enter its critical section next; this selection cannot be postponed indefinitely

- Bounded Waiting - There exists a bound, or limit, 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
  - We assume that each process executes at a nonzero speed
  - We make no assumption concerning relative speed of the N processes

Protecting Kernel Code from Race Conditions

- Various kernel code processes that perform the following activities must be safeguarded from possible race conditions
  - Maintaining a common list of open files
  - Updating structures for maintaining memory allocation
  - Updating structures for maintaining process lists
  - Updating structures used in interrupt handling
- Two general approaches are used to handle critical sections in operating systems
  - Preemptive kernels
  - Non-preemptive kernels

Preemptive and Non-preemptive Kernels

- Preemptive kernel
  - Allows a process to be preempted while it is running in kernel mode
  - Must be carefully designed to ensure that shared kernel data are free from race conditions
  - Is more suitable for real-time programming
  - May be more responsive because it won't be waiting for a kernel mode process to relinquish the CPU
  - Is implemented in Linux and Solaris
- Non-preemptive kernel
  - Does not allow a process to be preempted while it is running in kernel mode
  - Allows a process to run until the process does one of the following: exits kernel mode, blocks, or voluntarily gives up the CPU (i.e., terminates)
  - Is essentially free from race conditions on kernel data structure
  - Is implemented in Windows XP and traditional UNIX

6.3 Peterson's Solution

Peterson's Solution

- Peterson's solution is a classic software-based solution to the critical section problem
  - It provides a good algorithm description of solving the critical section problem
  - It illustrates some of the complexities involved in designing software that addresses the three solution requirements (mutual exclusion, progress, and bounded waiting)
  - Unfortunately, the assembly language implementations in modern computer architectures will not guarantee that Peterson's solution will work (i.e., LOAD and STORE are not atomic)

Peterson's Solution (continued)

- The solution is restricted to two processes that alternate execution between their critical sections and remainder sections
- It assumes that the LOAD and STORE instructions are atomic (they cannot be interrupted)
- The processes are numbered P_0 and P_1
  - For convenience, P_j denotes the other process, that is, j equals 1 - i
- The processes share two variables:
  - `int turn;`
  - `boolean flag[2];`
- The variable `turn` indicates whose turn it is to enter the critical section
  - It `turn == i`, then process P_i is allowed to execute in its critical section
- The `flag` array is used to indicate if a process is ready to enter its critical section
  - If `flag[i] == true`, then P_i is ready to enter its critical section

Algorithm for Process P_i

```
int turn;
boolean flag[2];
```

```
while (TRUE)
   {
   flag[i] = TRUE; // I am ready to enter my critical section
   turn = j; // But you can go first if you are ready
   while ( flag[j] && turn == j); // You're in your critical section
                                   // so I will just wait
   CRITICAL SECTION // I am now in my critical section

   flag[i] = FALSE; // I am done with my critical section

   REMAINDER SECTION // I have other processing to do now

   } // End while
```

6.4 Synchronization Hardware

The Concept of a Lock

- In general, any solution to the critical section problem requires some form of **lock**
- Race conditions are prevented by requiring that critical regions be protected by locks
  - A process must acquire a lock before entering its critical section
  - A process releases the lock when it exits its critical section

```
while (TRUE)
   {
   // Acquire lock
   // Critical section
   // Release lock
   // Remainder section
   } // End while
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Single processor – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this are not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptible
  - Either test memory word and set value
    `boolean TestAndSet (boolean *target);`
  - Or swap contents of two memory words
    `void Swap (boolean *a, boolean *b);`

TestAndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
{
boolean originalValue = *target;
*target = TRUE;
return originalValue;
} // End TestAndSet
```

- The function always sets the lock parameter to TRUE, but returns the original value of the lock

Solution using TestAndSet

- Shared Boolean variable named `lock`, initialized to FALSE
- Solution:

```
while (TRUE)
   {
   while ( TestAndSet (&lock )) ;    // do nothing

   // Critical section
   . . .
   lock = FALSE;

   // Remainder section
   . . .
   } // End while
```

Swap Instruction

- Definition:

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

Solution using Swap

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

```
while (TRUE)
   {
   key = TRUE;
   while ( key == TRUE)
      Swap (&lock, &key );

   //    critical section
   . . .
   lock = FALSE;

   //      remainder section
   . . .
   } // End while
```

6.5 Semaphores

Semaphore

- Another possible synchronization tool that is less complicated than `TestAndSet()` or `Swap()` is a **semaphore**
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal()
  - Originally these operations were called P() and V()
- The wait() and signal() operations are each executed indivisibly
  - wait(S)
    ```
    {
    while S <= 0
        ; // do nothing
    S--;
    } // End wait
    ```
  - signal(S)
    ```
    {
    S++;
    } // End signal
    ```

Semaphore as General Synchronization Tool

- Operating systems often distinguish between counting and binary semaphores
  - The value of a **counting semaphore** can range over an unrestricted domain
  - The value of a **binary semaphore** can range only between 0 and 1
    - On some systems, a binary semaphore is known as a **mutex lock**
- Binary semaphores can be used to deal with the critical section problem for multiple processes
  - The `n` processes share a semaphore, `mutex`, initialized to 1

```
while (TRUE)
   {
   wait(mutex);
   // Critical section

   signal(mutex);
   // Remainder section

   } // End while
```

Semaphore as General Synchronization Tool (continued)

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances
- The semaphore is initialized to the number of resources available
- Each process that wishes to use a resource performs a `wait()` operation on the semaphore (i.e., a decrement)
- When a process releases a resource, it performs a `signal()` operation on the semaphore (i.e., an increment)
- When the count for the semaphore reaches 0, all the resources are in use
  - Processes wanting to use a resource will block until the semaphore value becomes greater than 0
- Semaphores can also be used to solve various synchronization problems for concurrently running processes

Semaphore Implementation

- The main disadvantage of the semaphore definition given here so far is that is requires busy waiting
  - While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry section code
  - This semaphore implementation is called a **spinlock** because the process spins while waiting for the lock
- To overcome the need for busy waiting, we can modify the definition of the `wait()` and `signal()` semaphore operations
  - When a process executes the `wait()` operation and finds that the semaphore value is not positive, it must wait
  - However, rather than engaging in busy waiting, the process can block itself
  - The `block()` operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state
  - A process that is blocked, waiting on a semaphore, is restarted when some other process executes a `signal()` operation
    - The process is restarted by a `wakeup()` operation, which changes the process from the waiting state to the ready state, and places it in the ready queue

Semaphore Implementation with <u>no busy waiting</u>

- To implement semaphores under this new definition, we define a semaphore as a structure in C

```
typedef struct
   {
   int value;
   struct process *list;
   } semaphore;
```

- Each semaphore has an integer value and a list of processes
- When a process must wait on a semaphore, the `wait()` operation adds it to the list of processes
- A `signal()` operation removes one process from the list of waiting processes and awakens that process

Semaphore Implementation with no busy waiting (Continued)

- Implementation of `wait()` function:

```
void wait (semaphore *S)
   {
   S->value--;
   if (S->value < 0) {
       add this process to waiting queue
      block();   }
  } // End wait
```

- Implementation of `signal()` function:

```
void signal (semaphore *S)
   {
   S->value++;
   if (S->value <= 0) {
       remove a process P from the waiting queue
      wakeup(P);   }
  } // End signal
```

Semaphore Implementation with no busy waiting (Continued)

- The `block()` operation suspends the process that invokes it
- The `wakeup()` operation resumes the execution of a blocked process P
- These two operations would be provided by the operating system as basic system calls
- The critical aspect of semaphores is that they be executed atomically
  - We must guarantee that no two processes can execute the `wait()` and `signal()` operations on the same semaphore at the same time
  - In a single processor environment, we can solve this by simply inhibiting interrupts during the time the `wait()` and `signal()` operations are executing
  - In a multi-processor environment, interrupts must be disabled on every processor
    - This can be a difficult task and can seriously diminish performance
    - Consequently, alternative locking such as spinlocks are used instead

Process Deadlock

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes
  - The event in question is the execution of a `signal()` operation
  - This situation is called a **deadlock**
- To illustrate, let `S` and `Q` be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

  - P_0 executes `wait(S)` and then P_1 executes `wait(Q)`
  - When P_0 then executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`
  - Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`
  - Since these `signal()` operations cannot be executed, P_0 and P_1 are deadlocked

Process Starvation

- Another problem related to deadlocks is indefinite blocking, or **starvation**
- This is a situation in which processes wait indefinitely within the semaphore
- Starvation may occur if we add and remove processes from the list associated with a semaphore in a last in, first out (LIFO) order

6.6 Classic Problems of Synchronization

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true)  {

        //   produce an item

    wait (empty);
    wait (mutex);

        // add the item to the  buffer

    signal (mutex);
    signal (full);
}
```

- The structure of the consumer process

```
while (true) {
    wait (full);
    wait (mutex);

        //  remove an item from  buffer

    signal (mutex);
    signal (empty);

        //  consume the removed item

}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
}
```

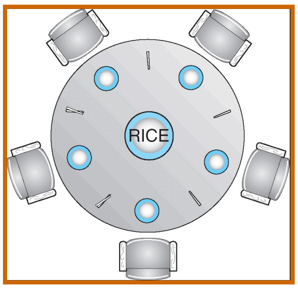
- The structure of a reader process

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readercount == 1)  wait (wrt) ;
        signal (mutex);

            // reading is performed

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

Dining-Philosophers Problem



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

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
while (true)  {
     wait ( chopstick[i] );
     wait ( chopStick[ (i + 1) % 5] );

            //  eat

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

              //  think

}
```

6.7 Monitors

Problems with Incorrect Use of Semaphores

- Incorrect use; order of signal and wait are reversed; several processes are in the critical section at the same time; this violates the mutual exclusion requirement
  - signal (mutex)
        // Critical section
    wait (mutex)
- Incorrect use; signal call replaced with wait call; deadlock will occur
  - wait (mutex)
        // Critical section
    wait (mutex) // ← This should have correctly been signal(mutex)
- Incorrect use; omitting of wait (mutex) or signal (mutex) (or both); mutual exclusion is violated or deadlock will occur
  - Normal section

    // Critical section

    Normal section

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

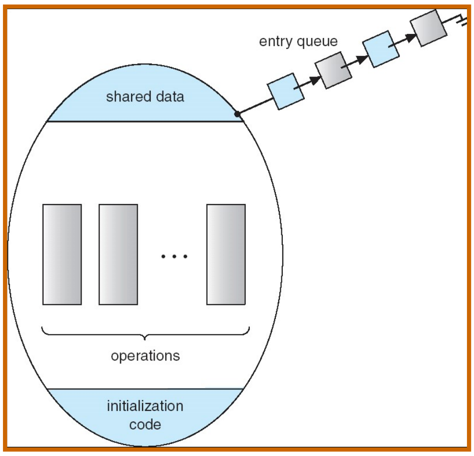
```
monitor monitor-name
{
 // shared variable declarations

 procedure P1 (…) { …. }
   …

 procedure Pn (…) {……}

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

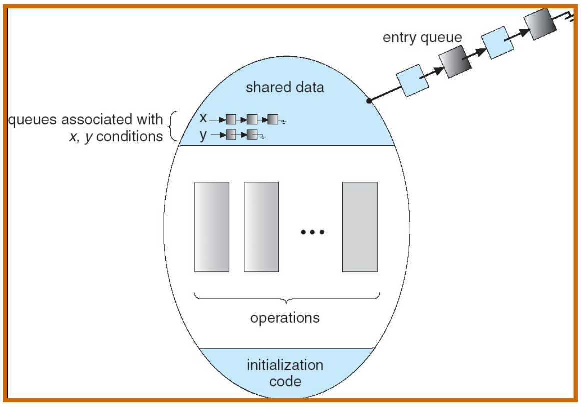
Schematic view of a Monitor



Condition Variables

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

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DiningPhilosophers
   {
    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);
           }
    }
```

Solution to Dining Philosophers (cont)

```
    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;
    }
}
```

Solution to Dining Philosophers (cont)

- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

```
dp.pickup (i)

    EAT

dp.putdown (i)
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
  …
        body of F;

  …
if (next-count > 0)
        signal(next)
else
        signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x-sem; // (initially  = 0)
int x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next-count > 0)
        signal(next);
else
        signal(mutex);
wait(x-sem);
x-count--;
```

Monitor Implementation

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
        next-count++;
        signal(x-sem);
        wait(next);
        next-count--;
}
```

6.8 Synchronization Examples

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads
- POSIX Semaphores

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

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

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Java Synchronization between threads

```
public class SomeClass
{
. . .
   public synchronized void safeMethod()
   {
   . . .
   }

}
```

```
SomeClass myObject = new SomeClass();

myObject.safeMethod();
```

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

Pthreads Synchronization (continued)

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, NULL);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

POSIX Semaphores

```
#include <pthread.h>

int sem_init(sem_t *semaphore, int pshared, unsigned int value);

The sem_init function initializes the semaphore.  The first
parameter is a pointer to a semaphore variable.  The second
parameter with a value of zero indicates that the semaphore is
shared among threads within a process.  The third parameter is
the value to which the semaphore variable is initialized.
Thus, this is a counting semaphore.

int sem_wait(sem_t *semaphore);  // WAIT

int sem_post(sem_t *semaphore);  // SIGNAL
```

POSIX Semaphores (Example)

```
#include <pthread.h>

sem_t theSemaphore;
. . .
. . .

sem_init(&theSemaphore, 0, 1);

. . .
. . .
sem_wait(&theSemaphore);  // WAIT

// Critical Region

sem_post(&theSemaphore);  // SIGNAL
```

End of Chapter 6

Chapter 7

Deadlocks

Chapter 7: Deadlocks

- 7.1 System Model
- 7.2 Deadlock Characterization
- 7.3 Methods for Handling Deadlocks
- 7.4 Deadlock Prevention
- 7.5 Deadlock Avoidance
- 7.6 Deadlock Detection
- 7.7 Recovery from Deadlock

Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing, avoiding, or detecting deadlocks in a computer system

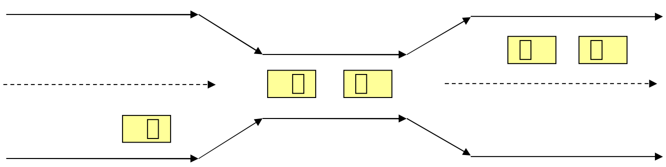
7.1 System Model

The Deadlock Problem

- A deadlock consists of a set of blocked processes, each holding a resource and waiting to acquire a resource held by another process in the set
- Example #1
  - A system has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs the other one
- Example #2
  - Semaphores A and B, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example



- Traffic only in one direction
- The resource is a one-lane bridge
- 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 $R_1, R_2, \ldots, R_m$

    CPU cycles, memory space, I/O devices
- Each resource type R_i has *1 or more* instances
- Each process utilizes a resource as follows:
    - request
    - use
    - release

7.2 Deadlock Characterization

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

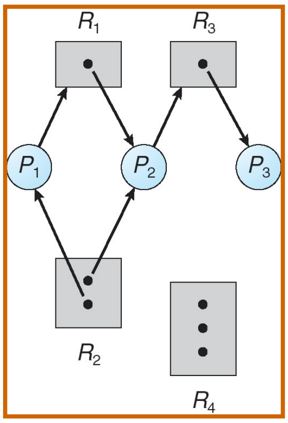
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it after that process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

Resource-Allocation Graph

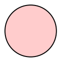
A set of vertices V and a set of edges E.

- 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 $R_j \rightarrow P_i$

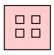


Resource-Allocation Graph (Cont.)

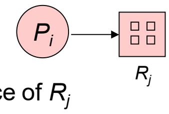
- Process



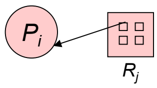
- Resource Type with 4 instances



- P_i requests instance of R_j

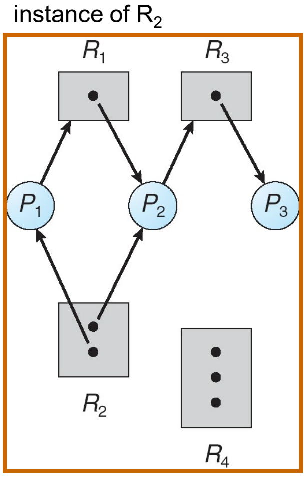


- P_i is holding an instance of R_j

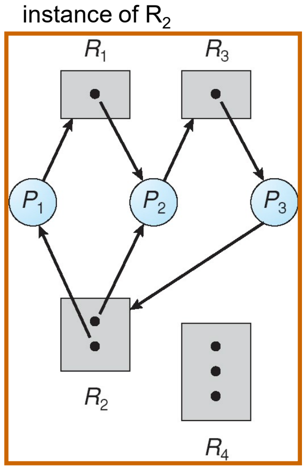


Resource Allocation Graph With A Deadlock

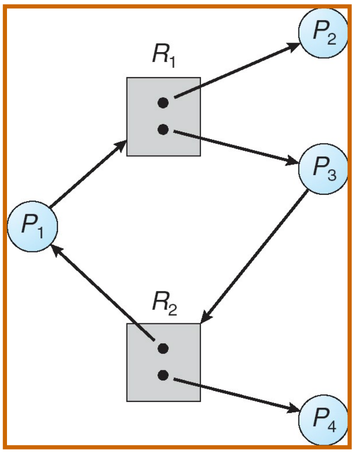
Before P_3 requested an instance of R_2



After P_3 requested an instance of R_2



Graph With A Cycle But No Deadlock



Process P_4 may release its instance of resource type R_2. That resource can then be allocated to P3, thereby breaking the cycle.

Relationship of cycles to deadlocks

- If a resource allocation graph contains no cycles ⇒ no deadlock
- If a resource allocation graph contains a cycle and if only one instance exists per resource type ⇒ deadlock
- If a resource allocation graph contains a cycle and and if several instances exists per resource type ⇒ possibility of deadlock

7.3 Methods for Handling Deadlocks

Methods for Handling Deadlocks

- **Prevention**
  - Ensure that the system will *never* enter a deadlock state
- **Avoidance**
  - Ensure that the system will *never* enter an unsafe state
- **Detection**
  - Allow the system to enter a deadlock state and then recover
- **Do Nothing**
  - Ignore the problem and let the user or system administrator respond to the problem; used by most operating systems, including Windows and UNIX

7.4 Deadlock Prevention

Deadlock Prevention

To prevent deadlock, we can restrain the ways that a request can be made

- **Mutual Exclusion** – The mutual-exclusion condition must hold for non-sharable resources
- **Hold and Wait** – we must guarantee that whenever a process requests a resource, it <u>does not</u> hold any other resources
  - Require a process to request and be allocated all its resources <u>before</u> it begins execution, or allow a process to request resources <u>only</u> when the process has none
  - Result: 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
  - A 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. For example:

  F(tape drive) = 1
  F(disk drive) = 5
  F(printer) = 12

7.5 Deadlock Avoidance

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

a priori: formed or conceived beforehand

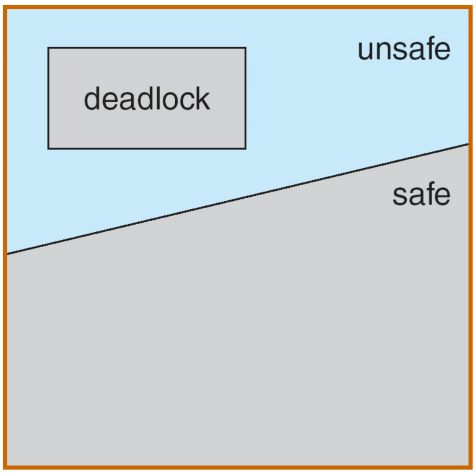
Safe State

- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a safe state
- A system is in a safe state only if there exists a safe sequence
- A sequence of processes $<P_1, P_2, \ldots, P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resource requests that P_i can still make, can be satisfied by currently available resources plus resources held by all P_j, with $j < i$.
- That is:
  - If the 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 State (continued)

- 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



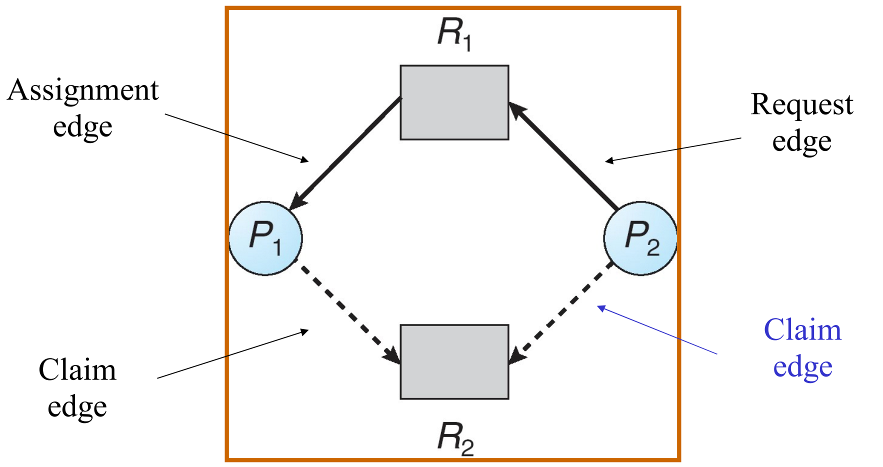
Avoidance algorithms

- For a single instance of a resource type, use a resource-allocation graph
- For multiple instances of a resource type, use the banker's algorithm

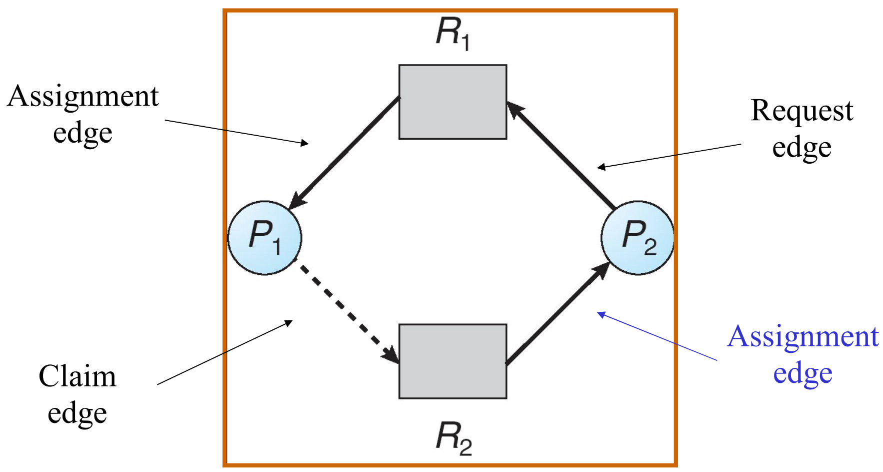
Resource-Allocation Graph Scheme

- Introduce a new kind of edge called a claim edge
- *Claim edge* $P_i \dashrightarrow R_j$ indicates that process P_j may request resource R_j; which is represented by a dashed line
- A claim edge converts to a request edge when a process **requests** a resource
- A request edge converts to an assignment edge when the resource is **allocated** to the process
- When a resource is **released** by a process, an assignment edge reconverts to a claim edge
- Resources must be **claimed *a priori*** in the system

Resource-Allocation Graph with Claim Edges



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

- Used when there exists **multiple** instances of a resource type
- Each process must **a priori** claim maximum use
- When a process requests a resource, it may have to wait
- When a process gets all its resources, it must return them in a finite amount of time

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* $[i,j]$

7.6 Deadlock Detection

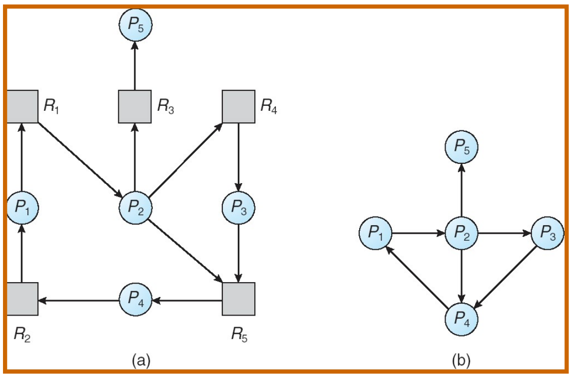
Deadlock Detection

- For deadlock detection, the system must provide
  - An algorithm that examines the state of the system to detect whether a deadlock has occurred
  - And an algorithm to recover from the deadlock
- A detection-and-recovery scheme requires various kinds of overhead
  - Run-time costs of maintaining necessary information and executing the detection algorithm
  - Potential losses inherent in recovering from a deadlock

Single Instance of Each Resource Type

- Requires the creation and maintenance of a *wait-for* graph
  - Consists of a variant of the resource-allocation graph
  - The graph is obtained by **removing** the resource nodes from a resource-allocation graph and **collapsing** the appropriate edges
  - Consequently; all nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph
  - If there is a cycle, there exists a deadlock
  - An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Multiple Instances of a Resource Type

Required data structures:

- ***Available**:* A vector of length *m* indicates the number of available resources of each type.
- ***Allocation**:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- ***Request**:* An *n* x *m* matrix indicates the current request of each process. If *Request* [i_j] = *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Detection-Algorithm Usage

- When, and how often, to invoke the detection algorithm depends on:
  - How often is a deadlock likely to occur?
  - How many processes will be affected by deadlock when it happens?
- If the detection algorithm is invoked arbitrarily, there may be **many** cycles in the resource graph and so we would not be able to tell **which one** of the many deadlocked processes "caused" the deadlock
- If the detection algorithm is invoked for every resource request, such an action will incur a considerable **overhead** in computation time
- A less expensive alternative is to invoke the algorithm when CPU utilization drops **below 40%**, for example
  - This is based on the observation that a deadlock eventually cripples system throughput and causes CPU utilization to drop

7.7 Recovery From Deadlock

Recovery from Deadlock

- Two Approaches
  - Process termination
  - Resource preemption

Recovery from Deadlock: Process Termination

- **Abort all deadlocked processes**
  - This approach will break the deadlock, but at great expense
- **Abort one process at a time until the deadlock cycle is eliminated**
  - This approach incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be re-invoked to determine whether any processes are still deadlocked
- **Many factors may affect which process is chosen for termination**
  - What is the priority of the process?
  - How long has the process run so far and how much longer will the process need to run before completing its task?
  - How many and what type of resources has the process used?
  - How many more resources does the process need in order to finish its task?
  - How many processes will need to be terminated?
  - Is the process interactive or batch?

Recovery from Deadlock: Resource Preemption

- With this approach, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- When preemption is required to deal with deadlocks, then three issues need to be addressed:
  - **Selecting a victim** – Which resources and which processes are to be preempted?
  - **Rollback** – If we preempt a resource from a process, what should be done with that process?
  - **Starvation** – How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

Summary

- Four necessary conditions must hold in the system for a deadlock to occur
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Four principal methods for dealing with deadlocks
  - Use some protocol to (1) **prevent** or (2) **avoid** deadlocks, ensuring that the system will never enter a deadlock state
  - Allow the system to enter a deadlock state, (3) **detect** it, **and** then **recover**
    - Recover by **process termination** or **resource preemption**
  - **(4) Do nothing;** ignore the problem altogether and pretend that deadlocks never occur in the system (used by Windows and Unix)
- To prevent deadlocks, we can ensure that **at least one** of the four necessary conditions **never holds**

End of Chapter 7