CS343 - Operating Systems

Module-3A

Inter Process Communication

Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

http://www.iitg.ac.in/johnjose/

Session Outline

- Multitasking/Multiprocessing Applications
- Review of process management functions
- Process creation and termination
- Inter Process Communication (IPC)
- Producer-Consumer problem
- IPC- shared memory
- IPC-message passing
- Direct vs indirect communiction

Multitasking in Mobile Systems

- Some mobile systems allow only one process to run, others suspended
- Due to screen space limits, user interface limits iOS provides for a
  - **Single foreground process-** controlled via user interface
  - **Multiple background processes**– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- **Android runs foreground and background**, with fewer limits
  - Background process uses a service to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

Multi-process Application

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash

- Google Chrome Browser is multiprocess with 3 types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript.
  - **Plug-in** process for each type of plug-in

Process Management

- Creating and deleting both user and system processes
- Suspending and resuming processes (context switching, scheduling)
- Providing mechanisms for process communication
- Providing mechanisms for process synchronization
- Providing mechanisms for deadlock handling

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent′s resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some OS do not allow child to exists if its parent has terminated.
- **Cascading termination:** If a process terminates, then all its children, grand children, etc. must also be terminated.
- The parent process may wait for termination of a child process by using the **`wait()`** system call.
- The call returns status information and the **`pid`** of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait`**, process is an **orphan**

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

Process Management

- Creating and deleting both user and system processes
- Suspending and resuming processes (context switching, scheduling)
- Providing mechanisms for process communication
- Providing mechanisms for process synchronization
- Providing mechanisms for deadlock handling

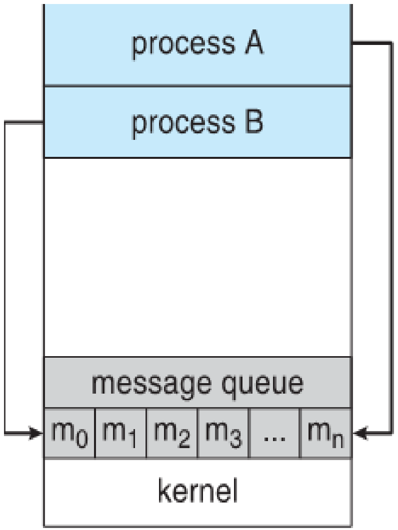
Inter-process Communication

- Processes within a system may be **independent** or **cooperating**
- **Independent process** cannot affect or be affected by the execution of another process
- **Cooperating process** can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

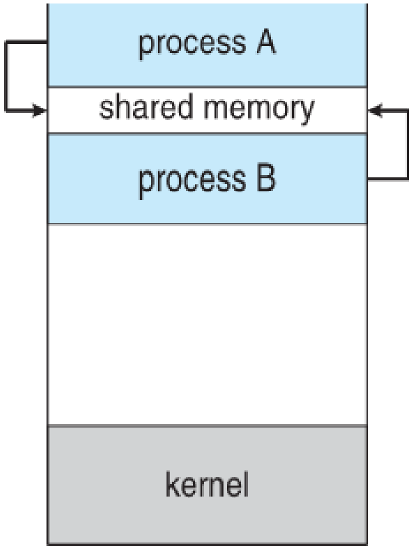
Communications Models

- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC:

Message passing



Shared memory



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

Bounded-Buffer – Producer & Consumer

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

Producer

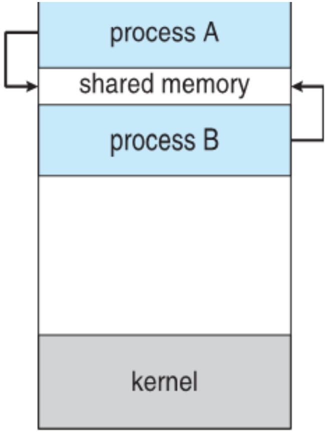
```
item next_produced;

while (true)

 {      /* produce an item in next
        produced */

    while(((in + 1)% BUFFER_SIZE)
    == out)

        ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;

}
```

Consumer

```
item next_consumed;

while (true)

 {
    while (in == out)

        ; /* do nothing */
    next_consumed = buffer[out];

    out = (out + 1) % BUFFER_SIZE;
    /* consume the item in next
consumed */

 }
```

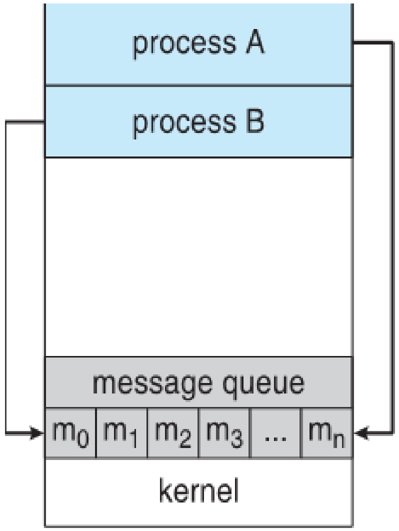
IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.



IPC – Message Passing

- 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)
  - **receive**(message)
- The message size is either fixed or variable



IPC – Message Passing

- If processes P and Q wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links between a pair of communicating processes?
  - What is the capacity of a link?
  - Unidirectional or bi-directional link?
  - Is the size of a message in the link fixed or variable?

IPC – Message Passing

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

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
  - Each mailbox has a unique id
  - 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

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
- **send**(A, message) – send a message to mailbox A
- **receive**(A, message) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available

Synchronization

- Message passing may be either blocking or non-blocking
- Non-blocking is considered asynchronous
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways

1. Zero capacity – no messages are queued on a link.
   Sender must wait for receiver
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full
3. Unbounded capacity – infinite length
   Sender never waits

Thank you

johnjose@iitg.ac.in
http://www.iitg.ac.in/johnjose/

CS343 - Operating Systems

Module-3B

IPC in Client Server Systems

Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

http://www.iitg.ac.in/johnjose/

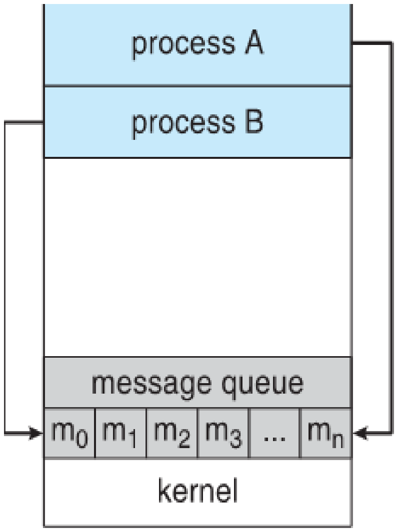
Session Outline

- Review of Inter Process Communication (IPC)
- Local Procedure Calls
- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation

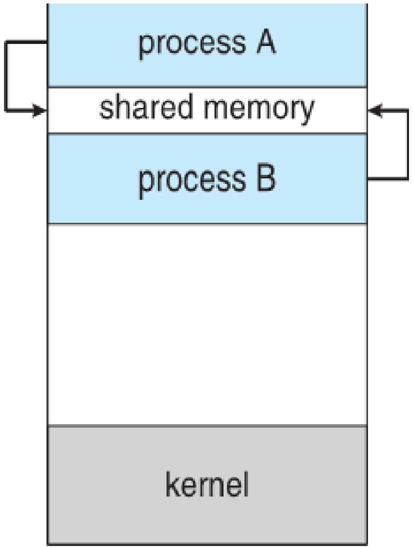
Communications Models

- ❖Cooperating processes need **interprocess communication** (**IPC**)
- ❖Two models of IPC:

Message passing

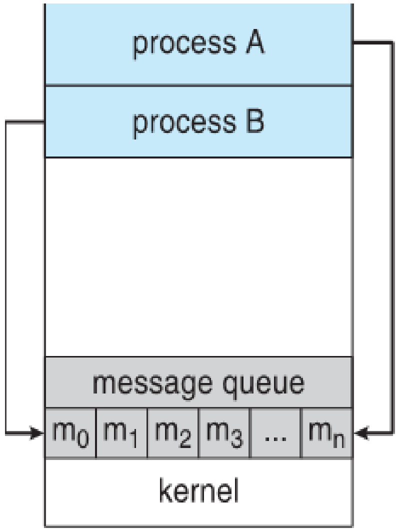


Shared memory

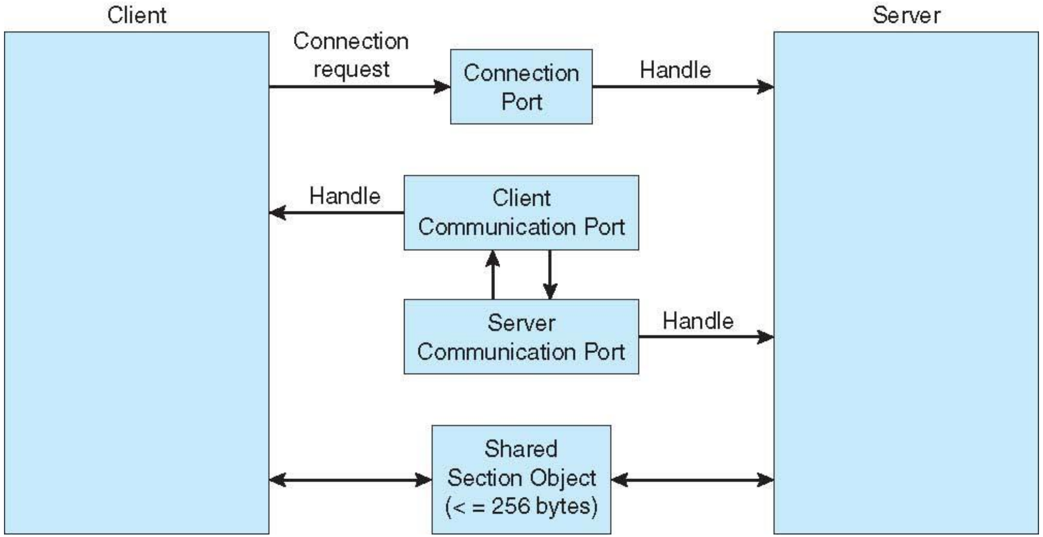


IPC – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- IPC using message passing facility provides two operations:
  - **send**(message)
  - **receive**(message)
- The message size is either fixed or variable
- Design Issues
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering



Local Procedure Calls in Windows



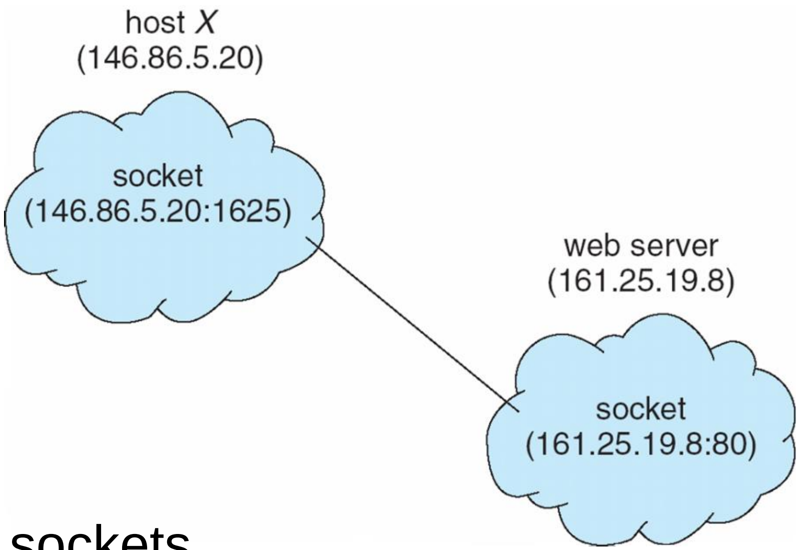
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are **well known**, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication

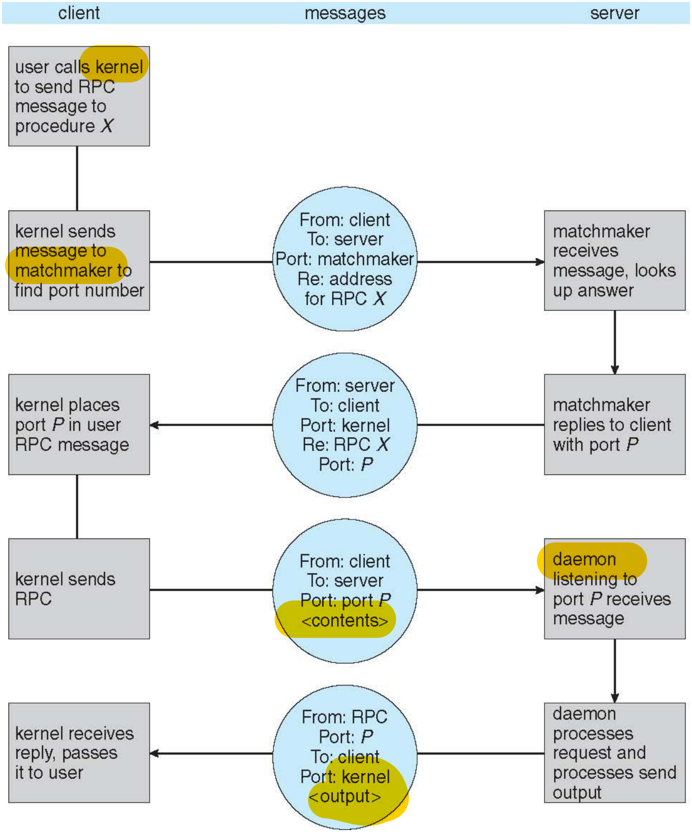


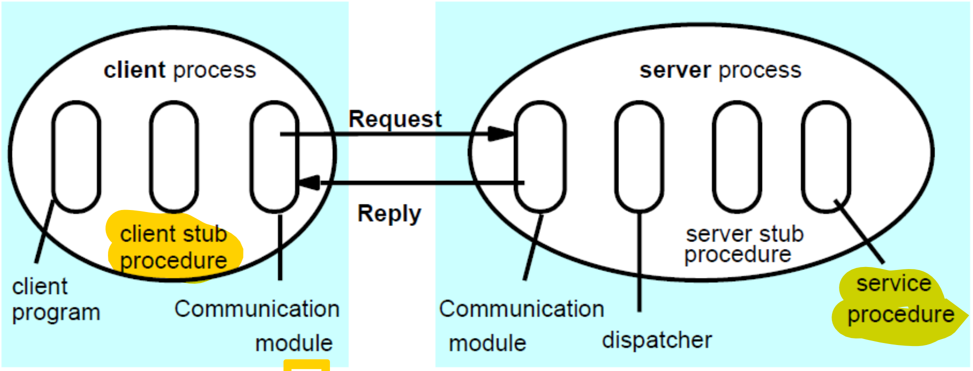
- Three types of sockets
  - **Connection-oriented (TCP)**
  - **Connectionless (UDP)**
  - **MulticastSocket** class– data can be sent to multiple recipients

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- RPC uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

Execution of RPC



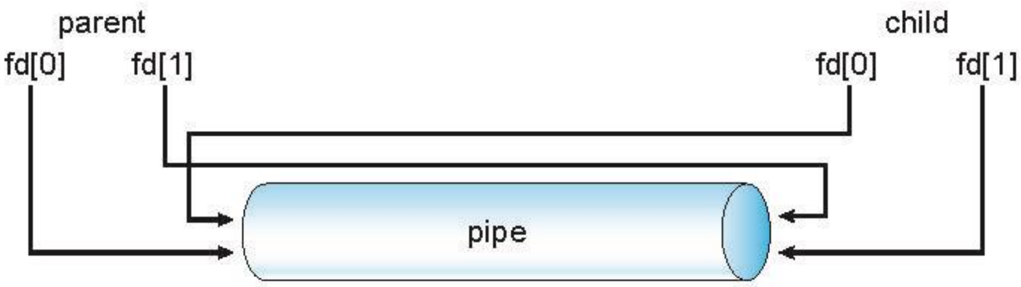


Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes

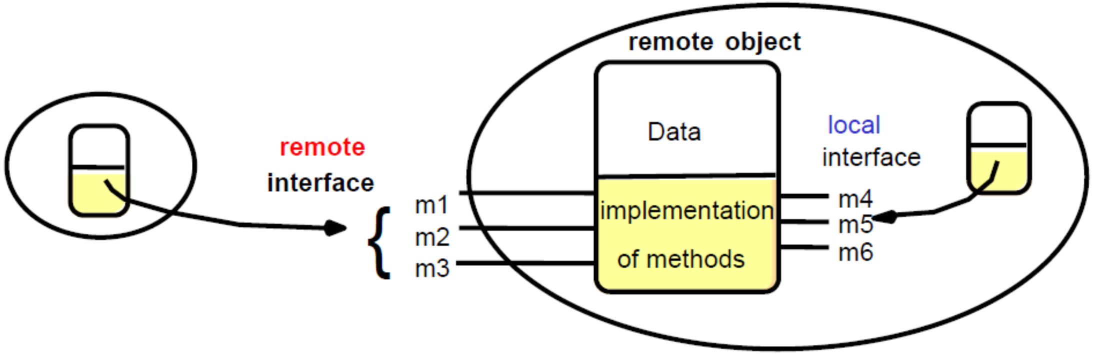


Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

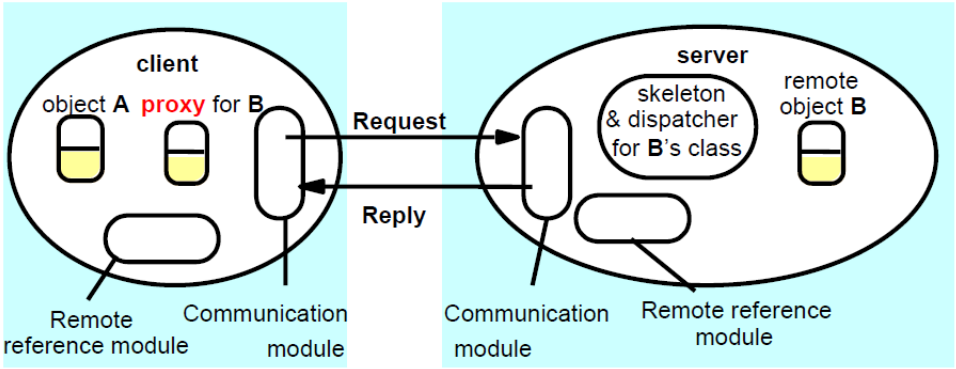
Remote Method Invocation

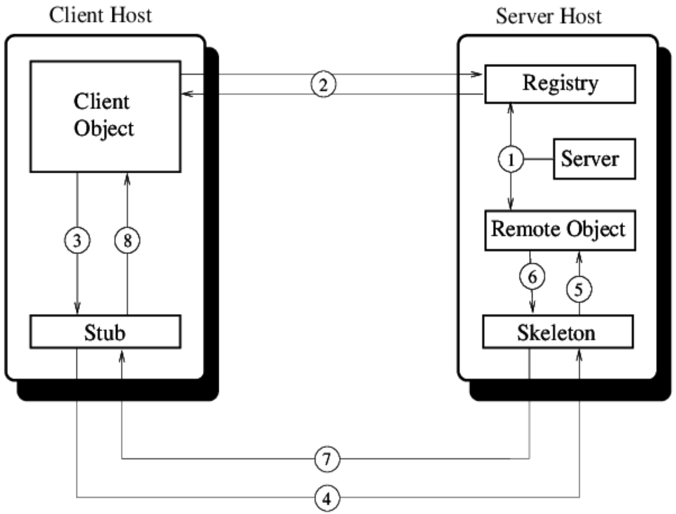
- RMI is a Java feature similar to RPCs.
- Allows a thread to invoke a method on a remote machine.
- RMI can be between two methods under two JVMs in the same machine.



Remote Method Invocation

- Client
- Stub and skeletons
- Parcel – remote method + marshalled parameters
- RMI registry





Thank you

johnjose@iitg.ac.in
http://www.iitg.ac.in/johnjose/

CS343 - Operating Systems

Module-3C

Process Synchronization – Critical Sections

Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

http://www.iitg.ac.in/johnjose/

Session Outline

- Background
- The Critical-Section Problem
- Peterson′s Solution
- Synchronization Hardware
- Mutex Locks

Objectives of Process Synchronization

- To introduce the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:

  Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers. We can do so by having an integer `counter` that keeps track of the number of full buffers. Initially, `counter` is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Bounded-Buffer – Producer & Consumer

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

Producer

```
while (true) {
    /* produce an item
    in next produced */

    while (counter == BUFFER_SIZE)
    ;        /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;

    counter++;

}
```

Consumer

```
while (true) {

    while (counter == 0)

        ; /* do nothing */

    next_consumed = buffer[out];

    out = (out + 1) % BUFFER_SIZE;

    counter--;

    /* consume the item in next
    consumed */

}
```

Race Condition

- **counter++** could be implemented as

  register1 = counter

  register1 = register1 + 1

  counter = register1

- **counter--** could be implemented as

  register2 = counter

  register2 = register2 - 1

  counter = register2

- Consider this execution interleaving with count = 5 initially:

  S0: producer execute **register1 = counter** {register1 = 5}
  S1: producer execute **register1 = register1 + 1** {register1 = 6}
  S2: consumer execute **register2 = counter** {register2 = 5}
  S3: consumer execute **register2 = register2 – 1** {register2 = 4}
  S4: producer execute **counter = register1** {counter = 6 }
  S5: consumer execute **counter = register2** {counter = 4}

Critical Section Problem

- Consider system of **n** processes {$\mathbf{p_0, p_1, \ldots\ p_{n-1}}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this

Critical Section

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

- General structure of process **P**

```
do {
    [entry section]
        critical section
    [exit section]
        remainder section
} while (true);
```

```
do {
 while (turn == j);
    critical section
 turn = j;
    remainder section
 } while (true);
```

Solution to Critical-Section Problem

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 section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

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

Peterson's Solution

- Applicable for two process solution
- Assume that the **load** and **store** machine-language 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 $\mathbf{P_i}$ is ready!

Peterson's Solution

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j]&&turn==j);

    critical section

    flag[i] = false;

    remainder section

  } while (true);
```

Algorithm for Process P_j

```
do {
    flag[j] = true;
    turn = i;
    while (flag[i]&&turn==i);

    critical section

    flag[j] = false;

    remainder section

  } while (true);
```

Peterson's Solution

- All three CS requirement are met:

1. Mutual exclusion is preserved

   P_i enters CS only if:

   either **flag[j] = false** or **turn = i**

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j]&&turn==j);

    critical section

    flag[i] = false;

    remainder section

  } while (true);
```

Synchronization Hardware - Locks

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

```
do {
 acquire lock
     critical section
 release lock
     remainder section
} while (TRUE);
```

Synchronization Using test_and_set Instruction

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE

```
do{
   while (test_and_set(&lock))
         ; /* do nothing */
         /* critical section */
         lock = false;
         /* remainder section */
 }while (true);
```

```
boolean test_and_set
 (boolean *lock)
{
   boolean rv = *lock;
   *lock = TRUE;
   return rv:
 }
```

Synchronization Using compare_and_swap Instruction

```
int compare_and_swap(int *value, int expected, int new_value)
 {   int temp = *value;
     if (*value == expected)
         *value = new_value;
     return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” to “new_value” only if “value” ==“expected”. That is, the swap takes place only under this condition.

Solution using compare_and_swap ()

- Shared integer lock initialized to 0;

```
do {
        while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */
         /* critical section */
        lock = 0;
         /* remainder section */
   } while (true);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Synchronization Using acquire() and release()

```
acquire()
 {
    while (!available)
       ; /* busy wait */
    available = false;;
 }
release()
 {
    available = true;
 }
```

```
do
 {
    acquire lock
       critical section
    release lock
      remainder section
 } while (true);
```

Thank you

johnjose@iitg.ac.in
http://www.iitg.ac.in/johnjose/

CS343 - Operating Systems

Module-3D

Process Synchronization – Semaphores & Monitors

Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

http://www.iitg.ac.in/johnjose/

Session Outline

- The Critical-Section Problem
- Semaphores
- Monitors
- Implementation of Semaphores and Monitors

Objectives of Process Synchronization

- To introduce the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- **To present both software and hardware solutions of the critical-section problem**
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Critical Section

- Each process must ask permission to enter **critical section** in **entry section**, may follow **critical section** with **exit section**, then **remainder section**

- General structure of process **P**

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

```
do {
 while (turn == j);
     critical section
 turn = j;
     remainder section
 } while (true);
```

Mutual Exclusion :: Progress :: Bounded Waiting

Semaphore

- Synchronization tool for processes to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations

```
wait(S)
{   while (S <= 0)
      ; // busy wait
      S--;
}
```

```
signal(S)
{
      S++;
}
```

Semaphore Usage

- **Binary semaphore** – value can range only between 0 and 1
  - Represents single access to a resource
- **Counting semaphore** – integer value (unrestricted range)
  - Represents a resource with N concurrent access
- Consider P_1 and P_2 that require S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
   S1;
   signal(synch);
```

```
P2:
   wait(synch);
   S2;
```

Semaphore Implementation

- With each semaphore there is an associated waiting queue
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation

- Semaphore uses two atomic operations
- Each semaphore has a queue of waiting processes
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- When signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread

```
wait(S)
{   while (S <= 0)
     ;// busy wait
     S--;
}
```

```
signal(S)
{
     S++;
}
```

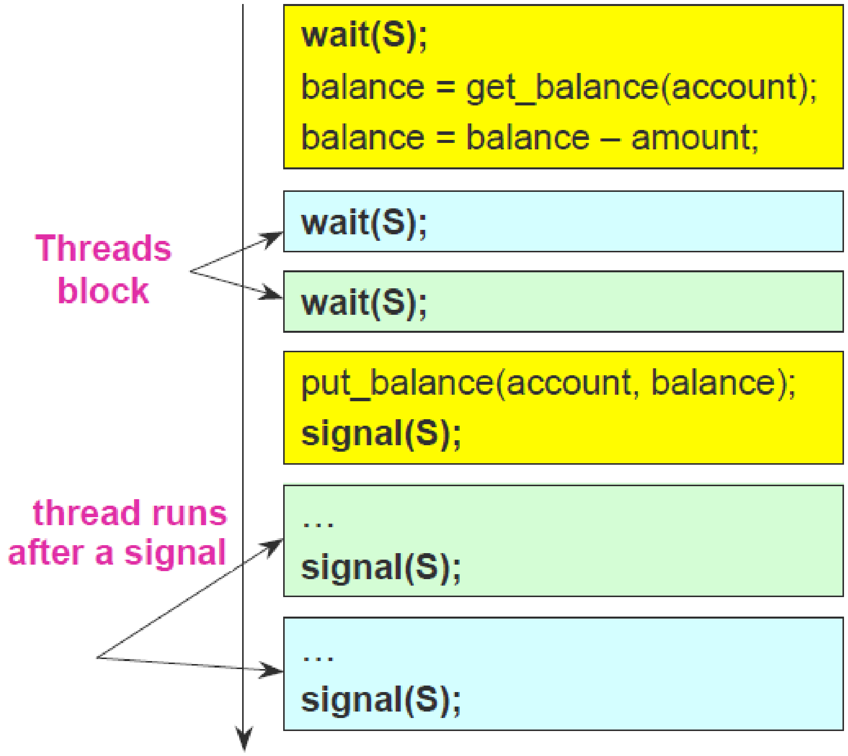
Semaphore Implementation

```
wait(semaphore *S)
{   S->value--;
    if (S->value < 0)
    {
        add this process to
        S->list;
        block();
    }
}
```

```
signal(semaphore *S)
{   S->value++;
    if (S->value <= 0)
    {
        remove a process P
        from S->list;
        wakeup(P);
    }
}
```

Semaphore Implementation

```
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```



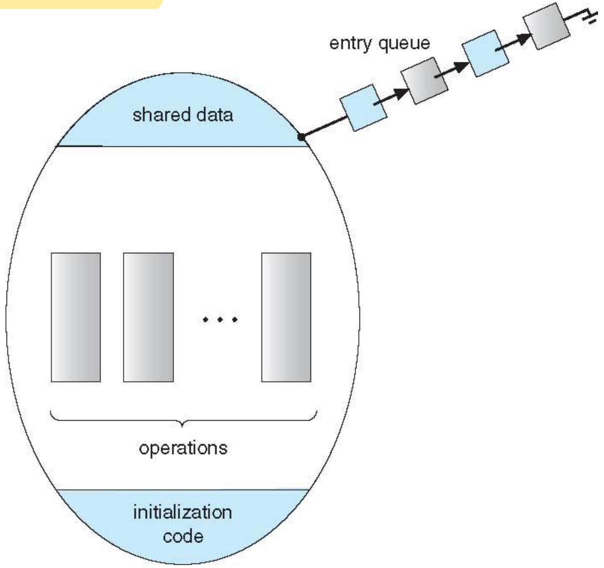
Monitors

- A monitor is a programming language construct that controls access to shared data
- Synchronization code added by compiler, enforced at runtime
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent procedure invocations
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways

Monitors

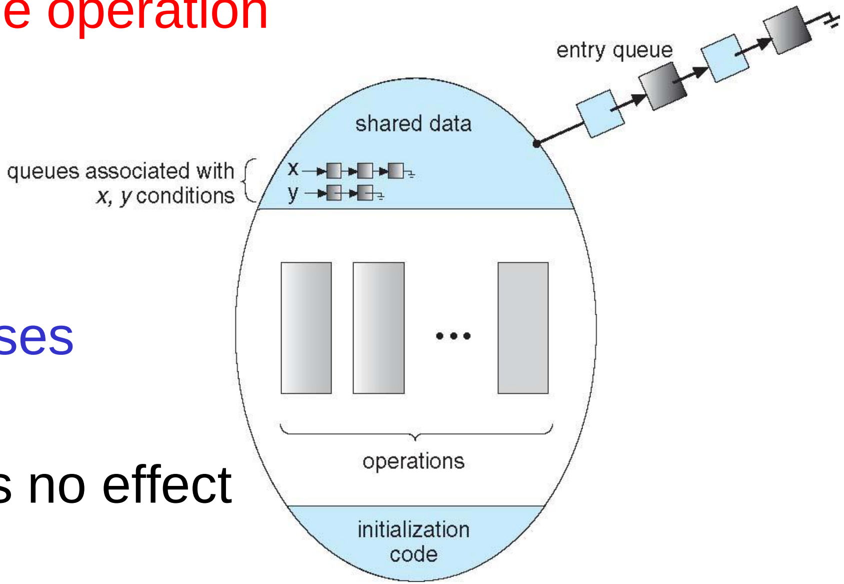
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- One process may be active within the monitor at a time

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



Condition Variables

- Two operations are allowed on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
- If no **x.wait()** on the variable, then it has no effect on the variable

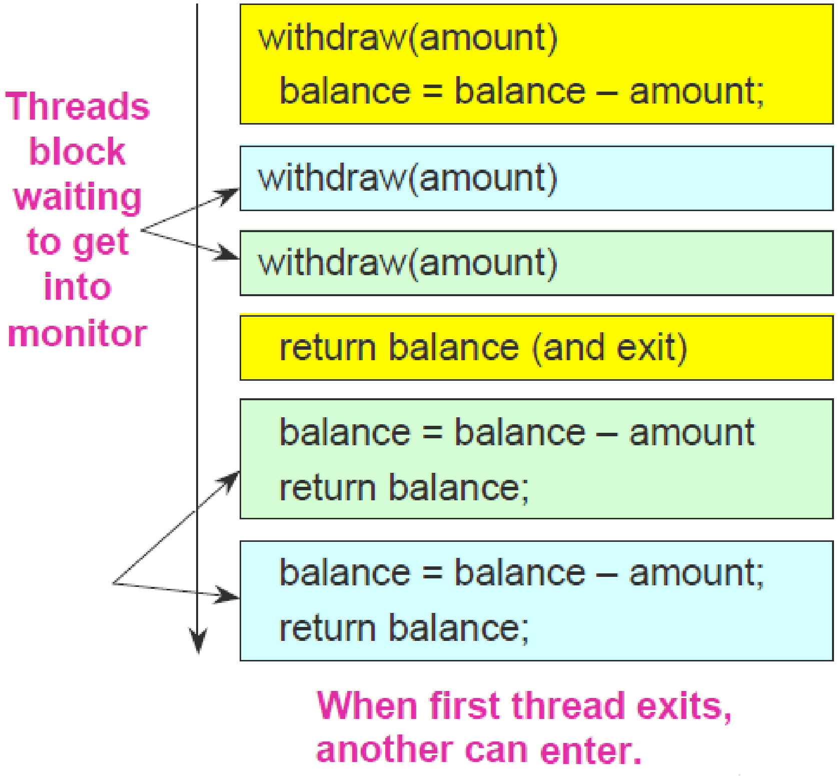


Condition Variables Choices

- If process P invokes **x.signal(),** and process Q is suspended in **x.wait()**, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition

Implementation using Monitors

```
Monitor account {
  double balance;

  double withdraw(amount) {
    balance = balance – amount;
    return balance;
  }
}
```



Thank you

johnjose@iitg.ac.in
http://www.iitg.ac.in/johnjose/

CS343 - Operating Systems

Module-3E

Classical Synchronization Problems

Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

http://www.iitg.ac.in/johnjose/

Session Outline

- ❖ Deadlock and Starvation Issues
- ❖ Bounded-Buffer Problem
- ❖ Readers and Writers Problem
- ❖ Dining-Philosophers Problem

Objectives of Process Synchronization

- To introduce the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- **To examine several classical process-synchronization problems**
- To explore several tools that are used to solve process synchronization problems

Deadlock and Starvation

- **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	P_1
wait(S);	**wait(Q);**
wait(Q);	**wait(S);**
...	**...**
signal(S);	**signal(Q);**
signal(Q);	**signal(S);**

```
wait(S)
{  while (S <= 0)
     ; // busy wait
     S--;
}
signal(S)
{  S++;
}
```

Deadlock and Starvation

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

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

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

mutex (1), full (0), empty (n)

Producer process

```
do {
    /* produce an item in */
    wait(empty);
    wait(mutex);
    /* add item to the buffer */
    signal(mutex);
    signal(full);
} while (true);
```

Consumer process

```
do {
    wait(full);
    wait(mutex);
    /* remove an item from buffer */
    signal(mutex);
    signal(empty);
    /* consume the item */
} while (true);
```

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
- 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 **rw_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read_count** initialized to 0

Readers-Writers Problem

First Readers Writers Problem

Second Reader Writer Problem

Writer process

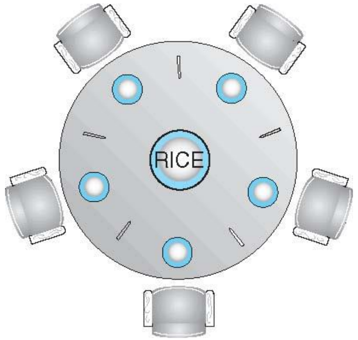
```
do {
     wait(rw_mutex);

      /* writing is performed */

     signal(rw_mutex);

    } while (true);
```

Reader process

```
do {

        wait(mutex);
        read_count++;
        if (read_count == 1)
                wait(rw_mutex);
        signal(mutex);
        /* reading is performed */
        wait(mutex);
        read count--;
        if (read_count == 0)
                signal(rw_mutex);
        signal(mutex);

} while (true);
```

Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem Algorithm

- The structure of Philosopher i:

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

What the limitations of this approach?

Dining-Philosophers Problem Algorithm contd..

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.)
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Monitor Solution to Dining Philosophers

```
monitor Dining Philosophers
{
    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.)

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

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

Monitor Implementation Using Semaphores

- Variables

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

- Mutual exclusion within a monitor is ensured

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

```
wait (mutex);
…
body of F;
…
if (next_count > 0)
signal (next);
else
signal (mutex);
```

Monitor Implementation – Condition Variables

For each condition variable ***x***,

```
semaphore x_sem; // (initially=0)

int x_count = 0;
```

x.signal

```
if (x_count > 0)
 { next_count++;
   signal(x_sem);
   wait(next);
   next_count--; }
```

x.wait

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

Thank you

johnjose@iitg.ac.in
http://www.iitg.ac.in/johnjose/

CS343 - Operating Systems

Module-3F
Introduction to Deadlocks

Dr. John Jose
Assistant Professor
Department of Computer Science & Engineering
Indian Institute of Technology Guwahati, Assam.
http://www.iitg.ac.in/johnjose/

Session Outline

- ❖System Model
- ❖Deadlock Characterization
- ❖Resource Allocation Graph
- ❖Methods for Handling Deadlocks
- ❖Deadlock Prevention

Objectives of Deadlock Management Unit

- 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 or avoiding deadlocks in a computer system

System Model

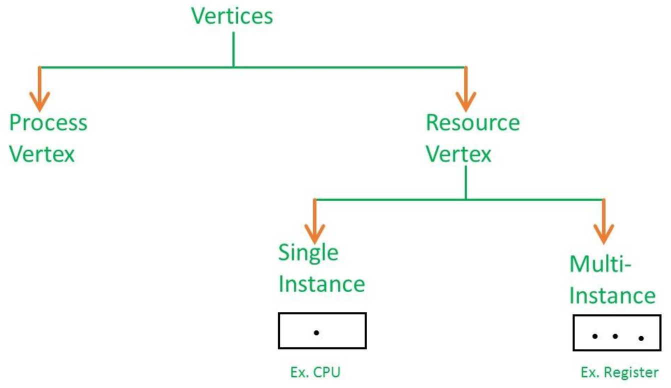
- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
    - *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
    - **request**
    - **use**
    - **release**

Deadlock Characterization

- Deadlock can arise if the following four conditions hold simultaneously.
- **Mutual exclusion**: Only one process at a time can use a resource
- **Hold and wait**: A process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: A resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, 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.

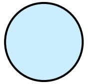
Resource-Allocation Graph

- 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 active processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

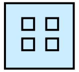


Resource-Allocation Graph

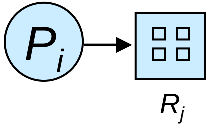
- Process



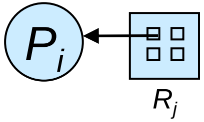
- Resource Type with 4 instances

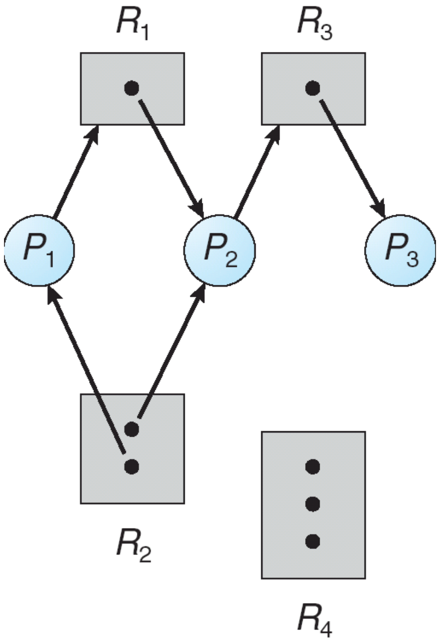


- P_i requests an instance of R_j

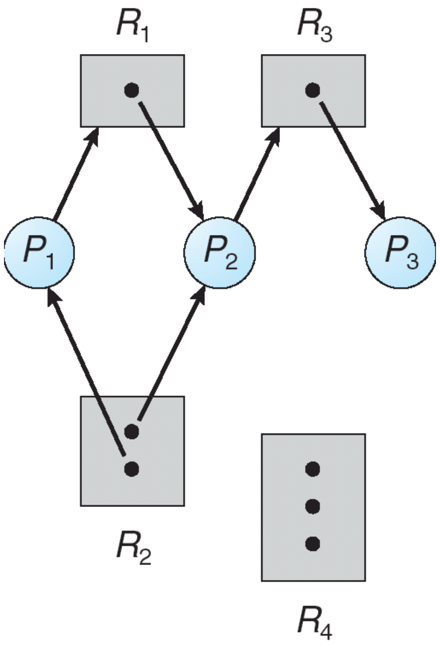


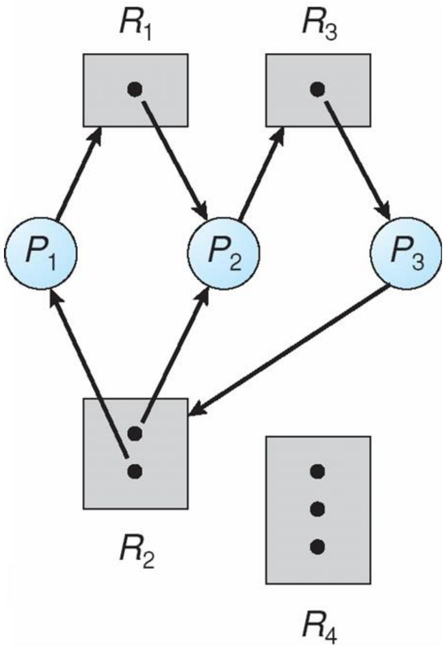
- P_i is holding an instance of R_j



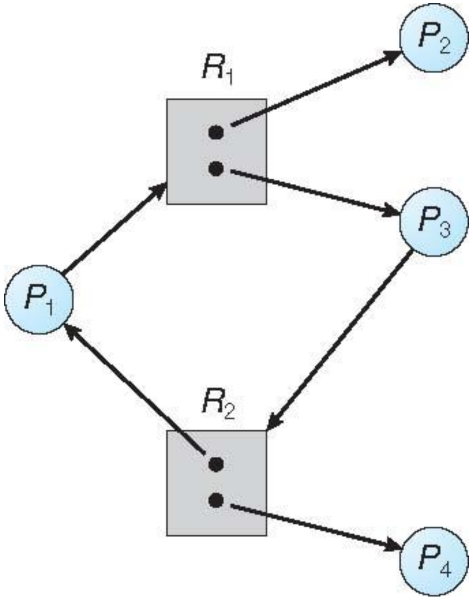


Resource-Allocation Graph





RAG with a deadlock

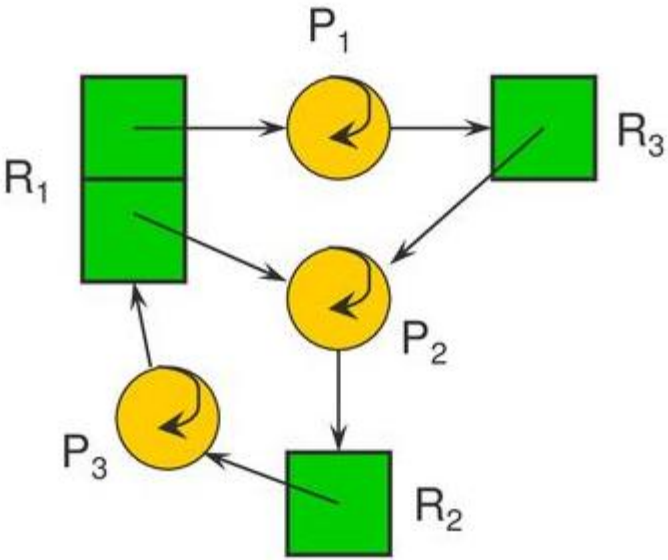


RAG without a deadlock

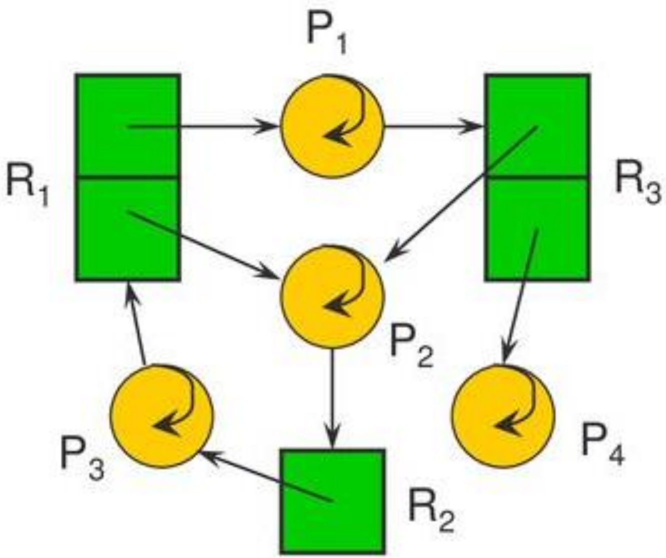
Deadlock detection in RAG

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Deadlock detection in RAG



A cycle...and deadlock!



Same cycle...but no deadlock. Why?

Methods for Handling Deadlocks

- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX
- Ensure that the system will **never** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover

Deadlock Prevention

- **Deadlock prevention is done by ensuring that at least one of the necessary 4 conditions for deadlock is not met.**
- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable 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 allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention

- **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 Example

```c
/* thread one runs in this function */

void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);

   /** * Do some work */

   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
```

```c
/* thread two runs in this function */

void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);

   /** * Do some work */

   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Thank you

johnjose@iitg.ac.in

http://www.iitg.ac.in/johnjose/

CS343 - Operating Systems

Module-3G

Deadlocks Avoidance, Detection & Recovery

Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

http://www.iitg.ac.in/johnjose/

Overview of Deadlock Management Section

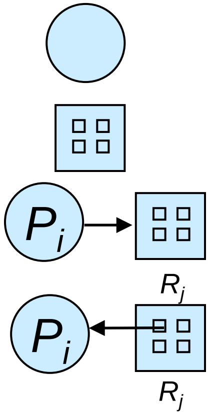
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

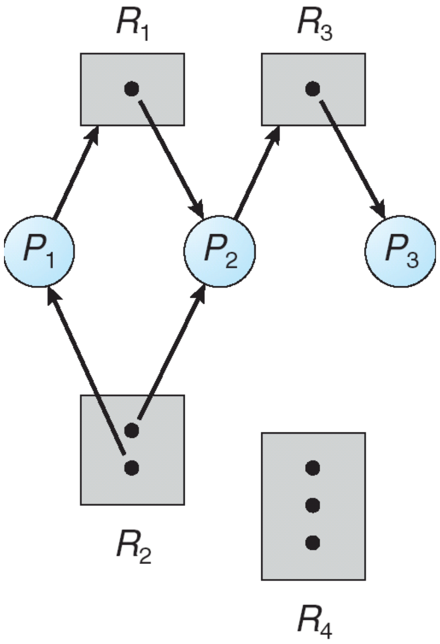
Deadlock Characterization

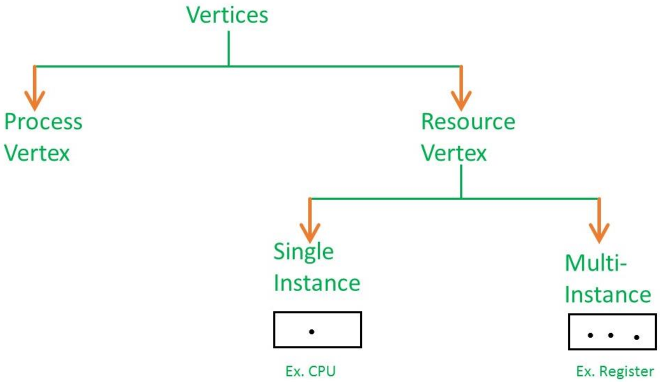
- Deadlock can arise if the following four conditions hold simultaneously.
- **Mutual exclusion**: Only one process at a time can use a resource
- **Hold and wait**: A process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: A resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, 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

- Process
- Resource Type with 4 instances
- P_i requests an instance of R_j
- P_i is holding an instance of R_j







Deadlock Avoidance

- Requires that the system has some additional ***a priori*** information available
- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

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

- System is in **safe state** if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in 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 $j < I$

- That is:

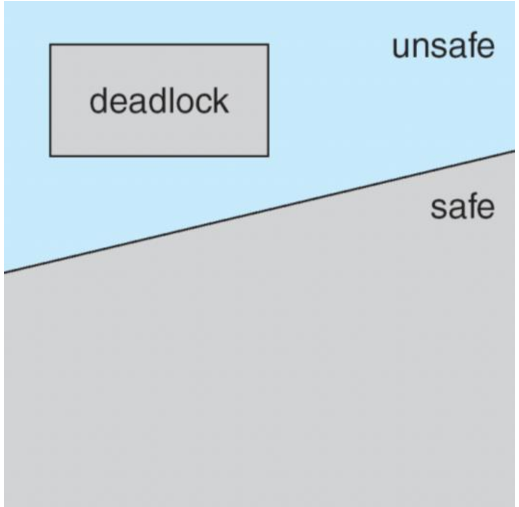
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished

  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate

  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Safe State & Deadlock

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.



Avoidance Algorithms

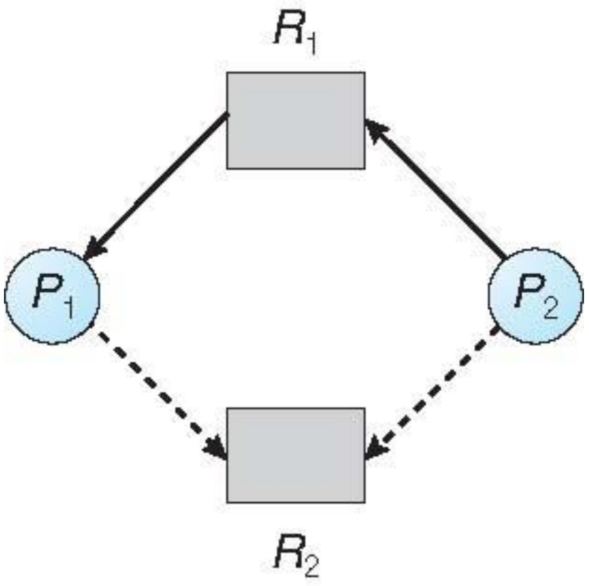
- Single instance of a resource type
  - Use a resource-allocation graph

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

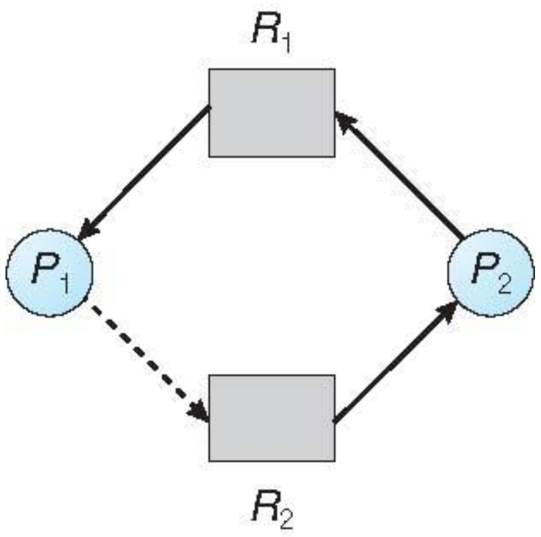
Resource-Allocation Graph Scheme

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

Resource-Allocation Graph & Unsafe State



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

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

Safety Algorithm

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

    Initialize: **Work = Available**

    Finish [i] = false for i = 0, 1, …, n- 1

2. Find an **i** such that both:

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

    (b) **$Need_i \leq$ Work**

    If no such **i** exists, go to step 4

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

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

Resource-Request Algorithm for Process P_i

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

1. If **$Request_i$** $\leq$ **$Need_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If **$Request_i$** $\leq$ **Available**, go to step 3. Otherwise **P_i** must wait, since resources are not available
3. Pretend to allocate requested resources to **P_i** by modifying the states

$$\mathbf{Available = Available - Request_i;}$$

$$\mathbf{Allocation_i = Allocation_i + Request_i;}$$

$$\mathbf{Need_i = Need_i - Request_i;}$$

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

Example of Banker's Algorithm

- 5 [P_0 -P_4] & 3 resource types: A (10), B (5), and C (7)
- Snapshot at time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example of Banker's Algorithm contd…

- The content of the matrix **Need** is defined to be **Max – Allocation**

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < $\mathbf{P_1, P_3, P_4, P_0, P_2}$> satisfies safety requirement

Deadlock Detection

- Allow system to enter deadlock state

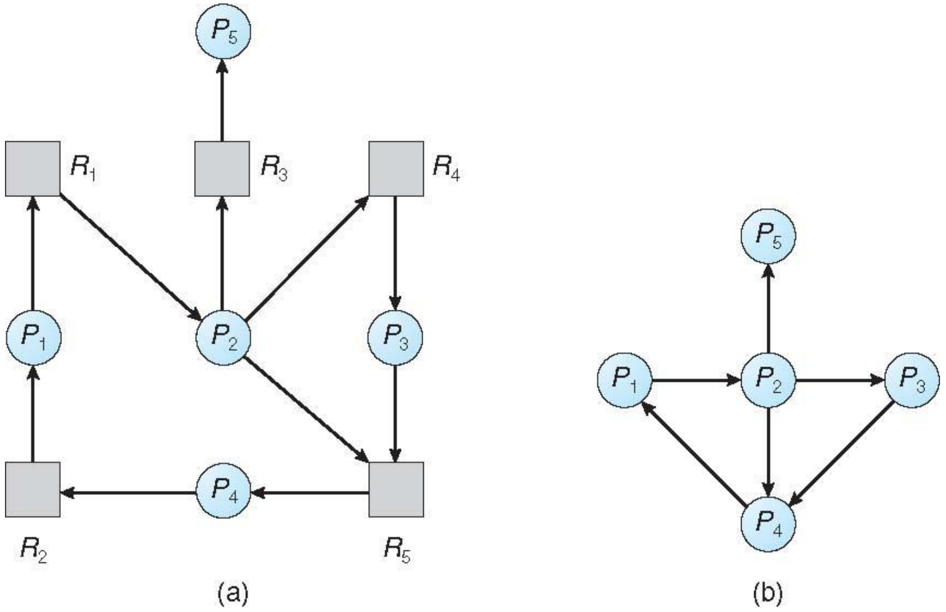
- Detection algorithm

- Recovery scheme

Detection in Single Instance Resource Types

- Maintain wait-for graph
  - Nodes are processes
  - $\mathbf{P_i} \rightarrow \mathbf{P_j}$ if $\mathbf{P_i}$ is waiting for $\mathbf{P_j}$
- 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 $\mathbf{n^2}$ 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

Several Instances of a Resource Type

- **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 $\mathbf{P_i}$ is requesting **k** more instances of resource type $\mathbf{R_j}$.

Detection Algorithm

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

    (a) **Work = Available**

    (b) For **i = 1,2, …, n**, if $\mathbf{Allocation_i \neq 0}$, then **Finish[i] = false**; otherwise, **Finish[i] = true**

2. Find an index **i** such that both:

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

    (b) $\mathbf{Request_i \leq Work}$

    If no such **i** exists, go to step 4

Detection Algorithm contd..

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

4. If **Finish[i] == false**, for some **i**, $1 \leq$ **i** $\leq$ **n**, then the system is in deadlock state. Moreover, if **Finish[i] == false**, then **P$_i$** is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes $\mathbf{P_0}$ through $\mathbf{P_4}$; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time $\mathbf{T_0}$:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Example of Detection Algorithm contd..

- $\boldsymbol{P_2}$ requests an additional instance of type ***C***

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system? :
- Can reclaim resources held by process $\boldsymbol{P_0}$, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes caused the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion?
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated?
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Thank you

johnjose@iitg.ac.in

http://www.iitg.ac.in/johnjose/