Process Synchronization (Topic 2-2)

홍 성 수

서울대학교 공과대학 전기 공학부
Real-Time Operating Systems Laboratory

Task Interactions (1)

- Why? - Independent processes:
  - One that can't affect or be affected by the rest of the universe
    - E.g., Processes running on different non-networked machines
  - Properties:
    - No shared state between processes
    - Deterministic: Input state alone determines results
    - Reproducible: Can stop and restart with no bad effects

Task Interactions (2)

- Why? - Cooperating processes:
  - Non-independent processes.
  - Processes that share state: not necessarily "cooperating"
    - E.g., Processes share a single file system
  - Properties:
    - Behavior is non-deterministic, maybe irreproducible

Task Interactions (3)

- Why permit processes to cooperate?

(1) Want to share resources:

One computer, many users.

One checking account file, many tellers.

(2) Want to do things faster:

Read next block while processing current one.

Divide jobs into sub-jobs, execute in parallel.

(3) Want to construct systems in modular fashion:

Unix example: tbl | eqn | troff

Case of Task Interaction: Data Sharing Problem (1)

- Interrupt routines and task code may share one or more variables that they can use to communicate with each other.
- This may cause a data sharing problem – a sort of synchronization problem.
  - Such a task is referred to as *non-reentrant code*
- Example: nuclear reactor monitoring system

Case of Task Interaction: Data Sharing Problem (2)

```
static int iTemperatures[2];

void interrupt vReadTemperatures (void) {
   iTemperatures[0] = !! Read in value from hardware
   iTemperatures[1] = !! Read in value from hardware
}
void main (void){
   int iTemp0, iTemp;

   while(TRUE){
        iTemp0 = iTemperatures[0];
        iTemp1 = iTemperatures[1];
        if (iTemp0 != iTemp1)
               !! set off howling alarm;
   }
}
```


Case of Task Interaction: Data Sharing Problem (3)

```
static int iTemperatures[2];

void interrupt vReadTemperatures (void) {
   iTemperatures[0] = !! Read in value from hardware
   iTemperatures[1] = !! Read in value from hardware
}
void main (void){
   int iTemp0, iTemp;

   while(TRUE){
        iTemp0 = iTemperatures[0];
        iTemp1 = iTemperatures[1];
        if (iTemp0 != iTemp1)
               !! set off howling alarm;
   }
}
```

If interrupt occurs between these two statements, iTemp0 and iTemp1 will differ and the system will set off the alarm, *even though the two measured temperatures were always the same*.

Case of Task Interaction: Data Sharing Problem (4)

```
static int iTemperatures[2];

void interrupt vReadTemperatures (void){
   iTemperatures[0] = !! Read in value from hardware
   iTemperatures[1] = !! Read in value from hardware
}

void main (void){
   while (TRUE){
        if (iTemperature[0] != iTemperatures[1])
               !! set off howling alarm;
   }
}
```

The same bug in previous page !

The problem is that the statement that compares iTemperatures[0] with iTemperatures[1] can be interrupted.

Case of Task Interaction: Data Sharing Problem (5)

- Solution
  - Disabling Interrupts
- In the previous code

```
void main (void){
   int iTemp0, iTemp;

   while(TRUE){
        disable(); /*Disable interrupts using array*/
        iTemp0 = iTemperatures[0];
        iTemp1 = iTemperatures[1];
        enable();
        if (iTemp0 != iTemp1)
               !! set off howling alarm;
   }
}
```


Obtaining Reentrancy (1)

- To handle cooperating processes, we needs the notion of atomic operations
  - Atomic means either it happens in its entirety or not at all.
  - The operation can not be interrupted in the middle.

    Examples:

    int A,B; A = B; // On most systems.

    (1) On uniprocessors, code between interrupts.

    (2) Test-and-set instruction in some architectures.

Obtaining Reentrancy (2)

- Need hardware support to provide atomic operations.
- Synchronization: Using atomic operations to ensure correct operation of cooperating processes.
- Example of cooperating processes that need synchronization:

  Two processes execute the following code:

```
if(BufferIsAvail) {
        BufferIsAvail = FALSE;
        UseBuffer();
        BufferIsAvail = TRUE;
}
```


Obtaining Reentrancy (3)

Time	Proc 1	Proc 2
0	if(BufferIsAvail)	
1		if(BufferIsAvail)
2	BufferIsAvail = FALSE	
3		BufferIsAvail = FALSE
4	UseBuffer();	
5		UseBuffer();

Problem : Both processes issue UseBuffer().

Obtaining Reentrancy (4)

- Lack of atomicity of “if” and “assignment.”
- Mutual exclusion:
  - Mechanisms that ensure that only one person or process is doing certain things at one time.
- Critical section:
  - A section of code, or collection of operations, in which only one process may be executing at a time.

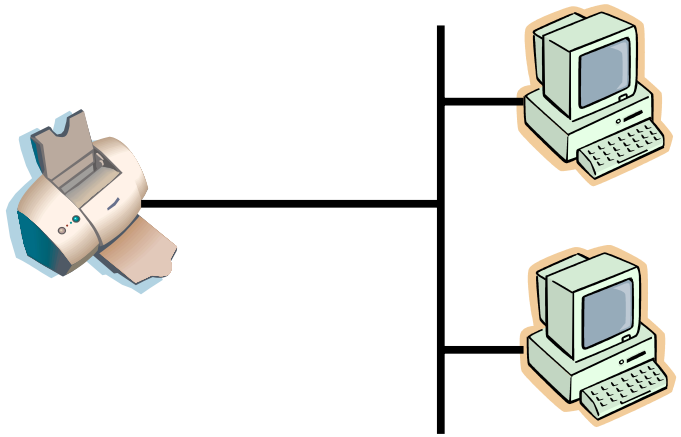
Obtaining Reentrancy (5)

- Requirements for a mutual exclusion mechanism.
  1) Only one process is allowed in a critical section at a time.
  2) If several requests at once, must allow one process to proceed.
  3) Must not depend on processes outside critical section.

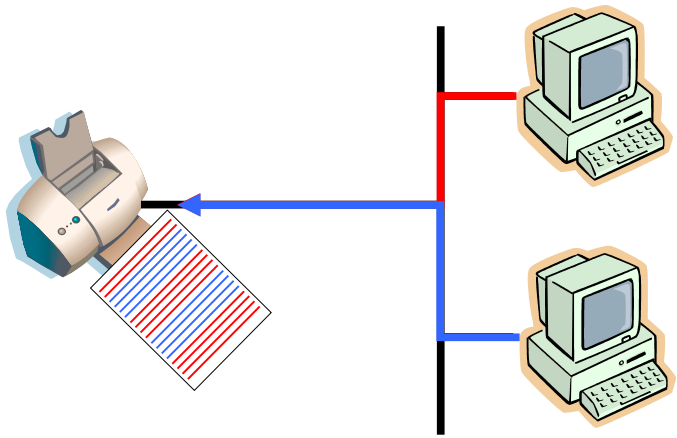
Obtaining Reentrancy (6)

- Desirable properties for a mutual exclusion mechanism:
  1) Don’t make a process wait forever.
  2) Efficient: Don’t use up substantial amounts of resources when waiting. (E.g., busy waiting.)
  3) Simple: Should be easy to use.

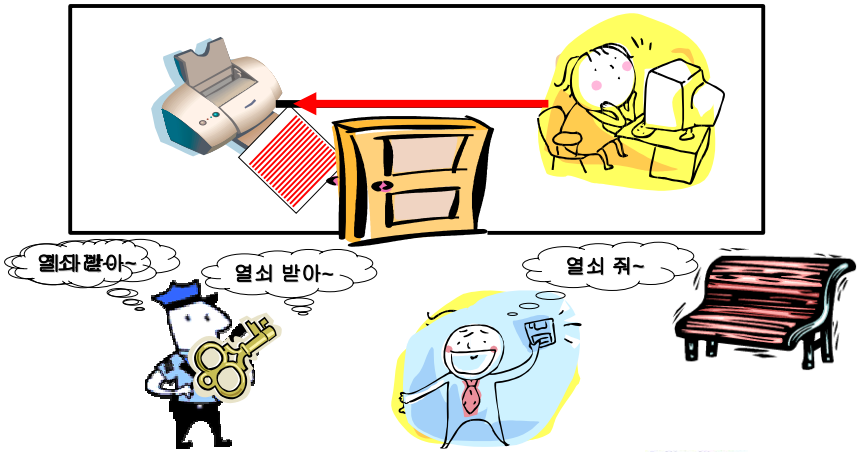
Semaphores (1): Necessity



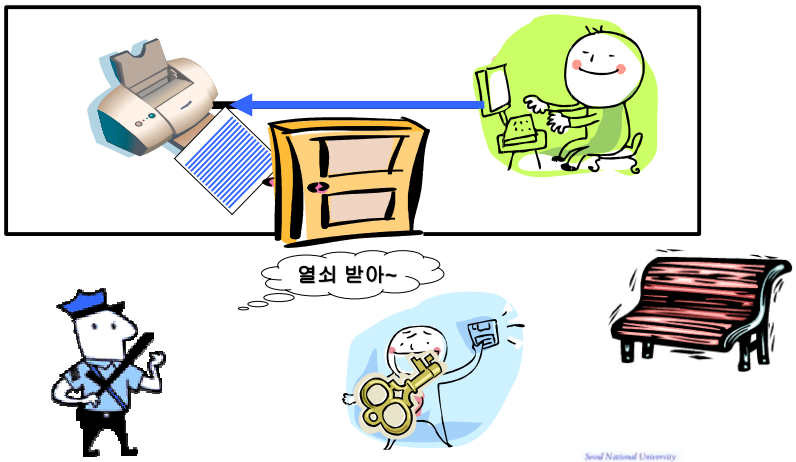
Semaphores (2): Necessity



Semaphores (3): Operation



Semaphores (3): Operation



Semaphores (4): Basics

- One synchronization mechanism
  - A sync. variable that takes on integer values.
  - *P(Semaphore)*: An atomic operation that waits for semaphore to become positive and then decrements it by one. (Also called *wait()*.)
  - *V(Semaphore)*: An atomic operation that increments semaphore by one. (Also called *signal()*.)
  - Semaphores are simple and elegant.
  - They do a lot more that just mutual exclusion.

Semaphores (5): Usage

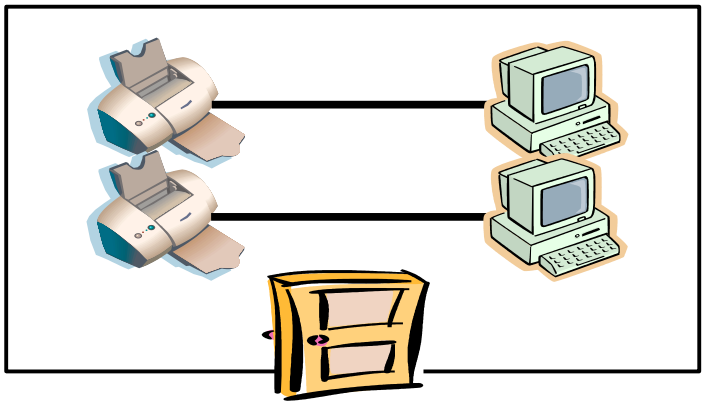
```
Task1(){
  P(S1)
     use pr;
  V(S1)
}
```

```
Task2(){
  P(S1)
     use pr;
  V(S1)
}
```

```
semaphore S = 1;
```


Semaphores (6): Initialization

- Binary semaphore vs Counting semaphore



Semaphores (6): Initialization

```
Task1(){
  P(S1)
     use pr;
  V(S1)
}
```

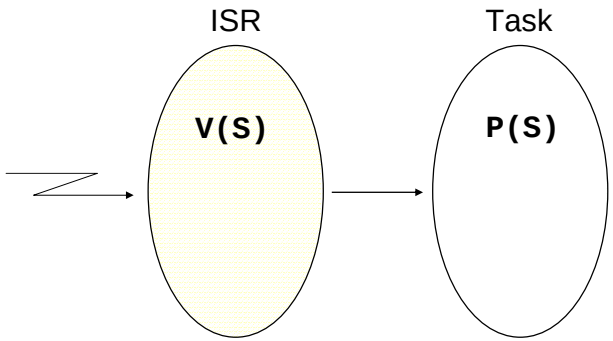
```
Task2(){
  P(S1)
     use pr;
  V(S1)
}
```

```
semaphore S = 2;
```


Semaphores (7): Basics

- Buffer example with semaphores:
  1) P(BufferIsAvail);
  2) UseBuffer();
  3) V(BufferIsAvail);
  Note: BufferIsAvail must be set to one
- What happens if BufferIsAvail is set to two? or zero?
- Uses of semaphores:
  - Mutual exclusion
  - Scheduling

Semaphore (8): Scheduling

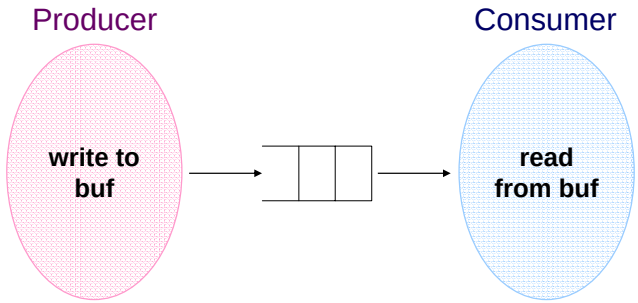



```
semaphore S = 0;
```


Semaphores (9): Producer/Consumer

- Scheduling with semaphores
  - Producer: Creates copies of a resource.
    Example: a user typing characters.
  - Consumer: Uses up (destroys) copies of a resources.
    Example: program reading users characters.
  - Buffers: Memory used to hold info after the producer has created it and before the consumer has used it.
    Allows producer to get ahead of consumer.
    Consumer and procedure don't operate in lock-step.

Semaphores (9): Producer/Consumer



Bounded buffer with blocking reads and writes

Semaphores (9): Producer/Consumer

- What is "correct" for this example?
- Constraints:
  - Consumer must wait for producer to fill buffer.
  - Producer must wait for consumer to empty buffers if all buffer space is filled.
- A separate semaphore is used for each constraint:
  - emptyBufferAvail — Initialized to numBuffers.
  - fullBufferAvail — Initialized to 0.

Semaphores (9): Producer/Consumer

```
Producer(){
  P(buf_avail)
     buf = data;
  V(data_avail)
}
```

```
Consumer(){
  P(data_avail)
     data = buf;
  V(buf_avail)
}
```

```
Semaphore   buf_avail = 1,
            data_avail = 0;
```


Semaphores (10): Disable Interrupts

```
Task1(){
  P(S1)
     use pr1;
  V(S1)
}
```

```
Task2(){
  P(S1)
     use pr1;
  V(S1)
}
```


Semaphores (10): Disable Interrupts

```
Task3(){
  P(S2)
     use pr2;
  V(S2)
}
```

```
Task1(){
  disable intr
     use pr1;
  enable intr
}
```

Turn all traffic lights in Seoul into red

Semaphores (11): Drawbacks

```
Producer(){
  P(S1)
     buf = data;
  V(S2)
}
```

```
Consumer(){
  P(S2)
     data = buf;
  V(S1)
}
```

Semaphore naming issue

Semaphores (11): Drawbacks

```
Task1(){
  P(S1)
     buf = data;
  V(S1)
}
```

```
Task2(){
  P(S2)
     data = buf;
  V(S1)
}
```

Is this a race condition or not?

Semaphores (12): Solution

```
Task1(){
  mutex_lock(S1)
     buf = data;
  mutex_unlock(S1)
}
```

```
Task2(){
  mutex_lock(S2)
     data = buf;
  mutex_unlcok(S1)
}
```

This is surely a race condition.

Semaphores (13): Implementation

- Note: V() is done when a resource is created and P() when destroyed.
- Producers and consumers are like Unix pipes:
  - Example: cat file | grep foo
- Semaphore implementation — Uniprocessors.

```
struct Semaphore {
        int value;
        Queue waitQ;
}
```

- Use disable of interrupts to get mutual exclusion.

Semaphores (13): Implementation

```
P(sema)
{
  disableInterrulpts();
  if(sema.value-- > 0) {
    enableInterrupts();
  } else {
    putMeToWaitQ(sema.waitQ);
    enableInterrupts();
    waitOn(sema.waitQ);
  }
}
```

```
V(sema)
{
  disbleInterrupts();
  if(sema.value++ >= 0) {
    enableInterrpts();
  } else {
    move1ToRdyList(sema.waitQ);
    enableInterrupts();
    yieldCPU();
  }
}
```

Semaphores (13): Implementation

- Multiprocessor solution: Mutual exclusion is harder
- Possibilities:

  (1) Turn off all other processors.

  (2) Use atomic hardware support.
  - Read-modify-write memory operations.
  - Example: test and Set (TAS) instruction

Semaphores (13): Implementation

- Common solution:
  - Change disableInterrupt(); to:

    disableInterrupt();

    while (TAS(lockMem) ! = 0) continue;
  - Change enableInterrupt(); to:

    lockMem = 0

    enableInterrupt();
- Note: Multiprocessors solution does some busy-waiting.

Semaphores (13): Implementation

- Important point: Implement some mechanism once, very carefully. Then always write programs that use that mechanism. Layering is very important.

Monitors (1)

- Monitors: High-level data abstraction tool that combines three features:

  (1) shared data.

  (2) Operations on the data.

  (3) Synchronization, scheduling.
- Monitors can be embedded in programming languages.
  - Example: Mesa/Cedar from Xerox and Java.

Monitors (2)

- Example monitor in C:

```
monitor QueueHandler:
  struct { ... } queue; //shared date.
  AddToQueue(val) { - code to add to shared queue - }
  int RemoveFromQueue() { - code to remove - }
endmonitor;
```

- One binary semaphore is associated with each monitor.
  - Implicit mutual exclusion.

Monitors (3)

- Monitors are easier to use and safer than semaphores.
  - Complier can check usage.
- Condition variables: things to wait on.
  - wait (condition): Release monitor lock, put process to sleep. Reacquire lock when waken.
  - signal (condition): Wake up one process waiting on the condition variable. If nobody waiting, do nothing.

    Note: No history in condition variables.
  - broadcast (condition): Wake up all processes waiting on the condition variable.

Monitors (4)

- There are several different variations on the wait/signal mechanism.
  - Who gets the monitor lock after a signal?
  - “Mesa semantics”;

    On signal, signaler keeps monitor lock.

    Awakened process waits for monitor lock.

    Must check again and be prepared to sleep.
- Unix internally uses a mechanism similar to wait/signal.
- Unix kernel critical sections are made by disabling interrupts:
  - “spl” calls.

Monitors (5): Usage

```
type resource_manager is monitor
    var busy: boolean; x: condition;

    procedure entry acquire()
    {
        if (busy) x.wait;
        busy = true;
    }
    procedure entry release()
    }
        busy = false;
        x.signal;
    }
    { /* Initialization */
        busy = false;
    }
```

Monitors (6): Realization

```
P(monitor_lock);
      Body of the Function;
if (sig_lock.cnt > 0) {
   V(sig_lock);
} else {
   V(monitor_lock);
}
```

```
/* x.wait*/
x_cnt++;
if (sig_lock.cnt > 0) {
   V(sig_lock);
} else {
   V(monitor_lock);
P(x_lock);
x_cnt--;
```

```
/* x.signal; */
if (x_cnt > 0) {
   sig_lock.cnt++;
   V(x_lock);
   P(sig_lock);
   sig_lock.cnt--;
}
```

monitor_lock	Semaphore for the monitor
sig_lock	Semaphore for signalers
x_lock	Semaphore for condition x

Communication with Messages (1)

- Up until now, discussion has been about communication with shared date.
  - Messages allow communication without shared data.
  - Only one process “owns” the data at a time.
- Definitions:
  - Message: A piece of information that is passed from one process to another.
  - Mailbox: A place where messages are stored between the time they are sent and the time they are received.

Communication with Messages (2)

- Operations on messages:
  - Send: Copy a message into mailbox.
  - Receive: Copy a message out of mailbox.
- Two general styles of message communication:
  - 1-way: Message flow back-and-forth.

    Example: Unix pipes, producer/consumer.
  - 2-way: Message flow back-and-forth.

    Example: Remote procedure call, client/server.

Communication with Messages (3)

- Producer/Consumer example:

```
Producer:
int msg1[1000];
{
   - - prepare msg 1 - -
   send(msg1, mbox);
}
```

```
Consumer:
int msg2[1000];
while(TRUE)
{
   recive(msg2, mbox);
   - - process msg2
}
```

Note: Buffer recycling is implicit.

Communication with Messages (4)

- Client/Server example:

```
Client:
char response[1000];

while(TRUE)
{
   send{"A CMD", mbox1);
   received(response, mbox2);
}
```

```
Server:
char command[100];
char answer[1000];

while(TRUE)
{
   receive(command, mbox1);
   - - decode command - -
   - - fill in answer
   send(answer, mbox2);
}
```

A lot like a procedure call & return

Communication with Messages (5)

- Why use messages?
  - Many applications fit into this model of processing a sequential flow of information.
    Example: Unix pipes.
  - Communication parties can be separate, except for the mailbox:
    Less error-prone: No invisible side effects.
    Parties might not trust each other (OS vs. user).
    Parties might be written at different times by different programmers.
    Parties might be running on different machines on a network.

Communication with Messages (6)

- Message systems vary along several dimensions.
- Relationship between mailboxes and processes:
  (1) One mailbox per process, uses process name in send, no name in receive (simple but restrictive) [RC4000, V].
  (2) No strict mailbox-process association, uses mailbox name (can have multiple mailboxes per process; can pass mailboxes from process to process; but trickier to implement) [Unix].

Communication with Messages (7)

(3) Extent of buffering:
  - Buffering (more efficient for large transfers when sender and receiver run at varying speeds).
  - None – rendezvous protocols (simple; OK for call-return type communication; knows that message was received).

- Blocking vs. non-blocking ops:
  - Blocking receive: Return message if mailbox isn't empty; otherwise, wait until message arrives.
  - Non-blocking receive: Return message if mailbox isn't empty; otherwise, return special "empty" value.

Communication with Messages (8)

  - Blocking send: Wait until mailbox has space.
  - Non-blocking send: Return “full” if no space in mailbox.
- Additional forms of waiting:
  - Almost all systems allow many processes to wait on the same mailbox at the same time. Messages get passed to processes in order.
  - A few systems allow a process to wait on several mailboxes at once (e.g. select in UNIX). The process gets the first message to arrive on any of the mailboxes.

    Useful for network services, window systems, etc.

Communication with Messages (9)

- Constraints on what get passed in messages:
  - None: Just a stream of bytes (Unix pipes).
  - Enforce message boundaries (send and receive in same chunks).
  - Protected objects (e.g. process id of sender, or a token for a mailbox).

Communication with Messages (10)

- Messages and shared-data approaches are equally powerful, but result in very different-looking styles of programming.

  Most people find shared-data approach easier to work with.
- We will revisit this material when we talk about networks.

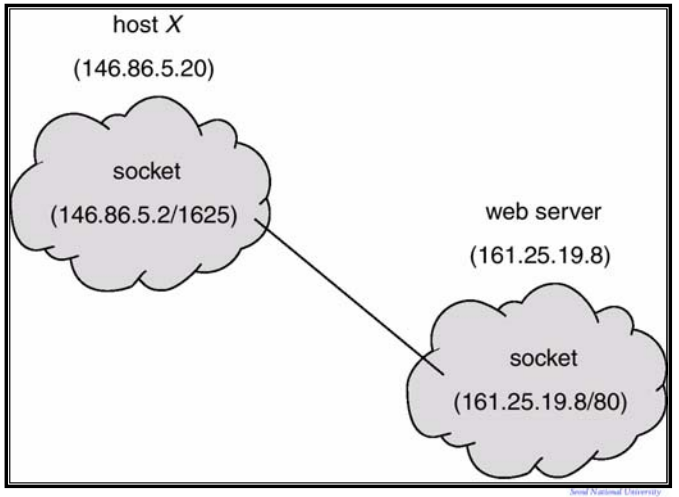
Client-Server Communication

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an *endpoint for communication*.
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets.

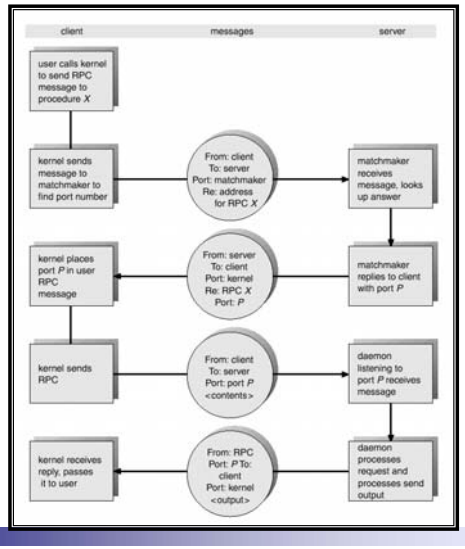
Socket Communication



Remote Procedure Calls

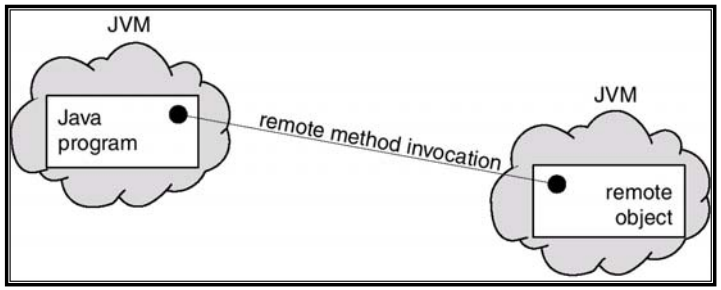
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.
- The server-side stub receives this message, unpacks the marshalled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

Execution of RPC

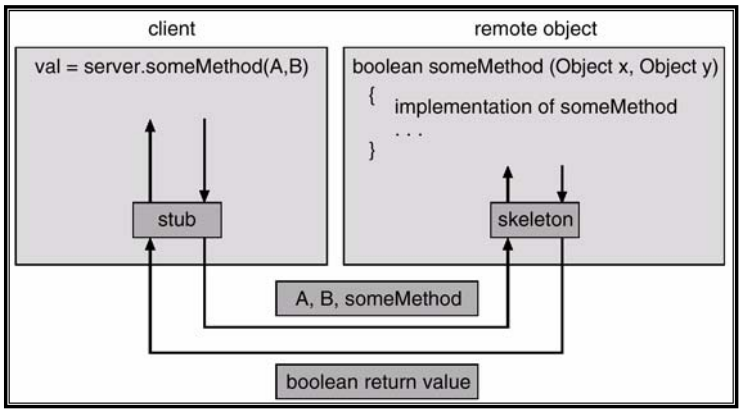


Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.



Marshalling Parameters



Deadlock (1)

- Deadlock is one area where there is a strong theory, but it is almost completely ignored in practice.
  - Reason: Solutions are expensive and/or require predicting the future.
- Deadlock example with semaphores.

  Process 0: Process 1:

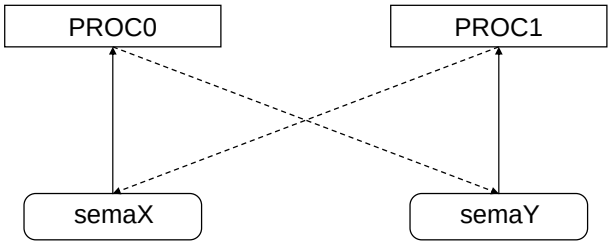
  P(semaX); P(semaY);

  P(semaY); P(semaX);

Deadlock (2)

- Define deadlock: A situation where each of a collection of processes is waiting for something from other processes in the collection. Since all are waiting,none can provide any of the things being waited for.

Deadlock (3): Example



Deadlock (4)

- The previous example was relatively simple-minded Things may be much more complicated:
  - In general, don't know in advance how many resources a process will need. If only we could predict the future...
  - Deadlock can occur over separate resources, as in semaphore example, or over pieces of a single resource, as in memory, or even over totally separate classes of resources (tape drives and memory).
  - Deadlock can occur over anything involving, for example, messages in a pipe system.
  - Hard for Os to control.

Deadlock (5)

- Four necessary conditions for deadlock:

(1) Mutual exclusion (limited access):

Resources cannot be shared.

(2) No preemption:

Once given, a resource cannot be taken away.

(3) Hold and wait (multiple independent requests):

Processes don't ask for resources all at once.

(4) Circular wait:

There is a circularity in the graph of who has what and who wants what.

Deadlock (6): Handling

- Solution to deadlock problem fall into two general categories:
  - Detection: Determine when the system is deadlocked, and then take drastic action. Requires termination of one or more processes in order to release their resources.
  - Prevention: Organize the system so that it is impossible for deadlock ever to occur. May lead to less efficient resource utilization in order to guarantee no deadlocks.

Deadlock Prevention (1)

- Mutual exclusion:
  - Don't allow exclusive access.

    This is probably not reasonable for many applications.
- No preemption:
  - Allow preemption. (E.g., Preempt your disk space?).

Deadlock Prevention (2)

- Hold and wait:
  - Make process ask for everything at once. Either get them all or wait for them all. Tricky to implement:

    Must be able to wait on many things without locking anything. Painful for process: May be difficult to predict, so must make very wasteful use of resources.

    This requires the process to predict the future.

Deadlock Prevention (3)

- Circular waiting:
  - Create enough resources so that there's always plenty for all.
  - Don't allow waiting. This punts the problem back to the user.

    (E.g., Phone company).
  - Make ordered or hierarchical requests. E.g., ask for all S's, then all T's etc. All processes must follow the same ordering scheme. Of course, for this you have to know in advance what is needed.

Deadlock Prevention (4)

- In general, prevention of deadlock is expensive and/or inefficient.

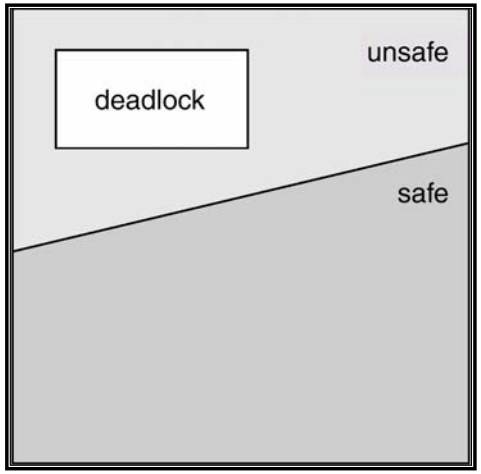
  Detection is also expensive and recovery is seldom possible

  (What if process has things in a weird state?).

Deadlock Avoidance (1): Safe State

- Safe state: The system can allocate resources to each process up to its maximum in some order and still avoid a deadlock.
- A safe sequence must exist from a safe state.
- Unsafe state: May lead to a deadlock.

Deadlock Avoidance (2): Safe State



Deadlock Avoidance (3): Safe State

- Example: A system with 12 magnetic drives.

Safe sequence: <P_1,P_0,P_2>

Processes	Max Needs	Current Allocations
P_0	10	5
P_1	4	2
P_2	9	2

Transition from a safe state to an unsafe one: < P'_2>

Processes	Max Needs	New Allocation (1 additional request)
P'_2	9	3

Deadlock Avoidance (4): Banker's Algorithm

- A new process must declare the maximum resource needs.
- When a process requests resources, the algorithm checks if the allocation will leave the system in a safe state.
- Grant the resources, if so.
- Have it wait until some other process releases enough resources.

Deadlock Avoidance (5): Banker's Algorithm (Notations)

- Available[1;m]: The number of available resources of each type.
- Max[1:n,1:m]: The maximum demand of each process.
- Allocation[1:n,1:m]: The number of resources of each type currently allocated to each process.

Deadlock Avoidance (6): Banker's Algorithm (Notations)

- Need[1:n,1,m]: The remaining resource need of each process.

  Max[I,j] = Allocation[I,j] + Need[I,j]

- For two vectors X and Y:
  - ∗ $X \leq Y$ iff $\forall i :: 1 \leq i \leq n :: X[i] \leq Y[i]$
  - ∗ $X < Y$ iff $X \leq Y$ and $X \neq Y$.

Deadlock Avoidance (7): Banker's Algorithm (Safety)

Step 0: Work[1:m] and Finish[1:n] are two vectors.

Step 1: Work:= Available and Finish[i]:= false for i = 1,2,...,n.

Step 2: Find an i such that both

- Finish[i] = false
- Need[i] ≤ Work

If no such i exists, go to Step 4.

Deadlock Avoidance (8): Banker's Algorithm (Safety)

Step 3: Work:= Work + Allocation[i]

Finish[i]:= true

Go to Step 2.

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

Banker's Algorithm (9): Resource Request for Pi

Step 0: Request[1:n, 1:m] is the resource request of each process.

Step 1: If Request[i] ≤ Need[i], go to step2.

Otherwise, raise an error condition.

Step 2: If Request[i] ≤ Available, go to step 3.

Otherwise, P_i must wait for the resource.

Banker's Algorithm (10): Resource Request for Pi

Step 3:

```
Available:= Available – Request[i];
Allocation[I]:= Allocation[i]+ Request[i];
Need[i]:= Need[i] – Request[i];
```

Step 4: If the resulting resource allocation is safe, the transaction is completed and P_i if allocated. Otherwise, P_i must wait and old resource allocation state is restored.

Deadlock Avoidance (11): Banker's Algorithm (Example)

- A state snapshot: Safe sequence <P_1,P_3,P_4,P_2,P_0>

Processes	Allocations	Max Needs	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	

- NewRequest[1] = (1,0,2):
  - Determine if the new state is safe.

Deadlock Detection (1)

- Single instance of each resource type:

  Existence of cycle is a necessary and sufficient condition for a deadlock.
- Multiple instances of a resource type:

  Use a deadlock detection algorithm similar to the banker's algorithm.

Deadlock Detection Algorithm (2)

Step 0: Work[1:m] and Finish[1:n] are two vectors.

Step 1: Work:= Available.

For I = 1,2,...,n, Finish[i]:= { false, if Allocation[i] ≠ 0; true, otherwise.

Step 2: Find an i such that both

- Finish[i] = false
- Request[i] ≤ Work

If no such exists, go to Step 4.

Deadlock Detection Algorithm (3)

Step 3: Work:= Work + Allocation[i]; Finish[i]:= true

Go to Step 2.

Step 4: If Finish[i] = false for some i, then the system is in a deadlock state. Such i (i.e., P_i) is a deadlocked process.

Deadlock Detection Algorithm (4): Example

- Deadlock involving P_1, P_2, P_3,P_4:

Processes	Allocations	Requests	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 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Deadlock Recovery

- Process termination
  - Abort all deadlocked processes
  - Abort one process one at a time until the deadlock cycle is eliminated.
- Resource Preemption
  - Select a victim
  - Rollback
  - Starvation