

CSE 120

Summer, 2005

Day 3
Inter-Process Communication (IPC)
Scheduling
Deadlock

Instructor: Neil Rhodes

Inter-Process Communication (IPC)

Inter-Process Communication

Cooperating processes must communicate

Need way to

- Transfer information
- Synchronize

Different types of IPC

- Shared Memory
 - Synchronization carried out via semaphores or monitors
- Message passing
- Remote Procedure Call (RPC)

Message Passing

One process sends a message, another receives it

- Variable-size / fixed-size message
- Direct or indirect communication
 - Directly specify other process
 - Symmetric: Send(P, message) / Receive(Q, &message)
 - Asymmetric: Send(P, message) / Receive(&processID, &message)
 - Indirect: use **mailbox** or **port**
 - send(MboxA, message) / Receive(MboxA, &message)
- Synchronous or asynchronous
 - Blocking send
 - Nonblocking send
 - Blocking receive
 - Nonblocking receive
 - Combo of blocking send and blocking receive yields a **rendezvous**
- Buffering: zero, bounded, or unbounded
 - Zero: sender blocks until receiver receives message
 - Bounded: If the buffer is full, sender blocks
 - Unbounded: sender never blocks

Remote Procedure Call (RPC)

Make a procedure call that invokes procedure in remote process

- Could be process on the same machine, or process across a network

How it works:

- Local stub **marshals** parameters
- May need to convert to machine-independent format
- Sends message with procedure ID and parameters
- Waits for receipt of message with return result

Remote Method Invocation (RMI)

- Used in Java as a way to cause a method to be executed on a remote object

Corba (Common Object Request Broker Architecture)

- Language- and platform-neutral way to execute methods on objects

5

Scheduling

Scheduling

Scheduler

- Part of the operating system that decides which ready process to run next

Scheduling Algorithm

- Algorithm the scheduler uses

Types of Processes

- I/O-bound
- CPU-bound

7

Scheduling Goals

Throughput

- Number of jobs per time period (or work/time period)

Fairness

- Some jobs aren't arbitrarily treated differently from others

Response Time

- Time till some output

Turnaround time

- Time until process complete

Predictability

- Low variance

Meeting deadlines

- Multimedia, for example

CPU utilization

- Don't waste if critical path

Proportionality

- Simple things are quicker than complicated things

8

Scheduling Algorithms

First-Come First-Served

- No preemption
- Easy to understand
- Low throughput and CPU utilization given I/O-bound processes

Shortest Process Next

- Look at CPU burst (not total CPU until completed)
- Let job run non-preemptively
- Predict based on history of CPU burst (T_i = time used for i th period. S_i = estimate of i th period)
 - Straight average: $S_{n+1} = 1/n T_n + (n-1)/n S_n$
 - Exponential average: $S_{n+1} = \alpha T_n + (1-\alpha) S_n$
- Reduces average turnaround time

Shortest Remaining Time

- Based on estimate of total time (given by user or estimated from history)-time spent so far
- Preemptive at time of arrival

Highest Response Ratio Next

- Response ratio = (waiting time + total time)/waiting time
- Chose job with highest response ratio

9

Scheduling Algorithms

Round-robin scheduling

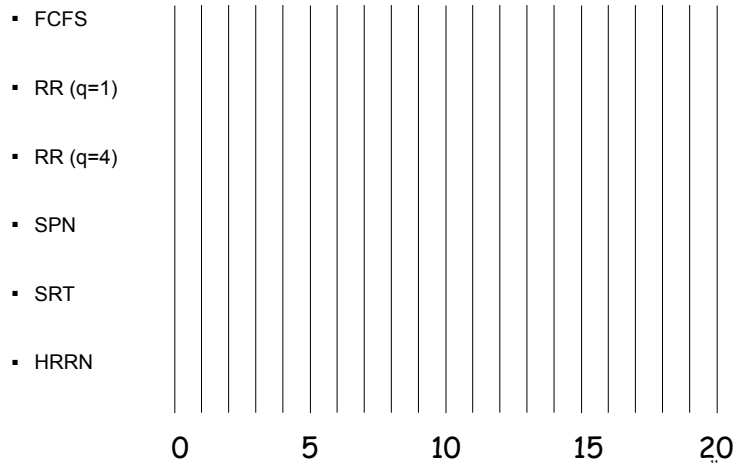
- Give each job a timeslice (quantum). Preempt if still running
- Put job at end of ready queue and run next job in ready queue
- Quantum should be large relative to context-switch time

10

Comparison

Four processes

- A: arrives at time 0, CPU time 3
- B: arrives at time 2, CPU time: 6
- C: arrives at time 4, CPU time: 5
- D: arrives at time 6: CPU time: 4



11

Scheduling Algorithms

Priority Scheduling

- Priority associated with each process
- High priority job runs before *any* lower priority jobs
- Starvation is a problem
 - Solution: Aging (slowly increase priority of waiting processes)

Multilevel Queue Scheduling

- Distinguish between different classes of processes
 - Student/Instructor/interactive/system
 - batch/interactive
- Queues can have different scheduling algorithms
- Need scheduling between queues

Multilevel Feedback-Queue Scheduling

- Different queues, processes move from queue to queue based on their history
- For example:
 - Queue 1: 1 quantum. If a process uses its entire quantum, it moves to next Queue
 - Queue 2: 2 quanta. if a process uses its entire quantum, it moves to next Queue
 - Queue 3: 4 quanta. ...
 - ...

12

Scheduling Algorithms

Fair-share scheduling

- Divide user community into a set of fair-share groups.
- Allocate a fraction of the processor resource to each group
- Each group gets its fair-share of the CPU time
- Priority of a process depends on:
 - How much CPU time its group has had recently
 - How much CPU time it itself has had recently
 - Base priority of the process
- Example
 - 3 process: A, B, C. A is in group 1, B&C are in group 2.
 - Assume fair-share is group 1: 50%, group 2: 50%
 - Possible scheduling sequence:
 - A B A C A B A C A B A C

13

Scheduling Algorithms

Lottery Scheduling

- Give processes lottery tickets
- Randomly choose a ticket: whoever has ticket gets to run
- Some processes can have more tickets than others
- If a process holds 20% of the tickets, in the long run, it'll get 20% of the CPU
- A process can give tickets to another process
 - Example: When a client makes a blocking request to a server, it give the server its tickets. Server doesn't normally need *any* tickets
- If a process doesn't use its entire quantum, give a compensation ticket that increases its tickets by a certain amount until next time it runs
 - For example, if process A and process B each have 400 tickets, but A uses its entire quantum and B uses 1/5, then A will get 5 times as much time as B
 - When B uses only 1/5 of a quantum, give compensation ticket worth 1600. Next lottery, B has 2000, A has 400. B is 5 times more likely to win the lottery

14

Priority Inversion

Image three levels of priority

- High
- Medium
- Low

We want high priority processes to run before any medium or low

L (a low priority process) holds a mutex.

H (a high priority process) blocks trying to obtain the mutex

M (a medium priority process) runs since:

- H is blocked
- L is of lower priority

Meanwhile, H can't run because it's waiting for L

Solution:

- Priority Donation:
 - While L holds a resource, it gets (temporarily) priority of higher processes waiting for it.

15

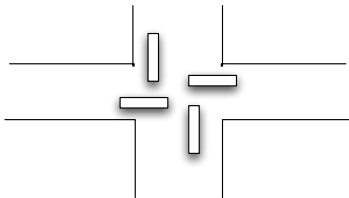
Deadlock

Deadlock

A chain of processes exist that are blocked waiting on one another

- Each process has requested a resource that another process is holding

Process A:	Process B:	Process C:	Process D:
Request(resourceA)	Request(resourceB)	Request(resourceC)	Request(resourceD)
Request(resourceB)	Request(resourceC)	Request(resourceD)	Request(resourceA)
Do Processing	Do Processing	Do Processing	Do Processing
Release(resourceB)	Release(resourceC)	Release(resourceD)	Release(resourceA)
Release(resourceA)	Release(resourceB)	Release(resourceC)	Release(resourceD)



17

Necessary Conditions for Deadlock

Can't have deadlock unless we have all four conditions:

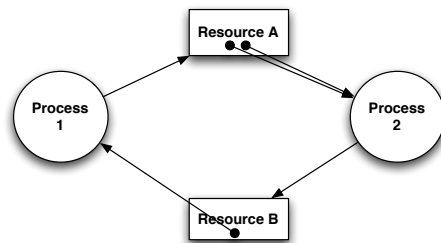
- Mutual Exclusion:** If process A requests a resource that process B is using, process A is blocked.
- Hold and wait:** At least one process must be holding a resource and waiting for another (blocked).
- No preemption:** A process can't be forced to release a process; it must do so voluntarily after it has finished its task
- Circular wait:** A set of resources $\{P_0, P_1, \dots, P_n\}$ of waiting processes must exist where P_0 is waiting for a resource held by P_1 , P_1 waiting for a resource held by P_2 , ... P_n waiting for a resource held by P_0 .

18

Resource-Allocation Graph

Arrow from resource instance to process if process has resource allocated

Arrow from process to resource if process has requested resource



19

Ways to Deal with Deadlock

Deadlock Prevention

- Change rules so deadlock can't happen

Deadlock Avoidance

- Check each allocation to see whether that could lead to a future deadlock situation

Deadlock Detection

- Detect
- Recover

20

Deadlock Prevention

Ensure no deadlocks by removing one of the 4 necessary conditions

- **Mutual Exclusion**
 - Some resources are sharable: read-only files, for example.
 - Others are intrinsically non-sharable (CD-burner, for example)
 - Can make some non-sharable resources sharable
 - Spooler for printer, for example
 - Can't remove this condition for all resources, though
- **Hold and Wait**
 - Require each process to allocate all resources before it begins execution
 - Request resources in a group: can only request if not holding any
- **No Preemption**
 - If process requests a resource but must wait, it gives up all its existing resources. These are added to list of resources it is waiting for
 - Or, if process is waiting holding a resource, resource may be preempted if needed by another process. First process wakes up only when *all* resources it needs are available
 - Problem: some resources state can't be maintained when taken away and then given back: in the middle of writing to a tape, for example.
- **Circular Wait**
 - Impose an ordering on the resources
 - Processes must request resources in increasing order

21

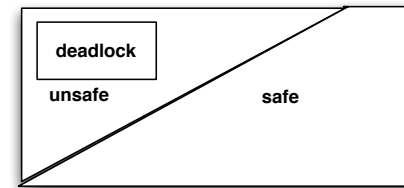
Deadlock Avoidance

Process provides additional information about their future resource usage

- for example, maximum number of resources of each type that it may need

System is kept in a **safe state**

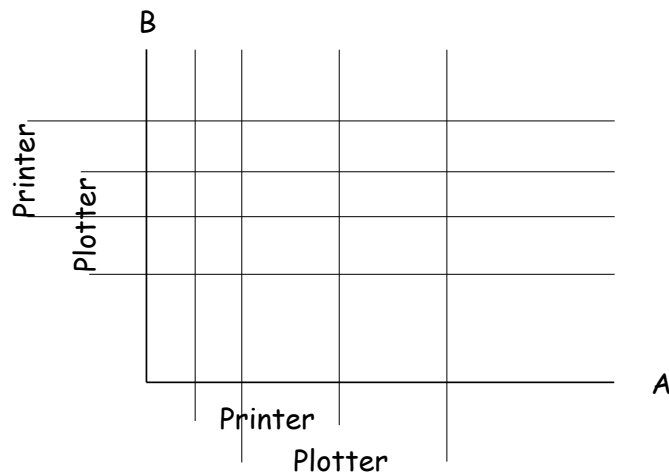
- Unsafe states can lead to deadlock
- If the state is safe now, based on the knowledge of possible future requests, it can be kept safe



22

Safe/Unsafe states

2 Processes each acquiring two resources



23

Bankers Algorithm

Imagine a bank with:

- n customers each with a line of credit
- A certain amount of money on hand

Want to make sure that if we lend any money to a customer

- We'll be able to satisfy all future requests up to the line of credit.
- Don't want to be in a situation where we've got no money and all customers with current loans out want more money
- Leave enough money-on-hand to satisfy all needs of at least one customer (if their payback is enough to satisfy one more customer, etc.)

24

Deadlock Avoidance

Don't grant resource request if it takes system to an unsafe state

- Even if the resource is currently available!

Banker's Algorithm

- Let n = number of processes, m = number of resource types
- Define $V[m]$: # of resources of a particular type currently available
- Define $D[n, m]$: Maximum demand of each process (specified when process begins)
- Define $A[n, m]$: current allocation of each process/type pair
- Define $N[n, m]$: How much more a process may need ($=M-A$)

When request is made:

- Assume temporarily that the request is granted
- Update state and determine whether it is still safe
- If not, restore the state and don't satisfy the request

Determining whether a state is safe

- Recursively:
 - Can some process' maximum demand be satisfied by their current allocation and the available resources?
 - If so, that process could finish, and return its resources. Update state as if resources were returned and continue
- If any processes still remaining, state is not safe

25

Bankers Algorithm

Assumptions:

- Maximum resource requirements for a process stated in advance
- Processes must be independent (A can't be waiting on B for anything other than a resource it has)
- Fixed number of resources
- A process can't exit while keeping its resources

26

Deadlock Detection

Detection

- Recursively:
 - Can some process' current request be satisfied by the available resources?
 - If so, that process could finish, and return its resources. Mark it, update available resources, and continue
- Any processes that remain unmarked are deadlocked.
- Must run algorithm to detect deadlock.
 - How often?
 - When system is slow
 - Once per time period
 - Every time a resource allocation would block?

Recovery

- Process Termination
 - Abort all deadlocked processes
 - Abort one process at a time until the deadlock cycle is eliminated
- Resource Preemption
 - Take away already allocated resources from some process
 - Rollback that process to known good state?

27