Comp 310
Computer Systems and Organization

Lecture #12
Process Management
(Deadlocks)

Prof. Joseph Vybihal

Announcements

- Oct 16 Midterm exam (in class)

- Tutorial times posted on web CT
  - Wednesday 12:00-13:00, Trottier – Theresa
  - TBD

- Old exam on web CT

Midterm Exam Format

- Four questions
  - Definition questions (1 to 2)
  - Analyze and fix (0 to 2)
  - Analyze and describe (1 to 2)
  - Pseudo-code (0 to 1)
  - Actual C code (0)
- Material on Exam
  - Material including this lecture
  - Historical and current OS architectures
  - The components of a modern OS
  - Implementation of the User Interface
  - Executing multiple processes & OS run-time states
  - Inter-process communication and synchronization problems
  - Thread implementations
  - OS Scheduling problems
  - Deadlock graphs, semaphore problems and algorithms

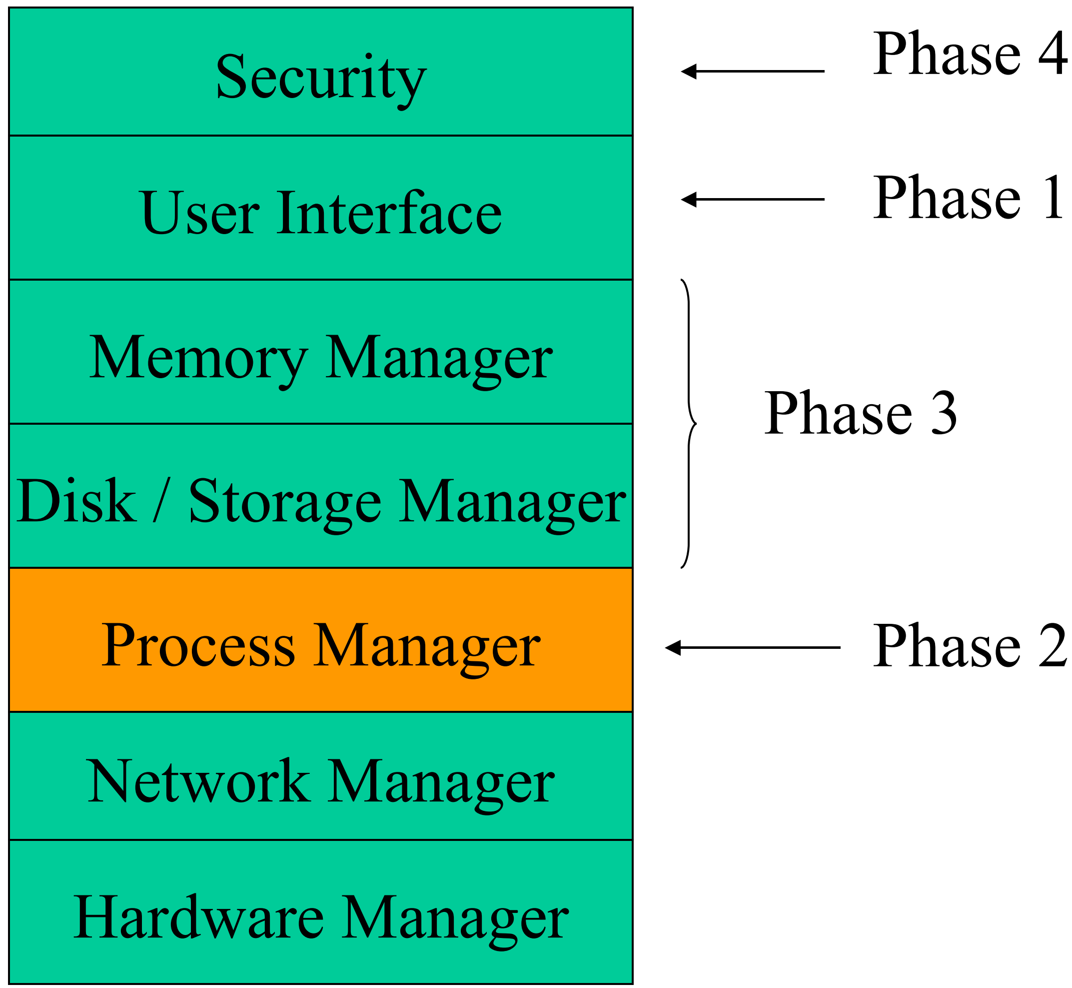
Architectures

User Interface

Process Management

Basic OS Architecture

(Course Table of Contents)



Kernel Design

```
while (!done)
{
        RunPtr = deQ(Ready);
                                                    //    0          1 to n        n+1 to m
        returnCode = contectSwitch(RunPtr); // quanta, interrupt, service request

        // Standard run-time overhead

        switch(returnCode)
        {
                case 0:
                        enQ(RunPtr);
                        RunPtr = NULL; // optional
                        break;
        }

        // System overhead

        detectDeadlock();
}
```

Part 1

Preventing Deadlocks

Three Methods

- Changing the necessary conditions, or
- Avoiding deadlocks, or
- Do nothing, then recover from a deadlock (detecting deadlocks)

Changing the necessary conditions

Necessary Conditions

- Mutual Exclusion
- Hold and Wait
  - A process must be holding at least one resource and waiting to acquire another that is being held
- No pre-emption
  - The OS does not permit pre-emption of held resources
- Circular wait
  - {P1, P2, P3} s.t. P1 → P2 → P3 → P1

Changing Mutual Exclusion

- Some resources are intrinsically non-sharable:
  - Printers, …
- Solution: Extreme exclusion
  - Single processor single process systems
    - One program runs until completion
    - Limited or no protection of resources
    - Trust programmer to code properly
  - Eg: MS DOS environment
- Is this a good solution? Why or why not?

Changing Hold and Wait

- Alternatives:
  - Use a *job-control-language* that tells the OS all the resources the program will need before it begins to run. The OS runs the program only after all those resources are available. They all get assigned to the process at once and are held until the process ends.
  - The *Unit Request* method uses a programming command to request a subset of resources all at once. These are held until not needed.
- Two Problems:
  - Low resource utilization
  - Starvation still possible (i.e. not deadlocked)

Changing No Preemption

- Alternatives:
  - The *wait-queue release* method automatically releases all held resources when a process is put on the wait-queue.
  - The *rob-a-resource* method locates the process on the ready-queue currently holding that resources and takes it away (puts it on wait-queue).
- Side effects:
  - e.g. Files take a long time to find and buffer
  - Used for resources who’s state can be quickly redefined

Changing Circular Wait

- Solution:
  - Rule 1: Enumerate all resources R={R1, R2,…Rn}
  - Rule 2: Access resource in increasing number
  - Rule 3: If a resource is needed from a lower number then process must release all higher numbers
- Proof:
  - If R0 to Ri in use and needs Ri+1
  - Implies Ri < Ri+1
  - But if Ri is Rn then circular and not legal (Ri > Ri+1)
  - Assumes modulo arithmetic
- Starvation still possible

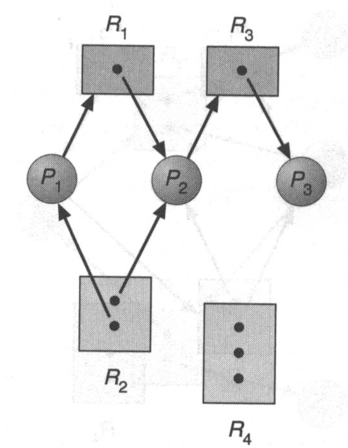
Part 2

Avoiding Deadlocks

The Safe State Concept

IDEA: The “Safe” State -

A sequence of resource requests exists so that no deadlock can result - called a “safe sequence”



METHOD:

A special OS resource table has extra fields initialized at run-time with the “Maximum Needed” resource of type X for process Y. OS tracks the max & current needs of all processes.

Tracking Avoidance

Assume we have a computer system that uses 12 File Buffers and that there is currently 3 processes running:

PROCESS	MAX NEEDS	CURRENT NEEDS at T_0
P0	10	5
P1	4	2
P2	9	2

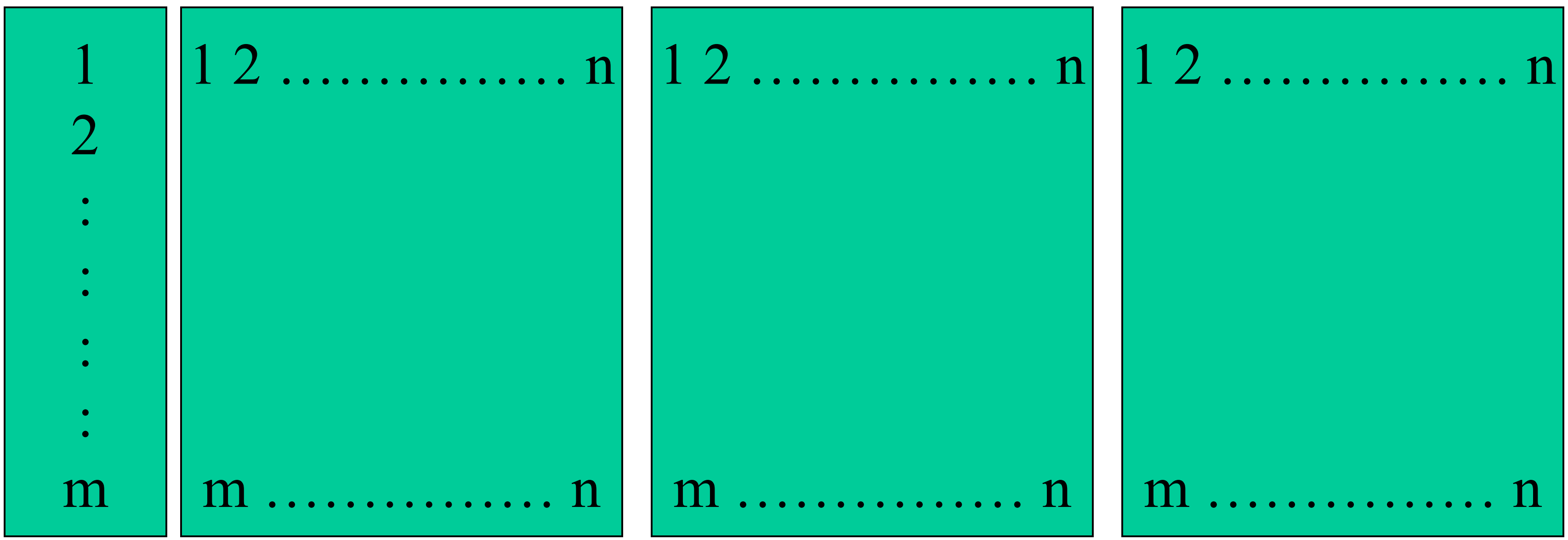
Note: Is in a safe state since <P1, P0, P2> terminates.

Assume: At T_0, P0 is allocated 1 more resource. We are now in an unsafe state - P1 can finish but not P0 and P2.

Banker's Algorithm

(Data structures)

Available Resources
in each type 1…m



Max Resources Needed
by each Process 1…n

Allocated Resources
by each Process 1…n

Needed Resources
by each Process 1…n

NOTE: Need[i,j] = Max[i,j] - Allocated[i,j]

<u>Banker's Algorithm safe state determination algorithm</u>:

1. Let WORK[1:m] = Available[1:m]
   Let FINISH[1:n] = false (for all i)

2. Locate i such that
   FINISH[i] = = false && Needed[i] <= WORK[i]
   if yes then goto step 3 else goto step 4

3. WORK[i] = WORK[i] + Allocated[i]
   FINISH[i] = true
   goto step 2

4. If (FINISH[i] = = true for all i) then return SafeState = true.

NOTE: $O(m * n^2)$

Banker's Resource-Request Algorithm:

1. Let Request[i] be the resources Pi wants to access (by type)

2. If (Request[i] <= Needs[i]) then goto step 3
   else error "Exceed max claim"

3. If (Request[i] <= Available[i]) then goto step 4
   else wait(Pi), not enough resources

4. Call Banker's Safe State Algorithm, given:
   Available = Available - Request (for i)
   Allocation = Allocation + Request (for i)
   Needs = Needs - Request (for i)

If returns TRUE then give resources else wait(Pi), not safe.

Example

Process	Allocation	MAX	Available	Need
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2	7 4 3
P1	2 0 0	3 2 2		1 2 2
P2	3 0 2	9 0 2		6 0 0
P3	2 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

Is this system in a safe state?

How can we make it into an unsafe state?

Deadlock Detection Algorithm:

1. Let Work[1:m] = Available
   Let Finish[1:n] = false when Allocated[i] <> 0, else true.

2. Locate i such that
   Finish[i] = = false && Request[i] <= Work[i]
   if yes the goto step 3 else goto step 4

3. Work[i] = Work[i] + Allocated[i]
   Finish[i] = true
   goto step 2

4. If (Finish[i] = = false) then return deadlock for Pi

NOTE: $O(m * n^2)$

What do we do with a deadlock?

If (Pi = = deadlock)

A) Terminate the process (easy), or

B) Resource preemption of all resources held by process.

(need data structure to remember resources,

will need to reallocate them once given CPU)

Part 3

Recovery from deadlocks

Three Solutions

- Prompt User
- Auto Process Termination
- Resource Preemption

Prompt User

- Resource reduction prompt
  - e.g. prompt user to close a file
  - does not specify which program or file
- Terminate a process prompt
  - ask the user to close a process
  - may or may not recommend a process
- User maintains control as to which program is destroyed

Auto Process Termination

- Abort all deadlocked chain of processes
  - expensive since a lot of work was done
- Abort one process and reevaluate
  - $O(m * n^2)$ each reevaluation
  - Other considerations:
    - Priority
    - Resource type
    - How many resources needed to terminate?
    - How long has process been running?
- No easy solution...

Resource Preemption

- Step 1: Selecting a victim
  - similar cost considerations as per last slide
- Step 2: Rollback
  - e.g. Once buffers are deleted the process cannot continue from where it left off … where then?
  - Total rollback i.e. delete and restart process
- Starvation is a problem…
  - Increase process priority (if that is a consideration)

Part 4

At Home

Things to try out

1. Prepare for midterm exam