CMSC421: Principles of Operating Systems

Nilanjan Banerjee

Assistant Professor, University of Maryland
Baltimore County
nilanb@umbc.edu
http://www.csee.umbc.edu/~nilanb/teaching/421/

Acknowledgments: Some of the slides are adapted from Prof. Mark Corner and Prof. Emery Berger's OS course at Umass Amherst

Announcements

- Homework 2 (due Oct 13th)
- Midterm (29th of October in class)
- Readings from Silberchatz [7th chapter]

Deadlock Prevention

- Instead of detection, ensure at least one of necessary conditions doesn’t hold
  - **Mutual exclusion**
  - **Hold and wait**
  - **No preemption**
  - **Circular wait**

Avoiding Deadlock

- Not ok – may deadlock.

```
lock (a);
lock (b);
unlock (b);
unlock (a);
```

```
lock (b);
lock (a);
unlock (a);
unlock (b);
```

- Solution: impose canonical order (acyclic)

```
lock (a);
lock (b);
unlock (b);
unlock (a);
```

```
lock (a);
lock (b);
unlock (b);
unlock (a);
```

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

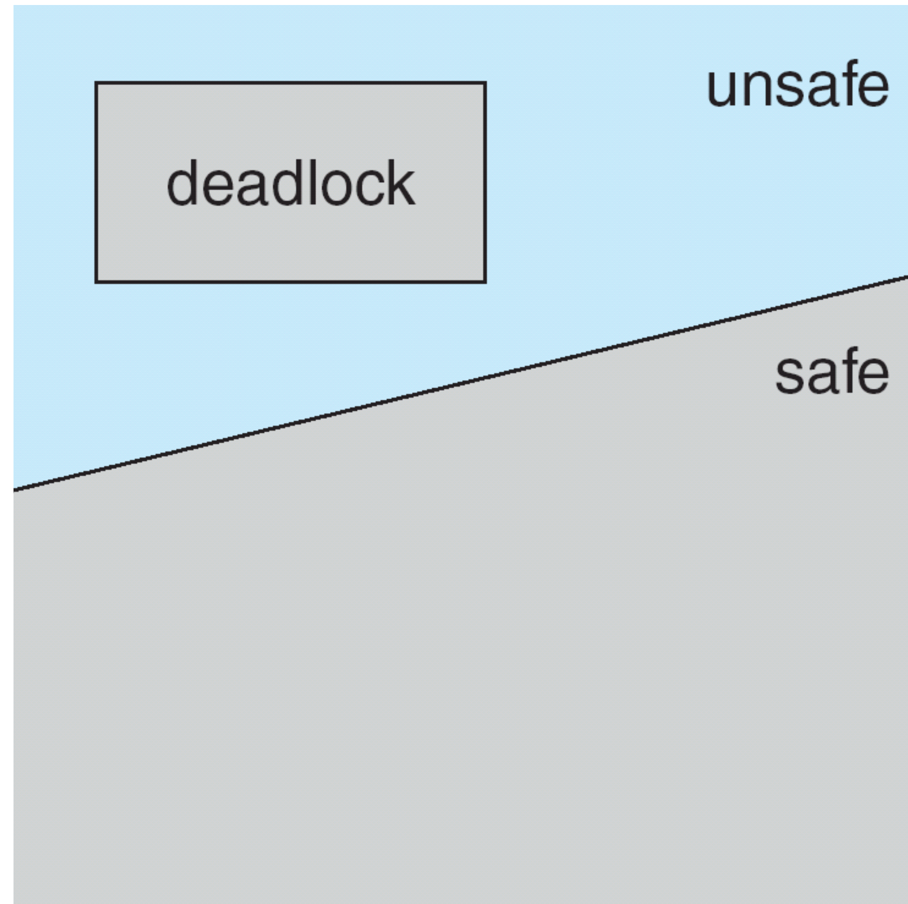
Safe State

- 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

Basic Facts

- 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.

Safe, Unsafe, Deadlock State



Avoidance algorithms

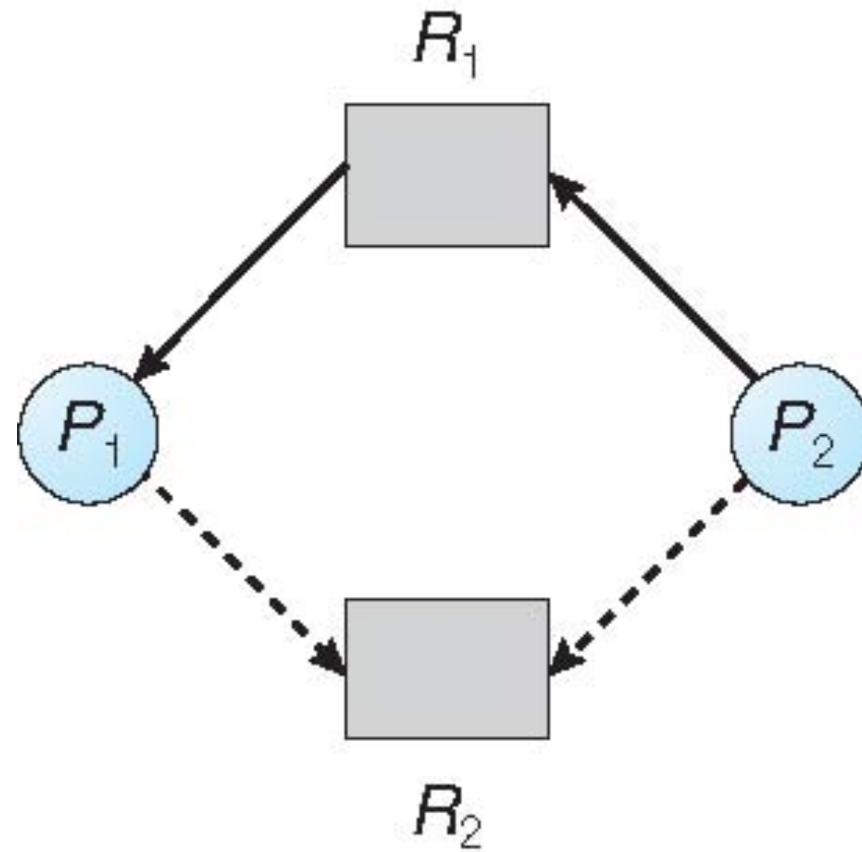
- Single instance of a resource type
  - Use a resource-allocation graph

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

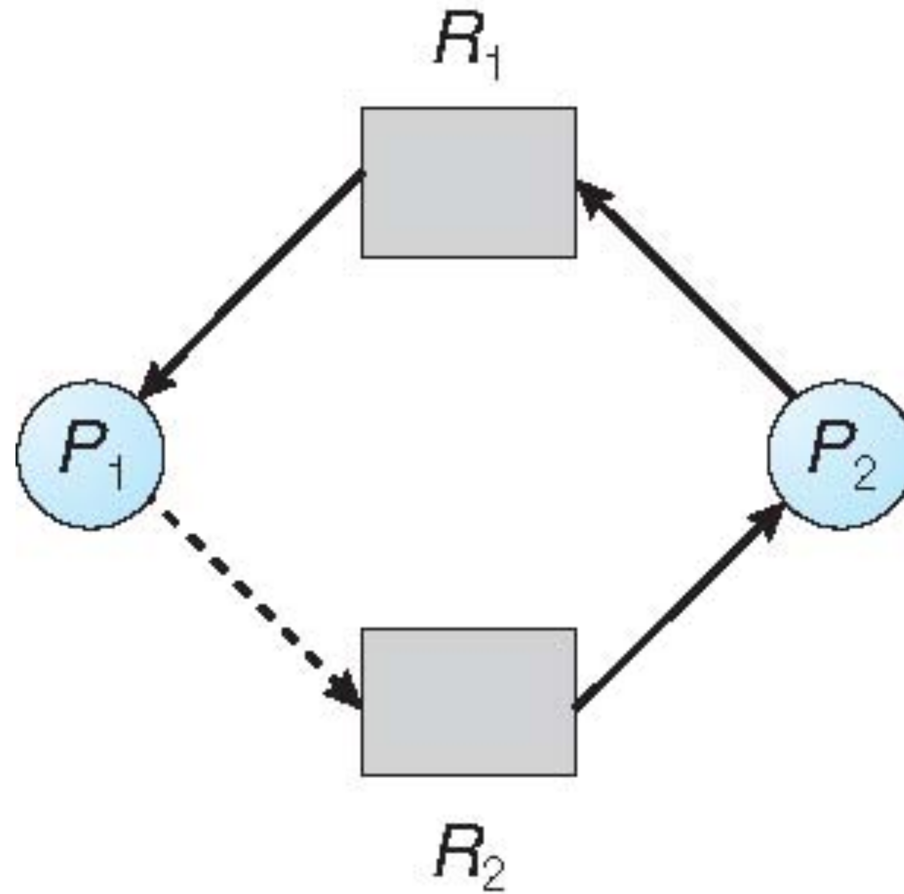
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; 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 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

Example of the Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5instances), and *C* (7 instances)

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 the Banker's Algorithm

- 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 of the Banker's Algorithm

- Process P0 requests (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 $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Scheduling (what is it?)

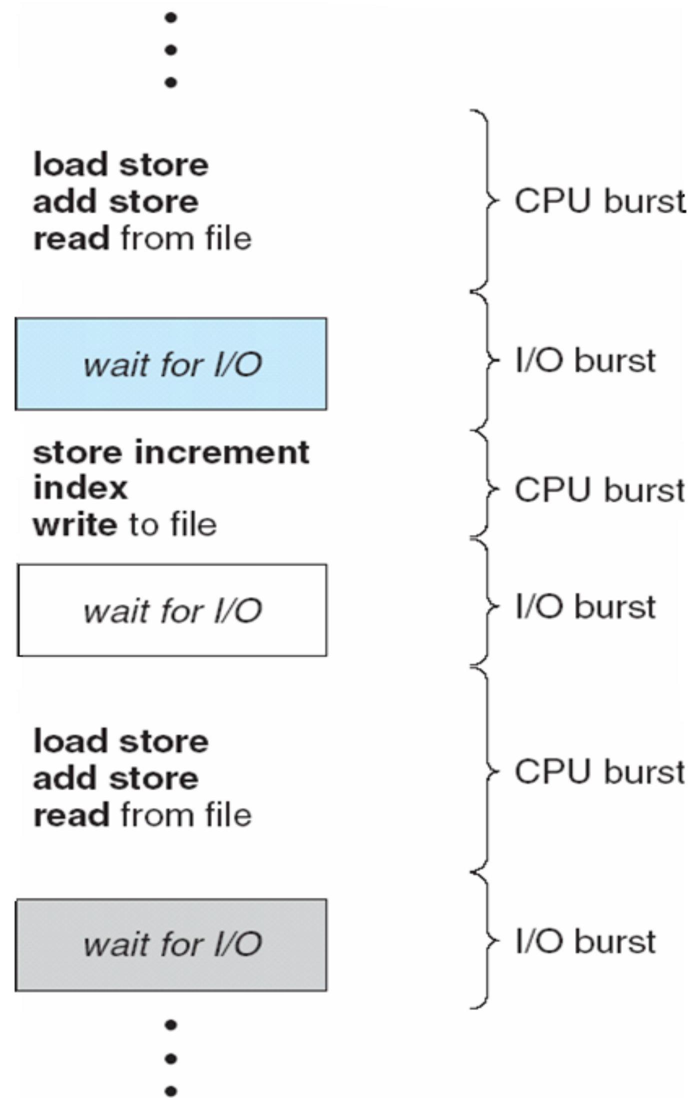
- Some number of tasks
  - Threads or processes
- One of more CPU cores to use
- CPU Scheduler decides on two things
  - How much time should each task execute on the CPU
  - In which order does the set of tasks execute
- In a multi-core system
  - Defacto: You take a set of tasks and always execute those set of tasks on one core
  - On a single core: you use more fine grained policies to decide which process to execute

Scheduling (Why do we need scheduling)

- Interactivity or multitasking
  - You cannot have one thread/process running on the CPU all the time
  - E.g. mouse pointer
- Performance
  - Processes might be waiting for stuff to happen
    - I/O bursts
    - Waiting on a sleep?

- If you have a compute intensive task (e.g., prime number calculation), scheduling will not yield any benefit

Ordinary task execution



How do you do scheduling?

- Cooperative Scheduling
  - A set of tasks cooperative amongst each other to do stuff
  - sched_yield();
- Preemptive Scheduling
  - The kernel schedules the process/thread on the CPU
  - The OS scheduler code determines the amount of time a task should be running on the CPU and the ordering of the task execution
  - How does the OS determine when to schedule a task
    - Timer interrupts
    - Interrupt the CPU
    - CPU executes an timer interrupt handler
    - Invokes the scheduler code

Cooperative Scheduling

- Pros
  - Very efficient
  - Easy to implement
    - Don’t need timers
    - Don’t need to be in the kernel
      - Green threads --- Java uses

- Cons
  - Greedy processes will take all the CPU time
  - Difficult to work with tight timing constraints
    - Real time system

Modern Operating Systems--- preemptive scheduling

- Selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling policy

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

An in-class discussion