Deadlocks

Arvind Krishnamurthy
Spring 2004

The deadlock problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process.
- Example
  - locks *A* and *B*

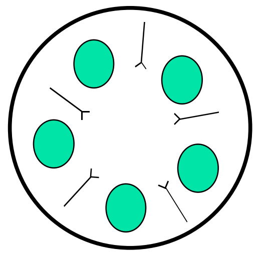
P_0	P_1
lock (A);	lock (B)
lock (B);	lock (A)

- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Deadlock implies starvation (opposite not true)

The dining philosophers problem

- Five philosophers around a table --- thinking or eating
- Five plates of food + five forks (placed between each plate)
- Each philosopher needs two forks to eat

```
void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork (i);
        take_fork ((i+1) % 5);
        eat();
        put_fork (i);
        put_fork ((i+1) % 5);
    }
}
```



Deadlock Conditions

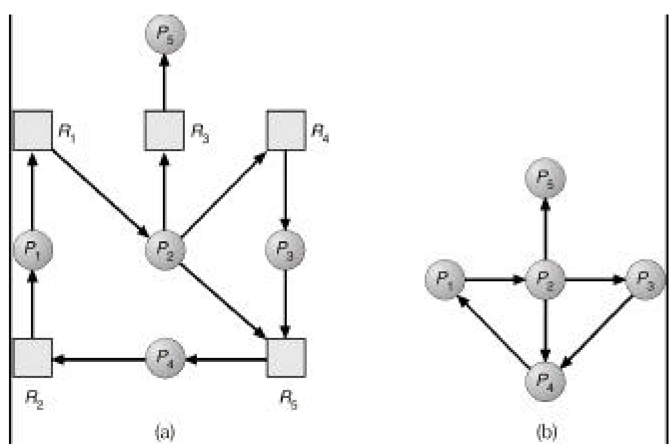
- **Deadlock can arise if four conditions hold simultaneously:**
  - **Mutual exclusion:** limited access to limited resources
  - **Hold and wait**
  - **No preemption:** a resource can be released only voluntarily
  - **Circular wait**

Resource-allocation graph

A set of vertices V and a set of edges E.

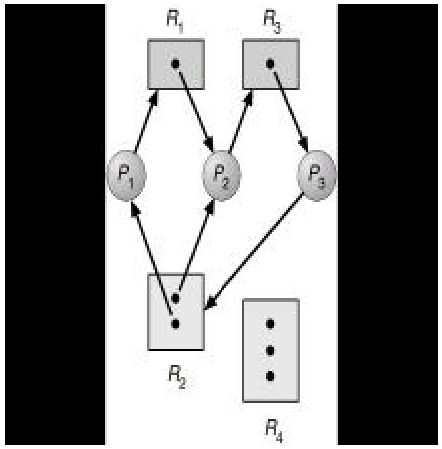
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system (CPU cycles, memory space, I/O devices)
  - Each resource type R_i has W_i instances.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation & Waits-For Graphs



Resource-Allocation Graph   Corresponding wait-for graph

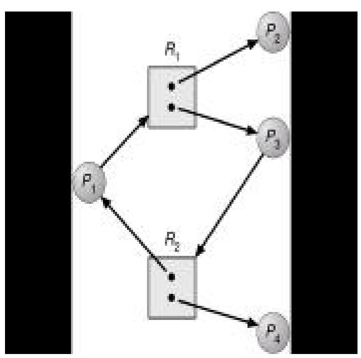
Deadlocks with multiple resources



P1 is waiting for P2, P2 is waiting for P3, P3 is waiting for P1 or P2

Is this a deadlock scenario?

Another example



- P1 is waiting for P2 or P3, P3 is waiting for P1 or P4
- Is there a deadlock situation here?

Announcements

- Midterm exam on Feb. 27th (Friday)
- Duration: 1:30 – 3:30??

Methods for handling deadlocks

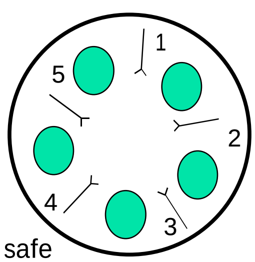
- Question: what options do we have in dealing with deadlocks?

Deadlock prevention

- Avoid mutual exclusion: have unlimited resources or unlimited access
- Do not hold and wait
  - must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request all its resources before it begins execution
  - Low resource utilization; starvation possible.
- Allow preemption
  - If a process 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 process needs
  - Process will be restarted only when it can regain all its resources
- Prevent circular wait
  - impose a total ordering of all resource types
  - require that each process requests resources in an increasing order

Preventing Circular Wait

- Dining Philosophers:
  - Number the forks from 1 thru' 5
  - First get the lower numbered fork
  - Then get the higher numbered fork



  - Sufficient to break the cycle
  - Any arbitrary numbering of resources is safe
  - What if resources are undistinguishable – let us say there is a pool of forks in the middle and philosophers want any two?
    - Ok, if both forks are requested simultaneously
    - How do we avoid deadlock if requests are made separately?

Banker's algorithm: Deadlock Avoidance

- More efficient than obtaining all resources at startup

- State maximum resources needed in advance
- Allocate resources dynamically when needed
  - Wait if granting request would lead to deadlock
  - Request can be granted if some sequential ordering of threads is deadlock free

- Sum of maximum resource needs can be greater than the total resources
- There just needs to be some way for all the threads to finish

- For example, allow a thread to proceed if:

  total available resources – # allocated >= max remaining needs of thread

Example

- Dining Philosophers: put forks in the middle of the table
- Rules:
  - If not last fork, grab it
  - If last fork and requesting thread needs only one more fork, let the thread grab the fork
  - Else, wait

- Another set of rules:
  - If not last fork, grab it
  - If last fork and requesting thread needs only one more fork, let the thread grab the fork
  - If last fork, but there is some thread who has two forks, let the requesting thread grab it
  - Else, wait

Data Structures for Banker's Algo

Let n = number of processes, and m = number of resources types.

- ***Max***: $n \times m$ matrix. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- ***Allocation***: $n \times m$ matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j.

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j].$$

- ***Need***: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task.
- ***Available***: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.

Resource Request

$Request_i$ = 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 state as follows:

$$Available := Available - Request_i;$$
$$Allocation_i := Allocation_i + Request_i;$$
$$Need_i := Need_i - Request_i;$$

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

Banker's algorithm: safety test

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively.
   Initialize:
   Work := *Available*
   Finish [*i*] = *false* for *i* =1,2, ..., *n.*

2. Find an *i* such that both:
   (a) Finish [i] = false
   (b) $\text{Need}_i \leq \text{Work}$
   If no such *i* exists, go to step 4.

3. $\text{Work} := \text{Work} + \text{Allocation}_i$
   Finish[i] := true
   go to step 2.

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

Example

- Three processes: p1, p2, p3
- Two resource types: r1, r2
  - Number of r1 resources = 3, number of r2 resources = 3
- Max requirements of processes:
  - P1's max = [2, 2]
  - P2's max = [2, 2]
  - P3's max = [2, 2]
- One deadlock scenario: everyone acquires one each of r1 and r2
- Satisfy request as they come – make sure that the deadlock scenarios cannot be reached

Scenario

- P1 requests [1, 1]
  - Grant it. Available: [2, 2]
- P2 requests [1, 1]
  - Request granted. Available: [1, 1]
- P3 requests [1, 1]
  - Request is not granted. P3 just blocks
- P2 requests [1, 0]
  - Request is granted. One safe sequence: P2 requests [0, 1], exhausts needs, finishes execution, releases resources, ...
- P1 requests [0, 1]
  - Request is not granted. P1 just blocks. System waits for P2 to request, finish & release resources

Deadlock detection and recovery

- Allow system to enter deadlock state
- Detection algorithm
  - use *wait-for* graph if single instance of each resource type
    - Nodes are processes.
    - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
  - periodically searches for a cycle in the graph; when and how often depends on:
    - How often a deadlock is likely to occur?
    - How many processes will need to be rolled back
  - harder if multiple instances of each resource type
- Recovery scheme
  - process termination
  - resource preemption, roll-back (used in databases)

Combined Approach

- Combine the three basic approaches
  - prevention
  - avoidance
  - detection

  allowing the use of the optimal approach for each of resources in the system.

- Partition resources into hierarchically ordered classes.

- Use most appropriate technique for handling deadlocks within each class.