Deadlocks

Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems
Uppsala University

Spring 2007

Outline

1. Recall the problem? Quickie on Synchronisation
   - Synchronisation
   - Solutions & Tools
   - Problem
2. Characterizing Deadlocks
   - Resources
   - Resource Allocation Graph
   - Conditions
3. Handling Deadlocks
   - Deadlock Prevention
   - Deadlock Avoidance
   - Deadlock Detection
   - Deadlock Recovery

Synchronisation

Example (El Classico)

producer-consumer on shared memory systems

Critical-section problem

- *n* processes share data
- each process has a *critical section* where the shared data are updated
- Make sure that only **one** process a a time is in the critical section

Synchronisation

Process structure

```
for(;;){ //Many times
    [Ask for permissions to enter critical section]
    // Do some work
    [Exit section] // Clean up, notify others
    // Do some work outside the CS
}
```

Requirements for a solution

- Mutual exclusion
- Progress
- Bounded waiting

Solutions and Tools

Peterson's Solution

Catch on that for yourself.
Recall that it is in case of exactly 2 processes

Hardware tools

- Test-And-Set(x)
  TAS sets x to TRUE and returns the previous value, atomically
- Compare-And-Swap(x,y,t)
  CAS swaps the content of x with y, if t is TRUE, atomically

Semaphores

Catch on for yourself. Invented by Dijkstra

Example (A bank account)

Balance $b = 0$ *sek*.
Person *A* does a deposit of 100 sek.
Person *B* does a deposit of 200 sek
$\Rightarrow$ Balance sould be 300 sek.

A	Balance *b*	*B*
	0	load R_4, b
load R_2, b	0	
add $R_2, \#100$		
store b, R_2	100	
		add $R_4, \#200$
	200	store b, R_4

Solution: Synchronisation (Semaphores,Monitors,Retry loops,...)

Main Problem

The programmer must use the semaphores correctly!!

Example (mutex semaphores S and Q)

P_1	P_2
wait(S)	wait(Q)
wait(Q)	wait(S)
...	....
signal(Q)	signal(Q)
signal(S)	signal(S)

Leads to deadlock: Both P_1 and P_2 are waiting for each other
Additionally, starvation may occur if semaphores are incorrectly implemented.

Resources

In shared memory systems, synchronisation deals with CPU allocation with respect to ...memory!

Other types of resources: CPU, disks, tapes, files, ...memory.

Each resource type may have a different number of instances
(example: 2 CPUs, 3 disks, *N* buffer places...
but instances may or may not be equivalent (e.g. printers on different floors).

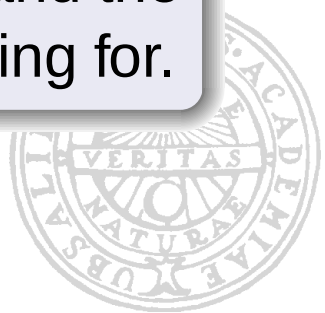
Resources

To protect against races, a resource may be

1. requested before use, waiting until it is available.
   Open file, allocate memory, move to ready queue...
2. used
3. released after use.
   Close file, deallocate memory, terminate/wait...

Deadlock

Arises when two or more processes wait for each other, and the only way out is if one releases a resource another is waiting for.



Resource Allocation Graph

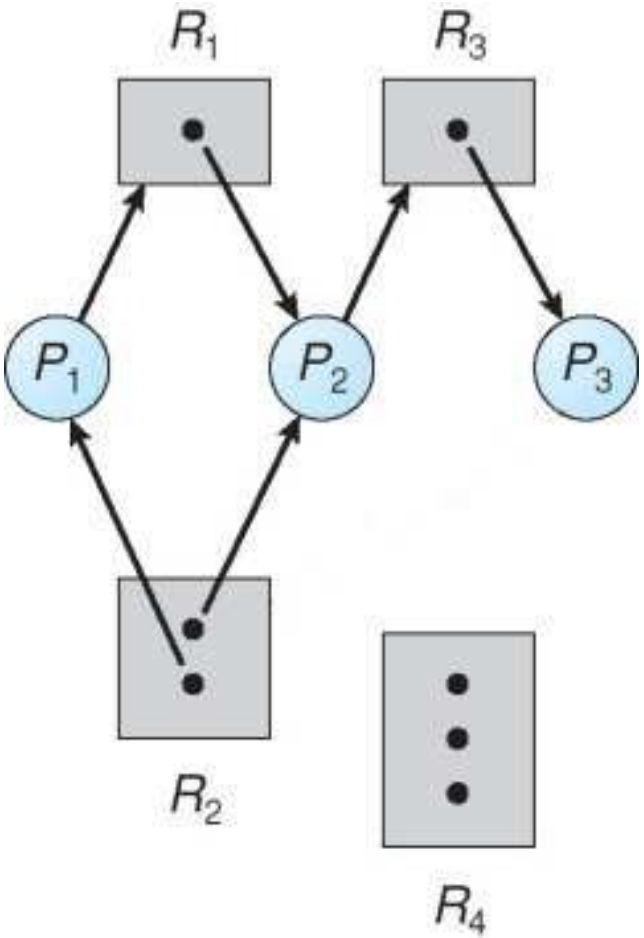
A graph

- a set of vertices υ
- a set of edges ϵ

System Resource Allocation Graph

- Active Processes $P = \{P_1, P_2, \ldots, P_n\}$ (Circles)
- Resource Types $R = \{R_1, R_2, \ldots, R_m\}$ (Boxes)
- Request edges ($P_i \rightarrow R_j$)
- Assignment edges ($R_j \rightarrow P_i$)
- Number of instances per resource type

Cycle in the Graph?



Deadlock?

- No cycle $\Rightarrow$ No process is deadlocked.
- If cycle, deadlock may exist

Cycle in the Graph?

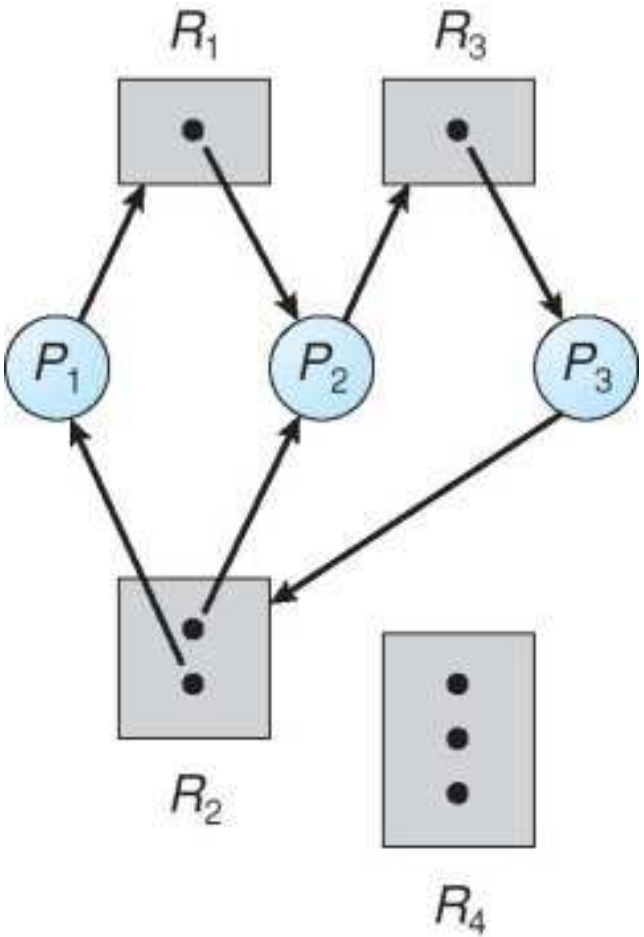
If each resource has ONE instance

- cycle $\Rightarrow$ deadlock
- Each process involved in the cycle is deadlocked
- Both necessary and sufficient condition for deadlock

If each resource has SEVERAL instance

- cycle $\not\Rightarrow$ deadlock
- Necessary but not sufficient condition for deadlock

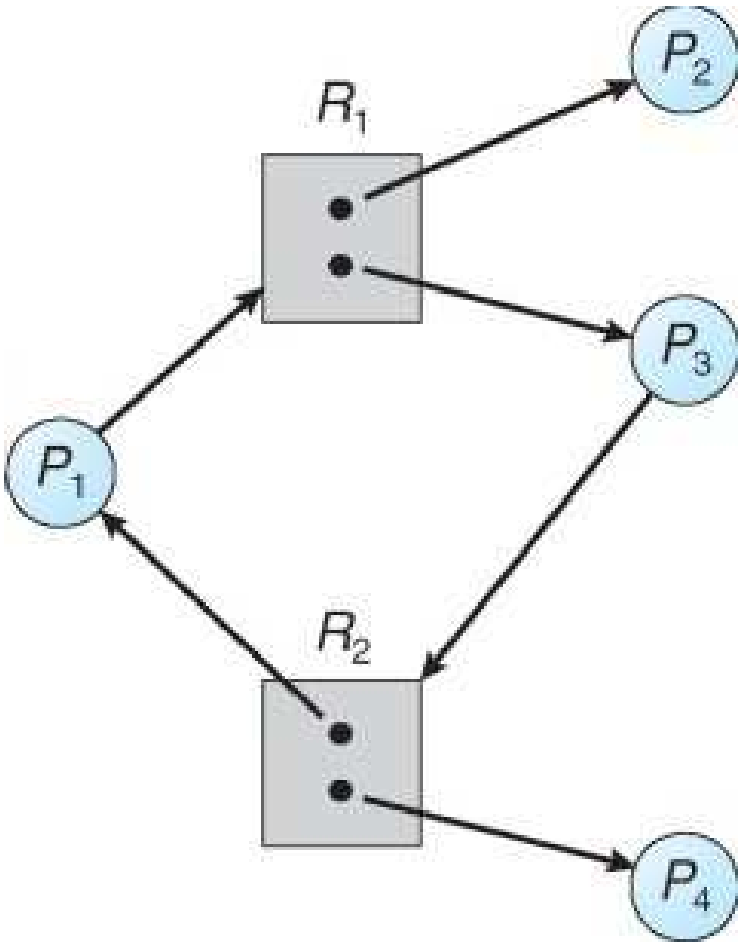
RAG example



- $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

P_1, P_2, P_3 are deadlocked

RAG example



$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

Not in deadlock state.

Important because we then can deal with deadlocks (Apply algorithms, induce choice,...)

p_4 may deallocate R_2

Conditions

Mutual exclusion

At least one resource must be nonsharable
(only one process can use it)

Hold and wait

At least one process holds at least one resource and waits for more resources which are held by other processes

No preemption

Only the process holding a resource can release it.

Circular wait

A set of processes are waiting for resources held by others in a circular manner
$< P_0, ..., P_n >$ where P_i waits for a resource held by $P_{i+1[n]}$.

How do we proceed?

1. Make sure they never occur
   - deadlock prevention
     (make sure at least one of the conditions above never hold)
   - deadlock avoidance
     (keeping more information about processes, allocate resources so that deadlocks cannot appear)
2. Deal with them
   - deadlock detection
   - deadlock recovery (work in pair)
3. Ignore them!

   if they don't appear too often, it's cheaper to restart the system/processes

Deadlock Prevention

Idea : Make sure one of the conditions is never satisfied.

Mutual exclusion

Difficult, since some resources cannot be shared (e.g. CPU).
For some resources it works fine (e.g. read-only files).

Deadlock Prevention

Hold and wait

Make sure that when a process requests a resource, it doesn't hold any other resources.

- allocate all resources *before* starting
  difficult to predict, waste of resources (low utilization)
- allow resource requests only when the process has none
  in some cases degenerates to the previous (e.g. printing files from two CDs directly after each other).

Risk for starvation for "popular" resources.

Deadlock Prevention

No preemption

Preempt (force release) allocated resources.

- Release all

  If the process holds a resource, requests another, and can't get it immediately, release all its resources (and have it wait for them all) - cf process scheduling!

- Release when waiting

  If the requested resources are not available, but allocated by a process which is waiting for some other resource, steal the resource (deallocate from waiting, allocate to requesting) and have the original holder wait for also this resource.

Solutions work for some resource types (e.g. CPU, memory - easy to store and reload state) but not for others (e.g. printers!)

Deadlock Prevention

Circular wait

Make a total ordering (<) of all resource types, and only allow requests in this order:

- If a process has R_1 and requests R_2, allow only if $R_1 < R_2$ (otherwise must deallocate R_1 first).
- If several instances of the same type are needed, must allocate all together.

Deadlock Avoidance

Deadlock prevention can lead to low resource utilization and reduced throughput.

If the system knows which resources will be requested by each process, and in which order, the system can order the requests so that deadlocks can not occur.

Yet difficult to describe, we could require *extra* information, to take better decision.

Deadlock Avoidance

Easiest solution: each process declares the maximum number of each resource type it will need (e.g. put in the PCB).

At each request, the system dynamically checks the resource-allocation state to make sure the circular wait condition will not be satisfied if the request is granted.

Deadlock Avoidance – Safe state

Algorithm idea

preserve a safe state of resource allocations.

safe state: if all processes can be given their maximum resource allocation (in some order) and avoid deadlock.
The order of allocations is a safe sequence:
$< P_1, P_2, ..., P_n >$ is a safe sequence for the current allocation state if

- for each P_i, its maximum resource requests can be satisfied using its current resources **plus** the resources held by all P_j where $j < i$.
- In order to use the resources of P_j, P_i may have to wait until they terminate, but P_i **can** get its resources!
- When P_i terminates, P_{i+1} can get its resources, etc.

Deadlock Avoidance – Safe state

If there is no safe sequence, the state is unsafe. An unsafe state may lead to deadlock, but doesn't have to. A safe sequence never leads to deadlock.

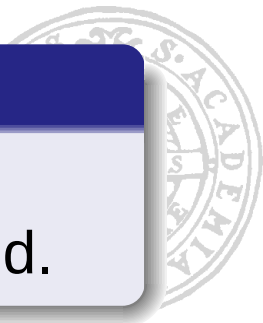
Refined algorithm idea

Allow only allocations which lead to a safe state.

- a process which requests a resource which is available may have to wait!
- Thus, the resource utilization may be less than optimal - but there will be no deadlocks.

Concrete algorithm

The Banker's Algorithm (section 7.5.3).
Never allocate the cash so customers can't get all they need.



Deadlock Detection

Need both detection and recovery.

- Detection algorithm are quite complex
  Take time and resources.
- Typically too expensive to do at each allocation

Run periodically or when CPU utilization goes down (which could be an indication of a deadlock).

How often? Depends on how often deadlocks appear: often means we should check frequently, otherwise the set of waiting process will grow and the cost of recovery will be larger.

Deadlock Detection

One instance of each resource

We can use resource allocation graphs (or wait-for graphs constructed from these) and check for cycles.

Need to *maintain* the RAG/WFG, and periodically *invoke* an algorithm: Overhead.

More instances

Use variant of Banker's algorithm (sec 7.6.2).

Deadlock Recovery

In the detection phase, we found which processes were involved in the circular wait.

Idea

By terminating processes, their resources are reclaimed by the system
$\Rightarrow$ the deadlock disappears

- Abort all deadlocked processes

  Very expensive, since many processes will have to recompute what they had done.
- Abort one process at a time until deadlock is broken

  Overhead: after each termination, re-run detection algorithm.

Deadlock Recovery – Process Termination

Both methods may result in e.g. inconsistent data in files.

If we abort one at a time, we must also consider which process to terminate (and at each stage) depending on the cost.

Example considerations:

- Process priority - especially if users pay to get priority
- Accumulated runtime - shorter runtime means less recomputations
- Current resource allocation - number (many means more resources will be returned to the system) and type (e.g. preemtable?)
- Future resource requirements

Deadlock Recovery – Resource preemption

Idea

Instead of terminating process,
Try preemting resources (by force releasing) until deadlock is broken.

1. **Selecting a victim**:
   Which resources? Which processes?
   $\Rightarrow$ minimize costs.
2. **Rollback**:
   The process must be able to continue running *normally*
   May require *backing* the process to a *safe state*
   safe state computation is difficult $\Rightarrow$ Total roll back.
3. **Starvation**:
   Avoid preempting/rolling back the same process again and again - cf priority scheduling and cost considerations.