Deadlock

Stepping on each other's feet - I

Thread T_1

```
b1:= allocate();
b2:= allocate();
……
release(b1);
release(b2);
```

Thread T_2

```
b1:= allocate();
b2:= allocate();
……
release(b1);
release(b2);
```

A serial execution is ok, but interdigitation can cause deadlock!

Stepping on each other's feet - II
The TENEX case

- If a process requests all systems buffers, operator console tries to print an error message
- To do so
  - lock the console
  - request a buffer

DUH!

Stepping on each other's feet - III

Thread T_1

```
open(f1, write);
open(f2, write);
    ......
close(f1,f2);
```

Thread T_2

```
open(f2, write);
open(f1, write);
    ......
close(f1,f2);
```

A serial execution is ok, but interdigitation can cause deadlock!

Issues

- How can we tell if a system is deadlocked?
- How can a deadlocked system recover?
- How can deadlocks be prevented in the first place?

Model

- A set of resources
  - e.g. files, buffers, *semaphores*?
- System is defined by:
  - a set of *states* Σ
    - each state represents the allocation status of the resources
  - a set of processes Π
    - each process p_i can request, acquire, release resources
    - each p_i can be described by the set of state transitions it can generate (p_i is a partial function from Σ to 2^{Σ})

Transitions

- If p_i can change the state of the system from σ_1 to σ_2 we write $\sigma_1 \rightarrow_i \sigma_2$
- We write $\sigma_a \rightarrow_* \sigma_b$ if σ_b can be reached from σ_a, i.e. if either:
  - $\sigma_a = \sigma_b$
  - $\exists\, p_i : \sigma_a \rightarrow_i \sigma_b$
  - $\exists\, \sigma_c , p_i : \sigma_a \rightarrow_i \sigma_c$ and $\sigma_c \rightarrow_* \sigma_b$

Key Definitions

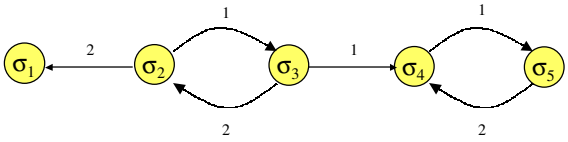
Blocked: p_i is *blocked* in σ_a if $\neg\exists\, \sigma_b : \sigma_a \rightarrow_i \sigma_b$

Deadlocked: p_i is *deadlocked* in σ_a if $\forall \sigma_b : \sigma_a \rightarrow_* \sigma_b$, p_i is *blocked* in σ_b

Deadlock state: σ_a is a *deadlock state* if $\exists\ p_i : p_i$ is deadlocked in σ_a

Safe state: σ_a is a *safe state* if $\forall\ \sigma_b : \sigma_a \rightarrow_* \sigma_b : \sigma_b$ is not a deadlock state

Example I



$\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5\}$ $\Pi = \{p_1, p_2\}$

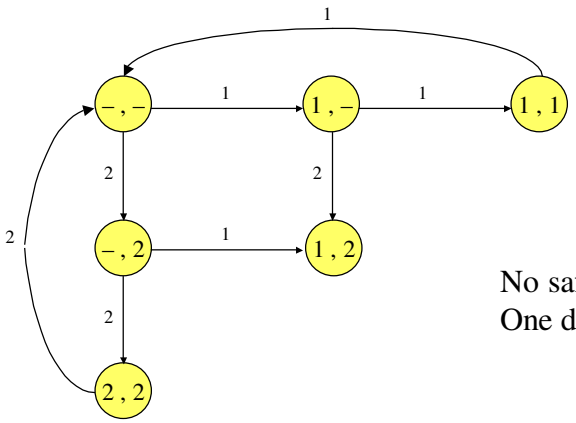
$p_1 : \{ (\sigma_2, \{\sigma_3\}) , (\sigma_4, \{\sigma_5\}) , (\sigma_3, \{\sigma_4\}) \}$

$p_2 : \{ (\sigma_2, \{\sigma_1\}) , (\sigma_3, \{\sigma_2\}) , (\sigma_5, \{\sigma_4\}) \}$

- σ_1 : p_1 deadlocked, p_2 deadlocked: deadlock state
- σ_2 : not safe (σ_1 deadlocked), not deadlock
- σ_3: not safe, not deadlock
- σ_4: p_2 blocked, safe state
- σ_5: p_2 blocked, safe state

Example 2

File allocation example from before:



No safe states
One deadlock state

Necessary Conditions for Deadlock

- Mutual exclusion
- Hold and Wait
- No preemption
- Circular waiting

How to deal with deadlock

Two basic approaches:

- *detect* deadlock and abort (breaks *no-preemption*)
- *prevent* deadlock (breaks one of the other conditions)

Reusable Resource Graphs

Direct bipartite graphs

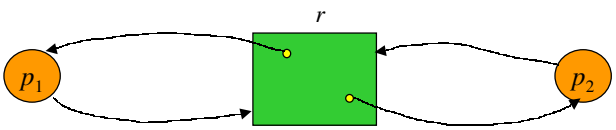
- process nodes are circles
- resource nodes are squares
- single resources as circles within the corresponding resource node square

- t_j = total number of resources at resource node r_j
- $|(a,b)|$ = number of edges from a to b

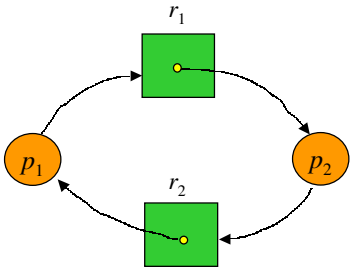
Two requirements:

- $\sum_i |(r_j, p_i)| \leq t_j$ (can't allocate more than available)
- $\forall\ i,j : |(p_i, r_j)| + |(r_j, p_i)| \leq t_j$ (no stupid deadlocks)

Examples



Requesting buffers



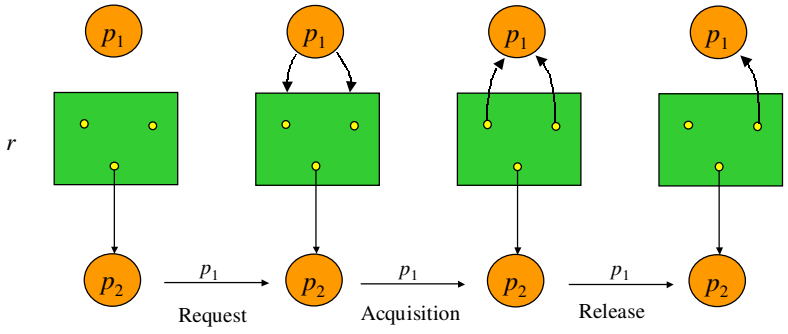
Allocating files

Evolving the Resource Graph

A resource graph changes as the state of the system changes:

- p_i requests r_j
  - p_i must have no outstanding requests
  - p_i may request multiple units (say u_i)
  - add u_i edges from p_i to r_j
- acquisition
  - p_i must have outstanding requests on r_j, and all such requests must be satisfiable
  - reverse the direction of all edges
- release
  - p_i must have no outstanding requests
  - p_i may release any nonempty set of resources
  - erase the appropriate subset of edges from r_j to p_i

Example



Deadlock Detection

- Deadlock implies that there is *no* sequence of state transitions that would allow processes to make progress
- Then, as long as there exists *one* sequence of actions through which some process can make progress, there is no deadlock

IDEA:

- Simulate the *most favorable* execution of each unblocked process. If that execution exibits deadlock, all will
- Simulate by *reducing* resource graph

Reducing the resource graph

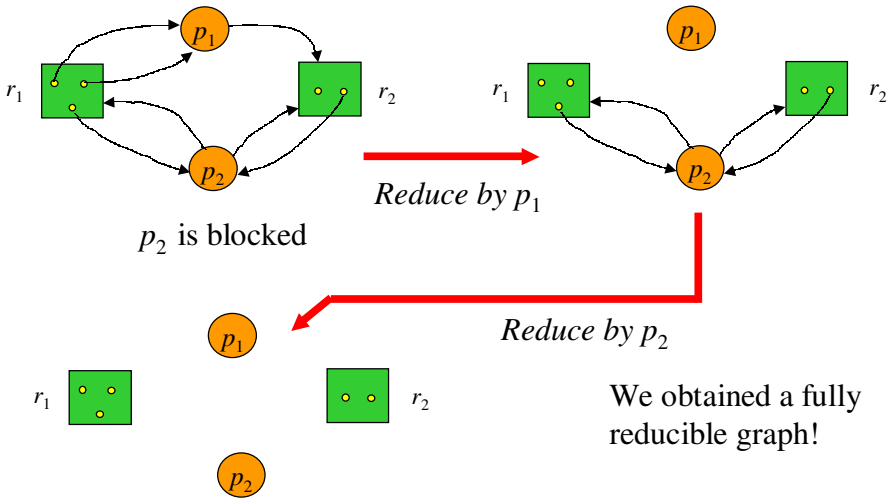
A process p_i is blocked if

$$|(p_i , r_j)| + \sum_k |(r_j , p_k)| > t_j \quad \text{for all } r_j$$

Reduction Step

- Find unblocked p_i which is not isolated
- Erase all edges incident on p_i : this is equivalent to acquiring and releasing all p_i's resources
- When there is no such p_i, the graph is *irreducible*
- An irreducible graph that contains no edges is *completely reducible*

Example



Deadlock Theorem

Theorem

A resource graph represents a deadlock state if and only if it is not completely reducible.

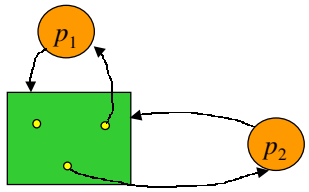
Is this really useful?

Yes, because, fortunately, the order in which reductions are performed does not affect the final outcome of the reduction process!

Observations: 1

- If deadlock, then cycle in resource graph

but the opposite is not true!

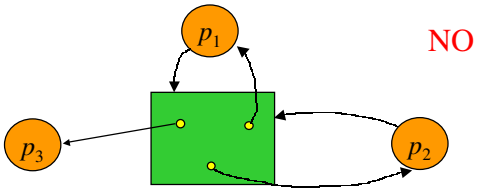


The resource node had some of its resources non allocated: what if all resources had been allocated?

Observations: 2

A graph is *expedient* if all of its resources have been allocated.

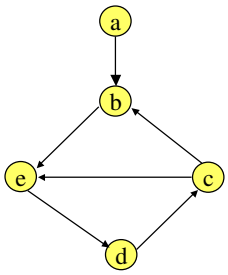
Is a cycle in an expedient graph a sufficient condition for deadlock?

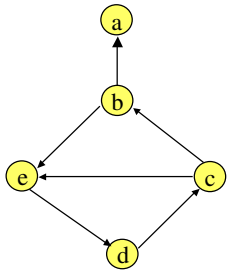


NO

Knots

A *knot* is a set of nodes that are mutually reachable from each other, and no other node is reachable from them.

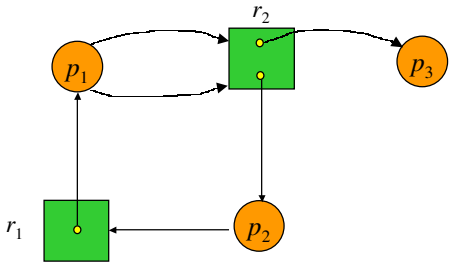




Observations: 3

- If expedient + knot, then deadlock

but not necessary!



Expedient, no knot, but deadlock!

Observations: 4

If resources are single unit, ($\forall j : t_j = 1$) then deadlock *iff* cycle

OK, we detected the deadlock: then, what?

- Select a victim and either
  - terminate the victim
    - costly
    - how is the victim selected?
  - preempt its resources
    - rollback
    - starvation

Deadlock Prevention - 1

Must make sure that at least one of the 4 necessary conditions does not hold

❶ Mutual Exclusion
- difficult to avoid: no collaboration!
- sometime possible (read-only files)

❷ Hold and Wait
- Force each process to allocate all resources at the beginning
- Allow a process to request resources only if it has none
  - low resource utilization
  - possible starvation for processes asking popular resources

Deadlock Prevention - 2

❸ No preemption
- preempt all resources held while waiting on a resource
- when requesting for a resource:
  - if it is available, allocate it
  - if it is not available, but it is held by another process that is waiting, preempt the resource

❹ Circular wait
- impose a total order on all resource types
- require each process to request resources in strictly increasing order

Deadlock Avoidance

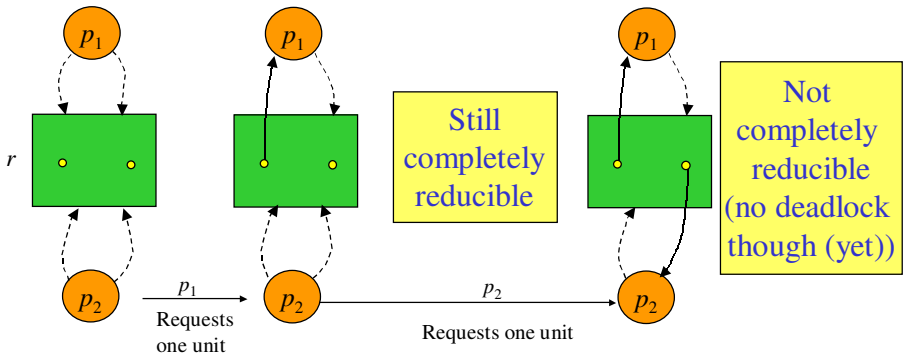
Determine whether, by allowing allocation, we could get a deadlock, i.e. check if by allowing allocation the system could enter a state that is not safe

Simple way to do it:

have each process declare the maximum number of resources of each type that it may need

Claims Graph

Claim graph: Resource graphs that also contains edges that represent *potential* requests (each process needs to specify max claim to each resource)



Banker's Algorithm (Dijkstra & Habermann)

Variables:

available[i]: units of r_i not allocated

c[a,i]: max claim of p_a on r_i

alloc[a,i]: number of units of r_i allocated to p_a

need[a,i]: number of potential requests: c[a,i] - alloc[a,i]

finish[i]: one entry per process, true if process can finish

work[i]: one entry per process

Let p_a attempt to allocate *request* units of r_i

```
if request > need[a,i] {error};
if request > available[i] {wait};
available[i] := available[i] - request;
alloc[a,i] := alloc[a,i] + request;
need[a,i] := need[a,i] - request;
if (not safe()) {undo request and wait};

safe()
    work[i] := available[i];
    while (∃ u: not finish[u] and
      (need(u,i) ≤ work[i])) {
            work[i] := work[i]+alloc[u];
            finish[u]:= true;
    }
    safe := ∀u: finish[u]
```