TTIT62 Real-time Process Control

Lecture 6: Deadlock

Simin Nadjm-Tehrani

Real-time Systems Laboratory
Department of Computer and Information Science
Linköping university

Recall from last lecture

- The ICP prevents deadlocks (How?)
- Moreover, it prevents starvation (How?)

Motivation

In a real-time system **liveness** is necessary – i.e. no presence of deadlock, starvation or livelock.
If this can be guaranteed then the system is live.

But not sufficient...

This lecture

- How immediate ceiling protocol prevents deadlock and starvation?
- But first some general review of deadlock related concepts...

---

4 necessary conditions

1. Mutual exclusion
Access to resource is limited to one (or a limited number of) process(es) at a time

2. Hold & wait
There are processes that hold a resource and wait for another resource(s) at the same time

---

3. Voluntary release
Resources can only be released by a process voluntarily

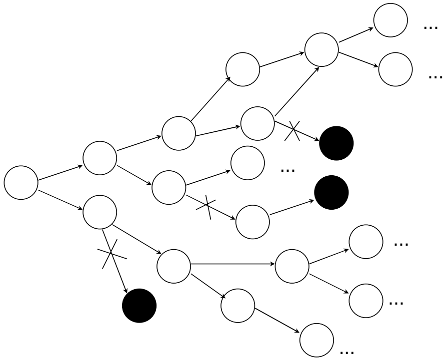
4. Circular wait
There is a chain of processes where each process holds a resource that is required by another resource

Deadlock elimination

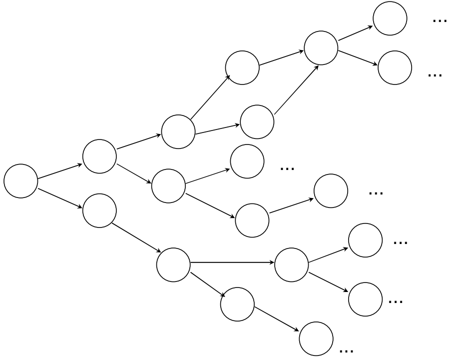
Repetition from OS course:

- Deadlock avoidance
- Deadlock prevention
- Deadlock detection and treatment

Deadlock avoidance

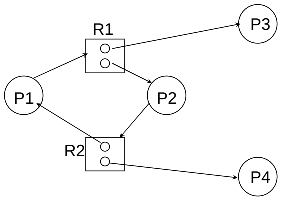


Deadlock prevention

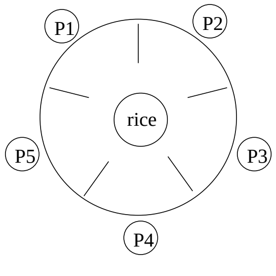


Detection and fixing

By building a dynamic resource allocation graph to detect deadlocks



Classic example



```
process Philosopher;
    loop
        think;
        <pick up left chopstick>
        <pick up right chopstick>
        eat;
        <put down right chopstick>
        <put down left chopstick>
    end loop
end;
```

Prevention / avoidance

- Avoidance
  - e.g. using banker's algorithm
- Prevention
  - e.g. allocate all necessary resources at once, before execution

Can lead to starvation!

Starvation

Starvation/lockout happens if some process never gets hold of the resources it needs despite the fact that the resources are not constantly engaged

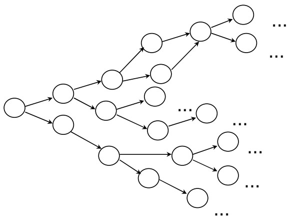
```
process Philosopher;
    loop
        think;
        <pick up left and right
        chopsticks if free>
        eat;
        <put down left and right
        chopsticks>
    end loop
end;
```

Consider following scenario

P1 wants to eat, takes left & right stick
P3 wants to eat, takes left & right stick
P2 wants to eat, must wait
P1 releases left & right stick
P1 thinks
P1 wants to eat, takes left & right stick
P3 releases left & right stick
P3 thinks
P3 wants to eat, …
…

Now back to scheduling…

- Immediate ceiling protocol (ICP) is deadlock preventing



Moreover…

- ICP prevents starvation (How?)

With no hard deadlines

- Banker's Algorithm: Technique for deadlock avoidance in presence of sharing multiple resources
- Question: Do you want an example run of Banker's algorithm?
  - Can be a topic for the resource session
  - Here is a few summary slides

Banker's algorithm

- Allocate multiple resources as and when processes ask for it, but only:
  - up to a predefined max value for each process and resource
  - provided that remaining resources together with potential future releases are enough for future allocations (up to the max value)

Implementation

For n processes and m resources we need following data structures:

`Max`: $n \times m$ matrix

`Max[i,j] = k` means that process i requires max k elements of resource type j

---

`Allocation`: $n \times m$ matrix

`Allocation[i,j] = k` means that process i has already been allocated k elements of resource type j

`Available`: m vector

`Available[i] = k` means that k elements of resource type i are available for allocation

---

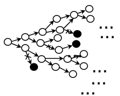
$Request_i$: m vector

process i:s request for resources

Notation:

$Allocation_i$: the i-th row in the `Allocation` matrix

State: instantiations of `Allocation`



Banker's algorithm

Input:
Matrix `Max`, vector `Available`,
a given state, and
$Request_i$ from some process i
Output:
Yes + new state, or
No + unchanged state
($Request_i$ can not be allocated now)

Algorithm:

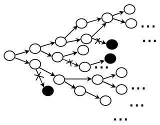
1. Need := Max - Allocation;
2. Check if
   $Request_i \leq Available$
   if not, return ”No”.
3. Pretend that resources in
   $Request_i$ are to be allocated,
   compute new state.

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

---

4. Test if the new state is
is deadlock-avoiding, in which
case return ”Yes”.

Otherwise, return ”No” -
roll back to the old state.



Testing for deadlock-avoidance

Start with a given *Allocation*
and check if it is deadlock-avoiding
According to the 3-step
algorithm below.

Finish: n vector with Boolean
values (initially false)

Work : m vector denotes
the changing resource set as
the processes become ready and release
resources (initially $Work := Available$)

1. Check if there is some process i
for which $Finish_i = false$ and
for which $Need_i \leq Work$. If there is no such
process i, go to step 3.

---

2. Free the resources that i has used to
get finished:
$Work := Work + Allocation_i$
$Finish_i := true$
continue from step 1.

3. If $Finish_i = true$ for all i then
the initial state is deadlock-avoiding,
otherwise it is not.