Earlier...

We have seen that
- Some solutions to mutual exclusion problem led to livelocks and deadlocks
- This lecture: Returning to deadlocks and dealing with them

Four necessary conditions for deadlock:

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

2. Hold & wait
   A process may hold a resource and wait for another resource 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

Three approaches:
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and treatment
Deadlock avoidance

Prevention and Avoidance

- Prevention
  - e.g. allocate all necessary resources at once, before execution

- Avoidance
  - e.g. using banker’s algorithm

Detection and fixing

Build a dynamic resource allocation graph to detect deadlocks

Sufficient condition

- With one resource instance of each type of resource
- Absence of cycles in the graph guarantees absence of deadlock

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

- Prevention
  - e.g. allocate all necessary resources at once, before execution

- Avoidance
  - e.g. using banker’s algorithm

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, ...
...

Starvation

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

Other approaches

- Can build deadlock prevention techniques in the scheduling algorithm
- We will return to this in the next course (real-time systems)
Prevention/avoidance

- Prevention
  - e.g. allocate all necessary resources at once, before execution

- Avoidance
  - e.g. using banker’s algorithm

Banker’s algorithm

- Upon each process request: Allocate multiple resources that the process requests, provided that:
  - the request is up to a predefined max value for each process and resource (cumulatively)
  - after each granting, the remaining resources together with potential future releases, are enough for all future allocations (up to the max values)

Rejecting a request

- When allocating a request does not lead to a new "safe" state:
  - Refuse to grant

- The request can be repeated in some future state and get granted

Implementation

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

- **Max**: \( n \times m \) matrix

  \( \text{Max}[i,j] = k \) means that process \( i \) requires \( k \) elements of resource type \( j \)

- **Allocation**: \( n \times m \) matrix

  \( \text{Allocation}[i,j] = k \) means that process \( i \) has already been allocated \( k \) elements of resource type \( j \)

- **Available**: \( m \) vector

  \( \text{Available}[i] = k \) means that \( k \) elements of resource type \( i \) are available for allocation

- **Request**: \( m \) vector

  process \( i \)'s request for resources

  Notation:

  \( \text{Allocation}_i \): the \( i \)-th row in the Allocation matrix

  **State**: instantiations of Allocation
The algorithm

Input:
- Matrix Max, vector Available, a given state Allocation, and Request for some process $i$ (* Request$_i$ <= Max$_i$ - Allocation$_i$ *)

Output:
- Yes + new state, or
- No + unchanged state (Request$_i$ can not be allocated now)

Algorithm:
1. Need := Max - Allocation
2. Check whether Request$_i$ <= 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 whether the new state is deadlock-avoiding (denoted safe), in which case return “Yes”.
   - Otherwise, return “No” - roll back to the old state.

Testing for safe state

Start with a given Allocation and check if it is safe (avoids future deadlocks) according to the 3-step algorithm below.

Illustration

- We’ll use an example to clarify how testing for the safe state test works!

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$_i$ <= Work. If there is no such process $i$, go to step 3.
2. Free the resources that $i$ has used to get finished:
   \[ \text{Work} := \text{Work} + \text{Allocation}_i \]
   \[ \text{Finish}_i := \text{true} \]
   continue from step 1.

3. If $\text{Finish}_i = \text{true}$ for all $i$ then the initial state is deadlock-avoiding, otherwise it is not.

Summary

- Liveness: Absence of deadlock, starvation or livelock
- We have looked at ways to implement live systems
- We will see in a future course that being live is necessary but not sufficient in real-time systems

Questions?