Karlsruhe Institute of Technology

Operating Systems

14. Deadlocks

Prof. Dr. Frank Bellosa | WT 2020/2021

Dining-Philosophers Problem

- Cyclic workflow of 5 philosophers
  1. Think
  2. Get hungry
  3. Grab for one chopstick
  4. Grab for other chopstick
  5. Eat
  6. Put down chopsticks
- Ground rules
  - No communication
  - No “atomic” grabbing of both chopsticks
  - No wrestling

- Models threads competing for limited number of resources (e.g., I/O devices) [TB15]

Dining-Philosophers Problem

- Naïve solution with **`mutex_t chopstick[5]`** representing the chopsticks
  - What happens if all philosophers grab their left chopstick at once?

```
void philosopher( int i )
{
   for(;;) // ever
   {
      mutex_lock( chopstick[i] );
      mutex_lock( chopstick[(i + 1) % 5] );
      //  eat
      mutex_unlock( chopstick[i] );
      mutex_unlock( chopstick[(i + 1) % 5] );
      //  think
   }
}
```

- Deadlock workarounds
  - Just 4 philosophers allowed at a table of 5 (example for deadlock avoidance)
  - Odd philosophers take left chopstick first, even philosophers take right chopstick first (example for deadlock prevention)

Deadlock Conditions

Deadlocks can arise if all four conditions hold simultaneously:

1. Mutual exclusion
   - Limited access to resource
   - Resource can only be shared with a finite amount of users
2. Hold and wait
   - Wait for next resource while already holding at least one
3. No preemption
   - Once the resource is granted, it cannot be taken away but only handed back voluntarily
4. Circular wait
   - Possibility of circularity in graph of requests

Example: Deadlock Conditions

1. Only one intersection
2. Cars block part of the intersection while waiting for the rest
3. Cars don't diminish into thin air
4. Every one of the four sides waits for the cars that come from the right to give way

Deadlock countermeasures

Three approaches to dealing with deadlocks:

- Prevention
  Pro-active, make deadlocks impossible to occur
- Avoidance
  Decide on allowed actions based on a-priori knowledge
- Detection
  React after deadlock happened (recovery)

Deadlock Prevention

Negate at least one of the required deadlock conditions:

1. Mutual exclusion
   - Buy more resources, split into pieces, virtualize $\rightarrow$ "infinite" # of instances
2. Hold and wait
   - Get all resources en-bloque
     $\rightarrow$ release all locks befor acquiring all locks necessary for next phase
   - Problem: resource utilization and starvation
3. No preemption
   - Virtualize to make preemptable
     (applicable for resources that can be saved and restored)
     - virtual vs. physical memory
4. Circular wait
   - Ordering of resources
     - E.g., mutex m_1 must be acquired before m_2

Deadlock Prevention

Negate at least one of the required deadlock conditions:

1. Mutual exclusion
   - Buy more resources, split into pieces, virtualize $\rightarrow$ "infinite" # of instances
2. Hold and wait
   - Get all resources en-bloque
     $\rightarrow$ release all locks befor acquiring all locks necessary for next phase
   - Problem: resource utilization and starvation
3. No preemption
   - Virtualize to make preemptable
     (applicable for resources that can be saved and restored)
     - virtual vs. physical memory
4. Circular wait
   - Ordering of resources
     - E.g., mutex m_1 must be acquired before m_2

Deadlock Prevention

Negate at least one of the required deadlock conditions:

1. Mutual exclusion
   - Buy more resources, split into pieces, virtualize $\rightarrow$ "infinite" # of instances
2. Hold and wait
   - Get all resources en-bloque
     $\rightarrow$ release all locks befor acquiring all locks necessary for next phase
   - Problem: resource utilization and starvation
3. No preemption
   - Virtualize to make preemptable
     (applicable for resources that can be saved and restored)
     - virtual vs. physical memory
4. Circular wait
   - Ordering of resources
     - E.g., mutex m_1 must be acquired before m_2

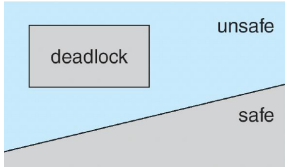
Deadlock Prevention

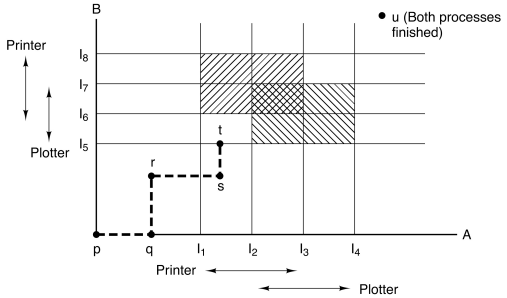
Negate at least one of the required deadlock conditions:

1. Mutual exclusion
   - Buy more resources, split into pieces, virtualize $\rightarrow$ “infinite” # of instances
2. Hold and wait
   - Get all resources en-bloque
     $\rightarrow$ release all locks befor acquiring all locks necessary for next phase
   - Problem: resource utilization and starvation
3. No preemption
   - Virtualize to make preemptable
     (applicable for resources that can be saved and restored)
     - virtual vs. physical memory
4. Circular wait
   - Ordering of resources
     - E.g., mutex m_1 must be acquired before m_2

Deadlock Avoidance

- If a system is in safe state
  → no deadlocks
- If a system is in unsafe state
  → deadlocks possible
  [SGG12]
- Deadlock Avoidance
  On every resource request:
  decide if system stays in
  safe state
  - Needs a-priori information
    (e.g., max resources needed) [TB15]

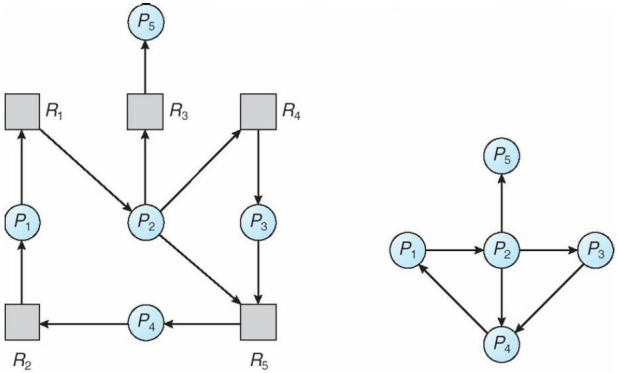




Deadlock Detection

Allow system to enter deadlock → detection → recovery scheme

- Maintain Wait-For Graph (WFG)
  - Nodes are processes
  - Edge represents “wait for” relationship



Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim
  - Minimize cost
- Rollback
  - Perform periodic snapshots
  - Abort process to preempt resources

  → Restart process from saved state
- Starvation
  - Same process may always be picked as victim

  → Include number of rollbacks in cost factor

References I

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 4th edition, 2015.