Deadlock

CS 537 Lecture

Deadlock: Why does it happen?

- When all entities (threads, processes) are waiting for a resource held by some other entity in a group
- None will release what they hold until they get what they are waiting for

Example: Unordered Mutex

- Two threads accessing two locks
  - Semaphore m[2] = {1,1}; // binary semaphore
  - Thread1 m[0].P(); m[1].P();
  - m[1].P(); m[0].P();
  - // access shared data
  - m[1].V(); m[0].V();
  - m[0].V(); m[1].V();
- What happens if Thread1 grabs m[0] and Thread2 grabs m[1]?  

Representing Deadlock

- “Waits-for” graph
- “Resource-allocation” graph

Four Necessary Conditions

- Mutual exclusion
  - >=1 resource held non-sharable
  - requests delayed until release
- Hold and wait
  - Exists a process that is holding >=1 resource, waiting for another that is held by some other process
- No preemption
  - Resources only released voluntarily
- Circular wait
  - Exists set of processes s.t. P0->P1->...->Pn->P0
- All 4 conditions must hold for deadlock to occur!

Handling Deadlock: Options

- Prevention
  - Ensure system never enters deadlock
  - (make sure >= 1 condition does not hold)
- Detection/Recovery
  - Allow deadlocks, but detect!
  - Somehow recover and continue
- Ignore
  - Fairly common approach, seems bad
  - (When could this be the right solution?)
**Prevention: Stopping 1 of 4**

1. **Mutual exclusion**
   - If not required, do not use (e.g., read-only file)
   - (but, sometimes needed, of course)
2. **Hold and wait**
   - Guarantee all P’s grab resources at once
   - (must be done atomically)
   - Why is this a bad idea (sometimes)?

**Prevention (cont.)**

3. **No preemption**
   - If holding some resources and trying to get others, must wait (could be a problem)
   - Instead, force others to release!
   - Why is this hard to do (in general?)
     - Must undo state of P that is preempted

**Prevention (cont.)**

4. **Circular wait**
   - Impose total order on locks
   - If all P’s follow order, no circular wait occurs
     - E.g., Locks M1, ..., Mn acquired in order only!
   - Advantages: Simple to follow, works
     - Common in practice
   - Disadvantages: Arbitrary ordering

**Avoidance**

- Different than prevention
  - By having knowledge of what processes will request, can schedule carefully so as to avoid deadlock
  - Must know maximal requests possible/process
  - E.g., Banker’s algorithm
  - Not commonly used: too much knowledge

**Detect and Recover**

- Detection
  - Notice waiting processes and dependencies
  - Inform human, or handle automatically
  - Might be expensive, so run infrequently
- Recovery: Abort processes!
  - Abort all that are deadlocked (good/bad?)
  - Abort one at a time until deadlock doesn’t exist
  - Why hard?
    - Must undo effects of process (lock1, remove $$ from account, lock2, put $$ in other account, release 2, release 1)
    - Could starve if repeatedly aborted (one that gets most locks)

**Summary**

- **Deadlock**
  - Mutual exclusion, Hold and wait, No preemption, and circular wait all required
- **Solve by**
  - Preventing one of four conditions
  - Avoidance via clever scheduling
  - Detect and recover by aborting processes
  - Ignoring altogether!