Concurrent Errors

Arnav Gupta

December 8, 2024

Contents

1 Race Condition

Race condition occurs when missing synchronization or mutual exclusion.

Happens when 2+ tasks race along assuming synchronization or mutual exclusion has occurred.

Can be very difficult to locate.

2 No Progress

2.1 Live-Lock

Indefinite postponement, "You go first" problem on simultaneous arrival.

Caused by poor scheduling in entry position.

Can always break time on simultaneous arrival by some mechanism that deals effectively with live-lock.

2.2 Starvation

Selection algorithm ignores 1+ tasks so they are never executed (lack of long-term fairness).

Long-term (infinite) starvation extremely rare, but short-term starvation can occur and is a problem.

Like live-lock, starving task might be ready any time, switching among active, ready, and possibly blocked states (consuming CPU).

2.3 Deadlock

State when 1+ processes wait for an event that will not occur.

Unlike live-lock/starvation, deadlocked task is blocked so not consuming CPU.

2.3.1 Synchronization Deadlock

Failure in cooperation, so blocked task never unblocked (stuck waiting).

```
int main() {
  uSemaphore s(0);  // closed
  s.P();            // wait for lock to open
}
```

2.3.2 Mutual Exclusion Deadlock

Failure to acquire resource protected by mutual exclusion.

5 conditions that must occur for a set of processes to deadlock:

1. concrete shared resource requiring mutual exclusion
2. process holding a resource while waiting for access to a resource held by another process (hold and wait)
3. once a process has gained access to a resource, runtime system cannot get it back (no preemption)
4. exists a circular wait of processes on resources
5. conditions must occur simultaneously

3 Deadlock Prevention

Eliminate 1+ conditions required for deadlock from an algorithm so deadlock can never occur.

3.1 Synchronization Prevention

Eliminate all synchronization from a program:

- no communication
- impossible in most cases

3.2 Mutual Exclusion Prevention

Prevent deadlock by eliminating 1 of 5 conditions:

1. no mutual exclusion
   (a) no shared resources, so impossible in most cases
2. no hold and wait: do not given any resource unless all resources can be given
   (a) poor resource utilization and possible starvation
3. allow preemption
   (a) preemption is dynamic, so cannot apply statically
4. no circular wait: by controlling order of resource allocations
   (a) use an **ordered resource** policy:
      i. divide all resources into classes $R_1, R_2, \ldots$

ii. rule: can only request a resource from class R_i if holding no resources from any class R_j for $j \geq i$

iii. unless each class contains only 1 resource, requires requesting several resources simultaneously

iv. denote highest class number for which T holds a resource by $h(T)$

v. if process T_1 is requesting a resource of class k and is blocked because that resource is held by process T_2, then $h(T_1) < k \leq h(T_2)$

vi. as preceding inequality is strict, circular wait is impossible

vii. in some cases, there is natural division of resources into classes that makes policy work nicely

viii. in other cases, some processes are forced to acquire resources in an unnatural sequence, complicating code and producing poor resource utilization

5. prevent simultaneous occurrence

   (a) show previous 4 rules cannot occur simultaneously

4 Deadlock Avoidance

Monitor all lock blocking and resource allocation to detect and potential formation of deadlock.

Achieve better resource utilization, but additional overhead to avoid deadlock.

4.1 Banker's Algorithm

Demonstrate safe sequence of resource allocations that give no deadlock. Requires a process state its maximum resource needs.

Check for safe order of execution that avoids deadlock should each process require maximum resource allocation.

If safe order exists, Banker's algorithm allows resource request.

If there is a choice of processes to choose for execution, it does not matter which path is taken.

Check for safe order can be performed for every allocation of a resource to a process (optimizations possible, like same thread asking for another resource).

4.2 Allocation Graphs

One method to check for potential deadlock is to graph processes and resource usage at each moment a resource is allocated.

Multiple instances are put into a resource so that a specific resource does not have to be requested. Instead, a generic request is made.

If graph contain no cycles, no process in the system is deadlocked.

If any resource has several instances, a cycle does not mean deadlock.

Can also create **isomorphic graph** without multiple instances (expensive and difficult).

If each resource has one instance, a cycle means a deadlock.

Can use graph reduction to locate deadlocks.

Problems with allocation graphs:

- when choices for tasks, selection is tricky (like isomorphic graph)
- for large graphs, detecting cycles is expensive
- many graphs to examine over time, one for each particular allocation state of the system

5 Detection and Recovery

Instead of avoiding deadlock, let it happen and recover. This requires ability to discover deadlock and preemption.

Discovering deadlock is difficult, since must build and check for cycles in allocation graph. Not done on each resource allocation, but every interval of seconds or every time a resource cannot be immediately allocated.

Recovery involves preemption of 1+ processes in a cycle:

- decision not easy and must prevent starvation
- preemption victim must be restarted, from beginning or from previous checkpoint, if no guarantee that all resources have not changed

- still might not be enough, since victim may have made changes before preemption

6 Method to Choose

Might be best to ignore the problem:

- if some process is blocked for a long time, assume it is deadlocked and abort it
- do this automatically in transaction processing system, manually elsewhere

Of techniques studied, only ordered resource policy has much practical value.