Concurrency: Deadlock and Starvation

Arnav Gupta

March 26, 2024

Contents

1 Principles of Deadlock

Deadlock: permanent blocking of a set of processes that either compete for system resources or communicate with each other.

No efficient solution exists in the general case.

Deadlock can be illustrated with a joint process diagram, which illustrates the progress of two processes competing for two resources.

- if an execution path on a joint process diagram enters the fatal region, deadlock is inevitable

1.1 Consumable Resources

Consumable resource: one that can be created and destroyed (ex. interrupts, signals, messages, IO buffer info)

When such a resource is acquired by a consuming process, the resource ceases to exist.

Can lead to deadlock, such as conflicting order of messages.

Common approaches to deal with deadlocks:

- **deadlock prevention**: disallow one of the three necessary conditions for deadlock occurrence (or prevent circular wait condition from happening)
- **deadlock avoidance**: do not grant a resource request if the allocation might lead to deadlock
- **deadlock detection**: grant resource requests when possible, but check for presence of deadlock and take action to recover

1.2 Resource Allocation Grpahs

Resource allocation graph: directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node, edges indicating requests for resources not yet granted

- each dot indicates one instance
- graph edge directed from a consumable resource node dot to a process indicates the process is the producer of the resource

1.3 Conditions for Deadlock

Conditions of policy (1-3) must be present for a deadlock to be possible:

1. **mutual exclusion**: only one process may use a resource at a time, no process may access a resource unit that has been allocated by another process
2. **hold and wait**: process may hold allocated resources while awaiting assignment of other resources
3. **no preemption**: no resource can be forcibly removed from a process holding it
4. **circular wait**: closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Fatal region in joint progress diagram only exists if all of the first 3 conditions above are met, otherwise no fatal region or deadlock occur.

Conditions 1-3 represent possibility of deadlock, conditions 1-4 represent existence of deadlock.

2 Deadlock Prevention

Strategy is to design a system such that the possibility of deadlock is excluded.

Two classes of deadlock prevention methods:

- **indirect method**: prevent the occurrence of 1 of the 3 necessary conditions
- **direct method**: prevent the occurence of a circular wait

2.1 Mutual Exclusion

Mutual exclusion cannot be disallowed, since some resources require it.

2.2 Hold and Wait

Can be prevented by requiring a process to request all required resources at once, which is quite inefficient:

- a process may be held up with these resource requests even though it does not need them all
- allocating resources to this process denies them to other processes
- a process may not know in advance all resources it requires

2.3 No Preemption

Can be prevented by:

- if a process holding certain resources is denied more, that process must release its original resources and request all necessary resources (including those it was denied) again
- if a process requests a resource held by another process, the OS may preempt the other process and require it to release its resources

Only practical to prevent when applied to resources whose state can be easily saved and restored (like the processor).

2.4 Circular Wait

Can be prevented by defining a linear ordering of resource types, where resources can only be requested in the order specified.

Preventing this may be inefficient, slow processes and deny resource access.

3 Deadlock Avoidance

Avoidance allows the 3 necessary conditions for deadlock, but makes choices to assure that deadlock is never reached.

- allows more concurrency than prevention
- requires knowledge of future process resource requests

3.1 Process Initiation Denial

Consider a system of n processes and m types of resources.

Let $R = (R_1, R_2, \ldots, R_m)$ be the total amount of each resource in the system. Let $V = (V_1, V_2, \ldots, V_m)$ be the total amount of each resource not allocated to any process.

Let the claim C be

$$C = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$$

where C_{ij} is the requirement of process i for resource j.

Let the allocation A be

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$$

where A_{ij} is the current allocation to process i of resource j.

The following relationships hold:

1. All resources are either available or allocated.

$$R_j = V_j + \sum_{i=1}^{n} A_{ij}, \forall j$$

2. No process can claim more than the total amount of resources in the system.

$$C_{ij} \leq R_j, \forall i, j$$

3. No process is allocated more resources of any type than the process originally claimed to need.

$$A_{ij} \leq C_j, \forall i, j$$

Using these, a deadlock avoidance policy can be defined that refuses to start a new process if resource requirements might lead to deadlock. The policy is to only start a new process P_{n+1} if

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}, \forall j$$

which means to only start a process if the maximum claim of all current processes plus the new process can be met.

3.2 Resource Allocation Denial

Strategy of resource allocation denial is called banker's algorithm.

Consider a system with a fixed number of processes and a fixed number of resources. The state of the system reflects the current allocation of resources to processes, consisting of vectors R and V and matrices C and A. Safe state is when there is some sequence of resource allocations to processes that does not result in deadlock.

To find if a state is safe, check if any process can be run to completion with the resources available. For some process i, this requires

$$C_{ij} - A_{ij} \leq V_j, \forall j$$

Once such a process is found, assume it runs to completion and make its resources available. Begin again.

The deadlock avoidance strategy is to ensure that the system is always in a safe state. If some resource request will put the system in an unsafe state, block the process until it is safe to grant the request, otherwise grant it.

Deadlock avoidance strategy does not predict deadlock with certainty, it just anticipates the possibility and assures this possibility never occurs.

Deadlock avoidance has the advantage that it is not necessary to preempt and rollback processes and is less restrictive than deadlock prevention. Restrictions on its use are:

- maximum resource requirement for each process must be stated in advance
- processes under consideration must be independent (order of execution must be unconstrained by synchronization requirements)
- must be a fixed number of resources to allocate
- no process may exit while holding resources

4 Deadlock Detection

Prevention is conservative, detection finds the circular wait condition.

4.1 Deadlock Detection Algorithm

Check for deadlocks can be done on each resource request or less frequently. Checking at each resource request leads to early detection and simple algorithm, but uses more processor time.

The Allocation matrix and Available vector are used along with a request matrix Q where Q_{ij} represents the amount of resources of type j requested by process i.

Initially all processes are unmarked. The following steps are performed to mark processes that are not part of a deadlocked set:

1. Mark each process that has a row in the Allocation matrix of all zeros, as a process that has no allocated resources cannot participate in a deadlock.
2. Initialize a temporary vector W to equal the Available vector.
3. Find an index i such that process i is currently unmarked and the \$i\$th row of Q is at most W. That is $Q_{ik} \leq W_k$ for $1 \leq k \leq m$. If no such row is found, terminate the algorithm.
4. If such a row is found, mark process i and add the corresponding row of the allocation matrix to W. That is, set $W_k = W_k + A_{ik}$ for $1 \leq k \leq m$. Return to step 3.

Deadlock exists if and only if there are unmarked processes at the end of the algorithm, as these are all deadlocked processes.

4.2 Recovery

Once deadlock is detected, must recover and the following are possible approaches, in increasing sophistication:

1. Abort all deadlocked processes.
2. Back up each deadlocked process to some previously defined checkpoint and restart all processes.
   (a) requires rollback and restart mechanisms
   (b) original deadlock could occur again
3. Successively abort deadlocked process until deadlock no longer exists.
   (a) order is on the basis of some criterion of minimum cost

(b) detection must be run after each abortion

4. Successively preempt resources until deadlock no longer exists.
   (a) order is on the basis of some criterion of minimum cost
   (b) detection must be run after each abortion
   (c) process that has resource preempted must be rolled back to a point prior to resource acquisition

Minimum cost could be:

- least amount of processor time consumed
- least amount of output produced
- most estimated time remaining
- least total resources allocated so far
- lowest priority

5 Integrated Deadlock Strategy

More efficient to use different deadlock strategies in different situations. One such approach:

- group resources into a number of different resource classes
- use the linear ordering strategy defined previously for the prevention of circular wait to prevent deadlocks between resource classes
- within a resource class, use the algorithm that is most appropriate for that class

Some classes could include swappable space (secondary memory), process resources (devices, tapes, files, etc), main memory, and internal resources (IO channels). This order is reasonable due to the sequence of steps that a process may follow in its lifetime.

Within each class, the following strategies could be used:

- **swappable space**: prevention of deadlocks by requiring that all required resources that may be used be allocated at one time (like hold-and-wait), which is reasonable if max storage requirements are known; deadlock avoidance also possible

- **process resources**: avoidance often effective because can expect processes to declare ahead of time the resources they require in this class; prevention also possible
- **main memory**: prevention by preemption most appropriate where preemption means a process is swapped
- **internal resources**: prevention by resource ordering used

6 Dining Philosophers Problem

Round table with spaghetti, 5 plates, 5 forks, and 5 philosophers that each require 2 forks to eat.

Problem illustrates basic problems in deadlock and starvation, difficulty in concurrent programming, and coordination of shared resources.

6.1 Solution Using Semaphores

Each philosopher first picks up the fork on the left, then the fork on the right. After eating, the forks are replaced on the table. This leads to deadlock.

Instead have an attendant that only allows 4 philosophers into the room, then at least 1 philosopher has 2 forks.

```
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait(room);
        wait(fork[i]);
        wait(fork[(i+1) % 5]);
        eat();
        signal(fork[(i+1) % 5]);
        signal(fork[i]);
        signal(room);
    }
}
void main()
```

```
{
    parbegin (
       philosopher(0),
       philosopher(1),
       philosopher(2),
       philosopher(3),
       philosopher(4),
        );
}
```

6.2 Solution Using a Monitor

Vector of 5 condition variables defined, 1 per fork. Used to enable a philosopher to wait for the availability of a fork. Further, there is a Boolean vector that records availability status of each fork.

Monitor consists of 2 procedures.

`get_forks` used to seize left and right forks, if either unavailable, philosopher process queued on the appropriate condition variable so more philosophers can enter the monitor.

`release_forks` used to make 2 forks available.

No deadlocks because only one process at a time may be in the monitor.

```
monitor dining_controller;
cond ForkReady[5];
boolean fork[5] = {true};

void get_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;

    if (!fork[left])
        cwait(ForkReady[left]);
    fork[left] = false;

    if (!fork[right])
        cwait(ForkReady[right]);
    fork[right] = false;
```

```
}
void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;

    if (empty(ForkReady[left])) // no one waiting for this fork
        fork[left] = true;
    else // awaken a process waiting on this fork
        csignal(ForkReady[left]);

    if (empty(ForkReady[right])) // no one waiting for this fork
        fork[right] = true;
    else // awaken a process waiting on this fork
        csignal(ForkReady[right]);
}

void philosopher(int k)
{
    while (true)
    {
        <think>;
        get_forks(k);
        <eat spaghetti>;
        release_forks(k);
    }
}
```