

Deadlock Recovery, Avoidance and Prevention

Recovering From Deadlock

Once a deadlock has been detected, one of the 4 conditions must be invalidated to remove the deadlock. Generally, the system acts to remove the circular wait, because making a system suddenly preemptive with respect to resources, or making a resource suddenly sharable is usually impractical. Because resources are generally not dynamic, the easiest way to break such a cycle is to terminate a process.

Usually the process is chosen at random, but if more is known about the processes, that information can be used. For example the largest or smallest process can be disabled. Or the one waiting the longest. Such discrimination is based on assumptions about the system workload.

Some systems facilitate deadlock recovery by implementing *checkpointing and rollback*. Checkpointing is saving enough state of a process so that the process can be restarted at the point in the computation where the checkpoint was taken. Autosaving file edits is a form of checkpointing. Checkpointing costs depend on the underlying algorithm. Very simple algorithms (like linear primality testing) can be checkpointed with a few words of data. More complicated processes may have to save all the process state and memory.

Checkpoints are taken less frequently than deadlock is checked for. If a deadlock is detected, one or more processes are restarted from their last checkpoint. The process of restarting a process from a checkpoint is called *rollback*. The hope is that the resource requests will not interleave again to produce deadlock.

Deadlock recovery is generally used when deadlocks are rare, and the cost of recovery (process termination or rollback) is low.

Process checkpointing can also be used to improve reliability (long running computations), assist in process migration (Sprite, Mach), or reduce startup costs (emacs).

Avoiding Deadlock

A quick word about resource trajectories: Tannenbaum spends some time on them. I spend none. I think that they're pedagogically uninteresting and practically unimplementable. If you want to hear about them, you should read Tannenbaum's discussion of them. I won't test you on them.

Deadlock avoidance is arranging how resources are granted at run time so as to avoid deadlock. Remember, knowing what resources processes need during their run is not enough to tell if there will be a deadlock, but knowing that upper bounds, the OS can steer the system safely through deadlock dangers.

The most common deadlock avoidance algorithm is Dijkstra's Banker's algorithm. The algorithm requires that we know *a priori* how many resources each process needs. From then on, the OS behaves like an honest small-town banker. It only loans out resources to processes if it has enough on hand to meet potential demand. We characterise this as a safe state. Here are a safe and unsafe state (considering only one resource):

Safe:

Available = 2

process	allocated	maximum
A	1	6
B	1	5
C	2	4
D	4	7

Unsafe:

Available = 1

process	allocated	maximum
A	1	6
B	2	5
C	2	4
D	4	7

An unsafe state does **not** mean that the system is deadlocked, merely that the possibility exists. If the system remains in safe states, deadlock never occurs.

The algorithm for determining if a state is safe is: pick the process closest to its limit. If it can finish, count **all** its resources as free, and repeat until either all processes have been considered, or until a state is found where no process can complete.

The algorithm generalizes to multiple resources:

- A Vector of available resources
- E Vector of existing resources
- C Matrix of committed resources
- R Matrix of resource requests (remaining)

$$\sum_{i=0}^m C_{ij} + A_j = E_j$$

if $\forall i: 0 \leq i \leq m: A_i < B_i$ then $A \leq B$

The Generalized Banker's Algorithm (safe state determination) is:

1. Find an unmarked process with the *i*th row of R less than or equal to A
2. Mark it and let $A = A +$ the *i*th row of C and goto 1
3. If all processes get marked the state is safe, if not, unsafe

GBA example:

Safe

$$E = (4, 2, 3, 1) \qquad A = (2, 1, 0, 0)$$

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 0 & 0 & 2 & 0 \end{bmatrix} \qquad R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Unsafe

$$E = (4, 2, 3, 1) \qquad A = (2, 1, 0, 0)$$

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 0 & 0 & 2 & 0 \end{bmatrix} \qquad R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{bmatrix}$$

Avoidance is appropriate when the system makes bounded resource requests and the additional per request overhead of running the Banker's Algorithm is acceptable.

Prevention

Deadlock prevention is the process of making it logically impossible for one of the 4 deadlock conditions to hold.

Relaxing mutual exclusion requires making all relevant resources sharable. Some resources can be made sharable. Spooling can make devices like printers or tape drives sharable. Spooling is storing the output on a shared medium, like disk, and using a single process to coordinate access to the shared resource. Print spooling is the canonical example.

There are generally some resources that cannot be spooled - semaphores, for example. Removing mutual exclusion is not always an option.

Relaxing Hold and Wait requires processes to acquire all their needed resources at once. To acquire new resources in such a system requires a process to relinquish all the processes it holds and try to reacquire all the resources it needs atomically.

Reasonable systems can be programmed this way, but in practice it's almost never done. Starvation is a real possibility, and its probability increases with the number of resources concurrently acquired.

Relaxing non-preemptability for resources other than the CPU is very non intuitive. The idea that a program might lose a lock at any time and have to reacquire it is very counterintuitive. If this is done with physical resources, things can be even more confusing. Other than very specialized cases, relaxing non-preemptability is almost never done.

The most common method of preventing deadlock is to prevent the circular wait. A simple way to do this, when possible, is to order the resources and always acquire them in order. Because a process can't be waiting on a lower numbered process while holding a higher numbered one, a cycle is impossible.

One can consider the Dining Philosophers to be a deadlock problem, and can apply deadlock prevention to it by numbering the forks and always acquiring the lowest numbered fork first.

```
#define N 5 /* Number of philosophers */
#define RIGHT(i) (((i)+1) %N)
#define LEFT(i) (((i)==N) ? 0 : (i)+1)

typedef enum { THINKING, HUNGRY, EATING } phil_state;

phil_state state[N];
semaphore mutex =1;
semaphore f[N]; /* one per fork, all 1*/

void get_forks(int i) {
    int max, min;

    if ( RIGHT(i) > LEFT(i) ) {
        max = RIGHT(i); min = LEFT(i);
    }
    else {
        min = RIGHT(i); max = LEFT(i);
    }
    P(f[min]);
    P(f[max]);
}

void put_forks(int i) {
    V(f[LEFT(i)]);
    V(f[RIGHT(i)]);
}

void philosopher(int process) {
    while(1) {
        think();
        get_forks(process);
        eat();
        put_forks(process);
    }
}
```

This solution doesn't get maximum parallelism, but it is an otherwise valid solution.