13 Deadlock

Deadlock is one area where there is a strong theory, but much of it is not relevant in practice. Reason: many solutions are expensive and/or require predicting the future.

Deadlock example: semaphores. Two threads: one does P(x) followed by P(y), the other does the reverse.

Deadlock: a situation where each of a collection of threads is waiting for something from other threads in the collection. Since all are waiting, none can provide any of the things being waited for.

These are relatively simple-minded cases. Things may be much more complicated:

- In general, don't know in advance how many resources a thread will need. If only we could predict the future....
- Deadlock can occur over separate resources, as in semaphore example, or over pieces of a single resource, as in memory, or even over totally separate classes of resources (e.g., printers and memory). Deadlock can occur over anything involving waiting, for example messages in a pipe system. Hard for OS to control.

In general, there are four conditions for deadlock:

- Mutually exclusive access: resources cannot be shared.
- No preemption. Once given, a resource cannot be taken away.
- Hold and wait: threads can hold resources while they are waiting for more resources.
- There is a circularity in the graph of who has what and who wants what. This graph shows threads as circles, resources as squares, arrows from thread to resource waited for, from resource to owning thread.

Solutions to the deadlock problem fall into two general categories:

- Detection : determine when the system is deadlocked and then take drastic action. Requires termination of one or more threads in order to release their resources. Impractical unless the system has special support for aborting threads and recovering (e.g., transactions).
- Prevention : organize the system so that it is impossible for deadlock ever to occur. May lead to less efficient resource utilization in order to guarantee no deadlocks.

Deadlock prevention: must find a way to eliminate one of the four necessary conditions for deadlock:

- Don't allow exclusive access. This is not reasonable for many applications.
- Create enough resources so that there's always plenty for all.
- Don't allow waiting. (If I can't get all the resources I need, give up all.) Not practical in many situations.
- Allow preemption. E.g. pre-empt students' thesis disk space?
- Make thread ask for everything at once. Either get them all or wait for them all. Tricky to implement: must be able to wait on many things without locking anything. Painful for programmer: may be difficult to predict future needs, so must make very wasteful use of resources.
- Banker's algorithm.
  1. state maximum resource needs in advance
  2. allocate resources when needed; delay when granting request could lead to deadlock

```
int Available[j]: Available resources of type j
int Max[i,j]: Maximal #resources of type j needed by thread i
int Allocation[i,j]: #resources of type j allocated to thread i

// request a resource
bool request(Thread i, Resource j, Num k) {
  if (Allocation[i,j] + k > Max[i,j] || Available[j] < k)
```

```
    return false;

  Available[j] -= k;
  Allocation[i,j] += k;
  // check if granting the resources is safe
  if (safe(j)) return TRUE;
  // unsafe, deny request
  Available[j] += k;
  Allocation[i,j] -= k;
  return FALSE;
}

// check if we're in a safe state (Banker's algorithm)
bool safe(Resource j) {
  int Work = Available[i,j];
  bool Finish[num_threads] = {FALSE, ..., FALSE};

  // seeks to find a sequence of threads that can execute even if
  // they ask for their maximal resources, thus ensuring progress
  while (1) {
    for (i=0; i<num_threads; i++) {
      if (!Finish[i] && Max[i,j] - Allocation[i,j]  <= Work) {
        // thread can execute
        // add the resources it will eventually release
        Work += Allocation[i,j];
        Finish[i] = TRUE;
        break;
      }
    }
    break;
  }
  if (Finish != {TRUE, ..., TRUE}) return FALSE;
  // all threads will be able to finish
  return TRUE;
}
```

Banker's algorithm provides interesting insight into dynamic deadlock prevention, but has little practical relevance (runtime cost, difficulty of predicting number of threads and their maximal resource needs).

- Require *ordered* requests. E.g. ask for resource of type S, then resources of type T, etc. If all threads allocate resource types in same order, there can be no cyclic dependencies. Widely used in practice.

  1. Resources that are never held by a thread at the same time can be requested in any order relative to each other.
  2. Finding a feasible ordering of resources requires design time knowledge of the needs of all threads.

In general, deadlock prevention is difficult, expensive or inefficient. Detection is also expensive and recovery requires special system support (thread may be in arbitrary state).