CIS520 – Operating Systems
Handout 4
Deadlock

- You may need to write code that acquires more than one lock. This opens up the possibility of deadlock. Consider the following piece of code:

```
Lock *l1, *l2;
void p() {
  l1->Acquire();
  l2->Acquire();
  code that manipulates data that l1 and l2 protect
  l2->Release();
  l1->Release();
}
void q() {
  l2->Acquire();
  l1->Acquire();
  code that manipulates data that l1 and l2 protect
  l1->Release();
  l2->Release();}
```

  If `p` and `q` execute concurrently, consider what may happen. First, `p` acquires `l1` and `q` acquires `l2`. Then, `p` waits to acquire `l2` and `q` waits to acquire `l1`. How long will they wait? Forever.

- This case is called deadlock. What are conditions for deadlock?
  - Mutual Exclusion: Only one thread can hold lock at a time.
  - Hold and Wait: At least one thread holds a lock and is waiting for another process to release a lock.
  - No preemption: Only the process holding the lock can release it.
  - Circular Wait: There is a set $t_1, \ldots, t_n$ such that t_1 is waiting for a lock held by t_2, ..., t_n is waiting for a lock held by t_1.
- How can `p` and `q` avoid deadlock? Order the locks, and always acquire the locks in that order. Eliminates the circular wait condition.
- Occasionally you may need to write code that needs to acquire locks in different orders. Here is what to do in this situation.
  - First, most locking abstractions offer an operation that tries to acquire the lock but returns if it cannot. We will call this operation `TryAcquire`. Use this operation to try to acquire the lock that you need to acquire out of order.
  - If the operation succeeds, fine. Once you've got the lock, there is no problem.
  - If the operation fails, your code will need to release all locks that come after the lock you are trying to acquire. Make sure the associated data structures are in a state where you can release the locks without crashing the system.

  - Release all of the locks that come after the lock you are trying to acquire, then reacquire all of the locks in the right order. When the code resumes, bear in mind that the data structures might have changed between the time when you released and reacquired the lock.

- Here is an example.

```
int d1, d2;
// The standard acquisition order for these two locks
// is l1, l2.
Lock *l1, // protects d1
     *l2; // protects d2
// Decrements d2, and if the result is 0, increments d1
void increment() {
  l2->Acquire();
  int t = d2;
  t--;
  if (t == 0) {
    if (l1->TryAcquire()) {
      d1++;
    } else {
      // Any modifications to d2 go here - in this case none
      l2->Release();
      l1->Acquire();
      l2->Acquire();
      t = d2;
      t--;
      // some other thread may have changed d2 - must recheck it
      if (t == 0) {
        d1++;
      }
    }
    l1->Release();
  }
  d2 = t;
  l2->Release();
}
```

  This example is somewhat contrived, but you will recognize the situation when it occurs.

- There is a generalization of the deadlock problem to situations in which processes need multiple resources, and the hardware may have multiple kinds of each resource - two printers, etc. See OSC Chapter 7. Seems kind of like a batch model - processes request resources, then system schedules process to run when resources are available.

- In this model, processes issue requests to OS for resources, and OS decides who gets which resource when. A lot of theory built up to handle this situation.

- Process first requests a resource, the OS issues it and the process uses the resource, then the process releases the resource back to the OS.

- Reason about resource allocation using resource allocation graph. Each resource is represented with a box, each process with a circle, and the individual instances of the resources with dots in the boxes. Arrows go from processes to resource boxes if the process is waiting for the resource. Arrows go from dots in resource box to processes if the process holds that instance of the resource. See Fig. 7.1.

- If graph contains no cycles, is no deadlock. If has a cycle, may or may not have deadlock. See Fig. 7.2, 7.3.

- System can either

  - Restrict the way in which processes will request resources to prevent deadlock.
  - Require processes to give advance information about which resources they will require, then use algorithms that schedule the processes in a way that avoids deadlock.
  - Detect and eliminate deadlock when it occurs.

- First consider prevention. Look at the deadlock conditions listed above.
  - Mutual Exclusion - To eliminate mutual exclusion, allow everybody to use the resource immediately if they want to. Unrealistic in general - do you want your printer output interleaved with someone elses?
  - Hold and Wait. To prevent hold and wait, ensure that when a process requests resources, does not hold any other resources. Either asks for all resources before executes, or dynamically asks for resources in chunks as needed, then releases all resources before asking for more. Two problems - processes may hold but not use resources for a long time because they will eventually hold them. Also, may have starvation. If a process asks for lots of resources, may never run because other processes always hold some subset of the resources.
  - Circular Wait. To prevent circular wait, order resources and require processes to request resources in that order.

- Deadlock avoidance. Simplest algorithm - each process tells max number of resources it will ever need. As process runs, it requests resources but never exceeds max number of resources. System schedules processes and allocates resoures in a way that ensures that no deadlock results.

- Example: system has 12 tape drives. System currently running P0 needs max 10 has 5, P1 needs max 4 has 2, P2 needs max 9 has 2.

- Can system prevent deadlock even if all processes request the max? Well, right now system has 3 free tape drives. If P1 runs first and completes, it will have 5 free tape drives. P0 can run to completion with those 5 free tape drives even if it requests max. Then P2 can complete. So, this schedule will execute without deadlock.

- If P2 requests two more tape drives, can system give it the drives? No, because cannot be sure it can run all jobs to completion with only 1 free drive. So, system must not give P2 2 more tape drives until P1 finishes. If P2 asks for 2 tape drives, system suspends P2 until P1 finishes.

- Concept: Safe Sequence. Is an ordering of processes such that all processes can execute to completion in that order even if all request maximum resources. Concept: Safe State - a state in which there exists a safe sequence. Deadlock avoidance algorithms always ensure that system stays in a safe state.

- How can you figure out if a system is in a safe state? Given the current and maximum allocation, find a safe sequence. System must maintain some information about the resources and how they are used. See OSC 7.5.3.

```
Avail[j] = number of resource j available
Max[i,j] = max number of resource j that process i will use
Alloc[i,j] = number of resource j that process i currently has
Need[i,j] = Max[i,j] - Alloc[i,j]
```

- Notation: `A<=B` if for all processes `i`, `A[i]<=B[i]`.

- Safety Algorithm: will try to find a safe sequence. Simulate evolution of system over time under worst case assumptions of resource demands.

```
1: Work = Avail;
   Finish[i] = False for all i;
2: Find i such that Finish[i] = False and Need[i] <= Work
   If no such i exists, goto 4
3: Work = Work + Alloc[i]; Finish[i] = True; goto 2
4: If Finish[i] = True for all i, system is in a safe state
```

- Now, can use safety algorithm to determine if we can satisfy a given resource demand. When a process demands additional resources, see if can give them to process and remain in a safe state. If not, suspend process until system can allocate resources and remain in a safe state. Need an additional data structure:

```
Request[i,j] = number of j resources that process i requests
```

- Here is algorithm. Assume process `i` has just requested additional resources.

```
1: If Request[i] <= Need[i] goto 2. Otherwise, process has
   violated its maximum resource claim.
2: If Request[i] <= Avail goto 3. Otherwise, i must wait
   because resources are not available.
3: Pretend to allocate resources as follows:
   Avail = Avail - Request[i]
   Alloc[i] = Alloc[i] + Request[i]
   Need[i] = Need[i] - Request[i]
   If this is a safe state, give the process the resources. Otherwise,
   suspend the process and restore the old state.
```

- When to check if a suspended process should be given the resources and resumed? Obvious choice - when some other process relinquishes its resources. Obvious problem - process starves because other processes with lower resource requirements are always taking freed resources.

- See Example in Section 7.5.3.3.

- Third alternative: deadlock detection and elimination. Just let deadlock happen. Detect when it does, and eliminate the deadlock by preempting resources.

- Here is deadlock detection algorithm. Is very similar to safe state detection algorithm.

```
1: Work = Avail;
   Finish[i] = False for all i;
2: Find i such that Finish[i] = False and Request[i] <= Work
   If no such i exists, goto 4
3: Work = Work + Alloc[i]; Finish[i] = True; goto 2
4: If Finish[i] = False for some i, system is deadlocked.
   Moreover, Finish[i] = False implies that process i is deadlocked.
```

- When to run deadlock detection algorithm? Obvious time: whenever a process requests more resources and suspends. If deadlock detection takes too much time, maybe run it less frequently.

- OK, now you've found a deadlock. What do you do? Must free up some resources so that some processes can run. So, preempt resources - take them away from processes. Several different preemption cases:
  - Can preempt some resources without killing job - for example, main memory. Can just swap out to disk and resume job later.
  - If job provides rollback points, can roll job back to point before acquired resources. At a later time, restart job from rollback point. Default rollback point - start of job.
  - For some resources must just kill job. All resources are then free. Can either kill processes one by one until your system is no longer deadlocked. Or, just go ahead and kill all deadlocked processes.

- In a real system, typically use different deadlock strategies for different situations based on resource characteristics.

- This whole topic has a sort of 60's and 70's batch mainframe feel to it. How come these topics never seem to arise in modern Unix systems?