Deadlocks
SGG: Chapter 7

I. Overview of Challenges

A. Finite resources, multiple processes competing for them

B. We've explored algorithms for scheduling and mechanisms for providing ordered/controlled access (through synchronization primitives) to those resources
   1. If a process needs a resource that's not available, it can be put into a wait queue, until the resource is available
   2. Perhaps waiting while holding on to other resources

C. If we're not careful about dependencies, we might create pathologies that lead to **deadlock**
   1. Like race conditions, may only happen sometimes, making them hard to identify and debug

D. Can we detect and resolve these situations? Can we prevent them?

E. Motivating Example 1: Three processes, three resources of the same type (DVD drive), each process requires two resources to make progress
   1. This can actually happen across a network as well, which is particularly hard to diagnose

F. Motivating Example 2: Two process creating a circular dependency on two different resources

`lock(&mutex1);`	`lock(&mutex2);`
`lock(&mutex2);`	`lock(&mutex1);`
`//Critical section`	`//Critical section`
`unlock(&mutex2);`	`unlock(&mutex1);`
`unlock(&mutex1);`	`unlock(&mutex2);`

II. Deadlock Characteristics [Coffman et al 1971]

A. Definition: A set of processes are **deadlocked** if each process in the set is waiting for an event that only another process in the set can cause.
   1. **Resource deadlock** is the most common (and what we'll study this week)
   2. **Communication deadlock**: e.g. Process A sends a message to B, then sleeps until reply; Process B sleeps until it receives a message from A; message is lost

B. Four necessary conditions (simultaneous)

C. **Mutual exclusion**: At least one resource can only be held by one process at a time; this can result in other processes waiting for that resources

D. **Hold and wait**: A process must be holding at least one resource while waiting for other resources (held by other processes)

E. **No preemption**: Resources can only be released voluntarily by a process; Resources cannot be revoked

F. **Circular wait**: A set of *n* waiting process $\{P_0, ..., P_n\}$ such that P_i is waiting for resources held by $P_{(i+1)\%n}$

G. All must be met, and some imply others: e.g. Circular wait implies hold and wait.

H. Each condition relates to a policy that may or may not be implemented
   1. Q: Can a resource be assigned to more than one process?
   2. Q: Can a process request multiple resources? Over what period of time?
   3. Q: Can resources be preempted, and how?

III. Deadlock Modeling [Holt 1972]

A. **Resource Allocation Graphs**: A directed graph, where process and resources are nodes (sometimes circles and squares respectively), and edges are resource allocations & requests
   1. Process → Resource: A resource request
   2. Resource → Process: An allocated resources

B. Example: Graph motivating example 2 above.

C. Cycles in a graph indicate deadlock
   1. When resources have only one instance: A cycle is both necessary and sufficient for deadlock
   2. If there are multiple instances: A cycle is necessary by not sufficient for a deadlock

D. Example: Graph above with two resource of one type
   1. $R_1 \rightarrow P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_1$

E. Example 2: $P_1 \rightarrow R_1 \rightarrow P_2$; $R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$; $R_2 \rightarrow P_4$
   1. Q: Is this deadlocked?
   2. No; There is no hold and wait/circular wait

F. No cycles, no deadlock; Cycles imply only a possible deadlock

IV. Handling Deadlock

A. OS can **prevent** deadlock: ensure deadlocks can never occur (by preventing one of the deadlock requirements from occurring)

B. OS can **avoid** deadlock: like prevention, but given more information about a processes resource needs

C. OS can **detect** deadlock: allow system to deadlock, detect and recover

D. OS can do **nothing**, and let applications handle it (common solution)
   1. **Ostrich Algorithm**: head in sand; wait

V. Prevention [Coffman et al 1971]

A. **Attacking Mutual exclusion**: No one resource can only be held by one process at a time; while some resources are preemptable (like memory, through swapping) preventing mutual exclusion is fundamentally difficult as so many resource are non-sharable: printers, files, disks, etc.
   1. We can create a monitor for non-preemptable resources (e.g. print spooler, disk drive buffer)

B. **Attacking Hold and wait**: When a process requests a resource, it cannot be holding any other resources
   1. A process must request **all** of its resources at one time
   2. Or, request a resource only when it has none; can even request and be granted multiple, just must release all when additional requests are made
      a) (Also a solution to circular wait)
   3. The first is easier to implement, but inefficient
      a) Example: Word processor that wants read a file from USB, edit it, and print it;
      b) Q: How long do we edit?
   4. The second is more complicated, but provides granularity
   5. In both cases, resource utilization can be low; most processes don't know their needs *a priori*
   6. Starvation is also possible

C. **Attacking No preemption**: Resources can be preempted/recalled.
   1. If a process is holding resources, and requests a busy resource, it releases all resources it holds
   2. Or, if a waiting process is holding resources another process needs, they can be reallocated
   3. Not all resources are preempt-able (e.g. printers, IO devices)
   4. Could also lead to starvation

D. **Attacking Circular wait**: Impose a total ordering of all resource types, require each process request resources in an increasing order

1. Resource set R = {R_0, R_1, ... R_{N-1}} (one-to-one); processes must request from R only in an increasing order
2. Example: Returning to the 2 mutex problem above; if we provide a total ordering, then P2 could NOT request in that order
3. The above logic holds for n > 2 processes
4. At any instance, one of the assigned resources will be the highest, and that process holding that resources will never ask for another resource

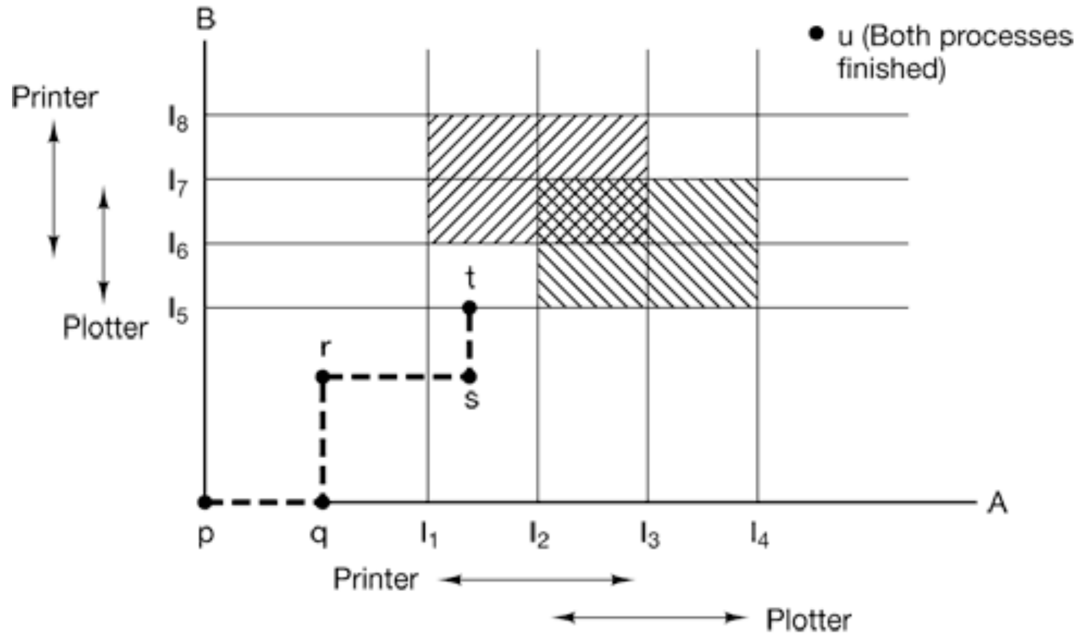
VI. Avoidance

A. All of the prevent algorithms regulate how resources are allocated

1. This is to prevent one of the four criteria for deadlocks occurring

B. Avoidance tried to achieve the same thing, but with fewer restrictions on how resources are allocated, in exchange for more information at request

1. Example: The order in which resources may be allocated; in what order they may be released
2. Some avoidance algorithm just requires the max. number of resources (of a given type)



C. **Important**: at point t, B requests a resource that will cause an unsafe state
D. Also, this is just a visualization: no algorithms based on graph visualization
E. **Safe State**: A state is **safe** if a system can allocate resources to each process in some order and still avoid deadlock. I.e. A safe allocation order. A safe state is not a deadlocked state.
F. A system is only in a safe state iff there exists a **safe sequence**
G. Resource Allocation Graph Algorithm
   1. Introduce a new kind of edge: **claim edg**e
      a) $P_i \rightarrow R_j$ indicates that P_i may request R_j in the future
      b) A request edge with a dashed line
   2. All claim edges must be established when Pi runs
   3. The system can maintain this graph, and only grant resources that will not create a cycle
      a) Example: $R_1 \rightarrow P_1 – C \rightarrow R_2$; $P_2 \rightarrow R_1$; $P_2 – C \rightarrow R_2$
   4. Only applicable to single instance resources (not multiple instances of resources of a single type)

H. Banker'sAlgorithm[Dijkstra65] (Avoiding Deadly Embrace)
   1. Process must declare max. number of resource instances of each type needed (can't exceed max. system resources)
   2. Analogy comes from a bank giving loans: will giving a particular loan put the bank in an unsafe state (e.g. not be able to meet their other obligations)
   3. Processes must wait if allocating those resources will leave the system in an unsafe state
   4. Required data structures:
      a) Available: an m-length vector of number of available resources; Available[i] = j then |Ri| = j; m = num resource types
      b) Max: an n x m matrix that defines max resource demand for each process; Max[i][j] = k, then Pi may request at most k from Rj; n = num processes
      c) Allocation: an n x m matrix that defines currently allocated resources to processes
      d) Need: an n x m matrix that defines remaining resource need
      e) This is an O(m x n^2) algorithm; and only worthy of academic discussions, because:
   5. In general, avoidance is nearly impossible: requires having prescience about resource allocation and a fixed number of processes, which is uncommon in general purpose OSes

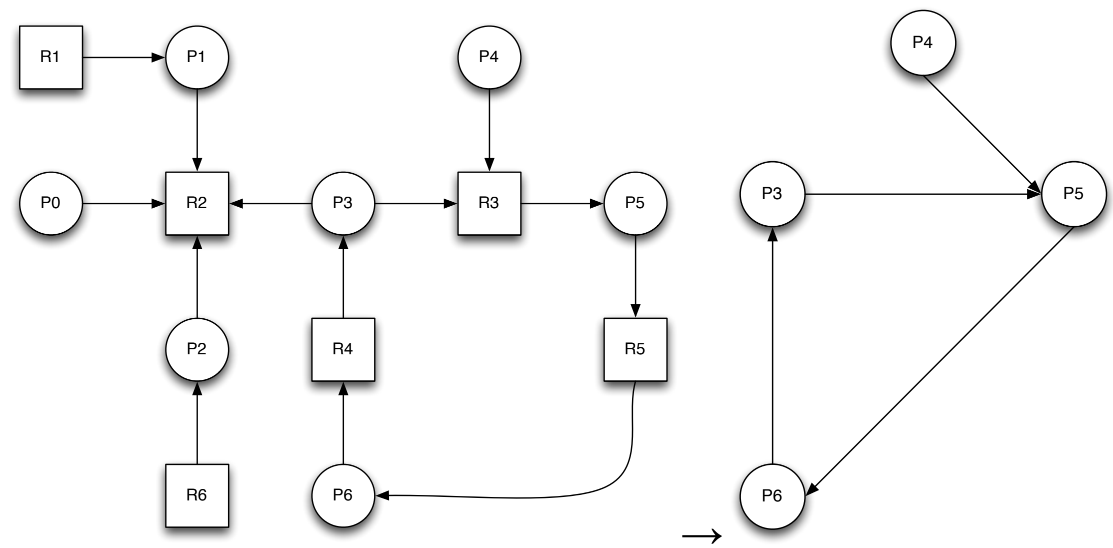
VII. Detection & Recovery

A. Here, we allow deadlocks occur, and try to **detect** them when they do; perhaps even try to **recover** from deadlock

B. **Wait-for graph**: A resource allocation graph variant
   1. It's a traditional resource allocation graph with removed resources, and collapsing appropriate edges
   2. Here, $P_i \rightarrow P_j$ implies P_i is waiting for P_j to release a resource it

needs; also implies there were two edges $P_i \rightarrow R_q \rightarrow P_j$

3. A deadlock exists if there is a cycle; easy to visualize, a little harder to implemented in software
4. An $O(n^2)$ operation (beyond the costs of maintaing the graph)
5. Example:



   a) Pick a process node at random, use as a root of a tree
   b) Perform depth-first search
   c) If you ever see a node twice, there's a cycle
   d) If you backtrack to the root upon completion, no cycle
   e) Repeat for all nodes

6. Only application with **single instance per resource type**; doesn't work with multiple instance of a resource type

C. If multiple instances, we must use a variant of the Banker's Algorithm (m x n^2 operations)

D. **Q:** When do we run these algorithms?

1. How often will deadlock occur?
2. How many processes will be affected by deadlock?
3. Run every process entry? Every resource allocation?
4. Heuristically: CPU utilization drops? every hour? who knows?

E. **Recovering** from deadlock

1. System **reset**
2. Process **termination**

a) Kill a process to release its resources
b) **Q:** how do we identify victims?
(1) In an identified cycle?
(2) not in cycle, but holding necessary resources
c) Again, all heuristical: kill a process that is low priority? can rerun easily (how do we know?)

3. Resource **preemption**
   a) We've seen this as a solution in prevention; highly dependent on the nature of resources
   b) Some resources are fundamentally difficult to preempt
4. **Rollback**
   a) Processes can checkpoint periodically
   b) Processes with their state (memory image and resource allocations) to a log
   c) When deadlock occurs:
      (1) Identify needed resources; identify processes that hold those resources; this process is rolled back to a point where it does not hold those resources (all progress beyond checkpoint is lost)
   d) similar to preemption: rewinding time may be fundamentally difficult (consider the 3D printer)

VIII. Communication Deadlock

A. Example: A sends a message to B and sleeps until a response; B sleeps until it receives a message; message is lost

B. This is different than resource deadlock: A does not possess a resource B wants; in the above example, there aren't even any resources
   1. they are blocked on an n (this still meets our definition of deadlock)

C. As such, cannot be prevented using resource-based solution (ordering, preemption, mutual exclusion, etc).

D. One solution: **timeouts**
   1. Timers go off after some "expected response" time
      a) At what interval? What do we do when timer goes off?

Retransmit? How many times?

2. If there’s delay, and not loss, recipient may receive the same message twice
   a) What happens in these circumstance?
3. Requires a **protocol** for handling (and largely out of the scope of this course)