Deadlocks

- …or, "the deadly embrace"
- As we have seen, potential critical section solutions must be *deadlock-free* — that is, they must never reach a situation where processes are mutually waiting for each other in order to proceed
- In general, deadlocks are a possibility in any situation where multiple processes are sharing a pool of resources; deadlock may occur if a process is waiting for resources that are themselves held by another process that is also in a waiting state

System Model

- Our discussion of deadlock uses a specific *system model* for multiprogramming and resource sharing
- We view a system as a finite number of *resources*, possibly partitioned into several *resource types* (memory, CPU, files, printers, drives, etc.)
- Multiple *processes* utilize these resources during execution in a three-phase sequence: *request*, *use*, and *release* — if a resource isn't immediately available upon *request*, then the process must wait for its *release* by another process; this is where deadlock can happen

Four Necessary Conditions

It turns out that deadlock has a precise characterization — it boils down to four *necessary conditions*, meaning that deadlock occurs if all four conditions are true:

- *Mutual exclusion* — There are resources that can only be used by one process at a time
- *Hold and wait* — A process is holding ("using") one resource while waiting for ("requesting") additional resources that are themselves being held by other processes in the system
- *No preemption* — A resource can only be released by the process that is holding it
- *Circular wait* — There exists a set of waiting processes $\{P_0, P_1, P_2, \ldots, P_n\}$ such that for $0 \leq k < n$, process P_k is waiting for a resource held by process P_{k+1}, and process P_n is waiting for a resource held by process P_0

Based on these four conditions and a *resource-allocation graph*, we can reason about deadlocks in a way that allows a system to deal with them in a number of ways: *prevention*, *avoidance*, *detection*, and *recovery*

Resource-Allocation Graphs

A *system resource-allocation graph* is a directed graph that is used for describing and reasoning about deadlock

- The graph's set of vertices V consists of two partitions: $P = \{P_1, P_2, P_3, \ldots, P_n\}$ is the set of active processes, while $R = \{R_1, R_2, R_3, \ldots, R_m\}$ is the set of available resource types
- The graph's set of edges E is also partitioned into two types: *request edges* ($P_i \rightarrow R_j$) indicate that P_i is waiting for an instance of R_j, while *assignment edges* ($R_j \rightarrow P_i$) indicate that R_j instance has been allocated to P_i

By expressing some combination of processes, resource types, requests, and allocations as a resource-allocation graph, certain conclusions can be drawn based on the graph's structure:

- If the graph has no cycles, then there is no deadlock
- If the graph *does* have a cycle, then there *might* be deadlock (i.e., the existence of a cycle is *necessary* but not *sufficient*):
  - If the resource types involved in the cycle have only one instance, then we have deadlock
  - In other words, the existence of a cycle becomes necessary *and* sufficient when we only have one instance per resource type in the cycle

Handling Deadlocks

- There are three overall ways to handling deadlocks:
  - *Prevent* or *avoid* deadlocks, so that we don't get there in the first place
  - *Recover* from deadlocks after they happen; of course, this means that we can *detect* them in the first place
  - Do nothing…just hope they don't happen when *you* need to get something important done!
- Many operating systems choose approach #3 — deadlocks are viewed as an "application-level" issue

- Deadlock "prevention" as opposed to "avoidance" sound somewhat similar; the distinction is that prevention is focused specifically on ensuring that at least one of the four necessary conditions for deadlock does not happen, while avoidance takes a higher-level view, asking for and using information about a process's *overall* resource needs when making resource allocation decisions
- The choice to ignore deadlocks altogether comes from their relative rarity — killing processes and/or forcing a reboot every so often may be viewed as "cheaper" than running anti-deadlock algorithms all the time — and the fact that systems can freeze for reasons *other* than deadlock; since freezes may happen anyway, we'll need escape routes even if the OS handles deadlocks

Deadlock Prevention

- As mentioned, the idea behind deadlock prevention is to make sure that the four necessary deadlock conditions cannot all be true
- Thus, it suffices to take each condition individually, finding ways to keep that condition from occurring — unfortunately, this is easier said than done
- To deny *mutual exclusion*, resources must be sharable
  - This is true for some resources, such as read-only files or databases
  - However, other types of resources, such as printers, just can't be shared, so it's impossible to eliminate mutual exclusion totally
- For *hold and wait*, we prohibit a process from requesting a resource if it's currently holding others
  - For multiple resources, a process must request *all* of them at once, or use them sequentially, releasing before requesting; this may result in low utilization or starvation
- Eliminating *no preemption* involves "taking away" resource(s) from a process, either when it requests an unavailable resource or when another process requests a resource that has been allocated to it
  - The ability to release resources behind a process's back requires some way to save the state of the resource at the time that it is taken away — possible for resources like memory, but not for many I/O devices
- For *circular wait*, we can impose a *total ordering* over the resource types, and require that processes request resources according to that order
  - This is provable! — But how do we ensure that a process respects this total order?

Deadlock Avoidance

- Restraining requests to keep at least one of the four necessary deadlock conditions from occurring is sound in principle, but may be viewed as too aggressive — it lowers device utilization and/or system throughput
- Alternatively, the *deadlock-avoidance* approach examines the current resource-allocation state and uses this information to make subsequent decisions
- One key piece of additional information: what is the *maximum number* of instances of a resource type that a process may need?

- The *resource-allocation state* of a system, then, consists of its available and allocated resources, as well as the maximum numbers needed by each process
- This state is *safe* if we can allocate resources up to each process's maximum without resulting in deadlock (i.e., circular wait)
- The *safe state* is equivalent to the existence of a *safe sequence* — a sequence $<P_1, P_2, P_3, \ldots, P_n>$ such that every P_i's resource needs can be satisfied by the currently available resources plus the resources used by P_j, $j < i$
- The idea, then, is to always keep the system in a safe state — grant a resource request only if the system stays in a safe state after this resource is allocated

Resource-Allocation-Graph Algorithm

- If there is only one instance of each resource type in a system, we can maintain a safe state by using a variation of the resource allocation graph
- We add a new type of process-to-resource edge, a *claim edge* $P_i \rightarrow R_j$ which indicates that P_i *might* request R_j sometime in the future — we know this because we know the maximum resource needs of each process
  - Note how this results in a "life cycle" of sorts for the edges in the graph: they all start as claim edges, then become request edges when an actual resource request is made
  - If a request is granted, then the request edge changes into an assignment edge; finally, when the process is done with a resource, the assignment edge returns to a claim edge
- We stay in a safe state by making sure that converting a request edge into an allocation edge (thus changing its direction) does not result in any cycles
- Cycle detection is $O(n^2)$ for n total processes
- If a cycle is detected, then the requesting process will have to wait, and the request edge stays that way until converting it into an allocation edge does not result in any cycles
- Note that the addition of claim edges in this resource allocation graph means that a cycle does not denote deadlock, but an unsafe state

Banker's Algorithm

- The *banker's algorithm* is more general than the resource-allocation-graph algorithm because it can handle multiple instances of a particular resource type; unfortunately, it is less efficient
- The algorithm requires the following data structures:
  - *Available* — A vector listing the number of available instances for each resource type
  - *Max* — A matrix that tracks the maximum need of each process for each resource type
  - *Allocation* — A matrix that tracks the number of instances of each resource type that is currently used by each process
  - *Need* — A matrix that indicates how many more of each resource type are needed by each process; note that $Need[i][j] = Max[i][j] - Allocation[i][j]$

- A core concept in the banker's algorithm is the idea of some vector X being "less than" another vector Y — this means $X[i] < Y[i] \; \forall \; i$
- The *safety algorithm* determines whether the system is in a safe state; it is $O(mn^2)$ where m is the number of resource types and n is the number of processes
  - In essence, the safety algorithm is a "safe sequence finder," testing whether the resource needs of all processes can be satisfied with the current resource availabilities and allocations; if so, the algorithm "finishes" the process by simulating the release of all of its resources, then tries to find the next process in the sequence
- The *resource-request algorithm* determines if a resource request can be granted (i.e., allocating the resource must keep the system in a safe state)
  - If a resource request can be granted (i.e., the requested resources are available), then we "simulate" an allocation, then perform the safety algorithm; if the resulting state remains safe, then we *actually* perform the allocation, else the requesting process needs to wait

Deadlock Detection

- If neither deadlock prevention nor avoidance are available, then deadlocks might happen — in this case, we must be able to *detect* such situations, then *recover* from them somehow
- When there is only one instance of each resource type, we can use cycle detection again, this time on a *wait-for graph* — a resource-allocation graph with the resource nodes and edges "collapsed" so that we only see which process is waiting for another process
  - Note how the system must maintain this graph then periodically check it for cycles
- When multiple instances exist for a resource type, we can employ a banker's algorithm-like approach using data structures for *Available*, *Allocation*, and (new for deadlock detection) *Request*, indicating how many of each resource type is being requested by each process
  - The algorithm resembles the test for safe state, but of course, it finds a deadlock state instead, since we are now dealing with actual allocations instead of a "what-if" situation
- In addition to the detection algorithm itself, there is the issue of *how often* this algorithm should be invoked, and *how many* processes may be involved in the deadlock at the time that it is detected
  - Frequency is a tradeoff between overhead and number of processes involved, since a deadlock tends to involve more and more processes as time goes on
  - Choices range from invoking the algorithm for every resource request (expensive, but should catch the "first" process that gets deadlocked) to less frequent heuristic rules like once every *t* units of time or whenever CPU utilization drops below some threshold

Deadlock Recovery

- Once deadlock is detected, *recovery* must then take place; an easy option is to "tell a human" via system logs or messages — let the person deal with it
- One option (which can be automated) is to terminate one or more of the processes involved in the circular wait, either one at a time (requiring subsequent deadlock checks plus a selection method for which deadlocked process should end first) or wholesale
  - Either way, we generally can't recover from this deadlock unscathed, with "costs of termination" ranging from lost work to incorrect or incomplete data

- The *resource preemption* approach tackles the no preemption condition for deadlock instead of circular wait; this involves forcibly taking resources away from a process to be used by others, and is subject to some significant issues:
  - Which process gets preempted (the "victim")? Even without termination, taking away resources likely incurs a cost (e.g., stopping a print job)
  - How should the process handle the loss of the resource? Since the process can't proceed without the resource, it may need to be *rolled back* — but to where? Worst case: total rollback by restarting the process
  - How can we avoid starvation? If we're not careful, the same process may always be picked as the victim whenever deadlock is detected, and so it never completes

Given this entire discussion, perhaps the reality that most operating systems *don't* deal with deadlocks at all is no longer that shocking!