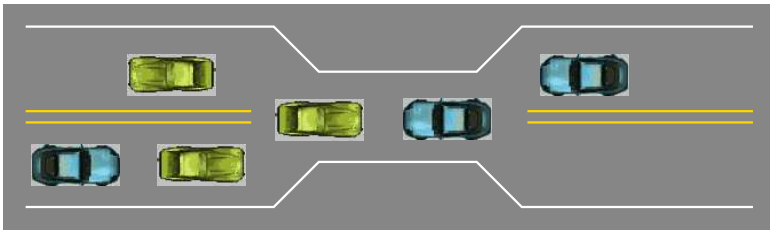
Topic Notes: Deadlock

The idea of *deadlock* has come up a few times so far:

A group of processes is *deadlocked* if each process is waiting for an event that can be caused only by one of the other waiting processes.

Consider a one-lane tunnel (see `http://www.teresco.org/pics/xc-19990722-0810/disk09/mvc-018f.jpg`) or underpass:



- One-lane traffic inside the tunnel
- The lane in the tunnel is a shared resource
- If a deadlock like this one occurs, we must tell one of the cars it has to give up its right to the resource and *rollback*
- This may mean more than one car has to back up
- Starvation is possible if we keep allowing cars from one direction to enter

The traffic analogy can become more complex with two-way intersections – **gridlock!**

We have seen the potential for deadlock when using semaphores:

P_0	P_1
`wait(Q);`	`wait(R);`
`wait(R);`	`wait(Q);`
...	...
`signal(R);`	`signal(Q);`
`signal(Q);`	`signal(R);`
...	...

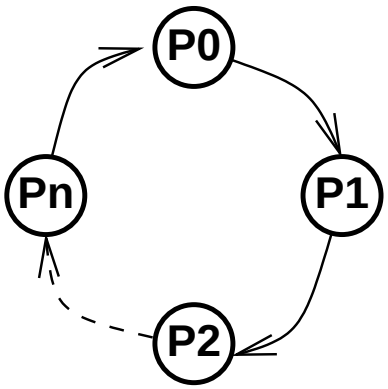
This is a situation that came up with the original "solution" to the dining philosophers.

We will consider processes that need access to more general *resources*. These could be any non-preemptable resources, such as tape drives or CD burners, in addition to things like the semaphores.

Requirements for Deadlock

Four conditions must hold simultaneously for deadlock to arise:

1. *Mutual exclusion*: only one process at a time can use a resource
2. *Hold and wait*: a process is holding, but not necessarily using, at least one resource and is waiting to acquire additional resources held by other processes
3. *No preemption*: a resource can be released only voluntarily by the process holding it, after that process has completed its task
4. *Circular wait*: there exists a circular chain of waiting processes, each of which it waiting for a resource held by the next process in the chain



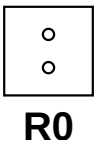
If any one of these conditions does not hold, deadlock cannot occur.

Resource Allocation Graphs

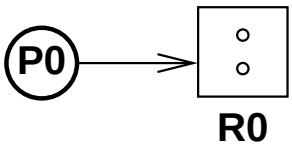
We model systems for our study of deadlock using directed graphs.

The vertices of our graph are *processes*, represented by circles,

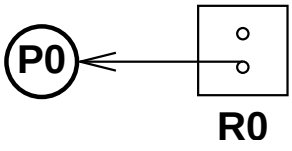
and *resources*, represented by squares, which may have a number of "dots" inside to indicate multiple equivalent *instances* of a resource.



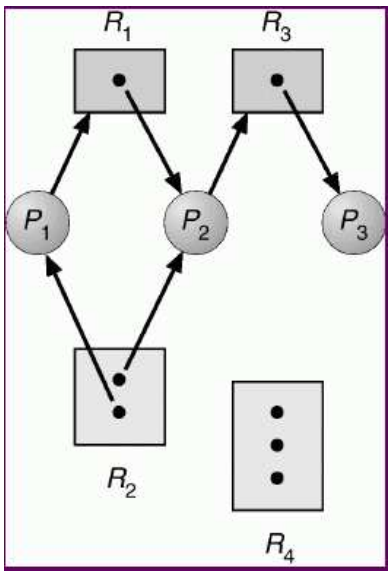
Edges from a process to a resource indicate a *request* by that process for an instance of that resource.



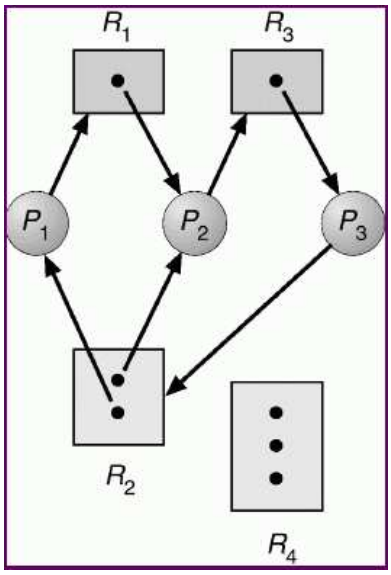
Edges from a resource to a process indicate that the resource has been *granted* to that process.



An example of a resource allocation graph:

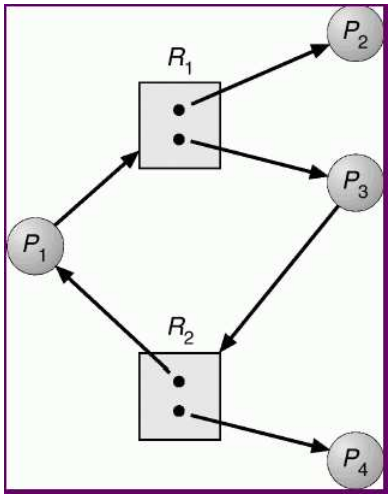


There are no cycles in this graph, so we are sure that there is no deadlock. There are two unfulfilled requests, P_1's request for R_1, and P_2's request for R_3, but we are sure things can continue. Since P_3 isn't waiting for anything, it will eventually release R_3. P_2 can then do its thing, allowing it to release R_1, which allows P_1 to do its thing. But what if P_3 requests an instance of R_2?



Deadlock! P_3 cannot continue until it acquires an R_2. But we know that P_2 can't continue until it gets an R_3, which isn't going to happen until P_3 can go. But P_1 is waiting for an R_1, which isn't available until P_2 can go... We have a problem. No process can continue.

However, a cycle alone is not enough to guarantee deadlock:



Here, there is a cycle, but P_2 and P_4 are not waiting for anything. That means that eventually they will free their resources, and the other processes can get what they need.

If there is a single instance of each resource, a cycle in the resource allocation graph means we have deadlock. When there are multiple instances, it only means there is a possibility of deadlock.

Dealing With Deadlocks

Most common approach: ignore the problem! It probably doesn't happen that much and it's hard to deal with, so forget about it and if it happens we'll just chalk it up to an OS bug or something. End of story.

But that's not good enough for us. We're going to look at three ways to deal with deadlock: *prevention*, *avoidance*, and *detection and recovery*.

Deadlock Prevention

Here, we make sure there is no deadlock by guaranteeing that one of the four necessary conditions cannot occur.

- *Mutual Exclusion*: This is a tough one to break, since a non-shareable resource is a non-shareable resource. But if nothing else, we should make sure that the resource really does require that concurrent access be restricted. But really, there's no help here.
- *Hold and Wait*: To break this, we'd have to require that a process that needs multiple resources requests them as a group. This may or may not be possible, and even if it is, it might cause a process that needs resource A for a long time, but resource B for a short time (concurrently), to hold on to B when it otherwise might be available for use by other processes. This can lead to low resource utilization and potentially to starvation.
- *No Preemption*: We could require that a process that fails to acquire a requested resource immediately gives back any other resource it already has. This is great, except in those cases where the resource really isn't preemptable. Taking away a CD burner when the CD is half written isn't a very good solution.
- *Circular Wait*: One possibility is to rank all resources, and only allow processes to request concurrent resources from low rank to high rank order. If you think about this a bit, you can convince yourself that this ensures that no cycles can form in the resource allocation graph. This may be difficult to implement, however, as an appropriate ranking may not be readily apparent.

Deadlock Avoidance

If we have some additional information available, we can use it to avoid deadlock.

For example, we could require that each process declare *a priori* the maximum number of each resource that it will ever need.

These values, along with the current number of resources allocated to each process and the number of resources available, form a resource allocation state.

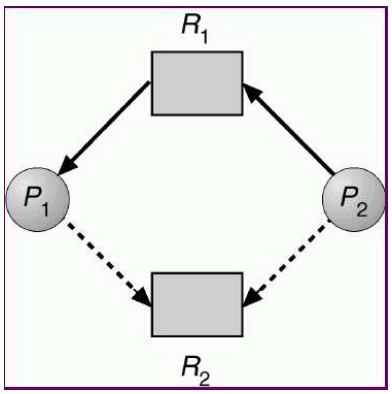
We want to ensure that the system stays in a *safe state*. A safe state is one in which, given the current allocations, there is guaranteed to be an ordering of the processes that will allow all processes to get the resources they need and (eventually) run to completion. The first process to run must be able to satisfy its resource needs with what it is allocated plus what is available. The next

process must be able to satisfy its needs with what it is allocated, with what is free, and what will become free when the first process releases its resources. And so on. We show that a state is safe by producing that *safe sequence*.

An unsafe state does not guarantee deadlock, as even if no such sequence exists, it's possible that one or more processes will not request the maximum allocation of a resource. So a process could complete and free resources, but maybe not.

By maintaining a safe state, we can ensure no deadlock.

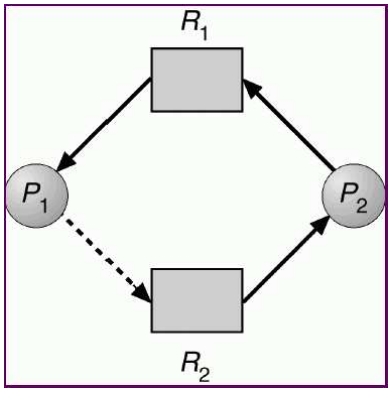
We can represent these "maximum claims" by augmenting our resource allocation graph to include *claim edges*:



Here, P_1 has been granted R_1 and has a claim edge indicating that it may also need R_2. P_2 is currently requesting R_1 and also may need R_2.

This is a safe state, as P_1, P_2 is a safe sequence.

Now suppose P_2 is granted R_2:



There is no deadlock yet, since there's a chance that P_1 will not need R_2 before completing, thereby freeing up R_1 for P_2. But the state is not safe, since we cannot guarantee a safe sequence. It's possible that P_1 *will* need R_2 before it releases R_1, and deadlock would follow.

For single-instance resources, we need only ensure that we do not grant any request that would introduce a cycle in the augmented resource graph.

For multiple instances of resources, we need something a little more complicated:

Banker's Algorithm (Dijkstra)

The decision to grant a resource or make a process wait even if the resource is available is made by checking if the state would become unsafe.

We use an example to see the Banker's Algorithm in action. (See class handout: bankers.pdf)

Deadlock Detection and Recovery

Since the Banker's Algorithm might be too expensive to run every time a resource is requested (in fact, at least one author, Tanenbaum, points out that no one uses it), we might just let processes get resources, then "once in a while" check for deadlock by running the "safety algorithm" or by checking for cycles in the resource allocation graph.

If we take this approach and discover that there is a deadlock, how to deal with it?

We could try to preempt a resource, though, that might force the process that had the resource to have to "undo" some of its computation and repeat it when it gets the resource back. This is complicated and can be costly (have you ever tried to "unwrite" a partially written CD?).

Since that option is so complex, we might just terminate processes to free their resources and have them restart. How do we choose a victim process? We could kill all deadlocked processes, but probably just one at a time until we can break the deadlock. Ideally we try to minimize the cost – avoid terminating a process that is almost done, or one that doesn't have enough resources to make much of a difference, or the boss' process.

Final Words on Deadlock

The statements that most systems do not worry about deadlock is a bit misleading. While no systems manage resources with something as elaborate as the Banker's Algorithm, many systems work to avoid deadlock in specific cases, such as with the allocation of file descriptors, process table entries, and memory.

While kernel deadlocks are infrequent in real systems, user process deadlocks are certainly common, which is why we study the 4 conditions.