Deadlocks

- A simple example
- Formal definition
- Resource allocation graph
- Deadlock Prevention
- Deadlock Avoidance
- The Banker's Algorithm
- Deadlock Detection and Recovery
- Starvation

A Simple example

```
void prod(int d) {
    P(mutex);
    P(empty);

    buf[in] = d;
    in = (in+1) % N;

    V(mutex);
    V(full);
}
```

```
#define N 8
int buf[N];
int in=0, out=0;
sem_t mutex=1;
sem_t empty=N;
sem_t full=0;
```

```
int cons(void) {
    int c;

    P(full);
    P(mutex);

    c = buf[out];
    out = (out+1) % N;

    V(mutex);
    V(empty);

    return c;
}
```

- This example differs from the producer-consumer example previously presented because the two semaphore operations in red have been exchanged
- Suppose that empty semaphore is zero (no free slots) and that a producer tries to produce a new item while no consumer is active
- When the producer is blocked waiting for P(empty) no consumer can consume data item since they should first acquire the mutex, blocked by the waiting producer and therefore cannot issue `V(empty)`
- The system hangs forever -> **Deadlock**

Formal Definition

- A deadlock can be defined formally as a situation in which a set of processes passively waits for an event that can be triggered only by another process in the same set.

- More specifically, when dealing with resources, there is a deadlock when all processes in a set are waiting for some resources previously allocated to other processes in the same set.

- A deadlock has therefore two kinds of adverse consequences:
  - the processes involved in the deadlock will no longer make any progress in their execution, that is, they will wait forever.
  - Any resource allocated to them will never be available to other processes in the system again.

- The following conditions are required for a deadlock to occur
  - ***Mutual exclusion***: Each resource can be assigned to, and used by, at most one process at a time. If any process requests a resource currently assigned to another process, it must wait.
  - ***Hold and Wait*** : For a deadlock to occur, the processes involved in the deadlock must have successfully obtained at least one resource in the past and have not released it yet, so that they hold those resources and then wait for additional resources.
  - ***Nonpreemption***: Any resource involved in a deadlock cannot be taken away from the process it has been assigned to without its consent, that is, unless the process voluntarily releases it.
  - ***Circular wait***: The processes and resources involved in a deadlock can be arranged in a circular chain, so that the first process waits for a resource assigned to the second one, the second process waits for a resource assigned to the third one, and so on up to the last process, which is waiting for a resource assigned to the first one.

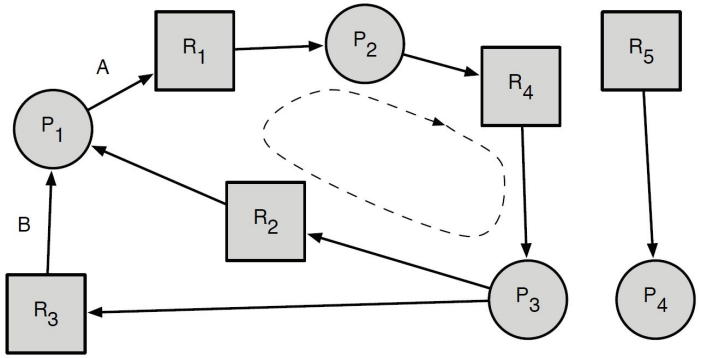
The Resource Allocation Graph

- A resource allocation graph is a directed graph with two kinds of nodes and two kinds of arcs.

- The two kinds of nodes represent the processes and resources of interest, respectively. In the most common notation,
  - Processes are shown as circles.
  - Resources are shown as squares.

- The two kinds of arcs express the request and ownership relations between processes and resources. In particular,
  - An arc directed from a process to a resource indicates that the process is waiting for the resource.
  - An arc directed from a resource to a process denotes that the resource is currently assigned to, or owned by, the process.

- If a cycle is found, then there is a deadlock in the system, and the deadlock involves precisely the set of processes and resources belonging to the cycle.



Deadlock Prevention

- The general idea behind all deadlock prevention techniques is to prevent deadlocks from occurring by making sure that (at least) one of the individually necessary conditions does not hold

- The ***mutual exclusion*** condition can be attacked by allowing multiple processes to use the same resource concurrently, without waiting when the resource has been assigned to another process.
  - This is not possible for several resources such as a lock protecting a critical section
  - Other resources such as printers, can be made shareable letting another task (a spooler) collect and enqueue requests for that resource

- The ***Hold and Wait*** condition is actually made of two parts that can be considered separately.
  - The wait part can be invalidated by making sure that no processes will ever wait for a resource.
    - One simple way of achieving this is to force processes to request all the resources they may possibly need during execution all together and right at the beginning of their execution.
  - The hold part may be invalidated by constraining processes to release all the resources they own before requesting new ones.
    - The new set of resources being requested can include, of course, some of the old ones if they are still needed, but the process must accept a temporary loss of resource ownership anyway. If a stateful resource, like for instance a printer, is lost and then reacquired, the resource state will be lost, too.

Deadlock Prevention - cont.

- The ***nonpreemption*** condition can be attacked by making provision for a resource preemption mechanism, that is, a way of forcibly take away a resource from a process against its will.
  - Like for the mutual exclusion condition, the difficulty of doing this heavily depends on the kind of resource to be handled.
    - Preempting a resource such as a processor is quite easy, and is usually done (by means of a context switch) by most operating systems.
    - Preempting ant operation already in progress in favor of another one entails losing a certain amount of work

- The ***circular wait*** condition can be invalidated by imposing a total ordering on all resource classes and imposing that, by design, all processes follow that order when they request resources. In other words, an integer-valued function f(Ri) is defined on all resource classes Ri and it has a unique value for each class. Then, if a process already owns a certain resource Rj , it can request an additional resource belonging to class Rk if, and only if, f(Rk) > f(Rj)

- Deadlock prevention implies putting into effect and enforcing appropriate design or implementation rules, or constraints. This may complicate life to programmers and reduces the flexibility in resource utilization (e.g. by imposing an ordering in the resources to be used)

- Such constraints can be relaxed when relying on ***Deadlock Avoidance*** schemas, that is, avoiding a resource allocation in case this may lead to a deadlock condition

Deadlock Avoidance

- Unlike deadlock prevention algorithms, deadlock avoidance algorithms take action later, while the system is running, rather than during system design. As in many other cases, this choice involves a trade-off:
  - on the one hand, it makes programmers happier and more productive because they are no longer constrained to obey any deadlock-related design rule.
  - On the other hand, it entails a certain amount of overhead.

- The general idea of any deadlock avoidance algorithm is to check resource allocation requests, as they come from processes, and determine whether they are safe or unsafe for what concerns deadlock.
  - If a request is deemed to be unsafe, it is postponed, even if the resources being requested are free.
  - The postponed request will be reconsidered in the future, and eventually granted if and when its safety can indeed be proved.

- Usually, deadlock avoidance algorithms also need a certain amount of preliminary information about process behavior to work properly.

- The Banker's algorithm can be used to check whether granting requested resources is safe

The Banker's Algorithm: used structures

A (column) vector **t**, representing the total number of resources of each class initially available in the system:

$$\mathbf{t} = \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix}$$

n=number of processes
m=number of resource classes

A matrix **C**, with m rows and n columns, that is, a column for each process and a row for each resource class, which holds the current resource allocation state:

$$C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & \dots & & \dots \\ \dots & & & \dots \\ c_{m1} & \dots & & c_{mn} \end{pmatrix}$$

A matrix **X**, also with m rows and n columns, containing information about the maximum number of resources that each process may possibly require, for each resource class, during its whole life:

$$X = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & \dots & & \dots \\ \dots & & & \dots \\ x_{m1} & \dots & & x_{mn} \end{pmatrix}$$

An auxiliary matrix **N**, representing the worst-case future resource needs of the processes. N = X − C

A vector **r**, representing the resources remaining in the system at any given time and a vector **qj** representing the request coming from the j-th process

$$\mathbf{r} = \begin{pmatrix} r_1 \\ \vdots \\ r_m \end{pmatrix} \qquad \mathbf{q_j} = \begin{pmatrix} q_{1j} \\ \vdots \\ q_{mj} \end{pmatrix}$$

Banker's Algorithm

1. It checks whether the request could, in principle, be granted immediately or not, depending on current resource availability. Since **r** represents the resources that currently remain available in the system, the test can be written as **qj ≤ r**
   a. If the test is satisfied, there are enough available resources in the system to grant the request and the banker proceeds with the next step of the algorithm.
   b. Otherwise the request cannot be granted immediately, due to lack of resources, and the requesting process has to wait.

2. The banker simulates the allocation and generates a new state that reflects the effect of granting the request on resource allocation (cj), future needs (nj), and availability (r), as follows:

$$\begin{cases} \mathbf{c_j}' &= \mathbf{c_j} + \mathbf{q_j} \\ \mathbf{n_j}' &= \mathbf{n_j} - \mathbf{q_j} \\ \mathbf{r}' &= \mathbf{r} - \mathbf{q_j} \end{cases}$$

3. Then, the new state is analyzed to determine whether it is safe or not for what concerns deadlock. If the new state is safe, then the request is granted and the simulated state becomes the new, actual state of the system Otherwise, the simulated state is discarded, the request is not granted immediately even if enough resources are available, and the requesting process has to wait.

Banker's Algorithm: checking whether a state is safe

- To assess the safety of a resource allocation state, the banker uses a conservative approach. It tries to compute at least one sequence of processes—called a ***safe sequence***—comprising all the n processes in the system and that, when followed, allows each process in turn to attain the worst-case resource need it declared, and thus successfully conclude its work.

- The safety assessment algorithm uses two auxiliary data structures:
  - A (column) vector **w** that is initially set to the currently available resources (i.e., **w** = **r** initially) and tracks the evolution of the available resources as the safe sequence is being constructed.
  - A (row) vector **f**, of n elements. The j-th element of the vector, **fj**, corresponds to the j-th process: fj = 0 if the j-th process has not yet been inserted into the safe sequence, fj = 1 otherwise. The initial value of f is zero, because the safe sequence is initially empty.

- The algorithm tries to find a new, suitable candidate to be appended to the safe sequence being constructed. In order to be a suitable candidate, a certain process Pj must not already be part of the sequence and it must be able to reach its worst-case resource need, given the current resource availability state, i.e. **fj = 0** (Pj is not in the safe sequence yet) $\wedge$ **nj ≤ w** (there are enough resources to satisfy nj)
  - If no suitable candidates can be found, the algorithm ends
  - After discovering a candidate, it must be appended to the safe sequence: **fj := 1** (Pj belongs to the safe sequence now) **w := w + cj** (it releases its resources upon termination)

- Then, the algorithm goes back to step 1, to extend the sequence with additional processes as much as possible.if fj = 1 $\forall$j, then all processes belong to the safe sequence and the simulated state is certainly safe for what concerns deadlock.

Deadlock Detection

- The deadlock prevention approach poses significant restrictions on system design, whereas the banker's algorithm presented requires information that could not be readily available and has a significant run-time overhead.

- To address these issues, a third family of methods acts even later than deadlock avoidance algorithms. That is, these methods allow the system to enter a deadlock condition but are able to detect this fact and react accordingly with an appropriate recovery action. For this reason, they are collectively known as ***deadlock detection and recovery*** algorithms.

- If there is only one resource for each resource class in the system, a straightforward way to detect a deadlock condition is to maintain a resource allocation graph, updating it whenever a resource is requested, allocated, and eventually released. Since this maintenance only involves adding and removing arcs from the graph, it is not computationally expensive and, with a good supporting data structure, can be performed in constant time.

- The resource allocation graph is examined at regular intervals, looking for cycles. The presence of a cycle is a necessary and sufficient indication that there is a deadlock in the system.

- If there are multiple resource instances belonging to the same resource class, this method cannot be applied. On its place we can, for instance, another algorithm, due to Shoshani and Coffman,similar to the banker's algorithm

- It can be executed at regular times and does not require knowing in advance the maximum number of resources that can be allocated by every process

Starvation

- Deadlock is only one aspect of a more general group of phenomenons, known as indefinite wait, indefinite postponement, or starvation.
- In the banker's algorithm discussed when more than one request can be granted safely but not all of them, a crucial point is how to pick the "right" request, so that no process is forced to wait indefinitely in favor of others.
- More in general, when several tasks cyclically compete for the same resource, it may happen that one 'unlucky' task, for any reason, will never acquire the resource because every time it tries accessing, there is another competing task that wins
- In other works, it is not possible to define an upper bound in the wait time for that resource
- Several algorithms take into account starvation and guarantee that any requesting task will eventually have its request satisfied
    - For example the Peterson Mutual Exclusion algorithm ensures that two competing tasks will gain alternatively access to the critical section