

Classic problem: dining philosophers

**Lecture 9
Deadlocks**

The last lecture mainly focus on how to **restrict** processes so that they can't interfere with each other and do something wrong.

by saying that two threads or processes must not be both running the same portion of the code, and if they try, one have to wait.

It turns out that it is possible to **restrict too much**: in such scenarios, all processes wait and cannot do useful work.

In this lecture, we study how the problem arises and how to prevent it.

References:

- [OSC] Chapter 8 (Deadlocks).

POS(0230A)

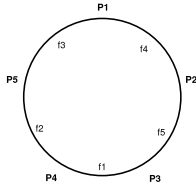
Consider the following classic problem:

Problem:

Five philosophers sits around a round table.
Placing around them are 5 forks.

The philosophers insist that they must hold the two forks besides him before eating.

At any time, a philosopher can be thinking, eating or hungry.



Why we are interested in it?

A lot of resources sharing problem are very similar to dining philosophers: the philosophers are the threads (or processes), the forks are the shared resources.

POS(0230A)-9.1

Deadlocks

<p>Solution</p> <p>Each philosopher uses the following algorithm when they are hungry:</p> <ul style="list-style-type: none"> • Wait until the left fork is available. • Pick up left fork. • Wait until the right fork is available. • Pick up right fork. 	<p>Evaluation</p> <p>Is the solution "right"? Yes, in that it does guarantee a philosopher holds two fork before eating.</p> <p>But... if all philosophers become hungry at the same time and pick up the left fork...</p> <p>No philosopher can eat, since each holds a fork waiting for another.</p>
--	---

This scenario is called **deadlock**. No progress can be made.

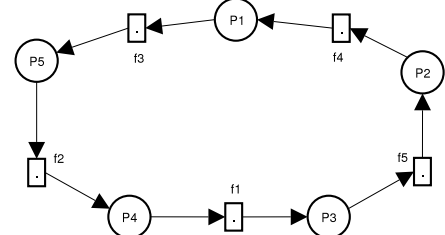
Back to the OS view, each process or thread involved is waiting for some resource that is held by another, so none makes any progress.

POS(0230A)-9.2

Abstraction of deadlock: RAG

What are the important ingredients for the deadlock scenario? They include the **resource**, the **processes**, the **requests**, and the **acquisition**.

The RAG (Resources allocation graph) represents exactly this.



If we have multiple copies of the same resources (so that a process requests for any of them), we have multiple dots in the same resources box, and the maximum number of out-going edges is the number of copies of the resources.

POS(0230A)-9.3

Requirements for deadlock

What is required before there can be a deadlock?

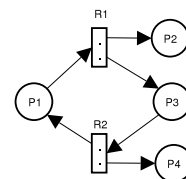
- **Mutual exclusion.** There must be some restrictions placed on the processes. In particular, the acquisition of a copy of a resource prevents others to use the same copy of resource.
- **Hold and wait.** There must be processes which hold a resource, not using it, but instead wait for another resource to become available.
- **No preemption.** The held resource cannot be preempted from waiting processes.
- **Circular wait.** The processes involved must form a cycle P1, P2, ..., Pn in which P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3, etc.

Without any of these ingredients, a deadlock can never occur. So one way to combat deadlocks is to deny one of them.

POS(0230A)-9.4

Cycle without deadlock

When multiple copies of a resource may be available, a cycle may not imply a deadlock:



P1, R1, P3, R2 forms a cycle. But **P2 and P4 will eventually complete** using R1 and R2 respectively, allowing P1 and P3 to get the resources they need.

How to detect deadlock when there may be multiple resources? OSC section 8.6.2 gives an algorithm. But soon we will learn a more generic algorithm anyway, so we skip it to avoid learning the same trick twice.

POS(0230A)-9.5

Handling deadlocks

Having all processes waiting is not good for any OS. How to deal with it?

- **Prevent deadlocks:** ensure that a system cannot be in a deadlock state, by making sure that one of the ingredients can never occur.
- Ask applications to provide more information about possible requests, thus allowing deadlocks to be **avoided by rejecting some requests**.
- Detect deadlocks and **recover from it** by terminating a process or by preempting some resources.
- **Ignore deadlocks**, and let users to deal with it by their own ways.

Within the kernel, most OSs have some quick prevention mechanisms.

Outside the kernel, applications must deal with them by their own means. Some OS detects deadlocks upon resources (e.g., semaphore) acquisition and deny the request in case deadlock is detected.

POS(0230A)-9.6

Prevention of deadlock 1: sharing and preemption

If the resource themselves are sharable or may be preempted, we don't have a problem of deadlock.

- **Sharable:** e.g., if many processes need to read the same file but not write to it, then all can be done at the same time.
- **Preemption allowed:** e.g., if many processes need some buffers in memory but the memory is not large enough, some memory may be moved to disk temporarily so that one process can continue.
So physical memory is in general not source of deadlock, but swap space on disk is.

When we talk about deadlocks, we focus on resources that cannot be shared or preempted.

E.g., one might want to acquire a mutex for accessing a linked list, or want to start using the printer, or want to lock some buffers to physical memory for disk transfer.

POS(0230A)-9.7

Prevention of deadlock 2: no wait on hold

One alternative is to attack the hold-and-wait requirement.

- If a process needs multiple resources, it gets them **one by one**.
- If any of the resources is not available, it **releases all** the resources that it has acquired, and try again after some time.

So whenever a process waits for resources, it hold no resources. So there is no deadlock. There is some serious drawbacks, though:

- Programs usually have to **acquire resources longer**.
E.g., for 3 resources A, B, C: if I need A uninterrupted for the whole day, B during in the morning, and C during the night, I have to acquire everything at in the morning, or I risk losing A when I acquire C.
- The system is **not very efficient**, and **starvation** may occur.
Imagine each philosopher pick up their left fork, put it down, and repeat.

In general, this is suitable if resources is **lightly used**.

POS(0230A)-9.8

Prevention of deadlock 3: no cycles in resources

One alternative to deal with deadlocks is to attack circular wait:

- All resources are arranged into a list of **priorities**.
- Whenever a process holds any resource of a low priority, it cannot request for resource of high priority.
E.g., always pick the even-numbered fork before the odd-numbered fork.

Now following a link in RAG always get from resources of high to low priority, so **a cycle can never be formed**: no deadlock.

It has some **disadvantage** as well:

- Resources must be acquired in an order determined by the system, not their usage. Some resources must be acquired before they are need.

Example use: most OS arranges all its internal mutex in some order. A system call requiring multiple of them must get them in this order.

POS(0230A)-9.9

Avoiding deadlocks

Deadlock prevention involves disallowing some request sequences. This generally **reduce throughput**, by requesting early or releasing late.

Alternative: **allow all request sequences**, but require extra information from processes like **maximum number of resources needed**.

When a resource is requested, the system **checks whether there is a danger of deadlock**, and if so **blocks the process**.

Example: suppose we have 12 tape drives, and processes A, B and C declares they need at most 10, 4 and 9 of them.

If process A, B and C requests for 5, 2 and 2 drives, the system is safe from deadlock: Even if all requests for maximum number of drives, we can serve B first, then A and finally C.

But if C now requests for one more drive, then the system is in danger. (Why?!) So **C is asked to wait**.

POS(0230A)-9.10

The Banker's Algorithm

How to know it is safe?

- The OS keeps a list **Available**, stating the number of copies of each resources that can be allocated.
- Each process declares a list **Max**, stating the maximum number of copies of resources that it may need.
Obviously, Max must be smaller than Available for each entry.
- The OS keeps a list **Allocation** for each process, stating the current number of allocated resources to the process.
- For each process, we can calculate a list **Need** by subtracting Allocation from Max, which is the maximum number of additional copies required.

Given the Available list, and the Max and Allocation list for all processes, we want to know whether it is possible to schedule the processes to completion.

POS(0230A)-9.11

The Banker's Algorithm, cont'd

The idea: we **assume the worst**, that each process does requests the maximum needed resources. We see **whether all processes can finish**.

- Repeatedly do the following:
 - Find all processes with $Need \leq Available$ for all entries. Delete these processes from consideration, and add their Allocation list to Available.

Intuitively, these processes can finish because the system can allocate the resources they need, if it does it one by one, and doesn't allow other processes to get new resources.
- The current situation is safe if and only if all processes are removed.

POS(0230A)-9.12

Example

Suppose we have the following resources: 10 A's, 5 B's, 7 C's. Currently there are 5 processes, with the following resources held/maximum requirements.

Process	A	B	C
0	0/7	1/5	0/3
1	2/3	0/2	0/2
2	3/9	0/0	2/2
3	2/2	1/2	1/2
4	0/4	0/3	2/3

Is it currently safe? What if process 0 now want 1 A and 2 C's?

POS(0230A)-9.13

Deadlock detection

How to detect deadlock has already occurred, without having each process declares the amount of resources it will later requests?

- Just **use the banker's algorithm**...
- But replace the Need array by the amount of resources each process is currently requesting.
- And use the Allocation array as usual.

In a sense, it is the optimistic view that **no process will ask for any more resources** after the current request.

POS(0230A)-9.14

Recovery from deadlocks

Sometimes maximum amount of resources are seldom used completely. In this case even the deadlock avoidance scheme is not efficient. It causes many process to wait unnecessarily.

Example: if a process allocates a large chunk of memory, that memory might need to be put into swap. But in reality, most of the memory might never be used, so need no swap. ("OOM": out-of-memory situation.)

One may choose to deal with deadlock by recovering from it:

- Detecting** a deadlock condition, usually by using the RAG.
- Selecting** a process to stop. If that does not get the system back to a state without some deadlock, this has to be repeated.
- Alternatively, one might **rollback** the process to some previous state when there is no deadlock. Linux 2.4 deals with OOM by a "OOM killer", killing processes to recover swap.

POS(0230A)-9.15

Summary

- A deadlock can occur if mutual exclusion, hold and wait, no preemption, circular wait all hold for some resources.
- A RAG can be used to check the current allocation status, and detect whether a deadlock has occurred.
- Depending on the frequency of deadlock conditions and the accuracy of maximum resources requirement, one may choose to ignore the deadlock, recover from it, avoid it or prevent it.
- Within the OS**, deadlock prevention is done to scarce resources like mutexes. This is done by denying circular wait.
- To abundant resources like swap space, OS usually either ignore deadlock, or detect it and recover from it.
- Outside the OS**, the application writers usually have to use their own ways to deal with deadlocks.

POS(0230A)-9.16