CS5600 Scribe notes

Ben Cordes

29 Sep 2010

1 Dining Philosophers

The *dining philosophers problem* is a classic synchronization problem. Picture five philosophers sitting around a dinner table. There's a large bowl of rice and five chopsticks on the table. Philosophers can be in three states: thinking, eating, and hungry. A philosopher can only eat if the chopstick to his left and the chopstick to his right are both available.

In synchronization parlance, each chopstick is a semaphore and every philosopher is trying to DOWN chopstick(i) and ($i + 1\%5$). But it's easy to see how all philosophers would get stuck in HUNGRY because they can only get one semaphore and they're stuck waiting for the other one. This is the concept of *deadlock*.

2 Synchronization: Transactions

We would like to be able to run multiple transactions at a time. If you put a mutex/lock around the transaction you can only have one at a time. But Transaction 1 could be modifying memory A while Transaction 2 modifies memory B, and we'd like these to be able to run in parallel but with some ordering.

Serializablilty implies that the result looks like there was some ordering of the transactions (1-2-3 or 1-3-2 or 2-3-1) as if they were atomic. Given n transactions there are $n!$ possible orderings assuming they are hitting the same memory location.

```
T0: rd(A) wr(A) rd(B) wr(B)
T1:                         rd(A) wr(A) rd(B) wr(B)
```

That *schedule* is serializable because it looks like T0 ran completely, then T1 ran completely.

```
T0: rd(A) wr(A)             rd(B) wr(B)
T1:             rd(A) wr(A)             rd(B) wr(B)
```

This schedule is also serializable; the result is the same as the previous schedule so it LOOKS like the transactions were ordered 0-1.

For a formal definition of serializable, we need to define *conflicting operations*. Operations i and j *conflict* if they are sequential (i.e. they follow each other in the schedule), from different transactions, they access the same data item, and at least one is a write. If two operations do not conflict, then we can reverse their ordering without problem. So each of these steps is okay:

```
T0: rd(A) wr(A)       rd(B)       wr(B)
T1:             rd(A)       wr(A)       rd(B) wr(B)

T0: rd(A) wr(A) rd(B)             wr(B)
T1:                   rd(A) wr(A)       rd(B) wr(B)

T0: rd(A) wr(A) rd(B) wr(B)
T1:                         rd(A) wr(A) rd(B) wr(B)
```

Thus the formal definition of serializable is that we can continually switch non-conflicting operations until we arrive at a schedule that is actually serialzed (i.e. the last schedule).

An example of a non-serializable schedule:

```
T0: rd(A) wr(A) rd(B)       wr(A)
T1:                   wr(A)       rd(B) wr(B)
```

The two writes to A can't be swapped so this schedule is not serializable – and that's bad. This is not a valid interleaving of operations.

Implementing transactions requires two types of locks, shared and exclusive. Every access to memory acquires a lock (reads use the shared lock, writes require an exclusive lock). Many threads can hold a shared lock at the same time but only one thread can hold an exclusive lock at once. But this isn't sufficient to guarantee serializability:

```
T0: rd(A) wr(A)             rd(A) wr(A)
T1:             rd(A) wr(A)             rd(A) wr(A)
```

This isn't serializable, but the locking mechanisms will work perfectly (if you lock before the read and release after the write).

We add the additional restriction that you can acquire as many locks as you like (the *growing phase*), but as soon as you release one lock you can never acquire any more (the *shrinking phase*). This gurantees serializability but does not prevent deadlock:

```
T0: rd(A) wr(A)             rd(B) wr(B)
T1:             rd(B) wr(B)             rd(A) wr(A)
```

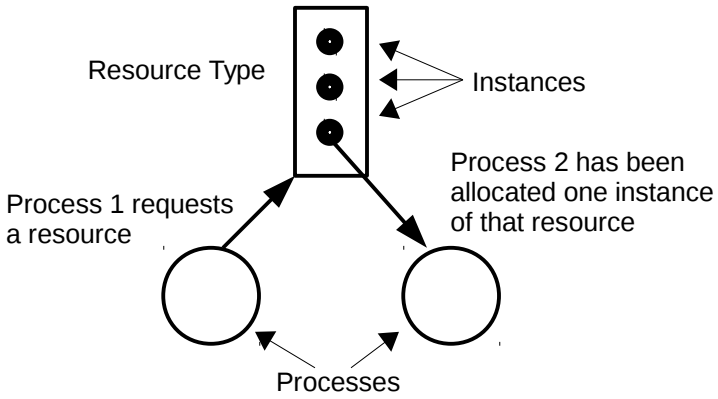

Figure 1: Deadlock graphs

2.1 Deadlock

More formally, deadlock is a set of processes, each holding a resource and each waiting to acquire a resource held by another process in the set. Given a set of processes $\{P_1 \dots P_n\}$ and a set of resources $\{R_1 \dots R_n\}$ which may have multiple instances which are identical (meaning that any instance of a resource is equivalent from the process's perspective, e.g. any empty page in memory). In order to have deadlock in this model, four things must be true:

- *mutual exclusion*, meaning that only a single process can use each instance of a resource
- *non-preemption*, meaning that there is no way to force a process to give up a resource
- *hold and wait*, meaning that there must a process holding one instance and waiting for another at the same time
- *circular wait*, meaning that there is a cycle of processes such that one process is waiting a resource held by the next

This model implies that we can visualize deadlock as a graph (see Figure 1). A graph G consists of vertices and edges (V, E), with processes P_n, resource types R_n, and instances $(W_n)_m$ as above. We then draw boxes of resource types with dots within for instances, and circles of processes. If a process needs a resource, we draw a *request edge* from the process P to the box for the resource type R. An *assignment edge* is drawn from an instance dot W to a process circle P. If you can find a cycle in this graph, there is deadlock.

Proof that a cycle is **necessary**: Assume that there can be a deadlock with no cycle. Pick one process. What resources is that process waiting on, and what

processes are the instances of those resource assigned to? Because there is no cycle, you can never end up at the process you started at, and wherever you end up, at least that process can make progress.

Proof that a cycle is **not sufficient**: If there are multiple instances of a resource and one of them is assigned to a process outside the cycle, it is possible for that thread to make progress and release the instance, at which case you can break the cycle. So having a cycle is sufficient if there is only one instance of each resource, but given multiple instances it is not sufficient. (If there are multiple instances of a resource, then deadlock exists if every process that has an instance of that resource is in a cycle.)

How do we ensure that deadlocks never happen? If we can design a system such that one of the four principles can never be true, then we'll never have deadlock.

- A system without mutual exclusion doesn't really work.
- A system without hold and wait implies that while waiting, you can't hold any resources. So you have to request all of your resources at once, and they're all granted at once and held for the entire time. This isn't great because maybe you only need one lock for a very short part of your critical period. It also potentially leads to starvation.
- A system with pre-emptable resources implies that the OS can take away your lock in the middle of your critical section, which is no good.
- The only thing we're left with is preventing circular wait. But searching the graph for a cycle is expensive. One implementation is to only allow threads to request resources in increasing priority. The proof says that if you have a cycle of processes that are waiting on resources in a loop, then you have a cycle of resources where each must have a higher priority than the next, which is impossible. This has a similar problem to the no-hold-and-wait case, plus you need to know the priorities in advance and make requests in the right order.

Deadlock prevention is not widely implemented because all of these methods are very cumbersome. So instead we implement *deadlock avoidance*. In general, the OS asks for extra information up front and only allows the process to proceed if doing so probably won't cause deadlock. For example, if we have 100 pages of memory and two processes that want to run request a maximum of 60 pages each, then we can't run both of them at the same time because we might cause deadlock.

More formally, we ask each process P for the maximum number of each resource R that it will need. Then we define that a state is a *safe state* if I can order all of the processes such that for each process P_i, its remaining requests can be satisfied by a combination of free resources and resources held by $P_0 \dots P_{i-1}$. That means that P_0's requests can be satisified by free resources only, which means that P_0 can run to completion because it's not waiting for anybody else. Then P_1's requests can either be satisfied immediately by free

resources, or by waiting until P_0 completes, and thus P_1 is guaranteed to be able to complete.

Safe states can not lead to deadlock. Unsafe states might lead to deadlock. Deadlocks are always unsafe states.

Given 12 available instances and three processes: P_0 needs 10 and has 5, P_1 needs 4 and has 2, and P_2 needs 9 and has 2. So 9 instances are currently used. But the ordering P_1, P_0, P_2 is a safe state. Now let's say we're considering giving P2 a third resource. But if we do so, we reach an unsafe state because although P_1 can still satisfy its requests with free resources, no one can after that.