Lecture [redacted]

[redacted]

[redacted]

Review of Deadlocks

Last class, we established that four conditions must occur simultaneously in order to have a deadlock:

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

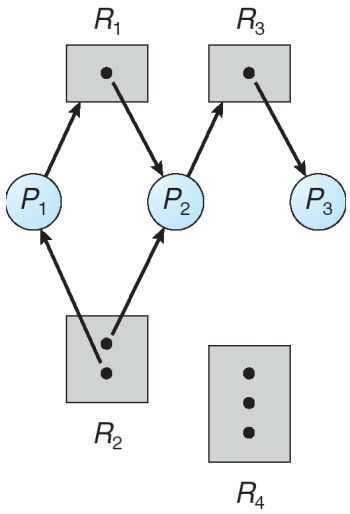
We will focus on circular wait, as it can be broken (more generally than the other three) in order to deal with deadlocks in a system.

Review of Resource Allocation Graphs

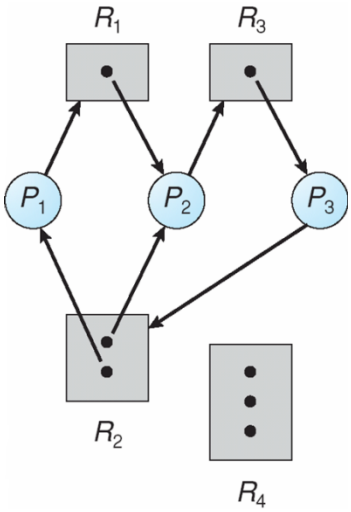
A resource allocation graph represents certain properties of which entities are holding or requesting which resources. It has two types of edges:

- Request made by a process Pi for a resource Rj (**request edge**): Pi → Rj
- Process Pi currently using a resource Rj (**assignment edge**): Rj → Pi

The presence of a cycle in the graph indicates circular wait and vice versa. We will use this property to deal with deadlocks. For example, the following graph shows no cycle, and therefore no circular wait:



On the other hand, the graph below contains two cycles, and therefore circular wait exists in the system. In this case, the processes end up exhausting R_2, resulting in a deadlock.



Basically, if the graph contains no cycles, there is no deadlock. If there is a cycle, the situation can be broken down into two cases: one instance of a resource (deadlock) and more than one instance of the resource (*possibility* of deadlock).

Handling Deadlocks

There are three ways to handle deadlocks:

1. We can ensure that the system will never enter a deadlock state. This method uses extra memory to maintain the resource allocation graph and extra CPU cycles to update the graph.
2. We can allow the system to enter a deadlock state and then recover (the "optimistic" solution). This method assumes that deadlocks are relatively rare.
3. We can ignore the problem and pretend that deadlocks never occur in the system. Most operating systems (including UNIX) use this method. We will not look at this method since there isn't much to implement.

Deadlock Prevention

The idea of deadlock prevention is to restrain the ways the request can be made such that one of the four conditions required for deadlock will not arise. We cannot prevent mutual exclusion for non-sharable resources (such as printers and critical sections), since these operations must be atomic.

To deal with hold and wait, we must require that whenever a process requests a resource, it does not hold any other resources. This can be accomplished by either (1) making sure that the process has been allocated all its resources before beginning execution, or (2) only allowing the process to request resources when it is not currently allocated any resources. With this method, some processes will end up waiting because only a subset of their required resources are available, even though it's possible that they could proceed given only that subset. This results in low resource utilization. It's also possible that a process could starve waiting for the perfect combination of available resources.

To eliminate no preemption, we must ensure that if a process holding some resources cannot be immediately allocated a requested resource, then that process releases all the resources it currently holds. These preempted resources are then added to the list of resources for which the process is waiting, and the process can only be restarted once it can regain its old resources, as well as the new one.

To prevent circular wait, we can impose a total ordering of all resource types, and then require that each process requests resources in an increasing order of enumeration.

Deadlock avoidance

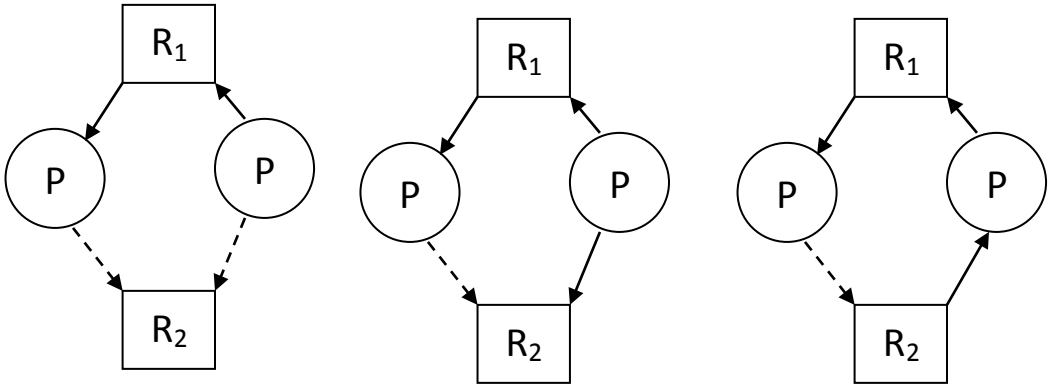
In order to avoid deadlocks, the system must have access to certain information about its processes and resources. The simplest avoidance model requires each process to declare a maximum number of resources for each resource type that it may need. The deadlock-avoidance algorithm dynamically examines the resource allocation graph to make sure that there are no cycles (circular wait). In this way, it ensures that the system always remains in a *safe state*. We define a safe state such that there exists a sequence of processes $<P_1, P_2, \ldots, P_n>$ of **all** the processes in the system such that for each P_i, the resources that P_i can still request can be satisfied by the currently available resources and the resources held by all the P_j, with $j < i$. That is, if the requested resources are not immediately available, P_i can wait until all the P_j have finished executing, at which point it will be allocated the requested resources.

If a system is in a safe state, then there are no deadlocks. On the other hand, if the system is in an unsafe state, then there is a possibility of a deadlock. The idea of deadlock avoidance is to make sure that the system never enters an unsafe state, and thus never incurs a deadlock. Unfortunately, this method can negatively impact performance because it is conservative: not *all* unsafe states will ultimately result in deadlock (the deadlock states are only a subset of the unsafe states). Even though it is not always strictly necessary to prevent an unsafe state, this method does it anyway.

When there is a single instance of a resource type, we can use the resource allocation graph to implement avoidance. For multiple instances of a resource type we use the banker's algorithm, which will be covered in the next class.

To use resource allocation graphs to implement deadlock avoidance, we must add a third edge type: the claim edge. This edge goes from P_i to R_j, and is represented by a dashed line. It indicates that P_i may request R_j. When the process actually requests the resource, the claim edge converts into a request edge (note that this may not actually happen). The request edge is eventually converted into an assignment edge when the process is allocated the requested resource, and finally the assignment edge reconverts to a claim edge when the process finishes and releases the resource.

In the following example, P_1 owns R_1, P_2 wants to use R_1, and both *may* request R_2. Then P_2, requests R_2, and finally P_2 is assigned R_2. This final picture shows a cycle in the resource allocation graph, indicating a possibility of deadlock. Note that in this case, there is no deadlock, but the system is still in an unsafe state.



We will revisit deadlocks in the next lecture.

Exam Review

The second part of the lecture is doing some practice for Mid-term exam.

The three themes for Mid-term exam are:

- T1: Architecture basics, hardware support for OS
- T2: Processes, Threads, Scheduling
- T3: Synchronization

The following are the sample questions we went through in class. The answer will be given with brief explanation for each question.

- **True/False with one sentence explanation:**

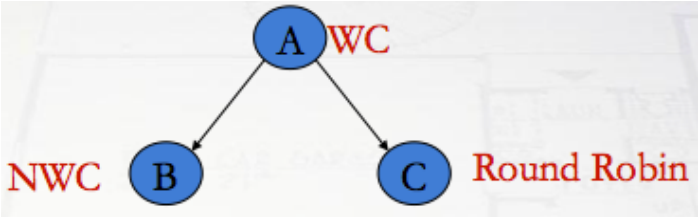
- Que: A second trap can occur during the handling of an initial trap
- Answer: Yes, if an instruction belonging to the trap handler for the first trap causes a trap, e.g., a segmentation fault.

For example, after the initial trap occurs, the trap handler will be called to deal with this trap. However, the trap handler is also a piece of code which could have mistakes and cause errors. If an instruction in this trap handler goes wrong, another trap is raised.

- Que: If any scheduler within a hierarchy of schedulers is non-work conserving (NWC), the entire hierarchy is also NWC.
- Answer: False. Use a counter-example.

WC means the resource (e.g., CPU) cannot be idle if there is some work to be done (e.g., at least one ready process).

NWC means that even if the resource is available, the work cannot be done due to some reasons, such as reservation.



In the above example, even if B is NWC, the entire hierarchy is WC. The reason is if B cannot utilize the resources right now (because B is NWC), scheduler A will immediately assign the resource to C to guarantee that the resource is not idle. But if C is also NWC, then the entire system is NWC. Because even if A assign the resource to C, C may not be able to utilize the resource at once due to NWC.

– Que: SJF offers better throughput than FCFS.
– Answer: False. Both offer the same throughput. Throughput means number of processes that complete their execution per time unit.

– Que: SRPT offers the same throughput as SJF. Assume that context switches take a finite, non-negligible amount of time.
– Answer: No. SRPT may cause more context switching than SJF, hence may lower throughput. The length of jobs is fixed, but the remaining time of the jobs will change. Thus SRPT may cause more context switching than SJF. The throughput will be lower if more time is spent on context switching.

– Que: The OS is made to reside in the address space of every process to allow improved usage of DRAM.
– Answer: False. This is done to reduce user/kernel switches from being as expensive as full context switches.

– Que: Threads in a concurrent program based on a user-level threads library can access each others stacks unlike those in a program based on kernel-level threads.
– Answer: True. In user-level threads, the stacks of all the threads reside in the same process' heap. Therefore they can access each other's stack. But in kernel-level threads, each thread own a separate stack. They cannot access each other's stack as user-level threads.

– Que: Semaphores completely eliminate busy waiting.

– Answer: No. They must employ locks to make their wait() and signal() functions atomic. They have to busy waiting for locks to be available.

- **Multiple-choice:**

– Que: Which of the following is essential to build a multi-programmed computer: (i) traps, (ii) interrupts, (iii) signals, (iv) TLB.
– Answer: (ii)
Only interrupts are strictly essential to build a multi-programmed computer. Interrupts give OS the chance to run itself and other processes on CPU. The system could die without traps (the running process may go wrong and stop there), but traps are not necessary for building multi-programmed computer.

Signals are used for communication between processes. TLB translates virtual addresses into physical addresses. They are both designed for better performance and efficiency of multi-programmed computer, but not necessary.

– Que: Which of the following liveness conditions implies the other two: (i) no starvation, (ii) no deadlock, (iii) bounded wait, (iv) none of the above
– Answer: (iii)
Bounded wait implies no starvation. No starvation implies no deadlock.

– Que: Which of the following styles of writing a concurrent program causes the most use of kernel memory: (i) user-level threads, (ii) kernel-level threads, (iii) processes, (iv) all are the same
– Answer: (iii)
If the program is implemented with n processes, it costs kernel memory n * sizeof (PCB) to keep PCBs.

If the program is implemented with n kernel-level threads, it costs kernel memory 1 * sizeof (PCB) + n * sizeof (TCB) to keep one PCB and n TCBs.

If the program is implemented with n user-level threads, it costs kernel memory 1 * sizeof (PCB) to keep only one PCB. This is because OS doesn't know the existence of user-level threads. In OS' eye, only one process exists.

- **Short-answer Questions**
  (Sample questions come from Practice Question Set)

– Question 1 from Set 1: Why is the CPU designed so that (in addition to executing instructions) its circuitry checks for the occurrence of traps or interrupts upon every clock cycle?
– Answer: This is designed to guarantee the correctness of the system. The CPU may continue to execute some wrong instructions if it does not check. By checking every clock cycle, CPU could promptly capture the traps and interrupts occurred and try to deal with them.

– Question 5 from Set 2: Consider a system employing lottery scheduling for its CPU. Suppose there are two processes P1 and P2 that are always ready to run and never enter the waiting state. Furthermore, we assign them CPU weights of 1 and 2, respectively. What is the probability that P1 will be scheduled to run for 10 successive quantums. Recall that a quantum is a duration for which a process is guaranteed to run (unless it yields the CPU) whenever it is chosen by the CPU scheduler.
– Answer: $\left(\frac{1}{3}\right)^{10}$.
Each time "P1 is scheduled to run" is independent discrete event.

So the probability that P1 be scheduled to run for once is $\frac{1}{3}$

The probability that P1 be scheduled to run for twice is $\frac{1}{3} * \frac{1}{3} = \left(\frac{1}{3}\right)^2$

….

….

The probability that P1 be scheduled to run for ten times is $\left(\frac{1}{3}\right)^{10}$

– Question 11 from Set 2 : Consider a faulty timer interrupt that, instead of interrupting the CPU once every 10 msec as requested by the OS at bootup time, interrupts it once every 20 msec. What do you think the implications of this would be on the functioning of the OS and the processes on this computer? Would the IO devices attached to this computer (say the hard disk drive or the network interface card) appear faster or slower than they actually are to the OS? (The answer is, obviuosly, yes. Can you figure out why? This is a phenomena that has been called "time dilation" in the research literature). Can you think of an OS dilating time in this fashion on purpose to achieve something useful? Hint: Imagine a computer with a 1Gbps network interface card where you want to test the working of some software written to work with a 10Gbps network interface.
– Answer: The processes appear to work faster than before. The IO devices attached to this computer appear faster than they actually are to the OS. Think it in this way: a process needs 200 msec to finish its work. 200 msec is 20 time quantum of CPU if the CPU is interrupted every 10 msec, but is 10 time quantum if CPU is interrupted every 20 msec. Therefore, from OS' point of view, the process now only needs 10 time quantum to finish its job, rather than 20 time quantum as before. OS would think process works much faster. The reason is the same with IO device.
We can use an analogy for easier understanding. You need 48 hours to finish a job. That's 2 days if one day has 24 hours. However, now we change the 24-hour interrupts to 48-hour interrupt, which means one day now has 48 hours. Then in the eye of God, you now only need one day to finish your job, instead of 2 days. Therefore you appear to work much faster.

We can purposefully use OS dilating time to do something useful. For example, you want to test the working of some software written to work with a 10Gbps network interface. But the computer only has a 1Gbps network interface card. If you change the time interrupt to 10 times longer, for example, interrupt the CPU once every 100 msec instead of interrupt it once every 10 msec, the network interface card would appears to work 10 times faster. Then the software would think it is working with a 10Gbps network interface instead of a 1Gbps network interface card.

- **Long-answer Questions**
  (Sample question from The Little Book of Semaphores: Dining Savages Problem)

A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.

Any number of savage threads run the following code (unsynchronized):

```
1  while True:
2      getServingFromPot()
3      eat()
```

And one cook thread runs this code (unsynchronized):

```
1  while True:
2      putServingsInPot(M)
```

Solution for Dining Savages Problem:

```
1  servings = 0
2  mutex = Semaphore(1)
3  emptyPot = Semaphore(0)
4  fullPot = Semaphore(0)
```

For the cook:

```
1  while True:
2      emptyPot.wait()
3      putServingsInPot(M)
4      fullPot.signal()
```

For the savage:

```
1  while True:
2      mutex.wait()
3          if servings == 0:
4              emptyPot.signal()
5              fullPot.wait()
6              servings = M
7          servings -= 1
8          getServingFromPot()
9      mutex.signal()
10
11     eat()
```