

Section: Deadlock and Scheduling

1 Announcements

- Midterm 1 is 10/12/05 from 5:30-8:30 in 10 Evans
- Review Session is on Sunday 10/9/05 from 4pm-6pm
- Nachos is due on Thursday night at 11:59pm
- Admin Questions?
- Design Docs ... sorry i didn't get them back to you earlier. I'll be better about it next time.
- I'll run through new material first and then get to Nachos stuff

Disclaimer: These notes aren't substitutes for attending lecture or reading the book. Those have been created with careful revision and review. These notes are mainly intended to be a more thorough index of the lectures and reading so that you can go off and read the full treatment on the topic and be able to ask the right questions while reading those two sources. If you have comments or suggestions please send me email and I'll fix them up.

2 Recap

So far we have learned a lot of different synchronization primitives that can be used for a variety of things. We have locks, semaphores, condition variables and monitors that can be used by threaded applications to make sure that they play together nicely. However there are a few set of problems that we have been glossing over that we will tackle head on today such as how different threads get scheduled and how to get around the problem of deadlock.

3 Deadlock

The book lists four conditions about why deadlock can happen. Lets take a look at each of these a bit more carefully in detail.

3.1 Conditions for Deadlock

1. Mutual exclusion: One thread is using a resource prevents which other threads from using that same resource. For example, if you hold the lock to a critical region then other threads can't share that lock to enter the critical region.

2. Hold and Wait: A process must be holding at least one resource and be waiting on other ones. If it doesn't have to wait for all its resources it will be able to run to completion and release all its resources and then other threads can get to run. Only if the thread holds a resource and is waiting on other resources can dead lock occur.
3. No preemption: resources can only be released voluntarily by the thread. If this wasn't the case we could just steal the resources and solve the deadlock problem
4. Circular Wait: Processes must have cyclic dependencies in order to cause dead lock. I.e. thread 0 is waiting on thread 1, thread 1 is waiting on thread 2, thread 2 is waiting on thread 3 and thread 3 is waiting on thread 0. Notice that you can have more than 2 threads that can cause deadlock.

3.2 Resource Allocation Diagrams

If any one of the above conditions doesn't hold then we can get out of deadlock. The resource diagrams presented in class and in the book are a good way of thinking about how cycles can occur. Threads are represented as circles and resources as boxes. The number of dots within a box represents the number of simultaneous copies of a certain type of resource exists. If we have two dots, that means that there are two indistinguishable copies of that resource. In the context of that diagram, a lock is a resource with one dot.

Question: Which of the three allocations in Figure 1 cause deadlock? The book has a couple more interesting examples.

3.3 Deadlock Prevention and the Bankers Algorithm

When we actually decide to prevent deadlock we need to determine if a resource allocation will cause deadlock. In order to do that we use the banker's algorithm to determine whether a resource allocation should or should not go through. We will explore this algorithm in a bit of detail to try to understand why it works. In order for the bankers algorithm to work we have a few variables that we need. Table 1 shows the variables used and their sizes. We also define $mat[i][:]$ to be all elements in row i of matrix mat . In addition we say that $X \leq Y$ if $x[i] \leq y[i]$ for all i . Algorithm 1 and Algorithm 2 run in conjunction to analyze whether it is safe to allocate resources. We call these algorithms the Bankers algorithm.

The algorithms may seem complicated but there is one basic premise. We first have to declare apriori the amount of resources we need. This declaration ahead of time helps us allocate resources because we know the max that any thread will request at any given time. The safety check algorithm (Algorithm 1) will check whether a system is in a safe state. So we can then "pretend" to make the allocation in Algorithm 2 and then check if the system is in a safe state. If its safe, then go ahead and allocate and update state, otherwise hold the allocation since the allocation is unsafe. I encourage you to walk through the code and then see how the algorithm works.

Question: What is one fundamental limitation this algorithm imposes that might be easier said than done?

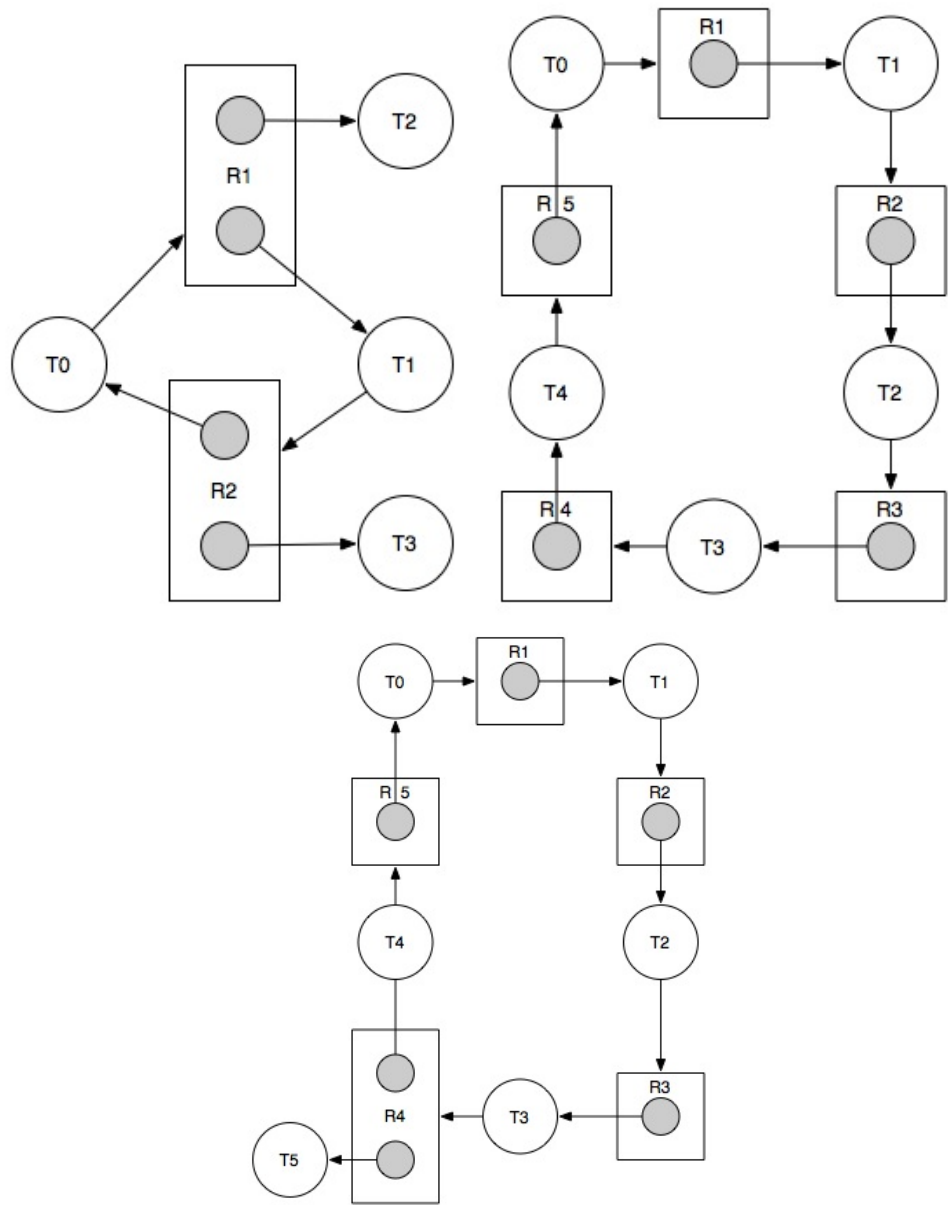


Figure 1: Sample Resource Allocation Diagrams

Name	Size	What it stores
avail	num resources x 1	avail[i] is the number of available and indistinguishable instances of the resource
Max	num threads x num resources	max[i][j] is the maximum number of instances of resource j that thread i will request
Allocation	num threads x num resources	allocation[i][j] is the number of instances of resource j that have been allocated to thread i
Need	num threads x num resources	need[i][j] is the max number of instances of resource j that thread i still needs to complete its task notice that need[i][:] = max[i][:]-allocation[i][:] when we don't overestimate the amount of resources we need

Table 1: Variables used in Bankers Algorithm

Question: If a system has 100s of resources and thousands of threads, how scalable are these algorithms and does it make sense to do a test on every allocation? Explain.

Algorithm 1 Algorithm for finding out if a system is in a safe state

```

let work = an array of length num resources
let finish = an array of length num threads
work = avail
for each thread i set finish[i] to false
while there exists a thread i such that finish[i] == false and need[i][:] ≤ work do
    work = work + allocation[i][:]
    finish[i] = true
end while
if finish[i] == true ∀i then
    system is safe
else
    deadlock is possible
end if

```

3.3.1 Examples

The following are different than book examples that will hopefully give you a chance to practice with the bankers algorithm.

Exercise: *Is the state represented by Table 2 safe?*

Exercise: *Is the state represented by Table 3 safe?*

Exercise: *Can we satisfy a request of $\langle 1,3,1 \rangle$ by thread 0 in state represented by Table 4?*

Algorithm 2 algorithm for requesting a resource allocation

```

let  $request[i][:]$  be the request vector for thread  $i$ 
if  $request[i][:] \leq need[i][:]$  then
  if  $request[i][:] \leq avail[i][:]$  then
    wait because there aren't enough free resources
  else
    pretend to modify the system
     $avail = avail - request[i][:]$ 
     $allocation[i][:] = allocation[i][:] + request[i]$ 
     $need[i][:] = need[i][:] - request[i][:]$ 
    if safety check algorithm says system is safe then
      allocate resources and complete transaction
    else
      undo the changes to  $avail$ ,  $allocation$ , and  $need$ 
      wait since allocation could cause deadlock and
      try the request later once resources have cleared up
    end if
  end if
else
  signal an error because we have requesting more than our max possible requests
end if

```

Thread ID	Allocation			Max			Need			Avail		
	A	B	C	A	B	C	A	B	C	A	B	C
0	2	0	0	7	2	1	5	2	1	3	3	4
1	2	3	6	3	5	9	1	2	3			
2	1	5	0	4	6	1	3	1	1			
3	0	0	0	2	3	3	2	3	1			

Table 2: bankers algorithm exercise 1

Thread ID	Allocation			Max			Need			Avail		
	A	B	C	A	B	C	A	B	C	A	B	C
0	2	3	1	5	11	3	3	8	2	2	4	5
1	1	3	0	3	4	4	2	1	4			
2	3	3	1	4	4	4	1	1	3			
3	1	2	2	6	3	4	5	1	2			

Table 3: bankers algorithm exercise 2

Thread ID	Allocation			Max			Need			Avail		
	A	B	C	A	B	C	A	B	C	A	B	C
0	1	0	0	5	11	3	4	11	3	3	7	6
1	1	3	0	3	4	4	2	1	4			
2	3	3	1	4	4	4	1	1	3			
3	1	2	2	6	3	4	5	1	2			

Table 4: bankers algorithm exercise 3

3.4 Deadlock Detection Algorithms

Deadlock detection algorithms work slightly different than deadlock avoidance algorithms. Deadlock detection algorithms don't do anything at allocate time, but they keep a wait graph to see which threads are waiting on other threads to complete. They basically use the standard CS 170 style approaches to finding cycles in graphs. Luckily here we don't have to find the largest or smallest cycle. We just need to find a cycle to declare that the system of threads is deadlocked. Once a deadlock is detected the action taken can range from simply reporting it to the user, to forcefully killing one of the threads to break the cycle, to restarting the entire system.

Question: What are the pros and cons of using deadlock detection versus deadlock avoidance?

4 Scheduling

The next major topic that we'll hit on is CPU scheduling. So far we've said that there is a magic algorithm that manages to pick the next running thread out of the ready queue, however choosing the right thread and the right frequency is a very tricky and interesting problem. There have been many years of research conducted on it and people are still coming up with new ideas. Fundamentally the problem is the optimal scheduler depends heavily on the application mix that is being run on that processor, however to design generic enough algorithms that aren't too complicated, to perform well on most application mixes is the tricky part.

There are many different criteria we can use for evaluating a scheduler and the importance of each one varies from application mix to application mix. As you read through, how different application mixes would place importance on the different criteria.

1. CPU Utilization: The goal since the dawn of computers was to keep the computer busy doing useful work and not let it spin doing wasteful work. To that end, we want to keep CPU utilization (i.e. running user code) as much as possible. The more time we spend switching the less user code we get to run.
2. Throughput: Another key design goal, is to make sure that we get as many processes in and out of the system in a given time frame. Think about a webserver and our performance of the webserver can be directly measured by how many requests we handle a second.

3. Turnaround time: This is a measure of how long the CPU and other hardware resources are taking to get back to our requests. We don't want a thread to be waiting for seconds for a memory request.
4. Waiting time we want to reduce the time the threads spends waiting in the ready queue as to maximize its utilization of the processor.
5. Response time: In interactive applications, this is a metric for understanding how long it takes for an application to respond to our requests.

Question: What is the most important criteria for compute bound applications like most scientific codes?

Question: What is the most important criteria for computer animators and other power users?

Question: What is the most important criteria for a database?

Question: What is the most important criteria for a webserver?

Question: What is the most important criteria for a normal computer user who just needs a machine to browse the web, write email, and edit word documents?

As you can see there are many different application mixes lead to different rankings on the criteria. Lets take a look at different scheduling algorithms and see what they optimize for.

4.1 First Come, First Serve (FCFS)

This one is relatively straight forward. The threads are chosen in the order in which they arrived. Since we are completely at the whim of random arrival times here, there is no real guarantee on the response time or the hold time of any specific thread. The advantage is that its easy to understand its behavior.

4.2 Shortest Job First (SJF)

This one again is a bit more complicated but is straightforward to understand. Each thread has to know how long it is going to run for and then the scheduler picks the thread with the shortest running time first, thus long jobs have the potential of getting stuck behind a lot of short jobs. The advantage of this is that we assume shorter jobs have higher importance on response time so in that sense we give advantage to them. Notice that there is little incentive to cheat because if you make your length too short then you won't finish the job and then you'd have to retry. However if you overestimate the time, then the you might get a slower response time than you would have hoped.

Process	Running Time	Priority	Arrival Time
P_1	10	3	2
P_2	4	1	0
P_3	2	3	1
P_4	1	4	4
P_5	5	2	2

Table 5: Running time and priorities for scheduling exercise

4.3 Priority Scheduling

I won't go too much into detail with this one since we you've thought about it through your projects. However i will say that priority scheduling can lead to starvation of lower priority threads if high priority threads constantly come into the system. We'll see a pretty cool way around this in your next project.

4.4 Round-Robin Scheduling

A round-robin scheduler breaks up the CPU usage into time quanta and then schedules each thread for a specific period of time before pre-empting a job and then choosing a new one. Thus longer running jobs will have to run for a lot more time quantas than smaller jobs. The main problem with pre-emptive scheduling schemes like round-robin is that we have to count the time it takes to switch and the more switches we have the more overhead we have leading to less useful work.

4.5 Sample Problem

I think the best way to explore the different algorithms is to actually work out a sample problem. I chose problem 5.4 from the seventh edition of the book and i reproduce it here and slightly modified it by adding in arrival time.

Exercise: *Given the table in Table 5 which shows running time, arrival time, and priorities, figure out the average wait time (start - arrival) and average turnaround time (finish - arrival) for all the schedules mentioned above. For the round-robin scheduler, use a time qunata of 1 unit*

5 Conclusion

This section covered a lot of material. The first thing was deadlock. Deadlock is an insidious problem and has been the bane of many computer scientists over the years. The reason is that deadlock is not deterministic and its very difficult to reproduce the right conditions for it. We thus create different methods for handling it. The one we will spend the most time on is the banker's algorithm, but realize that there are tons of other methods out there for dealing with it.

The next major topic was scheduling and here again scheduling is a very important problem but trying to pick the right scheduler for your application mix is very difficult. Understanding a standard set of schedulers will help you decide on the best way to write a scheduler that fits your own needs which is also much harder than it sounds.