CS162, Spring 2004 **Discussion #6** **Amir Kamil**
UC Berkeley **2/26/04**

Topics: Deadlock, Scheduling

1 Announcements

- Office hours today 3:30-4:30 in 611 Soda (6th floor alcove)
- Project 1 code due Thursday, March 4th, 2004, 11:59pm, final design and group evaluations due the next day
- Project clarifications:
  You are allowed to use global counter in boat problem; however, final design must not do so. See newsgroup for details.
  `KThread.join()` should return immediately if the target thread is already finished.
  You may not assume that a priority scheduler is being used for the boat problem.
- Midterm on Thursday, March 18th, 2004, 5:30-7:30pm, 155 Dwinelle

2 Comments on Design Docs

The average score on project 1 design documents was about 4/10. Here are a few specific flaws I noticed on your documents:

- giving no explanation
  The purpose of design is to explain how your stuff works, not to just give the code.
- leaving out code
  You should state all variables used, describe and provide pseudocode for all functions added or changed. Otherwise I can't tell if your design works or not. I don't care how trivial a function is, I want to see it.
- abstraction violations
  One group used `PriorityQueue.waitForAccess()` to recalculate effective priority, so they would call `waitForAccess()` from `setPriority()` for example. This is a clear abstraction violation. Each function should have exactly one purpose, so `waitForAccess()` should be used for adding a thread to a wait queue and a new function should be used for recalculating effective priority.
  Note also that your functions should be concise. In CS9*, the rule is that no function should be longer than 24 lines. In reality, on rare occasions it may make sense to break this rule.
- not reading specs carefully
  Some groups did not handle priority inversion in join(), and others did not have speak() when there are no listeners. These requirements are clearly stated in the specifications.
- not putting enough thought/effort into design
  Many groups did not think long enough about what could go wrong in each function, and others clearly put very little effort into the design. For example, if you gave me 15 lines of code for `PriorityScheduler`, something is definitely wrong.
- not proofreading
  Presentation is important also. Check word wrapping and indentation before turning in. Make sure that you do not have typos. One group turned in a design for `Boat` that checked whether or not the boat is at Oahu, when they meant the boat is *not* at Oahu.

Remember that you have to turn in a final design after your code is due, so make sure you fix all these problems before then.

3 Deadlock

Recall the four conditions for deadlock:

1. limited access
   Either there are a limited number of resources (e.g. disk space, printer) or the number of threads that can access a resource is restricted (e.g. mutex).

2. no preemption allowed
   There are two types of resources, preemptable and non-preemptable. The former include the CPU (scheduling, context switching) and memory (paging). The latter include disk, printer, and locks. Think of the consequences of, for example, taking the printer away from one thread in the middle of a print job and giving it to another.

3. multiple independent requests/holding and wait
   Holding some resources while waiting for another is possible.

4. circular chain of requests
   For example, thread A requires something that B holds, B requires something C holds, and C requires something A holds.

When a set of threads deadlock, none in the set can proceed, since all are waiting for something.

There are two approaches to deadlock. The first is to let it occur, detect it, and fix it, by killing one of the threads for example. This has a tendency to make people angry, if their thread is the one that got decapitated. A second solution is to avoid deadlock through safe resource allocation. The *banker's algorithm* is used to determine whether or not a resource request should be honored.

Before we go over the banker's algorithm, let's first define what it means for a resource request to be safe. A request is safe if all threads can still finish if the request is granted, through some sequence of task completions. Note that once a task is completed, resources can be released, so that threads that could not finish before may be able to finish with the released resources.

As a concrete example, suppose there are three threads, A, B, and C, with A using 100 MB of disk, B using 200 MB of disk, and C using 300 MB of space. Suppose that there is 1 GB total disk space, and that A needs 200 MB total to finish, B needs 700 MB, and C requires 600 MB. Now if B requests an additional 300 MB of space, should the request be granted? Yes, since there will still be 100 MB available, so A could allocate it and eventually finish, release 200 MB. Now B could take those 200 MB, finish, and release 700 MB. Finally, C could take 300 MB out of the available 700 MB and terminate as well.

Suppose instead that B requests 350 MB of space? If the request were granted, no thread would be able to get enough resources to finish. Thus the request should be denied.

The banker's algorithm formalizes the above technique, generalizing it to multiple types of resources. Each thread declares (but does not allocate) at startup the maximum amount of a resource that it would ever need. The algorithm keeps track of the following information for each thread `x`:

- `max(x)`, the maximum amount of the resource that `x` will use
- `alloc(x)`, the amount currently allocated to `x`
- `need(x)`, the amount needed by `x`, i.e. `max(x) - alloc(x)`
- `avail`, the amount currently available

When a thread `x` requests `k` amount of the resource, the algorithm first checks to make sure that it is no more than `need(x)`. If it is, than `x` is requesting to use more than its declared maximum, so the request is rejected. Otherwise, the algorithm checks if there is enough of the resource free to satisfy the request. If not, the thread must wait. If so, the algorithm computes whether or not it is safe to accept the request.

In order to decide whether or not a request is safe, the banker's algorithm implements the request and then determines whether or not there exists a sequence of thread completions that allows all threads to complete. This is done by repeatedly finding a running thread that can complete using its current allocation and the free resources, setting the thread to be finished, and deallocating its resources to be used by the

```
request(Thread x, Amount k):
  if (k + alloc(x,r) > max(x,r)) then:
    DENY REQUEST; // thread is trying to use more than its max
  else if (k > avail(r)) then:
    WAIT; // resource not currently available, must wait until more freed
  else:
    SAVE old state
    // allocate resource
    alloc(x) += k;
    need(x) -= k;
    avail -= k;
    // check if allocation is safe
    finish[1,...,n] = boolean array initialized to false, where n is the number of threads;
    tmp_avail = avail;
    while (true) do:
      for each thread x, do:
        if (finish(x) is false AND need(x) < tmp_avail) then:
          // this thread can finish, deallocate resources
          finish(x) = true;
          tmp_avail += alloc(x);
      // check if done
      if (finish(x) = true for all x) then:
        ACCEPT; // accept request
      else if (there is no x such that finish(x) is false AND need(x) < tmp_avail) then:
        // this is unsafe allocation; force thread to wait
        RESTORE old state;
        WAIT;
      continue;
```

Figure 1: The banker's algorithm for a single resource type.

remaining threads. At the end, if all threads are finished, the request is safe and is accepted, but if there are unfinished threads that cannot complete, the request is unsafe and the thread is forced to wait. Figure 1 provides the actual banker's algorithm.

As an example, let's take a look at the dining lawyers example with four lawyers and four chopsticks. Here, the resource is chopsticks. For each lawyer `x`, `max(x)` is 2, and the total number is 4.

Suppose that lawyer 1 is really fast and grabs 2 chopsticks. Now suppose the second lawyer asks for a chopstick. Should we give him one? If we do, then L1 can finish and release its chopsticks, resulting in 3 free. Then L2 can finish, resulting in 4 free sticks. Finally, L3 can take 2 and L4 can take the other 2 and both finish. Since everyone can finish, we allocate L2 the chopstick.

Now suppose that L3 asks for a chopstick. If we give it to him, L1 can finish, releasing 2, L2 and L3 can take one each and finish, and L4 can then take two and finish. So we can allocate the chopstick to L3. Note that this will not result in the most efficient solution, but it still results in a solution.

Consider another initial state. Suppose L4 is slow, and L1, L2, and L3 grab chopsticks. Can we now allocate one to L4? If we do so, no lawyer can eat. Thus we should deny L4's request for a chopstick.

The banker's algorithm can also be generalized to multiple resource types. It keeps track of the following information for each thread `x` and each resource type `r`:

- `max(x,r)`, the maximum amount of the resource `r` that `x` will use
- `alloc(x,r)`, the amount of `r` currently allocated to `x`
- `need(x,r)`, the amount of `r` still needed by `x`, i.e. `max(x,r) - alloc(x,r)`
- `avail(r)`, the amount of `r` currently available

```
request(Thread x, Resource r, Amount k):
  if (k + alloc(x,r) > max(x,r)) then:
    DENY REQUEST; // thread is trying to use more than its max
  else if (k > avail(r)) then:
    WAIT; // resource not currently available, must wait until more freed
  else:
    SAVE old state
    // allocate resource
    alloc(x,r) += k;
    need(x,r) -= k;
    avail(r) -= k;
    // check if allocation is safe
    finish[1,...,n] = boolean array initialized to false, where n is the number of threads;
    tmp_avail[1,...,m] = avail(1,...,m); // m is the number of types of resources
    while (true) do:
      for each thread x, do:
        if (finish(x) is false) then:
          // check if this thread can finish
          for each resource r, do:
            if (need(x,r) > tmp_avail(r)) then:
              if (x is not the last thread) then:
                GOTO next thread; // can't finish yet, move on to next thread
              else:
                // not all threads can finish, this is unsafe allocation
                RESTORE old state;
                WAIT;
          // can finish, deallocate resources
          finish(x) = true;
          for each resource r, do:
            tmp_avail(r) += alloc(x,r);
          break; // check if done now
      // check if done
      if (finish(x) = true for all x) then:
        ACCEPT; // accept request
      continue;
```

Figure 2: The general banker's algorithm.

Figure 2 gives the general banker's algorithm. Note that its complexity is $O(m \cdot n^2)$ where m is the number of types of resources and n is the number of threads.

4 Scheduling

There are multiple different scheduling algorithms, and we briefly review each one:

1. first in first out (FIFO)
   The threads run to completion in the order in which they arrive in the system.

2. round robin (RR)
   The threads take turns executing, each running for a small *time slice* or *quantum*.

3. shortest job first (SJF)
   The threads run to completion, with the currently shortest job running first. This can result in starvation, if short jobs keep entering the system. This is the optimal non-preemptive algorithm as far

Process	Arrival Time	CPU Time
A	0	4
B	2	5
C	3	1
D	6	3

Table 1: A set of jobs, their arrival times, and their processing requirements.

Time	FIFO	RR	SJF	SRPT
0	A	A	A	A
1	A	A	A	A
2	A	B	A	A
3	A	C	A	A
4	B	A	C	C
5	B	B	B	B
6	B	D	B	D
7	B	A	B	D
8	B	B	B	D
9	C	D	B	B
10	D	B	D	B
11	D	D	D	B
12	D	B	D	B

Table 2: The sequence of execution of the jobs in table 2 under different scheduling algorithms.

as average *turnaround time*, the time between a thread entering the system and finishing, is concerned, though it requires predicting how long a job will take.

4. shortest remaining processing time/shortest remaining time first (SRPT/SRTF)
   The threads take turns executing. After every quantum, the job with the least remaining time gets to run next. This is the optimal preemptive algorithm. However, long running jobs can starve, and it requires predicting the future.

5. multi-level feedback queues
   The processor switches between jobs in one level, moving on to the next only after they all complete. A job moves down a level if it runs for a whole quantum without yielding, and moves up if it runs for less than the whole quantum. This can result in starvation for threads in lower levels. This algorithm approximates SRPT without predicting the future.

6. lottery scheduler
   There still are multiple levels, but each thread now gets a number of tickets corresponding to which level it is in; the higher the level, the more tickets. The scheduler randomly picks a ticket at each quantum and runs the corresponding thread. Thus threads in higher levels are more likely to run, but threads in lower levels don't starve.

You should know well how the first four policies work, and qualitatively understand the last two.

An example question that may appear on an exam is to compute the sequence of execution of a set of threads under different scheduling policies. Suppose that we are using a time slice of 1 unit, and that preemptive algorithms schedule a new job immediately when they enter a system. Table 3 gives the sequence of execution for the threads in table 2 under these assumptions.

Another possible problem is to compute the average turnaround time for each algorithm. In the example above, FIFO has an average turnaround of 6.25, RR of 6.5, SJF of 5.25, and SRPT of 5. As you can see, SJF accomplishes the best non-preemptive time and SRPT accomplishes the overall best time.