

More Project Info

- Design Reviews were very good! Thank you!
 1. It was fun talking to everyone, and I was very pleased at the level of discussion.
- Design Documents: MUCH better than I had thought they would be.
 1. They have been graded and will be returned at the end of class.
 2. Overall mean was low. Do not freak out at this.
 1. You all now know how to do this process. The next one will be much better.
 3. Some points overall:
 2. Boat: consider how you are going to finish!
 3. DO NOT debug with random numbers! (why?)
 4. When testing, LIST EDGE CASES. Try and think of everything that can break your program, because the auto grader sure is going to try.
 5. Make sure to consider the problem fully before designing. Listing correctness constraints is helpful because makes you consider what can go wrong. Note that each section is NOT independent.
 6. TOO LONG. Your boss will not want to read verbose text. Use bulleted lists. Use simple sentences. Convey the idea efficiently.
 7. Name variables better :) Seriously, this is important.
- **CODE DUE THURSDAY**
- **DESIGN DOCUMENT REVISIONS AND QUESTIONS DUE FRIDAY.**
- **Unsolicited advice: Don't use your slip days right off the bat!**

Update on the Boat Problem

- What does begin() do??
 1. Start threads, and . . .
 2. Terminate nachos!
- Understand exactly how the boat problem starts and finished.
 1. Should only use global counters in begin();
 - The spec is ambiguous on this point, so the TAs have decided on this interpretation.
 2. If you have already finished an implementation that checks them in childIntinerary to finish, that's ok.
- In your final design document, MUST ADDRESS:
 1. How begin() works
 2. How you would revise your solution so that begin terminates it by checking the global counter.

Making a Development Schedule

- Why do this?

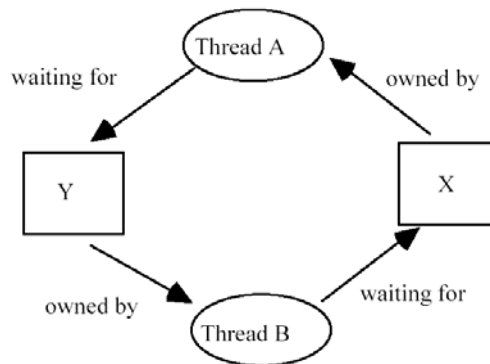
1. Answer: if you don't set goals and attempt to meet them, it is very easy to slip behind schedule and start munching slip days.
- What is a milestone?
 1. Software engineering talk for a GOAL: i.e., something must be done by a certain date.
 - Beta complete
 - Features frozen
 - Etc. . .
 2. Why bother having these?
 - Test schedule
 - Rolling integration tests—DO THIS!!
 - Off topic, but relevant:

Today's Menu:

- Deadlock
- Scheduling
- Quiz
- (If time) the Linux Scheduler

Deadlock

- **QUESTION:** What are the four conditions for deadlock?
- **ANSWER:**
 1. Limited resources
 2. No preemption (can't take away resources)
 3. Multiple independent requests
 4. Circular chain of requests
- What's that last one mean?
 1. It means there's a cycle in the *Resource Request Graph*:



2. Two ways around deadlock: detect and fix, or prevent.
- Detecting deadlock: the Graph Algorithm
 1. First, build resource request graph.
 2. Then scan the graph and look for cycles
 3. If cycle, deadlock is possible! Fix:
 - Kill thread

- Remove resources
- Example:
 1. Process X will attempt to acquire resources R1, R2, R3 in that order
 2. Process Y will attempt to acquire resources R3 then R1.
- **QUESTION:** Deadlock possible here?
- **ANSWER:** There's a cycle, so it's possible.
- Preventing deadlock: the Banker's Algorithm
 1. Important stuff; might be a midterm question!
 2. Source of name: Can be used by banks so they don't allocate their available cache without being able to satisfy the needs of all customers.
- When a process enters system, it declares maximum resources it may need.
 1. We build the following records:
 - Available[0..r] : Number of available resources of each type
 - Max[0..p][0..r] : Max demand of process P for resource R
 - Alloc[0..p][0..r] : Amount of R allocated to P
 - Need[0..p][0..r] : Amount of R *might be* needed by P
 - Need[p,r] = Max[p,r] - Alloc[p,r]
- Define notation:
 1. Vector $X \leq Y$ if $X[i] \leq Y[i]$ for all i
- Making a resource request:
 1. Process P states that it needs resources:
 - Request[p][0..r]
 2. If Request[p][0..r] > Need[p][0..r], ERROR
 - Process is requesting more than its stated maximum
 3. If Request[p][0..r] > Available[p][0..r], WAIT
 - All resources not available, so must wait
 4. Set:
 - Available[0..r] = Available[0..r] - Request[p][0..r]
 - Alloc[p][0..r] = Alloc[p][0..r] + Request[p][0..r]
 - Need[p][0..r] = Need[p][0..r] - Request[p][0..r]
- Now test if this resource-allocation state is "safe".
 1. If safe, proceed, otherwise revert to previous values of vectors and have P wait.
- Safety test: Available resources \geq Max needed by ANY process
 1. Implementation:
 - Work[0..r] = Available[0..r]
 - Finish[0..p] = false
- Overall:
 1. Find a process P such that:
 - Finish[P] = false && Need[p][0..r] < Work[0..r]
 - If no such P exists, goto 3
 2. Work[0..r] = Work[0..r] + Alloc[p][0..r]
 - Finish[p] = true

- Goto 1
- 3. If $\text{Finish}[p] = \text{true}$ for all procs P , system is safe
- Complexity: $O(r * p^2)$

- Example:

Suppose we have 12 available tape drives. After the following allocations...

Job	Needs	Allocated
1	8	3
2	7	2
3	7	1
4	2	0

... we have 6 tape drives left. Now suppose the following request from job 5 comes in. The request is for 4 tape drives immediately, and 2 more possibly.

Job	Needs	Needs Now
5	6	4

Using the Banker's Algorithm, show that the system is or isn't in a safe state.

- **QUESTION:** Is it safe? Can we finish safely?
- **ANSWER:** Yup. Here's how:
 1. Allocate the 4 drives
 - a. Available drives now = 2
 2. Can anybody finish? Yes. Job 4 can finish
 - a. Available drives now = 2
 3. Can anybody finish? Yes. Job 5 can finish
 - a. Available drives now = 6
 4. Can anybody finish? Yes. Job 1 can finish
 - a. Available drives now = 9
 5. Can anybody finish? Yes. Job 2 can finish
 - a. Available drives now = 11
 6. Can anybody finish? Yes. Job 3 can finish
 - a. Available drives now = 12

Scheduling

- Some scheduling vocab:
 1. Utilization/Efficiency: keep the CPU busy 100% of the time with useful work
 2. Throughput: maximize the number of jobs processed per hour.
 3. Turnaround time: from the time of submission to the time of completion, minimize the time batch users must wait for output
 4. Waiting time: Sum of times spent in ready queue - Minimize this

5. Average Response Time: average time over all jobs from submission till the first response is produced, minimize response time for interactive users
 6. Fairness: make sure each process gets a fair share of the CPU
 7. Average Flow Time: average time a job spends in the system, from its entrance to its finish.
- We've seen Round Robin, SJF, SRPT, FIFO, etc.
 - Quick review:
 1. Round Robin (pre-e)
 - Give each thread an equal slice of time
 - Pros: fair, easy
 - Cons: average waiting time can get bad, io-bound processes suffer
 - With high timeslice becomes FIFO.
 2. Shortest Job First (non pre-e)
 - Always run the shortest job first
 - Pros: optimal waiting time!
 - Cons: um, impossible to implement. . .
 - Also starvation!
 3. Shortest Remaining Processing Time (pre-e)
 - Always run the job with the shortest remaining processing time
 - Pros: Optimal responsiveness/waiting time
 - Cons: Same as SJF—no predicting future, can have starvation.
 4. First Come, First Served (non pre-e)
 - Run threads in FIFO order
 - Pros: easy, gets jobs done in order of submission
 - Cons: can have terrible response times
 5. Priority Scheduling (pre-e)
 - Always run the thread with the highest priority!
 - Pros: gives good control over what runs first
 - Cons: unfair
 - Now for a few more:
 1. Multilevel Feedback Queue (pre-e)
 - Adaptive scheduling policy!
 - Maintain multiple queues with different priorities
 - Jobs start out in high priority queue
 - As they run their quantum and still haven't finished, drop to lower level queues.
 - Pros: adaptive to load! (why?)
 - Cons: still unfair—can still have starvation.
 2. Lottery Scheduling (pre-e)
 - Do scheduling probabilistically through granting lottery tickets.
 - Give each job a ticket for each time slice, then generate a random number and hold a lottery.
 - For an example, see the lecture notes.

Quiz!

Linux Kernel Scheduler

- Basic description here, thanks to Emil Ong for making readable notes on this.
- Good description at: <http://www.people.fas.harvard.edu/~rross/cs261/paper.html>
- Recall multilevel feedback priority queues
 1. Used by most UNIX systems with some variation in details
 2. Multiple run queues for different priorities
 - BSD uses 32 run queues for 128 different priority levels;
 - so $\text{queuenum} = \text{priority}/4$
 - Each queue scheduled in round-robin fashion
 - Process increases priority if blocks
 - Process decreases priority if timeslice expires
- Linux uses same basic idea, but does not actually maintain queues
 1. Rather, computes "goodness" value for each runnable process in the system at each scheduling decision point
 - process with highest "goodness" is the next to run
 - $\text{goodness} = \text{priority} + \text{counter}$
 - priority indicates the priority of the process (how this is computed is given below). counter indicates length of time slice (in ticks) when it acquires the CPU. c
- Clock goes off every 10ms (on x86) - increments 'jiffies'
- Default counter (for priority = 20) is 20 ticks
 1. This actually corresponds to 210 ms, since you get one tick "for free"
 - Default timeslice is 21 ticks (or 210 ms)
 - Note that DEF_PRIORITY is 20 ticks, but, only reschedule after counter goes *below* zero (so you get one tick for free)
 - counter ranges from 0 (process has exhausted its time slice) to $2 * \text{priority}$.
- Priority calculated as $20 - \text{nice}$, where 'nice' is a value assigned to the process by the user or system admin. -20 corresponds to "highest priority" processes and +20 to "lowest priority" processes.
 1. Normal users can only increase the 'nice' value for their processes.
 - 'nice' goes back to early days of UNIX which explains why it is inverted.
 2. Process runs until its counter value dips below zero (note that $\text{counter} == 0$ means it will run for just one more tick)
 3. If time slice runs out: 'counter' reset to 'priority'
 4. If process blocks: Normally 'counter' stays where it is.
 - But, if no process has a nonzero counter value, then *all* processes (including blocked ones!) get the value:
 - $\text{counter} = \text{counter}/2 + \text{priority}$
 - This means that processes that block get higher counter values (asymptotically approaching $\text{priority} * 2$ according to the formula above).

- If process with higher goodness becomes runnable, preempts current process at next tick
- SMP issues
 1. When a process becomes runnable, tries to find an idle CPU to run it
 2. Weighs more towards same CPU that proc ran on before, or same CPU of process "related" to it (i.e. thread within same address space as the process leaving the CPU).
 - WHY? Better locality for page tables... TLB and cache should be flushed, right?
- Interesting features
 1. Processes with longer time slice get higher goodness, so run more often!
 2. Blocking processes eventually get higher counter values -> get higher goodness -> will run more often. Curious that a blocking process should get a longer timeslice.
 3. What would be the justification? (ANSWER: Perhaps that when the process becomes runnable again, will need the CPU for longer.)
 4. CPU bound process w/ priority 20 (default) gets 210 ms timeslice.
 - Highest priority is 40, so longest timeslice for CPU-bound procs is 420 ms.
 - Note that a blocked process can accrue counter up to $2 \times \text{priority}$ (so timeslice could be as high 840ms for a process which blocks for a long time on an idle system).
 5. Makes no attempt to reduce time-slices as run queues get longer
 - means that a loaded system with many CPU-bound processes will not become more reactive.