Real-Time Scheduling

- Priority based scheduling enables us to give better service to certain processes
- Priority scheduling is inherently a best effort approach
- If our task is competing with other high priority tasks, it may not get as much time as it requires. Sometimes best effort isn't good enough
    - Ex: Space shuttles are aerodynamically unstable. It is kept under control by, not the pilots, but by guidance computers that are collecting altitude and acceleration inputs
- There are many computer-controlled applications where delays in critical processing can have undesirable consequences

What are Real-Time Systems

- A real-time system is one whose correctness depends on timing as well as functionality
- Metrics for more traditional algorithms were turnaround-time (throughput) = T(completion) - T(arrival), fairness (sharing the amount of time between processes), and mean response time = T(first scheduled) - T(arrival)
- Real-time systems have different requirements that are characterized by different metrics
    - timeliness: how closely does it meet its timing requirements (e.g. ms/day of accumulated tardiness)
    - predictability: how much deviation is there in delivered timeliness
- New Concepts
    - feasibility: whether or not it is possible to meet the requirements for a particular task set
    - hard real-time: there are strong requirements that specified tasks be run a specified intervals. Failure to meet this requirement may result in system failure
    - soft real-time: we may want to provide very good response time. The only consequences of missing a deadline are degraded performances or recoverable failures
- Real-time systems have a few characteristics that make scheduling a bit easier
    - We may actually know how long each task will take to run -> enables intelligent scheduling
    - Starvation (of low priority tasks) may be acceptable. The space shuttle absolutely must sense attitude and acceleration and adjust spoiler positions. We must understand the criticality of each task
    - The workload may be relatively fixed. Normally high utilization implies long queuing delays. But if the traffic rate is relatively constant, we may be able to achieve high utilization and good response time

Real=Time Scheduling Algorithms

- In the simplest real time systems, if the tasks and their execution times are all known, there might not even be a scheduler.

- In a complex system with a larger number of tasks that do not function in a strictly pipeline fashion, it may be possible to do **static scheduling**
    - Based on the list of tasks to be run, and the expected completion time for each, we can define a fixed schedule that will ensure timely execution of all tasks
- However, the workload for most systems changes from moment to moment, so we need to have dynamic scheduling
    - How they choose the next (ready) task to run
        - shortest job first (SJF)
        - static priority: highest priority ready task
        - soonest start-time deadline first (ASAP)
        - soonest completion-time deadline first (STCF -> shortest time to completion first)
    - How they handle overload (infeasible requirements)
        - best effort
        - periodicity adjustments: run lower priority tasks less often
        - work shedding: stop running lower priority tasks entirely
- Preemption may also be a different issue in real-time systems. In ordinary time-sharing systems, preemption is a means of improving response time (RR, etc.). It also prevents buggy, possibly infinitely looping programs from taking over the CPU
    - The tradeoff between improved response time and increased overhead for adding context switches is almost always advantageous for preemptive scheduling
- However, this may not always be the case for real-time systems
    - preempting a running task will almost surely cause it to miss its completion deadline (hard real-time)
    - Since we often know the expected execution time for the process, we do not normally need to have a scheduler that utilizes preemption
- For the least demanding real time tasks, a sufficiently lightly loaded system might be reasonably successful in meeting its deadlines
    - However, this is simply because the frequency at which the task is run happens to be high enough to meet its real time requirements
    - A lightly loaded machine running a traditional scheduler can often display a video to a user's satisfaction, not because the scheduler knows that a frame must be rendered by a certain deadline, but because the machine has enough cycles and a low enough workload to render the frame before the deadline

Real-Time and Linux

- Linux was not designed to be an embedded or real-time OS, but many tasks that were once-considered embedded applications now require the capabilities of a general purpose OS
    - Linux supports a real-time scheduler which can be enabled with sched_setscheduler(2)

    - This real-time scheduler does not provide quite the same level of response-time guarantees that more traditional real-time OSs do
    - Windows favors general purpose throughput over meeting deadlines, as a rule, and therefore does not support real time schedulers

*Summary!: Overall remember that preempting a process may not be the best action for real-time systems because

Chapter 7: Scheduling: Introduction

- Scheduling Policies, also known as discipline, have been developed to handle which process the OS chooses to run and when

7.1: Workload Assumptions

- Refer to processes as jobs
    - 1) Each job runs for the same amount of time
    - 2) All jobs arrive at the same time
    - 3) Once started, each job runs to completion
    - 4) All jobs only use the CPU (i.e., they perform no I/O)
    - 5) The run-time of each job is known

7.2: Scheduling Metrics

- Metric: something that we use to measure something
- Turnaround time: The turnaround time of a job is defined as the time at which the job completes minus the time at which the job arrived in the system
    - T(turnaround) = T(completion) - T(arrived)
    - Because, for now, we have the assumption that all jobs arrive at the same time, we can say that T(arrived) = 0
- Turnaround time is a performance metric.
    - Another important metric is fairness
        - A scheduler may optimize performance but at the cost of preventing a few jobs from running, decreasing fairness

7.3: First In, First Out (FIFO)

- Example: Imagine that three jobs arrive in the system at roughly the same time (A, B, C Tarrival = 0)
    - Assume that A arrived a hair before B and B arrived a hair before C
    - Assuming each job runs for 10 seconds, what will the average turnaround time be for these jobs?
    - A finished at time 10, B finished at time 20, and C finished at time 30
        - Average turnaround time = (10+20+30)/3 = 20
    - However, if we relax one of our assumptions that each job runs for the same amount of time, FIFO has some clear disadvantages

    - Ex: If A runs for 100 seconds, and B and C run for 10, the average turnaround time will be 100+110+120/3 = 110, meaning the turnaround time is extremely high
        - This is known as the **convoy effect**, where a number of relatively short potential consumers of a resource get queued behind a heavyweight resource consumer

7.4: Shortest Job First (SJF)

- Shortest JOb First represents a general scheduling principle that can be applied to any system where the perceived turnaround time per customer matters
    - ex: grocery stores have a “ten items or less” line to ensure that shoppers with only a few things to purchase don’t get stuck behind people with lots of things to purchase
    - take previous example: if A is 10, B is 10 and C is 100 then the average turnaround time is (10+20+120)/3 = 50
    - Relax another assumption and say that jobs can arrive at any time instead of all at once
        - We can see that if the longest job arrives first, the scheduler has to take care of it and therefore the average turnaround time increases again

7.5: Shortest Time to Completion First (STCF)

- Relax assumption 3 that jobs must run to completion
- The scheduler can preempt job A and decide to run another job, perhaps continuing job A later
- This scheduler is also known as Preemptive Shortest Job First (PSJF)
- The STCP scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one. Thus, in our example, STCF would preempt A and run B and C to completion.
- (120-0) + (20-10) + (30-10)/ 3 = 50 average turnaround time

7.6: A New Metric: Response Time

- STCP would be a great way to handle processes if it was the only metric we had
- Response Time: Time from when job arrives to the first time it is scheduled
- T(response) = T(firstrun) - T(arrival)
- Ex: If we had A arriving at time 0 and B and C arriving at time 10, the response time for STCF would be 0 for A and B and 10 for C
    - The third job has to wait for the previous jobs to run in their entirety before it can be run

7.7 Round Robin

- Instead of running jobs to completion, RR runs a job for a time slice (sometimes called scheduling quantum) and then switches to the next job in the run queue

- The length of the time slice must be a multiple of the timer interrupt (the timer that calls an interrupt so that the OS can take control of the
- Assume three jobs A, B, and C arrive at the same time in the system, and that they each wish to run for 5 seconds.
    - With RR the response time will be 0+1+⅔ = 1
    - With SJF the response time will be 0+5+10/3 = 5
- If you make the time slice too short, the cost of context switching will become a problem.
- However, RR is one of the worst scheduling policies if turnaround time is our metric
- More generally, any policy that is fair, i.e., that evenly divides the CPU among active processes on a small time scale performs poorly on metrics such as turnaround time

Tip: Overlap enables higher utilizations: When possible, overlap operations to maximize the utilization of systems. Overlap is useful in many different domains, including when performing disk I/O or sending messages to remote machines

7.8: Incorporating I/O

- The currently-running process won't be using the CPU during the I/O because it is blocked waiting for the I/O completion
- The scheduler has to make the decision of which process to run while another process is performing an I/O request, and after the I/O completes
- Example: Assume that we have two jobs A and B which each need 50 ms of CPU time
    - A runs for 10ms and then performs an i/O while B does not perform an I/O
    - A common approach is to treat each 10ms sub-job of A as an independent job.
    - This means that its choice is to run a 10ms A or a 50ms B.
        - With STCF, we choose the shortest job, which is A
        - Then B begins running until it is preempted by a new sub-job of A

7.9: No More Oracle

- How can we build an OS if it doesn't know the length of each job? How will STCF and SJF work if it doesn't know the length?

Chapter 8: Scheduling: Multi-Level Feedback Queue (MLFQ)

- The fundamental problem that MLFQ tries to address is two-fold
    - it would like to optimize turnaround time (throughput) by running the shorter jobs first, however, the OS generally does not know how long a job will take to finish
    - Secondly, it would like to make a system feel responsive to users and therefore improve response time.

8.1: MLFQ: Basic Rules

- The MLFQ has a number of distinct queues, each assigned a different priority level
- MLFQ uses priorities to decide which job should run at a certain time
- A job with a higher priority would be on a higher queue
- Of course, more than one job may be on a given queue, and thus have the same priority. In this case, we will just use round-robin scheduling among those jobs

    - Rule 1: If priority(A) > priority(B), A runs and B doesn't
    - Rule 2: If priority(A) = priority(B), A&B run in RR (remember in RR, the time slices have to be multiples of the timer interrupts)
- MLFQ varies the priority of a job based on its observed behavior
    - Ex: If a job repeatedly relinquishes the CPU while waiting for input from the keyboard, MLFQ will keep the priority high
    - If a job uses the CPU intensively for a long period of time, a job may have a low priority
    - In this way, the MLFQ will try to learn about the processes as they run, and thus use the history of the object to predict its future behavior

8.2: Attempt #1: How to Change Priority

- Workload: a mix of interactive jobs that are short-running, (may frequently need to relinquish the CPU), and long-running (may need the CPU for extended periods of time) jobs
    - Rule 3: When a job enters the system, it is placed at the highest priority
    - Rule 4a: If a job uses up an entire time slice while running, its priority is reduced
    - Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level
- MLFQ tries to approximate SJF
- MLFQ assumes that the job might be a short job and it will run quickly and complete. If it is not, then it slowly move down the queues, and prove itself to be a long-running process
- Problems with a basic MLFQ with the above rules
    - **Starvation:** If there are too many interactive jobs (relinquishes CPU often), they will combine to consume all CPU time, and thus long-running jobs will never receive any CPU time (they starve)
    - A smart user could rewrite their program to **game the scheduler**
        - Gaming the scheduler generally refers to the idea of doing something sneaky to trick the scheduler **into giving you more than your fair share of the resource**
        - ex: before the time slice is over, issue an I/O request and thus relinquish the CPU. If we do this right before the time slice is over, (e.g. 99% of the time slice), then a job can monopolize the CPU

8.3: Attempt #2: The Priority Boost

- We must guarantee that CPU-bound jobs will make some progress
- Periodically boost the priority of all the jobs in a system
    - Rule 5: After some time period S, move all the jobs in the system to the topmost queue
- Our new rule guarantees that processes are not going to starve.
    - By sitting in the top queue, a job will share the CPU with other high-priority jobs in a round-robin fashion, and thus eventually receive service.

    - If a CPU-bound job has become interactive, the scheduler treats it properly once it has received the priority boost
    - Example: A long running job competing for the CPU with two, short running interactive jobs
- voodoo constants: The timer on the priority boost is very difficult to set and requires some “black magic”. If too high, long-running processes could potentially starve, while interactive ones may not get proper share of the CPU

8.4: Attempt #3: Better Accounting

- Rewrite Rules 4a/4b
    - Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced
- Example: Attempt to game the scheduler
    - Prevents the process from monopolizing the CPU

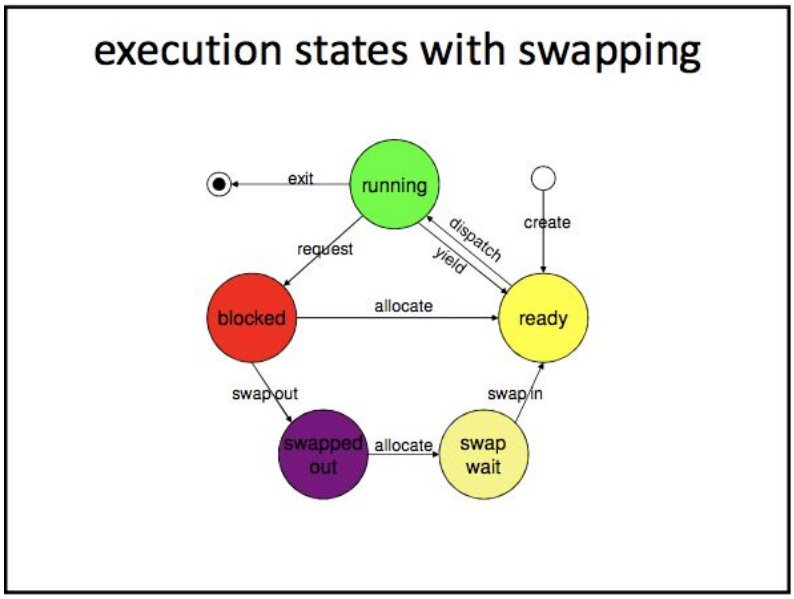
8.5: Tuning MLFQ and Other Issues

- A few other issues arise with MLFQ scheduling
    - How do we parametrize such a scheduler?
    - How big should the time slice per queue be?
    - How often should priority be boosted in order to avoid starvation
- Most MLFQ variants allow for varying time-slice length across different queues
    - high priority queues are usually given shorter time slices -> because the shorter running programs are given higher priority through SJF
- Tip: Avoid Voo-doo constants (Ousterhout’s Law)
    - Frequent result: A configuration file filled with default parameter values that a seasoned administrator can tweak when something isn’t quite right
- Solaris MLFQ Implementation
    - The time-sharing shecudling class (TS) is fairly easy to configure
        - provides a set of tables that determine exactly how the priority of a process is altered throughout its lifetime, how long each time slice is, and how often to boost the priority of a job
    - Default values: 60 queues, with slowly increasing time-slice lengths from 20ms (highest priority) to a few hundred ms (lowest) and priorites boosted around every second or so
- FreeBSD scheduler
    - uses mathematical formulae to adjust
    - calculates the current priority level of a job based on how much of the CPU it has used
    - Usage is decayed over time, so the desired priority boost is still given
- Some schedulers reserve the highest priority levels for OS work
    - typical user jobs can never obtain the highest level of priority

8.6: Summary of MLFQ

- Rule 1: If Priority (A) > Priority (B), then A runs and B doesn't
- Rule 2: If Priority (A) = Priority (B), then A & B run in RR
- Rule 3: When a job enters the system, it is placed at the highest priority
- Rule 4: Once a job uses up its time allocated at a given level, its priority is reduced
- Rule 5: After some time period S, move all jobs up to the highest priority queue

Lecture Slides
*Any serially reusable resource/partitioned resource is done in the scheduler



^5 state execution model -> yield = preemption

Un-dispatching a running process
- somehow we enter the OS
    - e.g. a via a yield system call or a clock interrupt
- State of the process has already been preserved
    - User mode PC, PS and registers are already saved on stack
    - Supervisor mode registers are saved on the supervisor mode stack
    - descriptions of address space, and pointers to code, data and stack segments, and all other resources are already stored in the process descriptor
- yield CPU-call scheduler to select next process

Re-dispatching a process
- decision to switch is made in supervisor mode
    - after state of current process has been saved
    - the scheduler has been called to yield the CPU
- select the next process to be run
    - get pointer to it process descriptor(s)
- locate and restore its saved state

    - restore code, data, and stack segments
    - restore saved registers, PS, and finally the PC
- and we are now executing the new process!

Blocking and Unblocking Processes

- Process needs an unavailable resource
    - data that has not yet been read in from disk
    - a message that has not yet been sent
    - a lock that has not yet been released
- Must be blocked until resource is available (when doing things like I/O requests)
    - change process state to blocked
- Un-block when resource becomes available
    - change process state to ready
- blocked/unblocked are merely notes to scheduler
    - blocked processes are not eligible to be dispatched
- anyone can set them, anyone can change them
- this usually happens in a resource manager
    - when process needs an unavailable resource
        - change process' scheduling state to "blocked"
        - call the scheduler and yield the CPU
    - when the required resource becomes available
        - change the process' scheduling data to "ready"
        - notify scheduler that a change has occurred
    - Running a new class is more expensive

Primary and Secondary Storage

- Primary = main (executable) memory
    - primary storage is expensive and very limited
    - only processes in primary storage can be run
- Secondary = non-executable (e.g. Disk)
    - blocked processes can be moved to secondary storage
    - swap out code, data, stack, and non-resident context
    - make room in primary for "ready" processes
- return to primary memory
    - process is copied back when it becomes unblocked

Why we swap

- Make the best use of limited memory
    - a process can only execute if it is in memory
    - max #of processes limited by memory size
    - if it isn't READY, it doesn't need to be in memory
- improve CPU utilization
    - when there are no READY processes, CPU is idle

    - Idle CPU time is waste, reduced throughput
    - we need READY processes in memory
- Swapping takes time and consumes I/O
    - we want to do it as little as we can

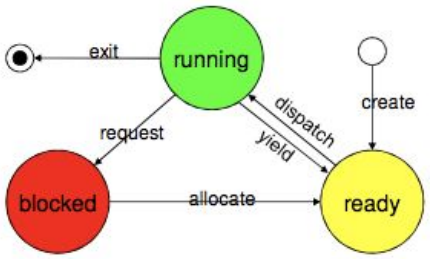
Swapping Out

- Process' state in main memory
    - code and data segment
    - non-resident process descriptor
- Copy them out to secondary
    - if we are lucky, some may still be there
- Update resident process descriptor
    - process is no longer in memory
    - pointer to location on secondary storage device
- Freed memory available for other processes to use

Swapping Back In

- Reallocate memory to contain process
    - code and data segments, non-resident process descriptor
- Read that data back from secondary storage
    - Change process state back to ready
- What about the state of the computations?
    - saved registers are on the stack
    - user-mode stack is in the saved data segments
    - supervisor-mode stack is in non-resident descriptor
- This involves a lot of time and I/P

Three State Scheduling Model

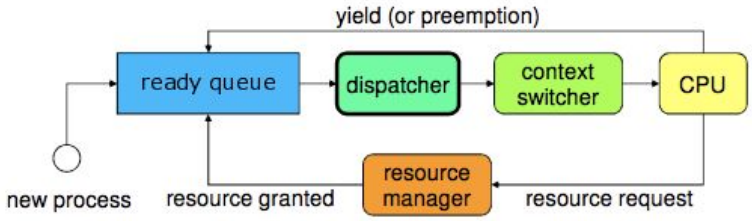


- a process may block to await
  - completion of a requested I/O operation
  - availability of an requested resource
  - some external event
- or a process can simply yield

Scheduling: Algorithms, Mechanisms and Performance 11

What is CPU Scheduling?

- Choosing which *ready* process to run next
- Goals:
  - keeping the CPU productively occupied
  - meeting the user's performance expectations



Scheduling: Algorithms, Mechanisms and Performance 12

Goals and Metrics

- Goals should be quantitative
  - if something is important it must be measurable
  - you cannot optimize hat you do not measure
- Metrics: the way and units which we measure
  - choose a characteristic to measure
    - it must correlate well with goodness/badness of service

        - it must be a characteristic we can measure or compute
    - find a unit to quantify that characteristic
    - define a process for measuring the characteristic

CPU Scheduling: Scheduling Metrics

- Mean time to completion (seconds)
    - for a particular job mix (benchmark)
- Throughput (operations per second)
    - for a particular activity or job mix
- Mean response time
    - time spent on the ready queue
- overall goodness
    - requires a customer specific weighting function
    - often stated in service level agreements

Different Kinds of Systems have Different Scheduling Goals

- Time sharing
    - Fast response time to interactive programs
    - Each user gets an equal share of the CPU
    - Execution favors higher priority processes
- Batch
    - Maximize total system throughput
    - Delays of individual processes are unimportant
- Real-Time
    - Critical operations must happen on time
    - non-critical operations may not happen at all

Non-Preemptive Scheduling

- Scheduled process runs until it yields CPU
    - may yield specifically to another process
    - may merely yield to “next” process
- works well for simple systems
    - small numbers of processes
    - with natural producer consumer relationships
- depends on each process to voluntarily yield
    - a piggy process can starve others
    - a buggy process can lock up the entire system

Non-Preemptive: First in First Out

- Algorithm:
    - run first process in queue until it blocks or yields
- Advantages
    - very simple to implement

    - seems intuitively fair
    - all process will eventually be served
- Problems
    - highly variable response time (delays)
    - a long task can force many others to wait (convoy)
        - if a long process comes first, then the other (possibly shorter) processes might have to wait
    - Starvation
        - a piggy process can starve other processes

Non-Preemptive: Shortest Job First

- Algorithm:
    - all processes declare their expected run time
    - run the shortest until it blocks or yields
- Advantages:
    - Likely to yield the fastest response time
- Problems:
    - Some processes may face unbounded wait times
        - example: If 99% of the processes have short run times but 1% of them has a long
    - ability to correctly estimate required run time dependent

Starvation

- Unbounded waiting times
    - not merely a CPU scheduling issue
    - it can happen with any controlled resource
- Caused by case-by-case discrimination
    - where it is possible to lose every time
- ways to prevent
    - strict (FIFO) queuing of prequests
        - credit for time spent waiting is equivalent
            - build up credit the longer you wait in line
        - ensure that individuals queues cannot be starved
    - input metering to limit queue lengths

Non-Preemptive: Priority

- Algorithm:
    - all processes are given a priority
    - run the highest priority until it blocks or yields
- Advantages:
    - Users control assignment of priorities
    - can optimize per-customer “goodness” function
- Problems

    - still subject to less arbitrary starvation
    - per process may not be fine enough control
- you can really just think about priority as the non-preemptive scheduling algorithm. The convoy problem is still prevalent

Preemptive Scheduling

- a process can be forced to yield at any time
    - if a higher priority process becomes ready
        - a perhaps as a result of an I/O completion interrupt
    - if running process' priority is lowered
- advantages
    - enables enforced "fair share" scheduling
- Problems
    - introduces gratuitous context switches
    - creates potential resource sharing problems

Forcing Processes to Yield

- need to take CPU away from process
    - process makes a system call, or clock interrupt
- consult scheduler before returning to process
    - if any ready process has had priority raised
    - if any process has been awakened
    - if current process has had priority lowered
- scheduler finds highest priority ready process
    - if current process, return as usual
    - if not, yield on behalf of the current process

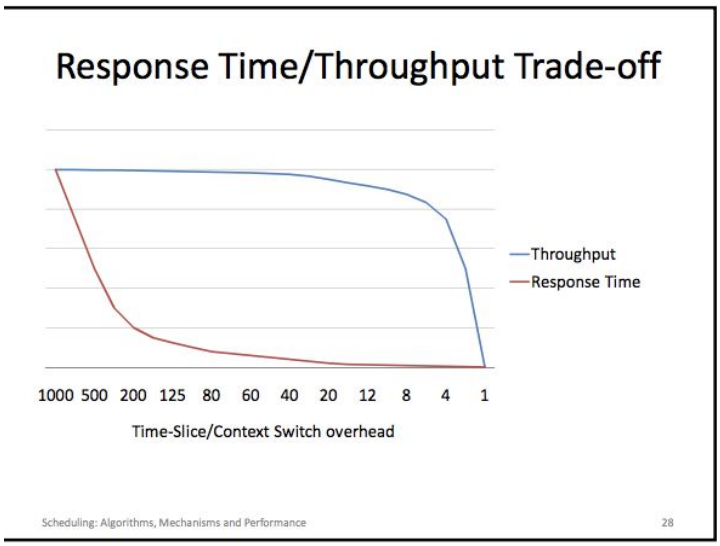
Preemptive: Round-Robin

- Algorithm
    - processes are run in (circular) queue order
    - each process is given a nominal time-slice
    - timer interrupts process if time-slice expires
        - time interrupt has to be a multiple of the time-slice
        - slowest possible interrupt rate is the time-slice
- Advantages
    - greatly reduced time from ready to running
    - intuitively fair
- Problems
    - some processes will need many time-slices
    - extra interrupts/context switches add overhead and can get expensive
- Gives better response time, at the cost of overhead and turnaround time

Costs of an extra context switch

*cost is roughly the same as the cost of a system call

- Entering the OS
    - taking interrupt, saving registers, calling scheduler
- Cycles to choose who to run
    - the scheduler/dispatcher does work to choose
- moving OS context to the new process
    - switch out old process, map in new process
- losing hard-earned L1 and L2 cache contents



So which approach is the best?

- preemptive has better response time
    - but what should we choose for our time-slice?
- non-preemptive has lower overhead
    - but how should we order our processes
- there is no one “best” algorithm
    - performance depends on the specific job mix
    - goodness is measured relative to specific goals
- a good scheduler must be adaptive
    - responding to automatically changing job loads
    - configurable to meet different requirements

*Remember how one would compare with one another depending on the situation

The “Natural” Time-Slice

- CPU share = time_slice x slices/second
    - 2% = 20ms/sec 2ms/slice x 10 slices/sec
    - 2% = 20ms/sec 5ms/slice x 4 slices/sec
- context switches are from free
    - they waste otherwise useful cycles
    - they introduce delay into useful computations

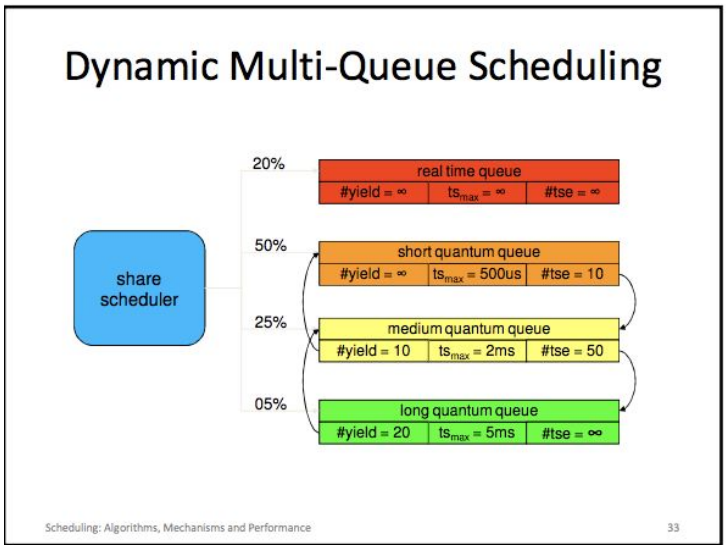
- natural rescheduling interval
    - when a process blocks for resources or I/O
    - optimal time-slice would be based on this period

Dynamic Multi-Queue Scheduling

- natural time-slice length is different for each process
    - create multiple ready queues
    - some with short time-slices that run more often
    - some with long time-slices that run infrequently
    - different queues may get different CPU shares
- Advantages
    - response time very similar to RR
    - relatively few gratuitous presumptions
- Problem
    - how do we know where a process belongs?

Dynamic Equilibrium

- Natural equilibria are seldom calibrated
- Usually the net result of
    - competing processes
    - negative feedback
- Once set in place these processes
    - are self calibrating
    - automatically adapt to changing circumstances
- The tuning is in rate and feedback constants
    - avoid overcorrection, ensure convergence



*multiple queues, each queue gets a different time slice/scheduling parameters
*if you exhibit good behavior, you get moved to a better queue, and if you exhibit bad behavior you get moved to a bad queue

*In the above, if you go over the allotted time slice or yield too many times,
*If you stop running before you preemption, it counts as a yield. If you are preempted, then it also counts as a yield
*Priority Boosting: The lower priority processes will not run if they aren't guaranteed to run, leading to starvation. If you spent a certain amount of time in the queue, you may receive a priority boost to higher level queue
*Mechanism is the system that reads the parameters, while the parameters themselves are the policies

*Arpaci's Queue vs Dynamic MLFQ (Kampe's)

- Arpaci's Queue uses priority boosting which intrinsically cause starvation. However, by creating a shared scheduler, with parameters/limits on yield numbers, time slice length, and number of time slices we eliminate the starvation problem
    - Priority boosting still exists as processes build more credit for being in a lower-quantum queue for too long. You may receive a priority boost to a higher level queue if you've been in the queue for a long time

Mechanism/Policy Separation

- simple built in scheduler mechanisms
    - always run the highest priority process
    - formulae to computer priority and time slice length
- controlled by user specifiable policy
    - per process (inheritable) parameters
        - initial, relative, minimum, maximum priorities
        - queue in which process should be started
        - these can be set based on user ID, or program being run
    - per queue parameters
        - maximum time slice length and number of time slices
        - priority changes per unit of run time and wait time
        - CPU share

Real-Time Schedulers

- Some things must happen at particular times
    - if you can't process the next sound sample in time, there will be a gap in the music
    - if you don't rivet the widget before the conveyor belt moves, you have a manufacturing error
    - if you can't adjust the spoilers quickly enough, the space shuttle goes out of control
- Real Time scheduling has deadlines
    - they can be either soft or hard

Hard Real-Time Schedulers

- The system absolutely must meet its deadlines
- By definition, system fails if a deadline is not met
- How can we ensure no missed deadlines?
- Typically by careful design-time analysis
    - prove no possible schedule misses a deadline
    - scheduling order may be hard-coded

Example question: Here is a situation, what algorithm would be best

Ensuring Hard Deadlines

- Requires deep understanding of all code
    - we know exactly how long it will take in every case
- Avoid complex operations with non-deterministic times
    - e.g. interrupts, garbage collection
- Predictability is more important than speed
    - non-preemptive, fixed execution order
    - no run time decision

Soft Real Time Schedulers

- highly desirable to meet your deadlines
    - some can occasionally be missed
- Goal of scheduler is to avoid missing deadlines
    - with the understanding that you might
    - sometimes called “best effort”
- May have different classes of deadlines
    - some “harder” than others
- May have more dynamic/variable traffic

Soft Real-Time algorithms?

- Most common is Earliest Deadline First
    - each job has a deadline associated with it
    - keep the job queue sorted by those deadlines
    - always run the first job on the queue
- Minimize total lateness
- Possible refinements
    - skip jobs that are already rate
    - drop low priority jobs when system is overloaded

Example of a Soft Real Time Scheduler

- A video playing device
- Frames arrive
- Each frame should rendered “on time”
    - to achieve highest user-perceived ability

- if a frame is late, skip it
    - rather than fall further behind