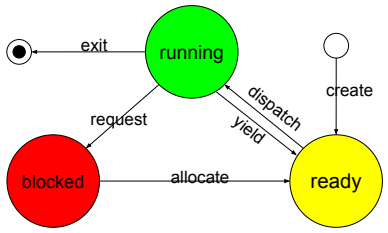
Scheduling

- 4A Introduction to scheduling
- 4B Non-preemptive scheduling
- 4C Preemptive scheduling
- 4D Adaptive scheduling
- 4E Scheduling and performance
- 4F Real-Time scheduling

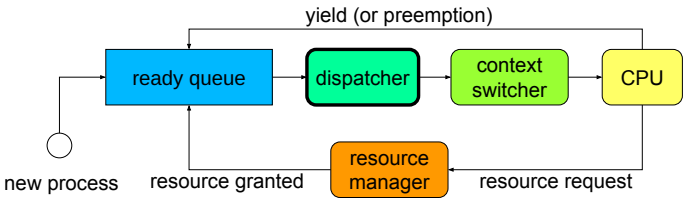
Basic Scheduling State Model



- 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

What is CPU Scheduling?

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



Goals and Metrics

- goals should be quantitative and measurable
  - if something is important, it must be measurable
  - if we want "goodness" we must be able to quantify it
  - you cannot optimize what you do not measure
- metrics ... the way & units in which we measure
  - choose a characteristic to be measured
    - 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: Proposed Metrics

- candidate metric: time to completion (seconds)
  - different processes require different run times
- candidate metric: throughput (procs/second)
  - same problem, different processes different time
- candidate metric: response time (milliseconds)
  - some delays are not the scheduler's fault
    - time to complete a service request, wait for a resource
- candidate metric: fairness (standard deviation)
  - per user? per process? are all equally important?

Rectified Scheduling Metrics

- mean time to completion (seconds)
  - for a particular job mix (benchmark)
- throughput (operations per second)
  - for a particular activity or job mix (benchmark)
- mean response time (milliseconds)
  - 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

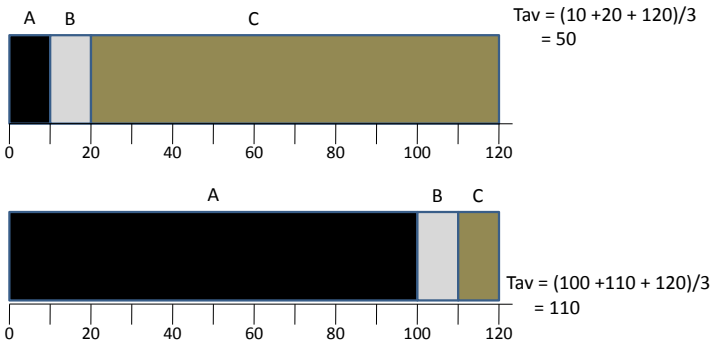
Non-Preepmtive Scheduling

- scheduled process runs until it yields CPU
  - may yield specifically to another process
  - may merely yield to "next" process
- simple, 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)

Example: First In First Out



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
    - Is this fair? Is this even “correct” scheduling?
  - ability to correctly estimate required run time

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 requests
    - credit for time spent waiting is equivalent
    - ensure that individual 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

Preemptive Scheduling

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

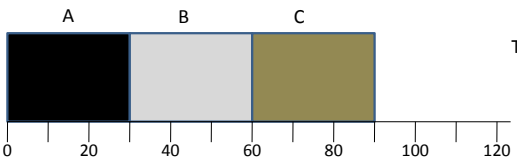
Forcing Processes to Yield

- need to take CPU away from process
  - e.g. 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

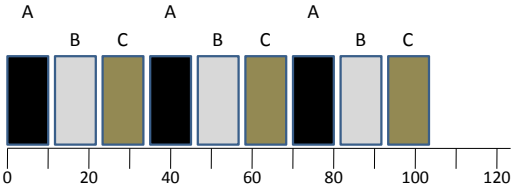
Preemptive: Round-Robin

- Algorithm
  - processes are run in (circular) queue order
  - each process is given a nominal time-slice
  - timer interrupts process if time-slice expires
- Advantages
  - greatly reduced time from *ready* to *running*
  - intuitively fair
- Problems
  - some processes will need many time-slices
  - extra interrupts/context-switches add overhead

Example: Round-Robbin



Trsp = (0 +30 + 60)/3
= 30

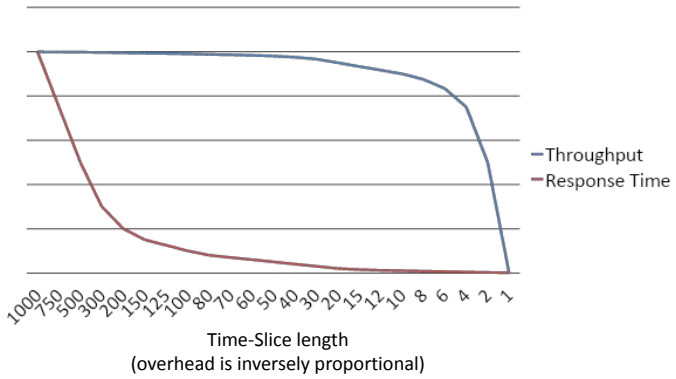


Trsp = (0 +11 + 22)/3
= 11

Costs of an extra context-switch

- 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 process descriptor, kernel stack
- switching process address spaces
  - map-out old process, map-in new process
- losing hard-earned L1 and L2 cache contents

Response Time/Throughput Trade-off



So which approach is 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 the 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 automatically to changing loads
  - configurable to meet different requirements

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 far 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

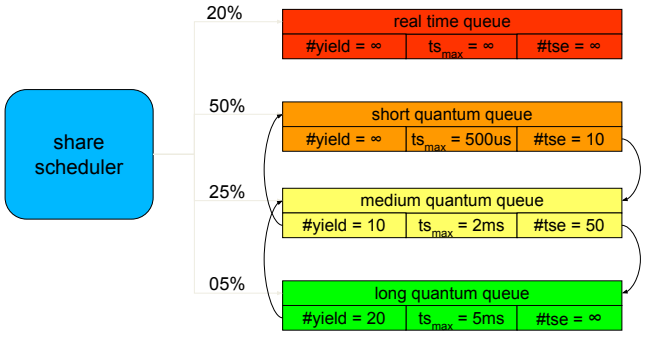
Dynamic Multi-Queue Scheduling

- natural time-slice 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 Round-Robin
  - fewer gratuitous preemptions
- 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 over-correction, ensure convergence

Dynamic Multi-Queue Scheduling



Mechanism/Policy Separation

- simple built-in scheduler mechanisms
  - always run the highest priority process
  - formulae to compute 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 (or resumed)
    - 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 change per unit of run time and wait time
    - CPU share (absolute or relative to other queues)

Real Time Schedulers

- Some things must happen at particular times
  - if we can't process the next sound sample in time, there will be a gap in the music
  - if we don't attach the widget before the conveyer belt moves, we have a manufacturing error
  - if we can't adjust the spoilers quickly enough during reentry, 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 deadline isn't met
  - e.g., controlling a supersonic fighter in flight . . .
- 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

Ensuring Hard Deadlines

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

Soft Real Time Schedulers

- Highly desirable to meet your deadlines
  - some (or any) 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
  - rendering up-front analysis impractical

Soft Real Time and Preemption

- All tasks need not always run to completion
  - we are allowed to miss some deadlines
- A high priority near-deadline task may arrive
  - it should preempt a lower priority task
- What if we miss (or cannot make) a deadline?
  - we fall behind, run it as soon as possible?
  - skip this invocation, we will catch it next time?
  - kill the task that missed its deadline?

  This is a policy question, let designers decide

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
- Minimizes *total lateness*
- Possible refinements
  - skip jobs that are already late
  - drop low priority jobs when system is overloaded

Example of a Soft Real Time Scheduler

- A video playing device
- Frames arrive (e.g. from disk or network)
- Each frame should be rendered "on time"
  - to achieve highest user-perceived quality
- If a frame is late, skip it
  - rather than fall further behind

Graceful Degradataion

- System overloads will happen
  - random fluctuations in traffic
  - load bursts from unanticipated events
  - additional work associated with errors
- What to do when the system is overloaded?
  - offer slower service to all clients?
  - allow deadlines to get later and later?
  - offer on-time service to fewer clients?
- We must choose (or allow clients to do so)

CPU Scheduling is not Enough

- CPU scheduler chooses a *ready* process
- memory scheduling
  - a process on secondary storage is not *ready*
- resource allocation
  - a process waiting for a resource is not *ready*
- I/O scheduling
  - a process waiting for I/O is not *ready*
- cache management
  - if process data is not cached, it will need more I/O

Reading and Assignments

Reading:

- Arpaci C12 … introduction to memory
- Arpaci C13 … address spaces
- Arpaci C14 … memory APIs
- Arpaci C17 … allocation algorithms
- Kampe: Garbage Collection

Projects:

- start looking at project 1B (compressed network I/O)

Supplementary Slides

Pros and Cons of Non-Preemptive Scheduling

+ Low scheduling overhead
+ Tends to produce high throughput
+ Conceptually very simple
− Poor response time for processes
− Bugs can cause machine to freeze up
  − If process contains infinite loop, e.g.
− Not good fairness (by most definitions)
− May make real time and priority scheduling difficult

First Come First Served Example

Dispatch Order	0, 1, 2, 3, 4				
Process	Duration		Start Time		End Time
0	350		0		350
1	125		350		475
2	475		475		950
3	250		950		1200
4	75		1200		1275
Average wait			595		

Total 1275

595

Note: Average is worse than total/5 because four other processes had to wait for the slow-poke who ran first.

When Would First Come First Served Work Well?

- FCFS scheduling is very simple
- It may deliver very poor response time
- Thus it makes the most sense:
  1. In batch systems, where response time is not important
  2. In embedded (e.g. telephone or set-top box) systems where computations are brief and/or exist in natural producer/consumer relationships

Priority Scheduling Algorithm

- Sometimes processes aren't all equally important
- We might want to preferentially run the more important processes first
- How would our scheduling algorithm work then?
- Assign each job a priority number
- Run according to priority number

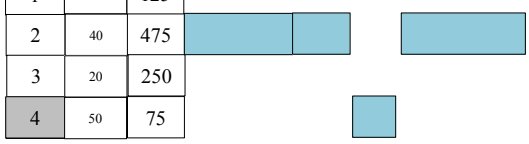
Priority and Preemption

- If non-preemptive, priority scheduling is just about ordering processes
- Much like shortest job first, but ordered by priority instead
- But what if scheduling is preemptive?
- In that case, when new process is created, it might preempt running process
  – If its priority is higher

Priority Scheduling Example

550 **Time**

Process	Priority	Length
0	10	350
1	30	125
2	40	475
3	20	250
4	50	75



Process 4 completes

So we go back to process 2

Process 3's priority is lower than running process

Process 4's priority is higher than running process

Problems With Priority Scheduling

- Possible starvation
- Can a low priority process ever run?
- If not, is that really the effect we wanted?
- May make more sense to adjust priorities
  - Processes that have run for a long time have priority temporarily lowered
  - Processes that have not been able to run have priority temporarily raised

Hard Priorities Vs. Soft Priorities

- What does a priority mean?
- That the higher priority has absolute precedence over the lower?
  - Hard priorities
  - That's what the example showed
- That the higher priority should get a larger share of the resource than the lower?
  - Soft priorities

Priority Scheduling in Linux

- Each process in Linux has a priority
  - Called a *nice* value
  - A soft priority describing share of CPU that a process should get
- Commands can be run to change process priorities
- Anyone can request lower priority for his processes
- Only privileged user can request higher

Priority Scheduling in Windows

- 32 different priority levels
  - Half for regular tasks, half for soft real time
  - Real time scheduling requires special privileges
  - Using a multi-queue approach
- Users can choose from 5 of these priority levels
- Kernel adjusts priorities based on process behavior
  - Goal of improving responsiveness