What is CPU Scheduling?

• Choosing which *ready* process to run next

**Goals:**
- keeping the CPU productively occupied
- meeting the user’s performance expectations

![CPU Scheduling Diagram](image)

**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, not different processes

• 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
Basic Scheduling State Model

- A process may block to await
  - Completion of a requested I/O operation
  - Availability of a requested resource
  - Some external event
- Or a process can simply yield

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 processes 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

\[ T_{av} = \frac{(10 + 20 + 120)}{3} = 50 \]

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
  - 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

\[
\text{Trsp} = \frac{0 + 30 + 60}{3} = 30
\]

\[
\text{Trsp} = \frac{0 + 11 + 22}{3} = 11
\]

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
  - relatively few 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

- short quantum queue
  - share = 20%
  - #yield = 10
  - #tse = 10
- medium quantum queue
  - share = 50%
  - #yield = 50
  - #tse = 50
- long quantum queue
  - share = 30%
  - #yield = 30
  - #tse = 30
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)

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

assignments

- reading for the next lecture
  - Arpaci ch 12 ... Introduction
  - Arpaci ch 13 ... Address Spaces
  - Arpaci ch 14 ... Memory API
  - Arpaci ch 15 ... Address Translation
  - Arpaci ch 16 ... Segmentation
  - Arpaci ch 17 ... Free Space Management

Quiz 5 is due before the lecture!
Have your project 1 issues ready for lab session

Charles Dickens on System Performance

"Annual income, twenty pounds; annual expenditure, nineteen, nineteen, six; Result ... happiness.
Annual income, twenty pounds; annual expenditure, twenty pounds ought & six; Result ... misery!"

Wilkins Micawber, David Copperfield

Performance: Throughput vs Load

throughput

ideal

typical

offered load
(why throughput falls off)

- dispatching processes is not free
  - it takes time to dispatch a process (overhead)
  - more dispatches means more overhead (lost time)
  - less time (per second) is available to run processes
- how to minimize the performance gap
  - reduce the overhead per dispatch
  - minimize the number of dispatches (per second)
    - allow longer time slices per task
    - increase the number of servers (e.g. CPUs)
- this phenomenon will be seen in many areas

(why response time grows w/o limit)

- response time is function of server & load
  - how long it takes to complete one request
  - how long the waiting line is
- length of the line is function of server & load
  - how long it takes to complete one request
  - the average inter-request arrival interval
- if requests arrive faster than they are serviced
  - the length of the waiting list grows
  - and the response time grows with it

Performance: response time vs load

Graceful Degradation

- when is a system "Overloaded"?
  - when it is no longer able to meet service goals
- what can we do when overloaded?
  - continue service, but with degraded performance
  - maintain acceptable performance by rejecting work
  - resume normal service when load drops to normal
- what can we not do when overloaded?
  - allow throughput to drop to zero (stop doing work)
  - allow response time to grow without limit