Scheduling

3F. Execution State Model
4A. Introduction to Scheduling
4B. Non-Preemptive Scheduling
4C. Preemptive Scheduling
4D. Dynamic Scheduling
4E. Real-Time Scheduling
4F. Scheduling and Performance

**un-dispatching a running process**
- somehow we enter the operating system
  - e.g. 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 also 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

**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
  - change process state to blocked
- Un-block when resource becomes available
  - change process state to ready

**execution states with swapping**

**Blocking and unblocking processes**
- 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’s scheduling state to “blocked”
      - call the scheduler and yield the CPU
    - when the required resource becomes available
      - change process’s scheduling state to “ready”
      - notify scheduler that a change has occurred

**re-)dispatching a process**
- decision to switch is made in supv 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 its process descriptor(s)
- locate and restore its saved state
  - restore code, data, stack segments
  - restore saved registers, PS, and finally the PC
- and we are now executing in a new process
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 other "ready" processes
• returning 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 wasted, reduced throughput
  – we need READY processes in memory
• Swapping takes time and consumes I/O
  – so we want to do it as little as possible

Swapping Out

• Process’ state is in main memory
  – code and data segments
  – non-resident process descriptor
• Copy them out to secondary storage
  – 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

Swapping Back In

• Re-allocate 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/O

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, 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

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)

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

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

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

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

<table>
<thead>
<tr>
<th>Throughput</th>
<th>Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>500</td>
</tr>
<tr>
<td>200</td>
<td>100</td>
</tr>
<tr>
<td>80</td>
<td>40</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

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 = \( \text{time_slice} \times \text{slices/second} \)
  - 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 and negative feedback
- Once set in place these processes are self-calibrating and 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 and 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—every 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—e.g., controlling a nuclear power plant . . .
- 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—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 the programmer 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 Degradation

- 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
Assignments

- Projects
  - try to get P1A running, take problems to lab
- Reading
  - A-D 12 (introduction to memory)
  - A-D 13 (address spaces)
  - A-D 14 (memory APIs)
  - A-D 17 (allocation algorithms)

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

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