Scheduling II

Today
- Proportional-share scheduling
- Multilevel-feedback queue
- Multiprocessor scheduling

Next Time
- Memory management
Scheduling with multiple goals

- What if you want both good turnaround time and responsiveness
- To optimize turnaround time, SJF
  - But you don’t know how long a job will run for
- To improve responsiveness, Round Robin
  - Which is terrible for turnaround time
Multilevel queue

- Ready queue partitioned into separate queues
- Each queue has its own scheduling algorithm
- Now must schedule between the queues
  - Fixed priority scheduling; (i.e., foreground first); starvation?
  - Time slice – each queue gets a fraction of CPU time which it can schedule amongst its processes
Multiple (feedback) queues ...

• Rather than assigning processes to queues, allow them to move between queues based on observed behavior
  – Job enters at highest priority
  – If uses entire slice while running, it moves down (priority reduced)
  – If it gives up CPU before quantum ends, stays in place

• How MLFQ approximates SJF
  – Since it doesn’t know whether the job will be short, assumes so and gives it high priority
  – If it turns out to be short, it will run quickly and be done
  – Else, it will move to lower queues and become more batch-like
MLFQ v1

- Two jobs: A, long-running CPU-intensive; B, short-running, interactive

- Some issues
  - Starvation again – if interactive jobs keep arriving ...
  - You can game the scheduler – issuing an I/O before your quantum is over and the job won’t move
  - What if the process changes behavior over time?
MLFQ v2

- **Addressing starvation**
  - Periodically boost priority; let’s say every time period $S$
  - Now another problem is setting $S$

- **CTSS – First MLFQ**
  - IBM 7094 had space for 1 process in memory (switch = swap)
  - Priority classes: class $i$ gets $2^i$ quantas
    - Scheduler executes first all processes in queue 0; if empty, all in queue 1, ...
  - What about process with long start but interactive after that?
    - Carriage-return hit → promote process to top class on the assumption it will become interactive
MLFQ v2

- Easy to game
  
  *Keep hitting that CR key!*

- Preventing gaming
  - Better accounting, instead of forgetting how much of its time-slice a process has used
  - Once a process has used its allotment, it is demoted to next priority queue
Scheduling challenges

- **Context**
  - Multiplex scarce resources
  - Among concurrently executing clients
  - Servicing request of varying importance

- **Priority scheduling**
  - Absolute control, but crude
  - Often ad-hoc
  - Resource rights don’t vary smoothly
  - Unable to control service rates of tasks
  - No modular abstraction

- **Proportional-share scheduling**
  - The execution rate of processes is proportional to the relative share that they are allocated – *fairness*
Proportional-share scheduling

- Lottery scheduling – a modern example
  - Randomized resource allocation
  - Each process gets lottery tickets for resources (CPU time)
  - Scheduling – lottery, i.e. randomly pick a ticket
  - Probabilistically fair

- A basic run – A: 75 tickets; B: 15 tickets
  - Hold the lottery, random[0..99], in our case
  - If 0-74, run A, else run B

A: 11/5 ~73%

Lottery scheduling

- CPU allocation and response time follow distributions with well-understood properties
  - Number of lotteries won by a process as a binomial distribution
    - Probability that a client with $t$ tickets will win a lottery is $p = t/T$ ($T$ is total tickets); after $n$ lotteries, expected number of wins $E[w] = np$
  - Number of lotteries for a first win has a geometric distribution
    - A process avg response time is inversely proportional to ticket allocation

- Other features
  - No starvation
  - Fair with number of tickets varying dynamically
  - Responsive to changes on ticket allocation
Lottery scheduling

- Simple implementation
  - You just need a good random number generator
  - A list of processes with their tickets
  - Pick a winner, walk the list adding up tickets until count exceed winner – current process is the winning process

```
Winner: 299
Count: 0
Count: 100
Count: 150
Count: 400
```

```
Winning process: C
```

- Any easy way to optimize this?
Lottery scheduling

● Some interesting features
  – Tickets can be used to insulate resource management policies of independent modules
  – Tickets transfer
    • Tickets can be treated as first class objects, so they can be transferred in messages
    • If you are blocked on someone else, give them your tickets
    • A client waiting on multiple clients, divide tickets among them
  – Ticket inflation/deflation is an alternative to transfer
    • A client can escalate its resource rights by creating more tickets
    • Only among mutually trusting clients
Lottery scheduling

- Some interesting features
  - Ticket currencies to isolate trust boundaries
    - A unique currency is used to denominate tickets within trust boundaries
    - Each currency is backed by tickets denominated in more primitive currencies – defining an acyclic graph
    - Effects of inflation can be contained with a base currency, that is conserved, and an exchange rate
  - Compensation tickets if a process consume a fraction of its allocated resource (blocking before quantum expires)
    - Issue tickets to inflate values in proportion to unused resource until next quantum
    - Ensure everybody gets its share of the CPU

- But how do you assign tickets?
  - Assume the user knows (but that’s really a non-solution)
And now a short break ...
Multiple-processor scheduling

- Scheduling more complex with multiple CPUs
  - From one to two dimensions – “Which process to run next?”
    - “Which process to run and where?”
  - Are all process related or are they independent?
  - When re-scheduling a process, what should we do with the data cached in the previous run?
Multiple-processor scheduling

- **Single-queue scheduling / time sharing**
  - Reuse known frameworks from single processor scheduling; automatic load-balancing
  - Contention for scheduling data
  - What if two processes share a lock and the one holding it ends its quantum?
    - Let the scheduler know setting a process-wide spin-lock flag?
  - Cache affinity?
    - Add some kind of affinity mechanism (e.g., a bitmask of CPUs allowed, like in Linux sched. 2.4 & 2.5)
Multiple-queue scheduling / space sharing

- Multiple queue, each with its own algorithm
- Process comes into a queue (which one?) and is scheduled from there
- Clearly more scalable – less contention for locks
- But … load imbalance
  - Job migration? Push or pull
Some other algorithms

- Guaranteed scheduling - e.g. proportional to number of processes
  - Priority = amount used / amount promised
  - Lower ratio $\rightarrow$ higher priority

- Fair-Share scheduling
  - Schedule aware of ownership
    - If user 1 starts up 9 processes and user 2 starts up 1, user 1 will get 90\% of the CPU
  - Owners get a \% of CPU, processes are picked to enforce it
  - Divide user set into fair-share groups
  - Scheduling is done based on priority
    - Accounting for underlying priority, recent usage, recent group usage
Some other algorithms – Real-time

- Different categories
  - *Soft or Hard RT* – Important or critical?
  - Hard: not on time ~ not at all
  - Scheduling can be static or dynamic

- Schedulable real-time system
  - Events a RT system may have to respond could be periodic (occurring at regular intervals) or aperiodic (occurring unpredictably)
  - Given a set of $m$ periodic events $i$ each with period $P_i$ and requiring $C_i$ seconds of CPU time, can the load be handle?

Schedulable
  - An admission-control policy could reject/accept a new job if schedulable
Solaris scheduling as an example

- Solaris is based on Unix System V Release 4 (SVR4)
  - As in all SVR4-based schedulers, two levels
    - Class-independent routines: dispatching and preemtping (mechanisms)
    - Class-dependent: setting priorities (policy)

- Three scheduling classes or priority-class types
  - Real time – priorities 100-159
  - System – priorities 60-99
  - Time sharing – priorities 0-59

- When a process is created, it inherits its parent’s priority class characteristics
  - i.e., priority class and global priority value
  - Most jobs will be running in the TS class
Solaris TS Scheduling Class

- A multi-level feedback queue scheduler
  - Jobs begins at priority 29 (range 0-59)
  - Priority is calculated from two proportional values, a kernel part and a user provided part for backward compatibility (nice)
  - Compute-bound jobs filter down to lower priorities
    - The priority of a process is lowered after it consumes its allocated timeslice
    - Schedule less frequently but for longer quantum
  - Interactive jobs move to higher priorities
Policy vs. mechanism

- Separate what is done from how it is done
  - Think of parent process with multiple children
  - Parent process may knows relative importance of children (if, for example, each one has a different task)

- None of the algorithms presented takes the parent process input for scheduling

- Scheduling algorithm parameterized
  - Mechanism in the kernel

- Parameters filled in by user processes
  - Policy set by user process
  - Parent controls scheduling w/o doing it
Next time

- We have discussed sharing CPU to improve utilization and turnaround time
- For that to happen we also need to share memory
- We’ll start with memory organization and basic management techniques (e.g. paging)