Scheduling II

To do ...

- Proportional-share scheduling
- Multilevel-feedback queue
- Multiprocessor scheduling
- Next Time: Memory management
Scheduling with multiple goals

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

\textit{\ldots different policies for different jobs?}
Multilevel queue

- Ready queue partitioned into separate queues
  - Each has its own scheduling algorithm
- Now we must schedule between queues
  - Fixed priority scheduling – starvation?
  - Time slice – each queue gets a fraction of CPU time which it can schedule amongst its processes

But how to decide where should a job go? Can the scheduler learn this on the fly?
Multiple (feedback) queues ...

- Rather than assigning processes to queues, move them around based on their behavior
  - Job enters at highest priority
  - If uses entire slice, it moves down (priority reduced)
  - If it gives up CPU before quantum ends, stays in place

- How MLFQ approximates SJF
  - Assumes any job is short and gives them highest 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 ...
- Gaming the scheduler – issuing an I/O before quantum is up and the job won’t move
- What if the process changes behavior over time?
CTSS – First MLFQ
- IBM 7094 had enough memory for 1 process (switch = swap)
- Priority classes: class \(i\) gets \(2^i\) quantas
  - Runs all processes in queue 0; if empty, all in queue 1, ...

Addressing changing behavior – what about a process with long start but interactive after that?
- Carriage-return hit → promote process to top class on the assumption it will become interactive

Problem – Easy to game!

*Keep hitting that CR key!*
MLFQ v2

- Addressing starvation and changing behavior
  - Periodically boost priority; let’s say every time period $S$; move every job to topmost queue
  - Setting $S$ is another problem

- Preventing gaming
  - Better accounting, instead of forgetting how much of its time-slice a process has used
    - Track time used, independently of how many times it gave up the CPU
  - Once a process has used its allotment, it is demoted to next priority queue
Scheduling challenges

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

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

- Don’t optimize turnaround or response time, ensure all processes get a fair share of CPU

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

- 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
And now a short break ...

<table>
<thead>
<tr>
<th>CRAZY PHENOMENON</th>
<th>IF IT WORKED, COMPANIES WOULD BE USING IT TO MAKE A KILLING IN...</th>
<th>ARE THEY?</th>
</tr>
</thead>
<tbody>
<tr>
<td>REMOTE VIEWING</td>
<td>OIL PROSPECTING</td>
<td></td>
</tr>
<tr>
<td>DOWSING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AURAS</td>
<td>HEALTH CARE COST REDUCTION</td>
<td></td>
</tr>
<tr>
<td>HOMEOPATHY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REMOTE PRAYER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASTROLOGY</td>
<td>FINANCIAL/BUSINESS PLANNING</td>
<td></td>
</tr>
<tr>
<td>TAROT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRYSTAL ENERGY</td>
<td>REGULAR ENERGY</td>
<td></td>
</tr>
<tr>
<td>CURSES, HEXES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RELATIVITY</td>
<td>GPS DEVICES</td>
<td>✔️</td>
</tr>
<tr>
<td>QUANTUM ELECTRODYNAMICS</td>
<td>SEMICONDUCTOR CIRCUIT DESIGN</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Eventually, arguing that these things work means arguing that modern capitalism isn't that ruthlessly profit-focused.
Lottery scheduling

- A basic run – A: 75 tickets; B: 25 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
  - Lotteries won by process as a binomial distribution
    - Probability of winning is proportional to number of tickets $t$
      $$ p = \frac{t}{T} \text{ where } T \text{ is total tickets} $$
      After $n$ draws, expected number of wins $E[w] = np$
  - 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
  - Need a good random number generator
  - A list of processes with their tickets
  - Pick a winner, walk the list adding up tickets, when count exceed winner, current process is the chosen one

![Diagram of processes and tickets]

- Any easy way to optimize this?
Lottery scheduling – Interesting features

- Tickets can help insulate resource management policies of independent modules

- Tickets as currency, ticket transfer
  - A process can distribute its tickets to its children; the system will adjust the tickets appropriately
  - Tickets as first class objects that can be transferred in msgs
  - If A is blocked on B, A could loans tickets to B

- Ticket inflation/deflation as an alternative to transfer
  - A client can escalate its resource rights by creating more tickets
  - Only among mutually trusting clients
Lottery scheduling – Interesting features

- Ticket currencies to isolate trust boundaries
  - A currency for 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
  - Issue tickets, proportional to unused resource, until next quantum

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

- Multiprocessors are everywhere
- So, how do you benefit from them?
- First, a typical application can’t – you will need to rewrite it to run in parallel
  - There are options but let’s not go there for now
- Then is the problem of how to schedule across multiple CPUs …
Multiple-processor scheduling

- Issues with multiprocessing – caches
  - Caches and locality (temporal and spatial)
  - Consistency, synchronization and affinity

- Scheduling complexity
  - From 1d to 2d: “Which process to run next?” → “Which process to run and where?”
  - Are processes related? With parallel make, scheduling all compilations but one makes little sense
  - When re-scheduling a process, what about the data cached?
Multiple-processor scheduling

- Single-queue scheduling / time sharing
  - Reuse known framework; automatic load-balancing
  - Contention for scheduling data
  - Cache affinity?
    - Add some kind of affinity mechanism (e.g., a bitmask of CPUs allowed, like in Linux sched. 2.4 & 2.5)
Multiple-processor scheduling

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 – 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**
  - A RT system may have to respond to periodic (at regular intervals) or aperiodic (unpredictable) events
  - Given a set of \( m \) periodic events \( i \) each with period \( P_i \) and requiring \( C_i \) seconds of CPU, can the load be handle? *Schedulable*
    - An admission-control policy could reject/accept a new job depending on this
Xv6 Scheduler

- In proc.c, there are a few functions related to scheduling
- scheduler is the entry point to the scheduler
- scheduler’s job is to switch context to a process, let it run, then resume
- Currently, a very simple round-robin implementation

With one CPU and, thus, one scheduler thread
void sched(void)
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}

void exit(void)
{
  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
void scheduler(void) {
    struct proc *p;

    for(;;){
        // Enable interrupts on this processor.
        sti();

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;

            // Switch to chosen process. It is the process’s job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            proc = p;
            switchuvvm(p);
            p->state = RUNNING;
            swtch(&cpu->scheduler, p->context);
            switchkvm();

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            proc = 0;
        }
        release(&ptable.lock);
    }
}
Solaris scheduling as an example

- Based on Unix System V Release 4 (SVR4)
  - Two parts, a class-independent – dispatching and preempting (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

- A process inherits its parent’s priority class characteristics
  - i.e., priority class and global priority value
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
    – Process priority is lowered after it consumes its quantum
    – Schedule less frequently but for longer
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 may know 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)