Scheduling

Today
- Introduction to scheduling
- Classical algorithms

Next Time
- Process interaction & communication
Scheduling

- **Problem**
  - Several ready processes and threads & much fewer CPUs

- **A choice has to be made**
  - By the *scheduler*, using a *scheduling algorithm*

- **The decision, scheduling, is policy**

- **Context switching is a mechanism**
Scheduling through time

- Early batch systems – Just run the next job in the tape
- Early timesharing systems – Scarce CPU time so scheduling is critical
- PCs – Commonly one active process so scheduling is easy; with fast & per-user CPU scheduling is not critical
- Networked workstations & servers – All back again, multiple ready processes & expensive CS, scheduling is critical
Environments and goals

- Different scheduling algorithms, with different goals, for different application areas

- Worth distinguishing
  - Batch
  - Interactive
  - Real-time

- Goals for all/most systems
  - Fairness – comparable processes getting comparable service
  - Policy enforcement – seeing that stated policy is carried out
  - Balance – keeping all parts of the system busy (mix pool of processes)
Environments and goals

- Batch systems
  - Throughput – max. jobs per hour
  - Turnaround time – min. time bet/ submission & termination
  - CPU utilization – keep CPU busy all time *(anything wrong?)*

- Interactive systems
  - Response time – respond to requests quickly (time to start responding)
  - Proportionality – meet users’ expectations

- Real-time system
  - Meeting deadlines – avoid losing data
  - Predictability – avoid quality degradation in multimedia systems

- Average, maximum, minimum or *variance*?
Process behavior

- **Task** – a request to be scheduled (a thread/process may be responsible for multiple tasks)
- **Workload** – a set of tasks for some systems to perform, the input to the scheduling algorithm
- **Bursts of CPU usage alternate with periods of I/O wait**
  - Key to scheduling – CPU-bound & I/O bound process
  - As CPU gets faster – more I/O bound processes

![Histogram of CPU-burst times]

- Large number of short CPU bursts
- Small number of long CPU bursts
When to make scheduling decisions?

1. At process creation
2. When a process exits
3. When a process blocks on I/O, a semaphore, etc
4. When an I/O interrupts occurs
5. A fixed period of time – Need a HW clock interrupting
When to schedule?

- A fixed periods of times … preemptive and non-preemptive
  - No-preemptive
    - Once a process gets the CPU, it doesn’t release it until the process terminates or switches to waiting
  - Preemptive
    - Using a timer, the OS can preempt the CPU even if the thread doesn’t relinquish it voluntarily
    - Of course, re-assignment involves overhead
And now some example policies

- Remember these are example *policies* – in practice, any real system uses some hybrid approach
- We will compare them based on average waiting time
  - minimize *average waiting time* (avg time from submission of request to start of execution)
- One of many possible metrics (sometimes in conflict)
  - maximize *CPU utilization*
  - maximize *throughput* (requests completed / s)
  - minimize *average response time* (avg time from submission of request to completion of response)
  - minimize *energy* (joules per instruction) subject to some constraint (e.g., frames/second)
First-Come First-Served scheduling

- First-Come First-Served (FCFS)
  - Simplest, easy to implement, non-preemptive

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

Change order of arrival ....

Average waiting time:
\[
(0 + 24 + 27)/3 = 17
\]

Average waiting time = 3
FCFS issues

- Potentially bad average response time
  - 1 CPU-bound process (burst of 1 sec.)
  - Many I/O-bound ones (needing to read 1000 records)
  - Each I/O-bound process reads one block per sec!

- May lead to poor utilization of resources
  - Poor overlap of CPU and I/O
Shortest Job/Remaining Time First sched.

- **Shortest-Job First**
  - Assumption – total time needed (or length of next CPU burst) is known
  - Provably optimal with respect to average response time
    - First job finishes at time $a$
    - Second job at time $a + b$
    - ...

  Mean turnaround time
  
  
  $$(4a + 3b + 2c + d)/4$$

- **A preemptive variation – Shortest Remaining Time First (or SRPT)**

<table>
<thead>
<tr>
<th>Job #</th>
<th>Finish time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a$</td>
</tr>
<tr>
<td>2</td>
<td>$b$</td>
</tr>
<tr>
<td>3</td>
<td>$c$</td>
</tr>
<tr>
<td>4</td>
<td>$d$</td>
</tr>
</tbody>
</table>

Biggest contributor

Preemptive or not?
SJF and SRTF

- **Shortest Job First – Non-preemptive**

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

  \[ \text{avg. waiting time} = \frac{(0 + 6 + 3 + 7)}{4} = 4 \]

- **Shortest Remaining (Processing) Time First – Preemptive**

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0.0</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>4.0</td>
<td>5</td>
</tr>
<tr>
<td>P3</td>
<td>11.0</td>
<td>7</td>
</tr>
<tr>
<td>P4</td>
<td>16.0</td>
<td>16</td>
</tr>
</tbody>
</table>

  \[ \text{avg. waiting time} = \frac{(9 + 1 + 0 +2)}{4} = 3 \]

- **Great, but what about**
  - Starvation? Burst time for a process?
Determining length of next CPU burst

- Can only *estimate* length
- Can be done using length of previous CPU bursts and exponential averaging

- $t_n = \text{actual length of } n^{th} \text{ CPU burst}$
- $\tau_{n+1} = \text{predicted value for the next CPU burst}$
- $\alpha, 0 \leq \alpha \leq 1$
- Define:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$ 

What's $\alpha$?
Scheduling the server-side of P2P systems

- P2P users’ response is dominated by download
  - >80% download requests in Kazaa are rejected due to capacity saturation at server peers
  - >50% of all requests for large objects (>100MB) take more than one day & ~20% take over one week to complete

- Most implementations use FCFS or PS

- Apply SRPT!
  Work from Northwestern

Mean response time of object download as a function of system load.
Priority scheduling

- SJF is a special case of priority-based scheduling
  - Priority = reverse of predicted next CPU burst
- Pick process with highest priority (lowest number)

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

avg. waiting time = \((6 + 0 + 16 + 18 + 1)/5\) = 8.2
Priority scheduling issues

- And how do you assign priorities?
- Starvation
  - With an endless supply of high priority jobs, low priority processes may never execute
- Solution
  - Increases priority with age, i.e. accumulated waiting time
  - Decrease priority as a function of accumulated processing time
  - Assigned maximum quantum
Round-robin scheduling

- Simple, fair, easy to implement, & widely-used
- Each process gets a fix *quantum* or *time slice*
- When quantum expires, if running preempt CPU
- With *n* processes & quantum *q*, each one gets 1/*n* of the CPU time, no-one waits more than (n-1) *q*

$q = 4$

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

avg. waiting time = (6 + 4 +7)/3 = 5.66

*Preemptive or not?*
Quantum & Turnaround time

- Length of quantum
  - Too short – low CPU efficiency (why?)
  - Too long – low response time (really long, what do you get?)
  - Commonly ~ 50-100 msec.

![Graph showing average turnaround time vs. time quantum](image)

<table>
<thead>
<tr>
<th>process</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1)</td>
<td>6</td>
</tr>
<tr>
<td>(P_2)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

Largest quantumsa don’t imply shortest turnaround times
Combining algorithms

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

---

![Diagram of multilevel queue](image.png)
Multiple (feedback) queues ...

- Multiple queues, allow processes to move between queues
- Example CTSS – Idea: separate processes based on CPU bursts
  - IBM 7094 had space for 1 process in memory (switch = swap)
  - Goals: low context switching cost & good response time
  - Priority classes: class $i$ gets $2^i$ quantas
  - Scheduler executes first all processes in queue 0; if empty, all in queue 1, ...
Multiple (feedback) queues

- **CTSS**
  - If process uses all its quanta → move to next lower queue (leave I/O-bound & interact. processes in high-priority queue)
  - E.g. a process that needs 100 quanta runs for 1, 2, 4, 8, 16, 32, 37 so 7 swaps instead of 7 with plain RR

  - 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
Thread scheduling

- Now add threads – user or kernel level?
- User-level (process-contention scope)
  - Context switch is cheaper
  - You can have an application-specific scheduler at user level
  - Kernel doesn’t know of your threads
- Kernel-level (system-contention scope)
  - Any scheduling of threads is possible (since the kernel knows of all)
  - Switching threads inside same process is cheaper than switching processes
Lottery scheduling

- Simple & predictable
- Basic idea
  - Each process gets lottery tickets for resources (CPU time)
  - Scheduling – lottery, i.e. randomly pick a ticket
  - Priority – more tickets means higher chance
- Interesting features
  - Tickets can be used to insulate resource management policies of independent modules
  - Tickets transfer – if you are blocked on someone else, give them your tickets
  - Ticket inflation rather than direct transfer
  - Ticket currencies to isolate trust boundaries
  - Compensation tickets to ensure everybody gets its share of the CPU even if it blocks before its quantum
Multiple-processor scheduling

- Scheduling more complex w/ multiple CPUs
- Asymmetric/symmetric (SMP) multiprocessing
  - Supported by most OSs (common or independent ready queues)
- Processor affinity – benefits of past history in a processor
- Load balancing – keep workload evenly distributed
  - Push migration – specific task pushes processes for balance
  - Pull migration – idle processor asks for/pulls work
- Symmetric multithreading (hyperthreading or SMT)
  - Multiple logical processors on a physical one
  - Each w/ own architecture state, supported by hardware
  - Shouldn’t require OS to know about it (but could benefit from)
Policy vs. mechanism

- Separate what is done from how it is done
  - Think of parent process with multiple children
  - Parent process may know the 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

- Process synchronization
  - Race condition & critical regions
  - Software and hardware solutions
  - Review of classical synchronization problems
  - ...

- What really happened on Mars?
  http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html
Some other algorithms

- **Guaranteed sched.** - e.g. proportional to # processes
  - Priority = amount used / amount promised
  - Lower ratio → higher priority

- **Fair-Share scheduling**
  - Schedule aware of ownership
  - Owners get a % of CPU, processes are picked to enforce it

- **Real-time scheduling**
  - 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 or aperiodic
    - 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
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;
            switchuvm(p);
            p->state = RUNNING;
            swtch(&cpu->scheduler, proc->context);
            switchkvm();

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            proc = 0;
        }
    }
    release(&ptable.lock);
}
Scheduling in xv6

```c
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;
}
```

// Give up the CPU for one scheduling round.
void
yield(void)
{
    acquire(&ptable.lock);
    proc->state = RUNNING;
    sched();
    release(&ptable.lock);
}

**Convention in xv6:** a process that wants to give up the CPU must acquire the process table loc, release any other lock it is holding, update its own state and call sched.
Scheduling in xv6

# Context switch
# void swtch(struct context **old, struct context *new);
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx

    # Save old callee-save registers
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi

    # Switch stacks
    movl %esp, (%eax)
    movl %edx, %esp

    # Load new callee-save registers
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret

Loads arguments off the stack into %eax and %edx before changing stack pointer

Switch stacks

New stack has same format, so just undo;
ret has the %eip at the top

Pushes register state creating a context structure on the current stack;
%esp is save implicitly to *old;
%eip was saved by call instruction that invoked swtch and is above %ebp
Pthread scheduling API

#include <pthread.h>
#include <stdio.h>
define NUM THREADS 5

/* Each thread begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread exit(0);
}

int main(int argc, char *argv[])
{
    int i;
    pthread_t tid[NUM THREADS]; pthread_attr_t attr;

    pthread_attr_init(&attr); /* get the default attributes */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM); /* set the sched algo */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER); /* set the sched policy */

    for (i = 0; i < NUM_THREADS; i++) /* create the threads */
        pthread_create(&tid[i],&attr,runner,NULL);

    for (i = 0; i < NUM_THREADS; i++) /* now join on each thread */
        pthread_join(tid[i], NULL);
}
Multilevel scheduling

- Batch systems allow scheduling at 3 levels

These two determine the degree of multiprogramming.

(1) Admission scheduler (Job Scheduler)

(2) Memory scheduler

(3) CPU scheduler

Select a ready process and allocate the CPU to it.

Too many processes & not enough memory – swap somebody out; when there's room: which one to bring in?

Look for mix of CPU- & I/O-bound jobs or Shortest job first.

These two determine the degree of multiprogramming.
Dispatcher

- Dispatcher module gives control of CPU to process selected by short-term scheduler
  - Switching context
  - Switching to user mode
  - Jumping to proper location in user program to restart it

- Dispatch latency – time for the dispatcher to stop one process & start another running
Algorithm evaluation

• First problem: criteria to be used in selection
  – E.g. Max CPU usage, but w/ max. response time of 1 sec.

• Evaluation forms
  – Analytic evaluation - deterministic modeling:
    • Given workload & algorithm → number or formula
    • Simple & fast, but workload specific
  – Queueing models
    • Computer system described as a network of servers
    • Load characterized by distributions
    • Applicable to limited number of algorithms – complicated math & questionable assumptions
  – Simulations
    • Distribution-driven or trace-based
  – Implementation
    • Highly accurate & equally expensive