SI 400: Lecture 09 9/22/05

Questions re. Threads and HW due Today

PU Scheduling
Basic Concepts: Multiprogramming

1. Tries to maximize **CPU Utilization** =

\[
\frac{\text{Time spend when CPU is running useful code (seconds or hours)}}{\text{Time system is up (seconds or hours)}} \times 100\%.
\]

This was the main factor to optimize in the days of expensive CPUs and batch job submissions.

2. Every time the current, running thread/process (=Linux **task**) has to wait, the (short term) CPU **scheduler** picks another ready thread/process/task to run. The **dispatcher** or “task switcher” reloads the hardware with the new process/thread’s state.

Dispatching includes virtual memory “reconfiguration” (changing of which page tables map virtual addresses into physical addresses), and is typically slow. Hardware cache holds less of the new process, so instruction executions take longer for a while.
Other resources, shared amongst processes, might be scheduled: (shared) disk or network subsystems: Hardware plus device drivers. (Requests for I/O are queued.

Multiprogramming is practical because typical processes alternate their CPU and I/O usage bursts.

- **CPU bound process**: Small number of long CPU bursts.
- **I/O bound**: Many short CPU bursts with most time spent on I/O.

Histogram plots for each duration of a CPU burst (X-axis) HOW MANY CPU bursts have that duration (Y-axis). Shape of histogram curve characterizes the statistics of burst duration.
kernel versus user threads (see AOS chapter 5):

Suppose the kernel doesn’t implement more than one “process control block” for each process (= virtual memory used by one run of a program). *You* (assembly language application programmers) can write code to

1. Change which array in memory is used for the execution stack (just modify the stack pointer register—*you cannot do that in C/C++/Java!*).

2. Save and restore registers to/from either per thread stack or other per thread register save area.

3. **Wrap** blocking system calls in routines that call their non-blocking variants and then switch threads. (Such routines are called **wrappers**.)

4. Maybe install a timer signal handler and call **kill()** to request timer signals from the kernel.

You thus write (more practically, buy/use) a **User Level Thread** library.

An advantage: might be faster than kernel threads.

Disadvantage: A real blocking system call (**action** performed by one user thread) will block the whole process. (Unless you use timer signals...).
Correction: POSIX threads can be either user level or, when the kernel provides
reads, kernel level. POSIX is a system interface specification, NOT an
implementation.
When is the scheduler called? For all scheduling schemes:

1. “When a process switches from the running state to the waiting [blocked] state (for example, I/O request, or invocation of wait for the termination of one of the child processes).”

2. “When [the] process terminates.”

**Nonpreemptive or cooperative** scheduling: These are the only two circumstances...so **while(true) ;** will hang your system!

**Preemptive scheduling** adds 2 more circumstances:

1. “When a process switches from the running state to the ready state (for example, when [a timer] interrupt occurs).” (Practical cooperative systems like some Java implementations have a **yield()** system call which makes this happen to the calling process.)

2. “When a process switches from the waiting state to the ready state (for example, [after an interrupt from] completion of I/O).”

Preemptive scheduling is used in modern, general purpose PCs and other computers. Cooperative scheduling may be used in simplified OSs of embedded real time systems, or “legacy” systems.
**Important disadvantage to preemption:** Even when there is only one hardware CPU,

- Concurrent processes can interfere with each other when they access common data.
- Computed results VARY with how luck varies the order of scheduling.

**Race:** Situation where the results VARY with the ORDER of INTERLEAVING of concurrent operations.

(remember: Parallel operations might happen at the same time; concurrency means this or interleaving of operations in a time-shared system.)

OS' plan: Teach scheduling strategies first, then how to deal with races later (Chapter 7 on Process Synchronization).
Criteria for choosing a scheduling strategy:

- **CPU utilization.** [40%-90%??]
- **Throughput** = Number of processes (or other jobs, more generally) finished per unit time (say hour).
- **Turnaround Time** = How long a particular process took, from submission to completion.
- **Waiting Time** = Sum of periods of time during which a particular process is in the READY state: Ready to run but “waiting” for a chance.
- **Response Time** = How long a particular process took from submission to its first response to the user. Often it means the time between the completion of a user input action and the process’s response to it.

Optimize: improve! (*Maximize* utilizations and throughputs, *minimize* times.)

Time is Money!

Sometimes *averages* (average waiting time e.g.) are important. Other goals:

*Minimize* the *maximum* time

*Guarantee* the *maximum* time is NEVER more than a specified period (e.g. in a hard **real time** system.)
heduling “Algorithms” (I prefer “strategies”.)

**First-Come First-Served:** Nonpreemptive; manage ready processes in a First-In-First-Out (Queue, from CSI310) manner.

**Round Robin:** The Preemptive variant of FCFS.

When any process becomes ready
---
- (the event it is waiting for occurs, it voluntarily calls **yield**() or
- it is running but is preempted because its time **quantum** expires)
---
the scheduler puts it at the TAIL of the queue (where insertion takes place.)
Shortest-Job-First Scheduling

The scheduling is done by examining the length of the next CPU-burst of a process, rather than its total length.” The ready process with shortest such length is scheduled (ties are broken by FCFS).

This is impossible to implement in general! “The real difficulty with the SJF algorithm is knowing the length of the next CPU request.”

Theorem (of mathematics): Given a set of processes with given (i.e., known) burst times that can each be started at any time, the SJF schedule has the minimum average waiting time among all choices of schedules.

We will demonstrate a proof of this theorem for the simpler, special case of non-preemptive scheduling. But first....
Three practical applications of the Shortest-Job-First strategy:

1. Long term scheduling where each user is asked to specify a time bound for each job. Jobs with smaller estimated times are run sooner...but, if you *underestimate* the time of your job, your job is terminated.

2. The scheduler uses records (gathered and provided by other kernel subsystems for timing and accounting) to calculate an *estimate* for the length of the next CPU burst based on *measured burst lengths of previous periods the process had been run*. The estimated burst length is used for the input to the SJF scheduler.

3. We can gather records of the actual CPU times used by jobs together with actual behavior of some scheduler. We can then, *retrospectively* calculate the SJF schedule’s average waiting time. The SJF’s avg. waiting time is known to be *ideal* from the above theorem. We can then compare the actual avg. waiting time with the ideal to *evaluate* “how good” the actual scheduler really is.
For each job $X$, the

**Turnaround time = Waiting time + Running time**

so, each job’s running time is *independent* of the scheduler’s decisions.

Therefore, for a set of jobs $A$, $B$, $C$, and $D$ with running times $a$, $b$, $c$, $d$ respectively, assuming they could all start at any time, the total sum of waiting times is minimized by the schedule that minimizes the total sum of turnaround times.

The average waiting time and average turnaround times are just the corresponding totals divided by 4, the number of jobs. So, we will prove the theorem by showing the SJF schedule minimizes the total turnaround time among all non-preemptive schedules.
Here is the Gantt chart for the schedule of the jobs run in the order $A$, $B$, $C$, $D$:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a$</td>
<td>$a+b$</td>
<td>$a+b+c$</td>
<td>$a+b+c+d$</td>
</tr>
</tbody>
</table>

Here are the jobs’ turnaround times for this schedule and their total:

<table>
<thead>
<tr>
<th>Job</th>
<th>Job Turnaround time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$a$</td>
</tr>
<tr>
<td>$B$</td>
<td>$a + b$</td>
</tr>
<tr>
<td>$C$</td>
<td>$a + b + c$</td>
</tr>
<tr>
<td>$D$</td>
<td>$a + b + c + d$</td>
</tr>
<tr>
<td>total</td>
<td>$4a + 3b + 2c + d$</td>
</tr>
<tr>
<td>Job</td>
<td>Job Turnaround time</td>
</tr>
<tr>
<td>-----</td>
<td>---------------------</td>
</tr>
<tr>
<td>$A$</td>
<td>$a$</td>
</tr>
<tr>
<td>$B$</td>
<td>$a + b$</td>
</tr>
<tr>
<td>$C$</td>
<td>$a + b + c$</td>
</tr>
<tr>
<td>$D$</td>
<td>$a + b + c + d$</td>
</tr>
<tr>
<td>total</td>
<td>$4a + 3b + 2c + d$</td>
</tr>
</tbody>
</table>

The running time $a$ of the first job scheduled $A$ is multiplied by 4, the total number of jobs, in the total sum of turnaround times. The general pattern is that the later any job is scheduled, the smaller is the multiplier on its turnaround time in the formula for the total of turnaround times.

This pattern indicates that to choose a sequence for the jobs which minimizes the total of turnaround times, we should schedule the jobs with larger run times later jobs with smaller run times. Why?? The later the job is scheduled, the smaller is the multiplier on the job’s run time in this formula.
We can therefore conclude (at least when jobs are not preempted) that the schedule that minimizes the average waiting time, being the one that minimizes the sum of the turnaround times, is the schedule where each job is completed before the completion of any job longer than it.

**Theorem:** Given a set of processes with given (i.e., known) burst times that can each be started at any time, the Shortest Job First schedule has the minimum average waiting time among all choices of schedules.

We demonstrated a proof of this theorem for the simpler, special case of non-preemptive scheduling.