Operating Systems
Lecture 6

Process life cycle
CPU Scheduling
Announcements

• HW1 is out
  □ A lot of “cool” problems
  □ Goal: deeper understanding of all concepts covered so far

• Note on academic integrity
  □ I take it VERY seriously
  □ If any indication of cheating
    □ Both the copy-er, and the copy-ee are responsible
    □ I reserve the right to invite you to an oral test

Homework may feel hard: start early!
Context for today’s lecture

- A lot of “algorithms”
  - Scheduling is an algorithmic problem
  - With some interesting OS bits …

- Focus on understanding tradeoffs
  - No algorithm will be perfect for every scenario
  - If you understand tradeoffs
    - You’ll be able to choose the “right” one

I don’t expect you to learn these by heart
Recall: Types of Interrupts

**Exceptions**
- process missteps (e.g. division by zero)
- attempt to perform a privileged instruction
  - sometime on purpose! (breakpoints)
- synchronous/non-maskable

**Interrupts**
- HW device requires OS service
  - timer, I/O device, interprocessor
- asynchronous/maskable

**System calls/traps**
- user program requests
- OS service
- synchronous/non-maskable
Recall: The Narrow Waist

System call interface

Portable OS Library

Portable OS Kernel

x86 ARM PowerPC

10Mbps/100Mbps/1Gbps Ethernet

1802.11 a/b/g/n SCSI

Graphics accelerators LCD Screens

Web Servers

Word Processing

Email

Web Browsers

Databases

Compilers

Email

Word Processing

Web Browsers

Databases

Compilers

Email

Word Processing

Web Browsers

Databases

Compilers
Recall: Executing a System Call

Process:
- Calls system call function in library
- Places arguments in registers and/or pushes them onto user stack
- Places syscall type in a dedicated register
- Executes `syscall` machine instruction

Kernel
- Executes `syscall` interrupt handler
- Places result in dedicated register
- Executes `RETURN_FROM_INTERRUPT`

Process:
- Executes `RETURN_FROM_FUNCTION`
Signals (Virtualized Interrupts)
## Signals (Virtualized Interrupts)

Asynchronous notifications in user space

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt (e.g., CTRL-C from keyboard)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
<tr>
<td>20</td>
<td>SIGSTP</td>
<td>Stop until SIGCONT</td>
<td>Stop signal from terminal (e.g., CTRL-Z from keyboard)</td>
</tr>
</tbody>
</table>
Receiving a Signal

Each signal prompts one of these default actions
- terminate the process
- ignore the signal
- terminate the process and dump core
- stop the process
- continue process if stopped

Signal can be caught by executing a user-level function called signal handler
- similar to exception handler invoked in response to an asynchronous interrupt
Context switch
How to Yield/Wait?

Must switch from executing the current process to executing some other READY process

- **Current** process: RUNNING → READY
- **Next** process: READY → RUNNING

1. Save kernel registers of **Current** on its interrupt stack
2. Save kernel Stack pointer of **Current** in its PCB
3. Restore kernel Stack pointer of **Next** from its PCB
4. Restore kernel registers of **Next** from its interrupt stack
Three Flavors of Context Switching

**Interrupt:** from user to kernel space
- on system call, exception, or interrupt
- Stack switch: $P_x$ user stack $\rightarrow P_x$ interrupt stack

**Yield:** between two processes, inside kernel
- from one PCB/interrupt stack to another
- Stack switch: $P_x$ interrupt stack $\rightarrow P_y$ interrupt stack

**Return from interrupt:** from kernel to user space
- with the homonymous instruction
- Stack switch: $P_x$ interrupt stack $\rightarrow P_x$ user stack
We are now ready to understand the life cycle of a Process
Process Life Cycle

- Init
- Ready
- Running
- Waiting
- Zombie
Process Life Cycle

- Init
- Ready
- Running
- Waiting
- Zombie

PCB: being created
Registers: uninitialized
Process Life Cycle

Init

Admitted to the Ready queue

Ready

Running

Zombie

PCB: being created
Registers: uninitialized

Waiting
Process Life Cycle

Init → Admitted to the Ready queue → Ready → Running → Zombie

PCB: on the Ready queue
Registers: pushed by kernel code onto interrupt stack
Process Life Cycle

Init → Admitted to the Ready queue → Ready → Dispatch → Running → Zombie

PCB: currently executing
Registers: popped from interrupt stack into CPU
Process Life Cycle

- **Init**: Admitted to the Ready queue
- **Ready**: Dispatch
- **Running**: Yield
- **Waiting**: PCB: on Ready queue
- **Zombie**: Registers: pushed onto interrupt stack (SP saved in PCB)
Process Life Cycle

- **Init**: Admitted to the Ready queue
- **Ready**: Dispatch
- **Running**: Yield
- **Waiting**: currently executing
- **Zombie**: 

**PCB**: currently executing

**Registers**: SP restored from PCB; others restored from stack
Process Life Cycle

- **Init**: Admitted to the Ready queue
- **Ready**: Dispatch
- **Running**: Yield
- **Waiting**: blocking call (e.g., read(), wait())
- **Zombie**: (No transition shown)

**PCB**: on specific waiting queue (I/O device, lock, etc.)

**Registers**: on interrupt stack
Process Life Cycle

- **Init**
  - Admitted to the Ready queue

- **Ready**
  - PCB: on Ready queue
  - Registers: on interrupt stack
  - Blocking call completion
  - Dispatch

- **Running**
  - Blocking call e.g., read(), wait()

- **Waiting**
  - Yield

- **Zombie**
Process Life Cycle

- **Init**: Admitted to the Ready queue
- **Ready**: Dispatch, Yield, blocking call completion
- **Running**: Dispatch, blocking call e.g., read(), wait()
- **Waiting**: Yield, blocking call completion
- **Zombie**: Admitted to the Ready queue

**PCB**: currently executing
**Registers**: restored from PCB (SP) and interrupt stack into CPU
Process Life Cycle

- **Init**: Admitted to the Ready queue
- **Ready**: Dispatch
- **Running**: Yield
- **Waiting**: Blocking call completion
- **Zombie**: done exit()

PCB: on Finished queue, ultimately deleted
Registers: no longer needed
CPU scheduling
Mechanism and Policy

Mechanism
- enables a functionality

Policy
- determines how that functionality should be used

*Mechanisms should not determine policies!*
Kernel Operation
(conceptual, simplified)

Initialize devices
Initialize “first process”
while (TRUE) {
  while device interrupts pending
    - handle device interrupts
  while system calls pending
    - handle system calls
  if run queue is non-empty
    - select a runnable process and switch to it
  otherwise
    - wait for device interrupt
}
The Problem

You are the cook at the State Street Diner

- Customers enter and place orders 24 hours a day
- Dishes take varying amounts of time to prepare

What are your goals?

- Minimize average turnaround time?
- Minimize maximum turnaround time?

Which strategy achieves your goal?
Context matters!

What if instead you are:

- Google, serving millions of users?
- the head nurse managing the waiting room of an emergency room
- a student who has to do homework in various classes, hang out with other students, eat, and (occasionally) sleep
Schedulers in the OS

- **CPU scheduler** selects next process to run from the ready queue
- **Disk scheduler** selects next read/write operation
- **Network scheduler** selects next packet to send or process
- **Page Replacement scheduler** selects page to evict
Scheduling processes

- OS keeps PCBs on different queues
  - Ready processes are on ready queue - OS chooses one to dispatch
  - Processes waiting for I/O are on appropriate device queue
  - Processes waiting on a condition are on an appropriate condition variable queue

- OS regulates PCB migration during life cycle of corresponding process
Why scheduling is challenging

- Processes are not created equal!
  - CPU-bound process: long CPU bursts
    - mp3 encoding, compilation, scientific applications
  - I/O-bound process: short CPU bursts
    - index a file system, browse small web pages

- Problem?
  - don’t know process type before running it
  - Process behavior can change over time
Job Characteristics

**Job**: A task that needs a period of CPU time
- A user request: e.g., mouse click, web request, shell command...

**Defined by:**
- Arrival time
  - When the job was first submitted
- Execution time
  - Time needed to run the task in isolation
- Deadline
  - By when the task must have completed (e.g. for videos, car brakes...)

**Metrics**

- **Response time**
  - How long between job’s arrival and first time job runs?

- **Total waiting time**
  - How much time on ready queue but not running?
  - Sum of “red” intervals below

- **Execution time**: sum of “green” intervals

- **Turnaround time**: “red” + “green”
  - Time between a job’s arrival and its completion

- **Throughput**: jobs completed/unit of time
Other Concerns

- **Fairness**: Who get the resources?
  - Equitable division of resources

- **Starvation**: How bad can it get?
  - Lack of progress by some job

- **Overhead**: How much useless work?
  - Time wasted switching between jobs

- **Predictability**: How consistent?
  - Low variance in response time for repeated requests
The Perfect Scheduler

- Minimizes response time and turnaround time for each job
- Maximizes overall throughput
- Maximizes resource utilization ("work conserving")
- Meets all deadlines
- Is fair: everyone makes progress, no one starves
- Has zero overhead

Alas, no such scheduler exists...
When does the Scheduler Run?

- **Non-preemptive**
  - job runs until it voluntarily yields the CPU
  - process blocks on an event (e.g., I/O or P(sem))
  - process explicitly yields
  - process terminates

- **Preemptive**
  - all of the above, plus timer and other interrupts
  - when processes can’t be trusted
  - incurs some context switching overhead
Context switch overhead

- Cost of saving registers
- Cost of scheduler determining which process to run next
- Cost of restoring register
- Cost of flushing caches
  - L1, L2, L3, TLB
Basic Scheduling Algorithms

- FIFO (First In First Out)
- SJF (Shortest Job First)
- EDF (Earliest Deadline First)
- Round Robin
- Shortest Remaining Time First (SRTF)
FIFO

Jobs $J_1, J_2, J_3$ with compute time 12, 3, 3

Job arrival $J_1, J_2, J_3$

Average Turnaround Time: 
$(12+15+18)/3 = 15$
FIFO

Jobs $J_1, J_2, J_3$ with compute time 12, 3, 3

- Job arrival $J_1, J_2, J_3$

Average turnaround time: \((12+15+18)/3 = 15\)

- Job arrival $J_2, J_3, J_1$

Average turnaround time: \((3+6+18)/3 = 9\)

Average turnaround time very sensitive to arrival time!
FIFO Roundup

The Good
- Simple
- Low overhead
- No starvation

The Bad
- Average turnaround time
- Very sensitive to arrival time

The Ugly
- Not responsive to interactive tasks
How to minimize average turnaround time?
SJF: Shortest Job First

Schedule jobs in order of estimated completion time
SJF: Shortest Job First

Schedule jobs in order of estimated completion time
SJF: Shortest Job First

Schedule jobs in order of estimated completion time

Average Turnaround time (att): 39/6 = 6.5

Would a different schedule produce a lower turnaround time?

Consider where \( c_i < c_j \)

\[
\text{att} = \frac{(c_j + (c_i + c_j))}{2}
\]
SJF: Shortest Job First

- Schedule jobs in order of estimated completion time

- Average Turnaround time (att): \( \frac{39}{6} = 6.5 \)

- Would a different schedule produce a lower turnaround time?

Consider

\[
\text{att} = \frac{(c_i + (c_i + c_j))}{2}
\]

where \( c_i < c_j \)

\[
\text{att} = \frac{(c_j + (c_i + c_j))}{2}
\]
SJF Roundup

The Good

Optimal average turnaround time

The Bad

Need to estimate execution time

The Ugly

Can starve long jobs
Earliest Deadline First (EDF)

- Schedule in order of earliest deadline
- If a schedule exists that meets all deadlines, then EDF will generate that schedule!
  - does not even need to know the execution times of the jobs
Earliest Deadline First (EDF)

- Schedule in order of earliest deadline
- If a schedule exists that meets all deadlines, then EDF will generate that schedule!
  - does not even need to know the execution times of the jobs
EDF Roundup

**The Good**
- Meets deadlines if possible (but...)
- Free of starvation

**The Bad**
- Does not optimize other metrics

**The Ugly**
- Cannot decide when to run jobs without deadlines
Round Robin

- Each process is allowed to run for a **quantum**
- Context is switched (at the latest) at the end of the quantum — **preemption!**
- Next job to run is the one that hasn’t run for the longest amount of time

- What is a good quantum size?
  - Too long, and it morphs into FIFO
  - Too short, and much time lost context switching
  - Typical quantum: about 100X cost of context switch (~100ms vs. << 1ms)
Round Robin Roundup

The Good
- No starvation
- Can reduce response time

The Bad
- Overhead of context switching
- Mix of I/O and CPU bound

The Ugly
- Particularly bad average turnaround for simultaneous, equal length jobs
SJF

$J_1$ arrives at time 0; $J_2, J_3$ arrive at time 10

Average Turnaround Time:
$\frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33$
 SJF + Preemption

$J_1$ arrives at time 0; $J_2, J_3$ arrive at time 10

With a preemptive scheduler — SRTF

At end of each quantum, scheduler selects job with the least remaining time to run next

Average Turnaround Time:

$$\text{Average Turnaround Time: } \frac{100+(110-10)+(120-10)}{3} = 103.33$$

Shortest Remaining Time First

- Often same job is selected, avoiding a context switch...
- ...but new short jobs see improved response time

Average Turnaround Time:

$$\text{Average Turnaround Time: } \frac{(120-0)+(20-10)+(30-10)}{3} = 50$$
SRTF Roundup

The Good
Good response time and turnaround time of I/O bound processes

The Bad
Bad turnaround time and response time for CPU bound processes
Need estimate of execution for each job

The Ugly
Starvation
Priority Scheduling

Assign a number (priority) to each job and schedule jobs in priority order.

Reduces to SRTF when using as priority the estimate of the execution time.

To avoid starvation:
- change job’s priority with time (aging)
- select jobs randomly, weighted by priority
Multi-level Feedback Queue (MFQ)

- Scheduler learns characteristics of the jobs it is managing
  - Uses the past to predict the future
- Favors jobs that used little CPU...
  - ...but can adapt when the job changes its pattern of CPU usage
The Basic Structure

Queues correspond to different priority levels
- higher is better

Scheduler runs job in queue $i$ if no other job in higher queues

Each queue runs RR

Parameter:
- how many queues?

How are jobs assigned to a queue?
Moving down

- Job starts at the top level
- If it uses full quantum before giving up CPU, moves down
- Otherwise, it stays where it is

What about I/O?
- Job with frequent I/O will not finish its quantum and stay at the same level

Parameter
- quantum size for each queue
Moving Up

A job’s behavior can change
- After a CPU-bound interval, process may become I/O bound

Must allow jobs to climb up the priority ladder...
- As simple as periodically placing all jobs in the top queue, until they percolate down again

Parameter
- time before jobs are moved up
Sneeeeakyyy…

- Say that I have a job that requires a lot of CPU
  - Start at the top queue
  - If I finish my quantum, I’ll be demoted…
    - …just give up the CPU before my quantum expires!

Better accounting
  - Fix a job’s time budget at each level, no matter how it is used
Linux’s “Completely Fair Scheduler” (CFS)

- Let “Spent Execution Time” (SET) to be the amount of time that a process has been executing.
- Scheduler selects process with lowest SET.
- Let \( \Delta \) be some time (typically, 50ms or so).
- Let \( N \) be the number of processes on the run queue.
- Process runs for \( \Delta/N \) time.
  - There is a minimum value too.
- If it uses up this quantum, reinsert into the queue.
  - \( \text{SET} += \Delta/N \)
- Computing of elapsed SET can be weighed by priority value.
- Processes that move to a waiting queue, upon returning to the READY queue, have SET initialized to the minimum SET of any process on the READY queue.
CPU scheduling

- Many possible “algorithms”
  - Scheduling is an algorithmic problem
  - With some interesting OS bits ...

- Knowing tradeoffs is important
  - No algorithm will be perfect for every scenario
  - If you understand tradeoffs
    - You’ll be able to choose the “right” one

I don’t expect you to learn these by heart