Scheduling: an overview

Azza Abouzied

From Last Class: Process vs. Thread

What is a process

A bundle of things:

1. A thread of execution
2. Program counter, registers, stack (execution history)
3. Address space
4. Resources managed by the kernel
    - Pointers to system resources: e.g. file descriptors
    - Privileges & User
    - Current working directory
    - Bookkeeping stuff: resource tracking; address-space management (e.g. process id, current state, ...)

What happens on a context switch?

Context switching, the switching from one runnable task to another.

Occurs at interrupts, system calls, any preemption

Kernel sets up the environment for the process to continue working.

Is the process the right level of abstraction?

A Thread vs. a process

What do you need to take care of during a context switch between one user process and another and between two threads in a user process?

Process	Thread
Everything in the thread	Program Counter
Address space	Stack
Global variables	Registers
Open files	State
Child processes	
Pending alarms	
Registered Interrupts/Signals & handlers	
Bookkeeping stuff (pids, priveleges, ...)	

Event-driven or Multithreaded?

Not an easy debate to settle:

- 1995, **Why threads are a bad idea?** John Ousterhout (UC Berkeley, Sun Labs)

- 2003, **Why events are a bad idea?** Van Behren, Condit, Brewer (UC Berkeley)

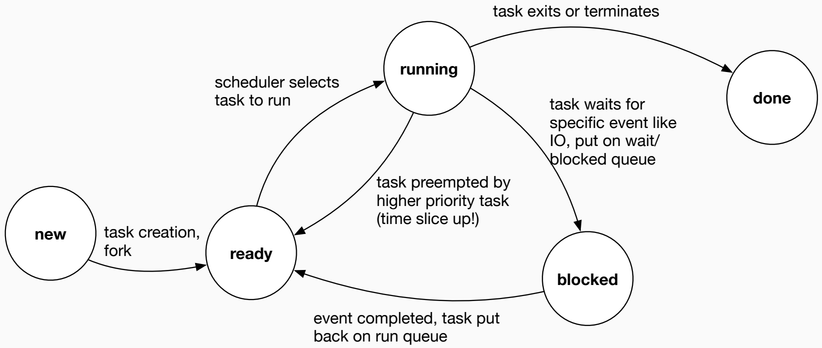
Why did we favor events in 1995?

Which ones are easier to program? *2003 paper argues event-driven is hard: stack-ripping, can't track control flow (all over the place!)*

Which ones require less resources? *ED: A stack per processor; Less synchronization but how would you scale them to SMP systems*

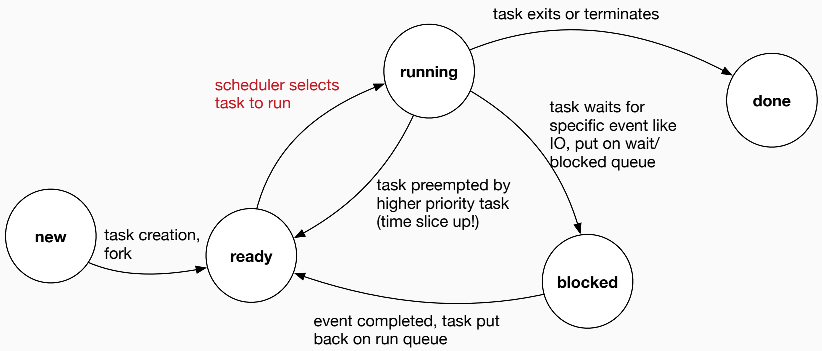
The Process Life Cycle

The Process Life Cycle



task exits or terminates
running
done
scheduler selects task to run
task waits for specific event like IO, put on wait/ blocked queue
task preempted by higher priority task (time slice up!)
new
task creation, fork
ready
blocked
event completed, task put back on run queue

Today's Lecture



task exits or terminates
running
done
scheduler selects task to run
task waits for specific event like IO, put on wait/ blocked queue
task preempted by higher priority task (time slice up!)
new
task creation, fork
ready
blocked
event completed, task put back on run queue

Multitasking

A Multitasking OS interleaves the execution of more than one process.

Multitasking allows processes to block or sleep: a process is in memory but not runnable.

Preemptive Multitasking: Schedule decides when to put a process on hold and when to run a process. *Involuntary suspension = Preemption*

Cooperative Multitasking: a process runs until it decide to give up the processor. *Voluntary suspension = Yielding*

Scheduling Policy

What are you scheduling?

IO Bound (Interactive Processes) Most of the time spent making blocking calls and thus run for very short amounts of time (usually IO)

GUIs, text editor, ...

CPU Bound (Batch Processes/Processor hogs...) Most of time spent executing code.

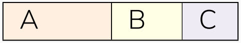
ssh-keygen, MATLAB, video-encoder, that one time your editor goes on spell/grammar check rampage.

Real time processes: SHUT THE REACTOR NOW!

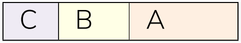
What is the right policy for each class?

Three processes: A: 100ms, B: 50ms, C: 30ms

Schedule 1:



Schedule 2:



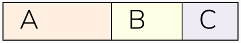
1. How many jobs are completed per second?

2. How long does it take for a job to complete on average?

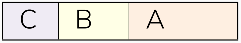
What is the right policy for each class?

Three processes: A: 100ms, B: 50ms, C: 30ms

Schedule 1: Throughput: 16.7 pps; $\mu(RT)$ = 143ms



Schedule 2: Throughput: 16.7 pps; $\mu(RT)$ = 97ms



1. How many jobs are completed per second? **Throughput**
2. How long does it take for a job to complete on average? **Response time**

Scheduler Policy

Strike a balance between two conflicting goals: Low latency (fast response time) vs. High throughput (maximum system (processor) utilization).

Why do the two goals conflict?

What does this boil down to in terms of policy for IO-bound vs. CPU-bound processes?

Other goals

1. High priority processes get to run more
2. Fairness: Low priority processes do not starve
3. Resource Utilization: Keep disks busy 100%; Keep all processors busy; ...
4. Predictability:
    - Scheduler guarantees for e.g. process *x* will run once every 10 ms.
    - Low response time variance or maximum response time thresholds, etc

Some Scheduling algorithms

The basics: FCFS

First come first served mimics the checkout line.

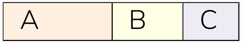
Non-preemptive policy
What about

1. Response time

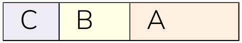
2. Fairness

3. Throughput

Schedule 1: Throughput: 16.7 pps; $\mu(RT)$ = 143ms



Schedule 2: Throughput: 16.7 pps; $\mu(RT)$ = 97ms



The basics: Round-robin

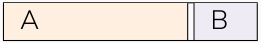
Each job gets a *time slice*, once used up, go back of the line.

What about

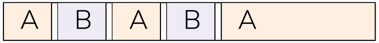
1. Throughput
2. Response time
3. Fairness

Consider two jobs: A: 100ms, B: 20ms
Time slice: 10ms

FCFS



RR



What about the time slice?

What is a good default value?

Too short:

- wasted overhead on context-switch
- CPU-bound need long time slices to keep caches hot!

Too long:

- IO-bound don't need long time slices
- Wait around for the hogs to use up their slices!

Many OS set their timeslices to be short: 10ms.

The Habachi Grill Break

Figure: Order: 3 steaks; Grill can only take two; 5 minutes on each side. What is the optimal schedule? Optimal in what sense?

The Shortest Job First (SJF)

Provably Optimal

- Get short jobs out of the way to minimize the overall number of jobs waiting.
- Longest jobs do the least damage to the wait times of others

Fairness

- Think about ‘Cutting in Line’
- What about Shortest Remaining Time to Completion First: a new short process preempts the current process

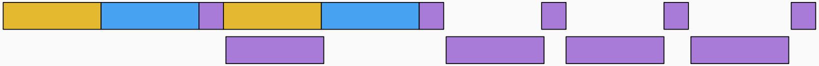
Practicality?

Hybrids are almost always better

Suppose you have

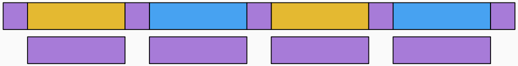
A: 80ms, B: 80ms, C: 50ms cpu; 160ms io

RR only



CPU utilization: 67%; Disk utilization: 40%

RR + FCFS queue for IO-bound tasks. Priority goes to IO task.



CPU utilization: 100%; Disk utilization: 60%

Design Exercise

Design an elevator

- m elevators: start with $m = 1$ then scale. Be careful of concurrency control.
- n floors
- R_t a request at time time t. Requests are either up or down.

What are you optimizing for? How did that influence your policy?

Expand your design: Introduce priority.

Questions?