Lecture: Scheduling

Questions:

- Difference between alloc / scheduling?
- Pre-emptive vs. non?
- Different policies: how behave?
  (+'s , -'s)

Review

Processes + low-level mechanisms
how to preempt, save/restore ctxt

create:
2 styles: VMS/VMS ⇒
Spawn: NT
create Process
Clone: fork(), exec()
e.g.
1 process ↓
2 processes ↓↓

```
int c=3
int rc=fork();
if (rc<0)
  // exit
else if (rc==0)
  exec()
else {
  wait()
}
```

⇒ flexible
cat > file
close(STDOUT_FILENO)
fd = open("file", O_WRONLY);
pipes

⇒ can be costly
huge copy before everything

⇒ Resources: what OS's manage
things that process needs/uses
(CPU, disk, memory)

Today: CPU ~ resource
scheduling

Resources: Not all alike

Preemptible
Can take resource away, use for something else
give back later
e.g. CPU

Non-preemptible
Once given away, can't really take back
(until volunteered)
e.g. Disk block, terminal

Key: Given resource/demands ⇒ determine how to manage it

Decisions about Resources

Allocation: which process gets which resources?
[admission control]

Space Sharing: Divide up resource, give some to each
when resources not easily preemptible
e.g. File space

Scheduling: how long process keeps resource
(best order)

Time sharing: more resources can be requested than available
Resource must be pre-emptible
e.g. CPU scheduling

could have both space + time sharing of CPU in parallel system, memory

Role of Dispatcher v. Role of Sched

Dispatcher: low level mechanism
must ctxt switch
{ save state in PCB for old
load state state of new }

Scheduler: higher-level policy
which to run

Performance Metrics

> Minimize cumulative waiting time: ready, not running
  init wait time → ("response time")

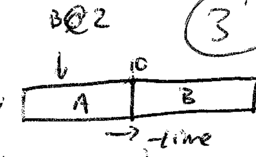
> Minimize turn-around time

> Minimize "cumulative wait" → total time spent in ready queue

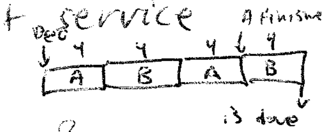
> Minimize overhead
  ↓ ctxt switches (costly?)

> maximize Fairness
  each process ⇒ same % of CPU

"response" wait time: time 'til first service



turnaround
from enter → exit



service time (exec time): 8
total/turn: 16
slowdown/stretch: 2

When does Sched make decisions?

Minimal: Non-preemptive

1) Process blocks on I/O
2) Process terminates
⇒ (remain scheduled until relinquished)

Also: Pre-emptive
\+ (I/O interrupt)
1) event completes: blocked → ready
2) timer interrupts
⇒ Can run what it wants to

Algorithms: Overview

① Process: CPU activity, I/O activity interleaved

if CPU bound: long CPU bursts
I/O bound: short CPU bursts

② might view "job" as □ □ □ □

Best Algorithm depends on workload, environment

Specialized: only certain kinds of jobs, much knowledge about them

general purpose: want to do well for all types

Projects	Book	Discussion

FCFS

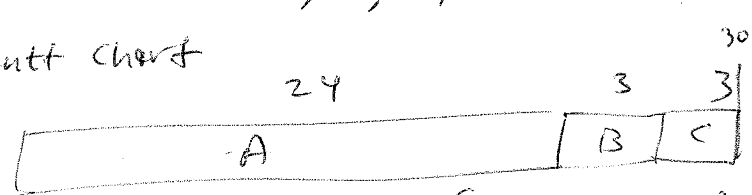
> Simplest algorithm:
> Not preemptive

> How to implement?
FIFO queue ⇒ simple

> why bad?
wait/~~service~~ turnaround time depends on order of arrival
⇒ unfair to later jobs

e.g. 3 jobs (A, B, C) arrive @ ~same time

Gantt Chart



Avg cum. wait time: $(0+24+27)/3 \Rightarrow 17$

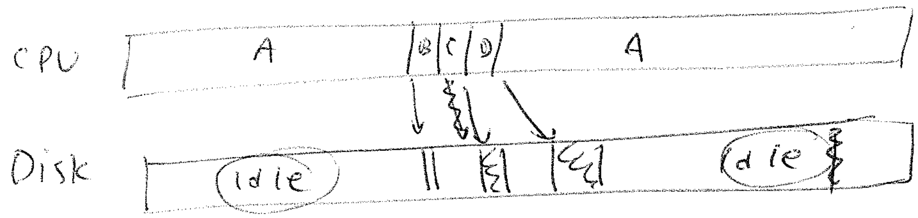
Turnaround: $\frac{24+27+30}{3} \Rightarrow 27$

10/04 What worked well here:
simple 2 job example
A: 100
B: 20
use throughout

Note: in multiprogramed world, can put job @ end of Q upon I/O

⇒ Instance of Convoy Effect
(Slow truck in front of fast cars)

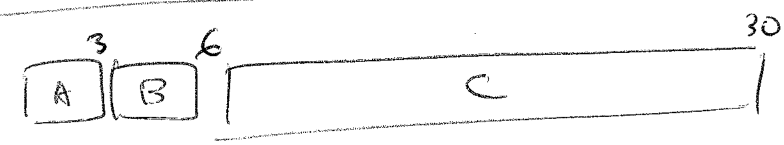
A: CPU bound
B, C, D : I/O bound



general problems
⇒ can also reduce I/O device utilization
⇒ the short job waiting time

SJF (Shortest Job First)

⇒ Should minimize wait / turnaround time



Avg wait: $(0+3+6)/3 = \boxed{3}$

Avg Turn Around: $\frac{3+6+30}{3} \Rightarrow 13$

⇒ if $T_A = T_B = T_C$ ⇒ FCFS

10/01

> Provably optimal (w/o preemption)

(supermarket example)

insight: moving short jobs up improves their standing A LOT while only hurting the long jobs A LITTLE

> But, not often practical
How to know job length?
(maybe can use past ⇒ predict future)

STtoCF (Shortest Time to Completion First)

STCF = SJF w/ preemption

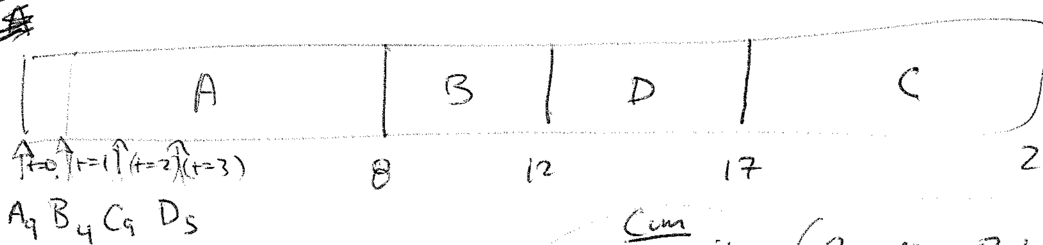
Problem: SJF is fine, IF all jobs are in system

BUT, some arrive later!

Avg Com wait: (0 + 7 + 9 + 15)/4

⇒ 7.75

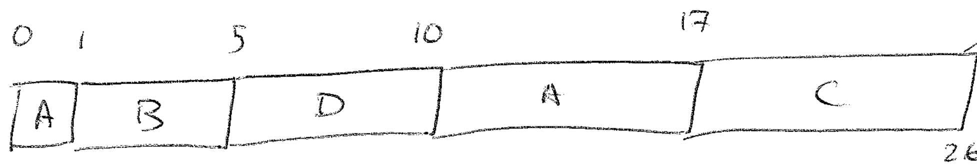
SJF



Avg Com wait: (9 + 0 + 2 + 15)/4

⇒ 6.5

w/ preemption!



10/04

What worked well:

let them figure it out

A :100 , B : 20

A has run for: { 2 s, 99 s }

Note: total time is always same (no magic here)

RR

~~Avg Com wait~~

⇒ more practical time sharing approach

⇒ run for time slice, then back of FIFO Q

⇒ preempted if still running @ end

Advantages

Fair, low average wait if job lengths vary widely

> good interactive perf (low response time)

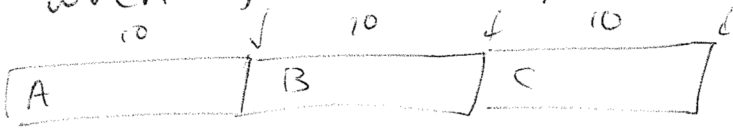
⇒ if N jobs time slice T ms

longest wait for service? (N-1) T

avg $\frac{(N-1)T}{2}$

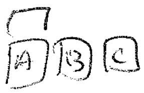
But, RR has problems

Poor when jobs nearly identical



Even FCFS is better!

Avg turnaround time = (10 + 20 + 30) / 3 = 20



= (28 + 29 + 30) / 3 = 29

But, avg wait/response time is good!

(0 + 1 + 2) / 3 ⇒ 1 ⇒ more interactive

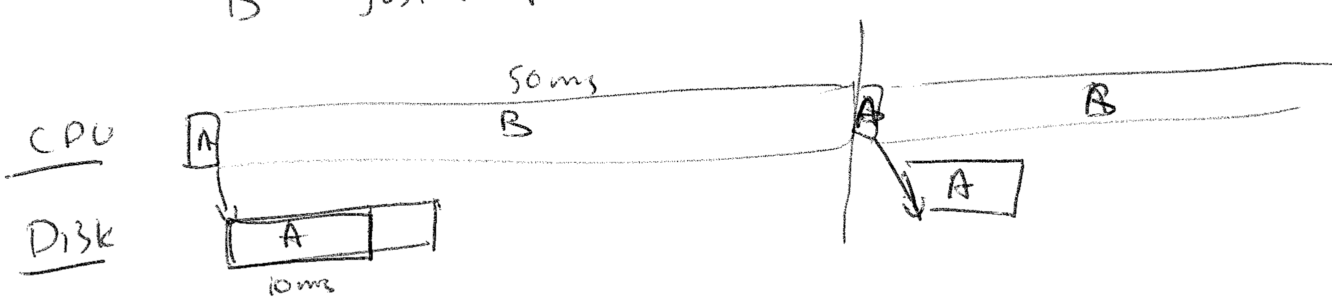
FCFS : (0 + 10 + 20) / 3 ⇒ 10

Problem #2 : Perf depends on length of time slice

if too high ⇒ FCFS

e.g. A 1 ms compute, 10 ms I/O

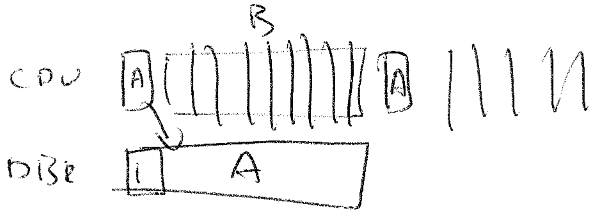
B just compute



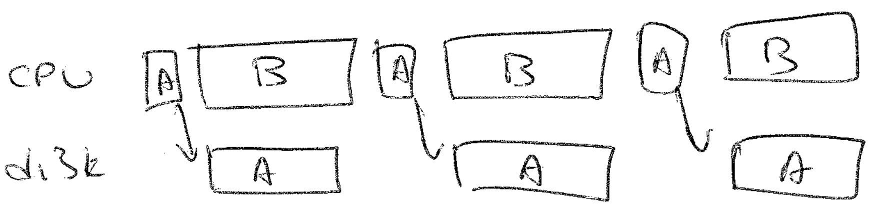
RR cont

> if time-slice too low, ctxt switch overhead

TS = 1 ms



But STCF is still better



> How to approximate? ⇒ ~~Priorities~~ Dynamic Adaptive w/ priorities

---

> Each process has priority
  run highest, RR among equal