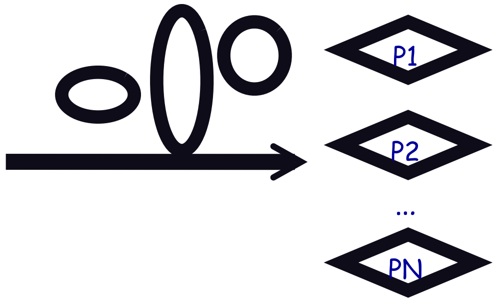
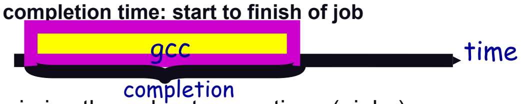
Scheduling -- Overview

- Simple: put a variety of jobs on N processors



Scheduling: what job to run?

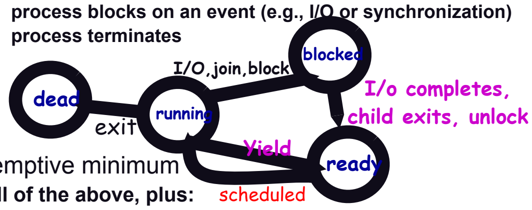
- We'll have three main goals (many others possible)
- minimize response/completion time
  - **response time = what the user sees: elapsed time to echo keystroke to editor (acceptable delay ~50-100 ms)**
  - **completion time: start to finish of job**



- Maximize throughput: operations (=jobs) per second
  - **minimize overhead (context switching)**
  - **efficient use of resources (CPU, disk, cache, …)**
- Fairness: share CPU "equitably"
  - **Tension: unfair makes system faster...**

When does scheduler make decisions?

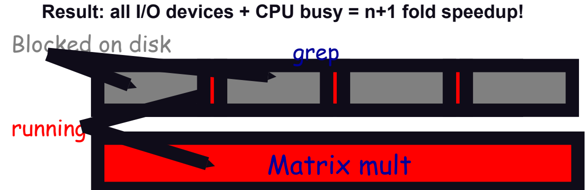
- Non preemptive minimum:
  - **process runs until voluntarily relinquish CPU:**
    - **process blocks on an event (e.g., I/O or synchronization)**
    - **process terminates**



- Preemptive minimum
  - **All of the above, plus:**
  - **Event completes: process moves from blocked to ready**
  - **Timer interrupts**
  - **Impl: process can be interrupted in favor of another**

Implicit insight: I/O device = special CPU

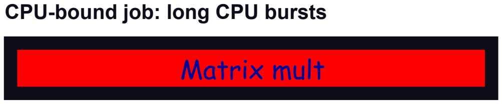
- I/O device ~ one special purpose CPU
  - **"special purpose" = disk drive can only run a disk job, tape drive a tape job, …**
- Implication: computer system with n I/O devices ~ n+1 CPU multiprocessor
  - **Result: all I/O devices + CPU busy = n+1 fold speedup!**



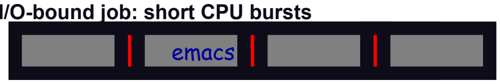
overlap them just right? ave. completion time ~ halved.

Process *model*

- Process alternates between CPU and I/O bursts
  - **CPU-bound job: long CPU bursts**



  - **I/O-bound job: short CPU bursts**



  - **I/O burst = process idle, switch to another "for free"**
  - **Problem: don't know job's type before running**
- An underlying assumption:
  - **"response time" most important for interactive jobs, which will be I/O bound**

Universal scheduling theme

- General multiplexing theme: what's "the best way" to run n processes on k nodes? (k < n)
  - **we're (probably) always going to do a bad job**
- Problem 1: mutually exclusive objectives
  - **no one best way**
  - **latency vs. throughput conflicts**
  - **speed vs. fairness**
- Problem 2: incomplete knowledge
  - **User determines what's most important. Can't mind read.**
  - **Need future knowledge to make decision and evaluate impact**
- Problem 3: real systems = mathematically intractable
  - **Scheduling very ad hoc. "Try and see"**

Scheduling

- Until now: Processes. From now on: resources
  - **Resources are things operated on by processes**
  - **e.g., CPU time, disk blocks, memory page, network bufs**
- Two ways to categorize resources:
  - **Non-preemptible: once given, can't be reused until process gives back. Locks, disk space for files, terminal.**
  - **Preemptible: once given, can be taken away and returned. Register file, CPU, memory.**
- A bit arbitrary, since you can frequently convert non-pre-emptible to preemptible:
  - **create a copy & use indirection to rename**
  - **e.g., Physical memory pages: use virtual memory to allow transparent movement of page contents to/from disk.**

How to allocate resources?

- Space sharing (horizontal):
  - **How should the resource split up?**
  - **Used for resources not easily pre-emptible**
    - **e.g., disk space, terminal**
  - **Or when not *cheaply* preemptible**
    - **e.g., divide memory up rather than swap entire thing to disk on context switch.**
- Time sharing (vertical):
  - **Given some partitioning, who gets to use a given piece (and for how long)?**
  - **Happens whenever there are more requests than can be immediately granted**
  - **implication: resource cannot be divided further (CPU, disk arm) or it's easily/cheaply pre-emptible (e.g., registers)**

Goals of "the perfect CPU scheduler"

- **Minimize latency: metrics = response time (user time scales ~50-100 ms) or job completion time**
- **Maximize throughput: Maximize jobs / time.**
- **Maximize utilization: keep I/O devices busy. Recurring theme with OS scheduling**
- **Fairness: everyone gets to make progress, no one starves**

Problem cases

- **I/O goes idle because of blindness about job types**
- **Optimization involves favoring jobs of type "A" over "B". Lots of A's? B's starve.**
- **Interactive process trapped behind others. Response time suffers for no (good?) reason.**
- **Priorities: A depends on B. A's priority > B's. B never runs.**

First come first served (FCFS or FIFO)

- **Simplest scheduling algorithm:**
  - **Run jobs in order that they arrive**
  - **Uni-programming: Run until done (non-preemptive)**
  - **Multi-programming: put job at back of queue when blocks on I/O (we'll assume this)**
  - **Advantage: dirt simple**

More FCFS

- **Disadvantage: wait time depends on arrival order**
- **unfair to later jobs (worst case: long job arrives first)**
  - **example: three jobs (times: A=100, B=1, C=2) arrive nearly simultaneously – what's the average completion time?**

And now?

FCFS Convoy effect

- **A CPU bound job will hold CPU until done, or it causes an I/O burst (rare occurrence, since the thread is CPU-bound)**
  - **long periods where no I/O requests issued, and CPU held**
  - **Result: poor I/O device utilization**
- **Example: one CPU bound job, many I/O bound**
  - **CPU bound runs (I/O devices idle)**
  - **CPU bound blocks**
  - **I/O bound job(s) run, quickly block on I/O**
  - **CPU bound runs again**
  - **I/O completes**
  - **CPU bound still runs while I/O devices idle (continues…)**
  - **Simple hack: run process whose I/O completed?**
    - **What is a potential problem?**

Round robin (RR)

- Solution to job monopolizing CPU? Interrupt it.
  - **Run job for some "time slice," when time is up, or it blocks, it moves to back of a FIFO queue**
  - **most systems do some flavor of this**
- Advantage:
  - **fair allocation of CPU across jobs**
  - **low average waiting time when job lengths vary:**

What is avg completion time?

Round Robin's Big Disadvantage

- **Varying sized jobs are good, but what about same-sized jobs? Assume 2 jobs of time=100 each:**

Avg. completion time?

How does this compare with FCFS for same two jobs?

RR Time slice tradeoffs

- Performance depends on length of the timeslice
  - **Context switching isn't a free operation.**
  - **If timeslice time is set too high (attempting to amortize context switch cost), you get FCFS. (i.e. processes will finish or block before their slice is up anyway)**
  - **If it's set too low you're spending all of your time context switching between threads.**
  - **Timeslice frequently set to ~100 milliseconds**
  - **Context switches typically cost < 1 millisecond**
  - **Moral: context switching is usually negligible (< 1% per timeslice in above example) unless you context switch too frequently and lose all productivity.**

Priority scheduling

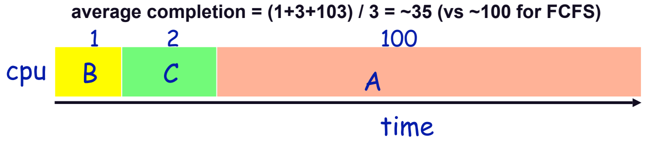
- Obvious: not all jobs equal
  - **So: rank them.**
- Each process has a priority
  - **run highest priority ready job in system round robin among processes of equal priority**
  - **Priorities can be static or dynamic (Or both: Unix)**
  - **Most systems use some variant of this**
- Common use: couple priority to job characteristic
  - **Fight starvation? Increase priority as (time last ran)**
  - **Keep I/O busy? Increase priority for jobs that often block on I/O**
- Priorities can create deadlock.
  - **Fact: high priority always runs over low priority.**
  - **So?**

Handling thread dependencies

- Priority inversion, e.g. T1 at high priority, T2 at low
  - **T2 acquires lock L.**
  - **Scenario 1: T1 tries to acquire L, fails, spins. T2 never gets to run.**
  - **Scenario 2: T1 tries to acquire L, fails, blocks. T3 enters system at medium priority. T2 never gets to run.**
- Scheduling = deciding who should make progress
  - **Obvious: a thread's importance should increase with the importance of those that depend on it.**
  - **Naïve priority schemes violate this**
- "Priority donation"
  - **Thread's priority scales w/ priority of dependent threads**

Shortest time to completion first (STCF)

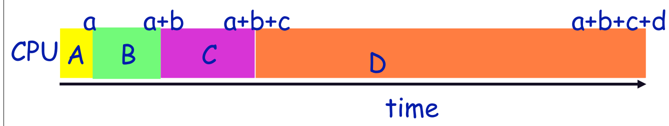
- STCF (or shortest-job-first)
  - **run whatever job has least amount of stuff to do**
  - **can be pre-emptive or non-pre-emptive**
- Example: same jobs (given jobs A, B, C)
  - **average completion = (1+3+103) / 3 = ~35 (vs ~100 for FCFS)**



- Provably optimal: moving shorter job before longer job improves waiting time of short job more than harms waiting time for long job.

STCF Optimality Intuition

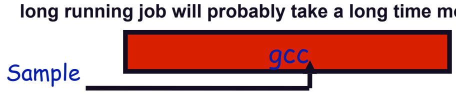
- consider 4 jobs, a, b, c, d, run in lexical order



the first (a) finishes at time a
the second (b) finishes at time a+b
the third (c) finishes at time a+b+c
the fourth (d) finishes at time a+b+c+d
therefore average completion = (4a+3b+2c+d)/4
minimizing this requires a <= b <= c <= d.

How to know job length?

- Have user tell us. If they lie, kill the job.
  - **Not so useful in practice (though used in batch queues)**
- Use the past to predict the future #1:
  - **long running job will probably take a long time more**

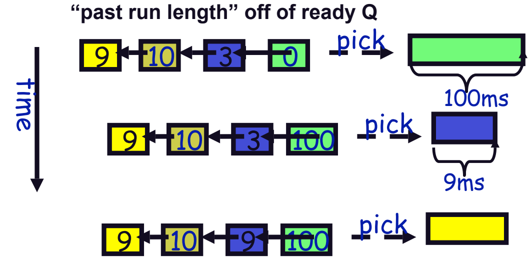


- Use the past to predict the future #2:
  - **view job as sequence of sequentially alternating CPU and I/O jobs**

  - **If previous CPU jobs in the sequence have run quickly, future ones will to ("usually")**
  - **What to do if past != future?**

Approximate STCF

- ~STCF: predict length of current CPU burst using length of previous burst
  - **record length of previous burst (0 when just created)**
  - **At scheduling event (unblock, block, exit, …) pick smallest "past run length" off of ready Q**

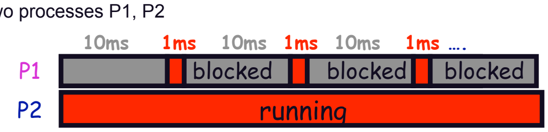


Practical STCF

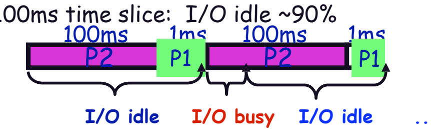
- Disk: can predict length of next "job"!
  - **Job = Request from disk.**
  - **Job length ~ cost of moving disk arm to position of the requested disk block. (Farther away = more costly.)**
- STCF for disks: shortest-seek-time-first (SSTF)
  - **Do read/write request closest to current position**
  - **Pre-emptive: if new jobs arrive that can be serviced on the way, do these too.**
- Problem:
  - **Problem?** **Solution:**
  - **Elevator algorithm: Disk arm has direction, do closest request in that direction. Sweeps from one end to other**

~STCF vs RR

- Two processes P1, P2



RR with 100ms time slice: I/O idle ~90%

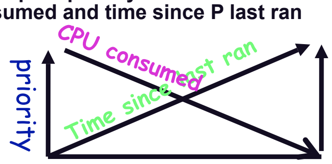


1ms time slice? RR would interrupt P1 9 times for no reason (since it would still be blocked on I/O)

- **~STCF Offers better I/O utilization**

Generalizing: priorities + history

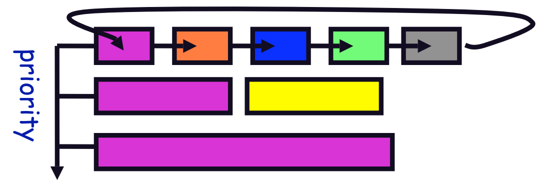
- ~STCF good core idea but doesn't have enough state
  - **The usual STCF problem: starvation (when?)**
  - **Sol'n: compute priority as a function of both CPU time P has consumed and time since P last ran**



- Multi-level feedback queue (or exponential Q)
  - **Priority scheme where adjust priorities to penalize CPU intensive programs and favor I/O intensive**
  - **Pioneered by CTSS (MIT in 1962)**
  - **Implemented by you (or should be!)**

A simple multi-level feedback queue

- Attacks both efficiency and response time problems
  - **efficiency: long time quanta = low switching overhead**
  - **response time: quickly run after becoming unblocked**
- Priority queue organization: one ready queue for each pri. level



  - **process created: give high priority and short time slice**
  - **if process uses up the time slice without blocking:**
    - **priority = priority - 1; time_slice = time_slice * 2;**

Some problems

- Can't low priority threads starve?
  - **Ad hoc: when skipped over, increase priority**
- What about when past doesn't predict future?
  - **E.g., CPU bound switches to I/O bound**
  - **Want past predictions to "age" and count less towards current view of the world.**

Summary

- FIFO:
  - **+ simple**
  - **- short jobs can get stuck behind long ones; poor I/O**
- RR:
  - **+ better for short jobs**
  - **- poor when jobs are the same length**
- STCF:
  - **+ optimal (avg. response time, avg. time-to-completion)**
  - **- hard to predict the future**
  - **- unfair**
- Multi-level feedback:
  - **+ approximate STCF**
  - **- unfair to long running jobs**