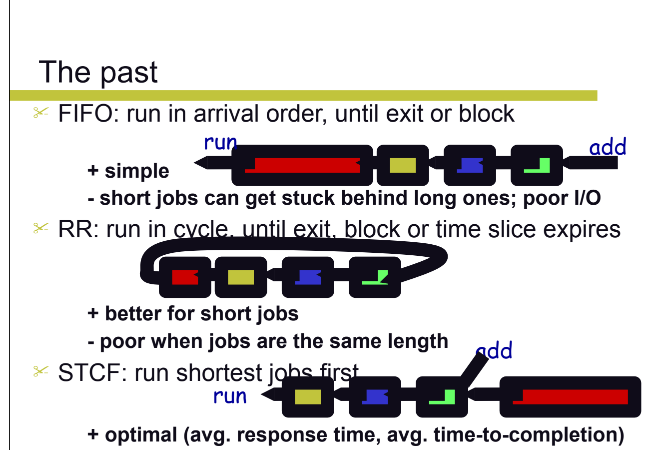
## Today's big adventure

- Multi-level feedback in the real world
  - **Unix**
- Lottery scheduling:
  - **Clever use of randomness to get simplicity**
- Retro-perspectives on scheduling
- Reading:
  - **7th Ed. Chapter 5**
  - **6th Ed. Chapter 6**

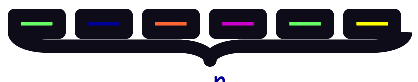
## The past




- FIFO: run in arrival order, until exit or block
  - **+ simple**
  - **- short jobs can get stuck behind long ones; poor I/O**
- RR: run in cycle, until exit, block or time slice expires
  - **+ better for short jobs**
  - **- poor when jobs are the same length**
- STCF: run shortest jobs first
  - **+ optimal (avg. response time, avg. time-to-completion)**
  - **- hard to predict the future. Unfair.**

## Understanding scheduling

- You add the nth process to system
  - **when will it run at ~1/n of speed of CPU?**
  - **< 1/n?**
  - **> 1/n?**




- Scheduling in real world
  - **Where RR used? FIFO? SJF? Hybrids?**
  - **Why so much FIFO?**
  - **When priorities?**
  - **Time Slicing? What's common cswitch overhead?**
  - **Real world scheduling not covered by RR, FIFO, STCF?**

## Multi-level Unix (SysV, BSD)

0..3
4..7
8..11
....
124..127

**Priorities go from 0..127 (0 = highest priority)**
**32 run queues, 4 priority levels each**
**Run highest priority job always (even if just ran)**
**Favor jobs that haven't run recently**

## Multi-level in real world: Unix SVR3

- Keep history of recent CPU usage for each process
  - **Give highest priority to process that has used the least CPU time "recently"**
- Every process has two fields:
  - **p_cpu field to track usage**
  - **usr_pri field to track priority**
- Every clock tick
  - **increment current job's p_cpu by 1**
- Every second recompute every job's priority and usage
  - **p_cpu = p_cpu / 2 (escape tyranny of past!)**
  - **p_priority = p_cpu / 4 + PUSER + 2 * nice**
- What happens:
  - **to interactive jobs? CPU jobs? Under high system load?**

## Multi-level in real world: BSD 4.3

- Like previous slide, but decay p_cpu using:
  - **decay = (2 * load_average) / (2 * load_average + 1)**
  - **load_average = ave size of runq over last sec**
  - **p_cpu = p_cpu * decay;**

**Why does this fix our problem?**

## Some Unix scheduling problems

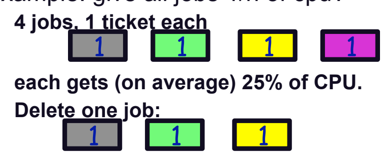
- How does the priority scheme scale with number of processes?
- How to give a process a given percentage of CPU?
- OS implementation problem:
  - **OS takes precedence over user process**
  - **user process can create lots of kernel work: e.g., many network packets come in, OS has to process. When do a read or write system call, ….**
  - **So?**

## Lottery scheduling: random simplicity

- Problem: this whole priority thing is really ad hoc.
  - **How to ensure that processes will be equally penalized under load? That system doesn't have a pathological failure mode?**
- Lottery scheduling! Dirt simple:
  - **give each process some number of tickets**
  - **each scheduling event, randomly pick ticket**
  - **run winning process**
  - **to give P n% of CPU, give it (total tickets)* n%**
- How to use?
  - **Approximate priority: low-priority, give few tickets, high-priority give many**
  - **Approximate STCF: give short jobs more tickets, long jobs fewer. Key: If job has at least 1, will not starve**

## Grace under load change

- Add or delete jobs (and their tickets):
  - **affect all proportionally**
- Example: give all jobs 1/n of cpu?
  - **4 jobs, 1 ticket each**



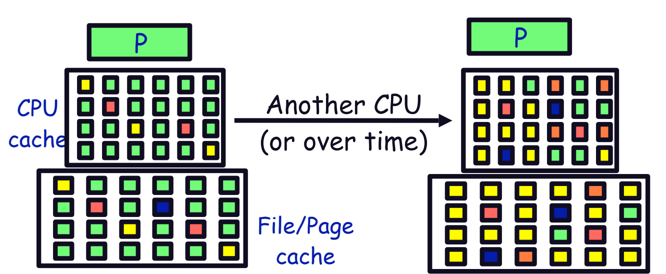

  - **each gets (on average) 25% of CPU.**
  - **Delete one job:**
  - **automatically adjusts to 33% of CPU!**
- Easy priority donation:
  - **Donate tickets to process you're waiting on.**
  - **Its CPU% scales with tickets of all waiters.**

## Changing Assumptions

- Real time: processes are not time insensitive
  - **missed deadline = incorrect behavior**
  - **soft real time: display video frame every 30th of sec**
  - **hard real time: "apply-breaks" process in your car**
- Scheduling more than one thing:
  - **memory, network bandwidth, CPU all at once**
- Distributed systems: System not contained in 1 room:
  - **How to track load in system of 1000 nodes?**
  - **Migrate jobs from one node to another? Migration cost non-trivial: must be factored into scheduling**
- So far: assumed past = one process invocation
  - **gcc behaves pretty much the same from run to run.**
  - **Research: How to exploit?**

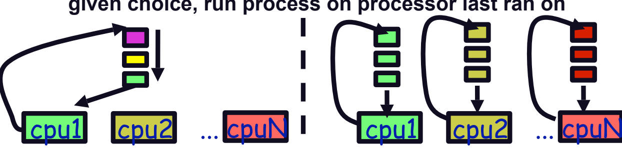
## A less simplistic view of context switching

- Brute cswitch cost:
  - **saving and restoring: registers, control block, page table, …**
- Less obvious: lose cache(s). Can give 2-10x slowdown



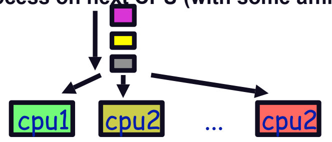

## Context switch cost aware scheduling

- Two level scheduling:
  - **if process swapped out to disk, then "context switching" very very expensive: must fault in many pages from disk**
  - **One disk access costs ~10ms. On 500Mhz machine, 10ms = 5 million cycles!**
  - **So run in core subset for "awhile", then move some between disk and memory. (How to pick subset?)**
- Multi-processor: processor affinity
  - **given choice, run process on processor last ran on**

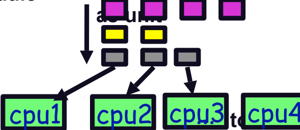


## Parallel systems: gang scheduling

- N independent processes: load-balance
  - **run process on next CPU (with some affinity)**

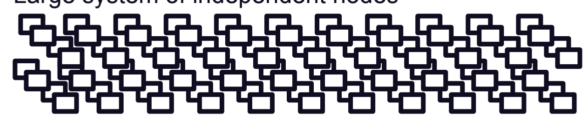



- N cooperating processes: run at same time
  - **cluster into groups, schedule**
  - **can be much faster:**
  - **Share caches**
  - **No context switching**
  - **communicate**




## Distributed system load balancing

- Large system of independent nodes

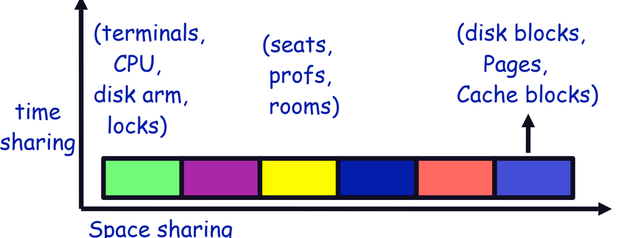


- Want: run job on lightly loaded node
  - **Querying each node too expensive**
- Instead randomly pick one
  - **(used by lots of internet servers)**
- Mitzenmacher: Then randomly pick one other one!
  - **Send job to shortest run queue**
  - **Result? Really close to optimal (w/ a few assumptions ;-)**
  - **Exponential convergence =picking 3 doesn't get you much**

## The universality of scheduling

- Used to let m requests share n resources
  - **issues same: fairness, prioritizing, optimization**
- Disk arm: which read/write request to do next?
  - Opt: close requests = faster
  - Fair: don't starve far requests
- Memory scheduling: who to take page from?
  - Opt: past=future? take from least-recently-used
  - Fair: equal share of memory space/bandwidth
- Printer: what job to print?
  - People = fairness paramount: uses FIFO rather than SJF. "admission control" to combat long jobs

## How to allocate resources?




- Space sharing (sometimes): split up. When to stop?
- Time-sharing (always): how long do you give out piece?
  - **Pre-emptable (CPU, memory) vs. non-preemptable (locks, files, terminals)**

## Postscript

- In principle, scheduling decisions can be arbitrary since the system should produce the same results in any event
  - **Good: rare that "the best" process can be calculated.**
- Unfortunately, algorithms have strong effects on system's overhead, efficiency and response time
- The best schemes are adaptive. To do absolutely best we'd have to predict the future.
  - Most current algorithms tend to give the highest priority to the processes that need the least!
  - Scheduling has gotten *increasingly* ad hoc over the years. 1960s papers very math heavy, now mostly "tweak and see"