CSL373: Lecture 7
Advanced Scheduling

Today

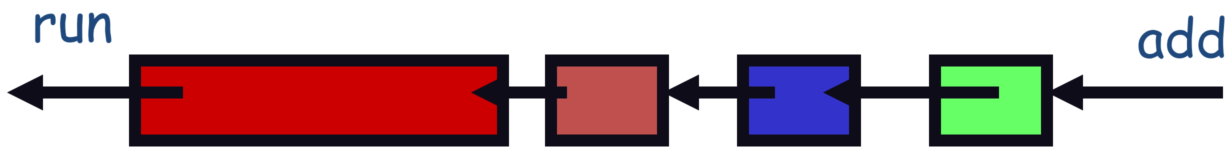
- Multi-level feedback in the real world
    - UNIX
- Lottery scheduling:

    Clever use of randomness to get simplicity
- Retro-perspectives on scheduling

- Reading: Chapter 5

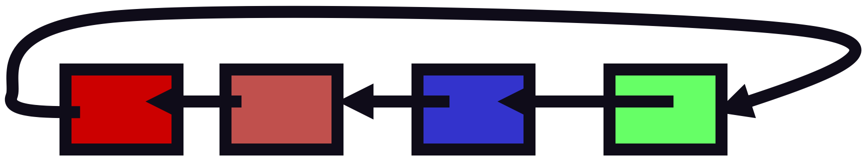
Scheduling Review

- FIFO: run in arrival order, until exit or block



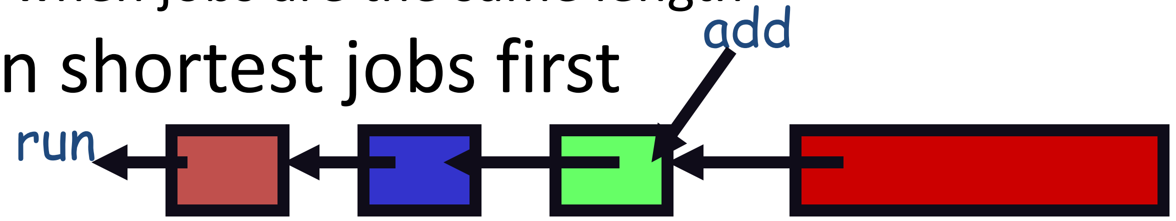
  + simple
  - Short jobs get stuck behind long ones; poor I/O

- 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. Possible starvation

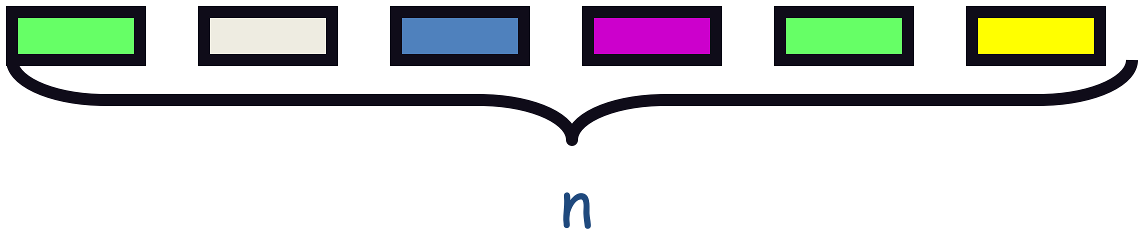
Understanding scheduling

- You add the nth process to the 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?

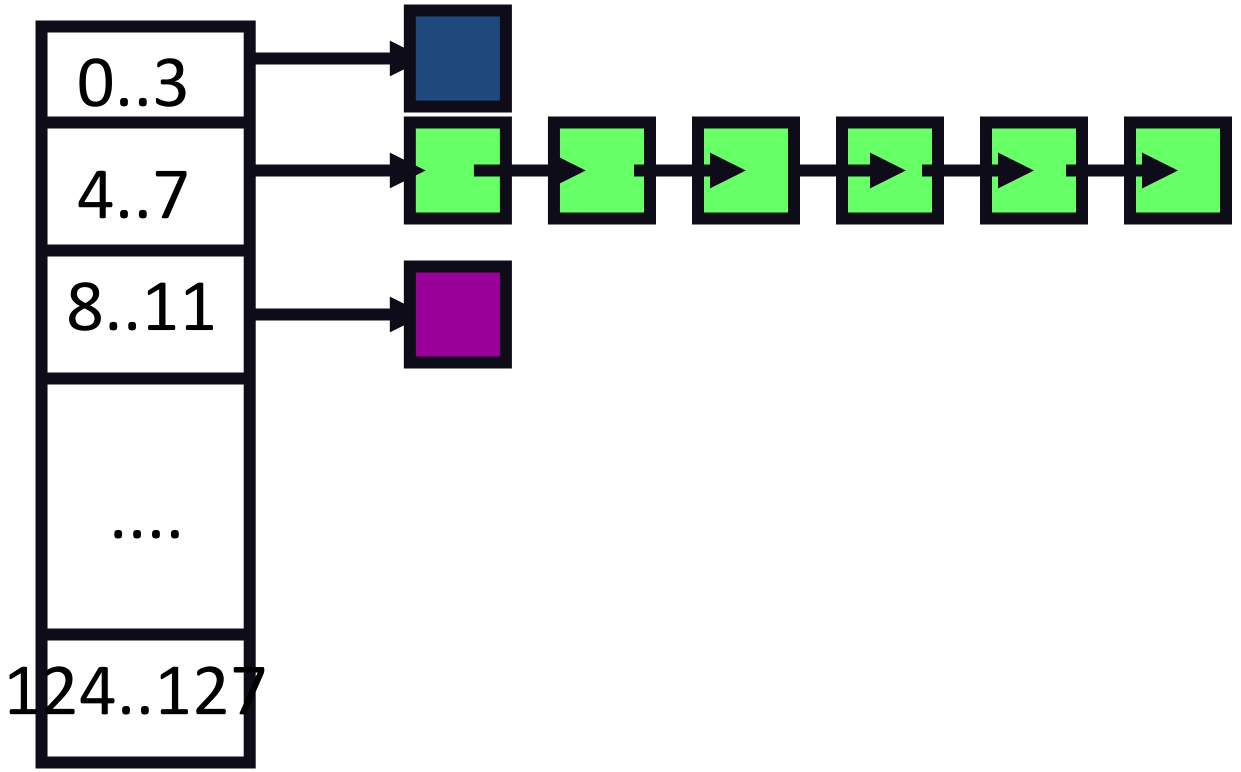
  Why so much FIFO?

  When priorities?

  Time slicing? What’s common cswitch overhead?

  Real world scheduling not covered by RR, FIFO, STCF?

Multi-level UNIX (SysV, BSD)



- 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 (lower value => higher priority)
- Every clock tick (how frequent?)

  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?

Some UNIX scheduling problems

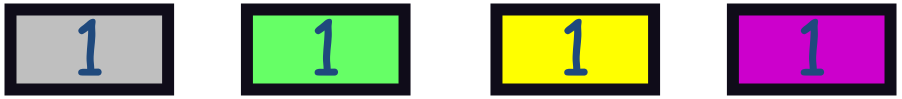
- How does the priority scheme scale with number of processes?

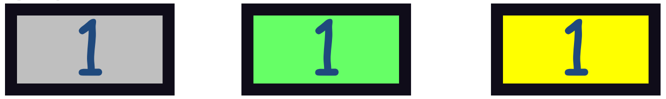
- How to give a process a given percentage of CPU?

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 bad case where most processes suffer?
- 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 on all waiters.

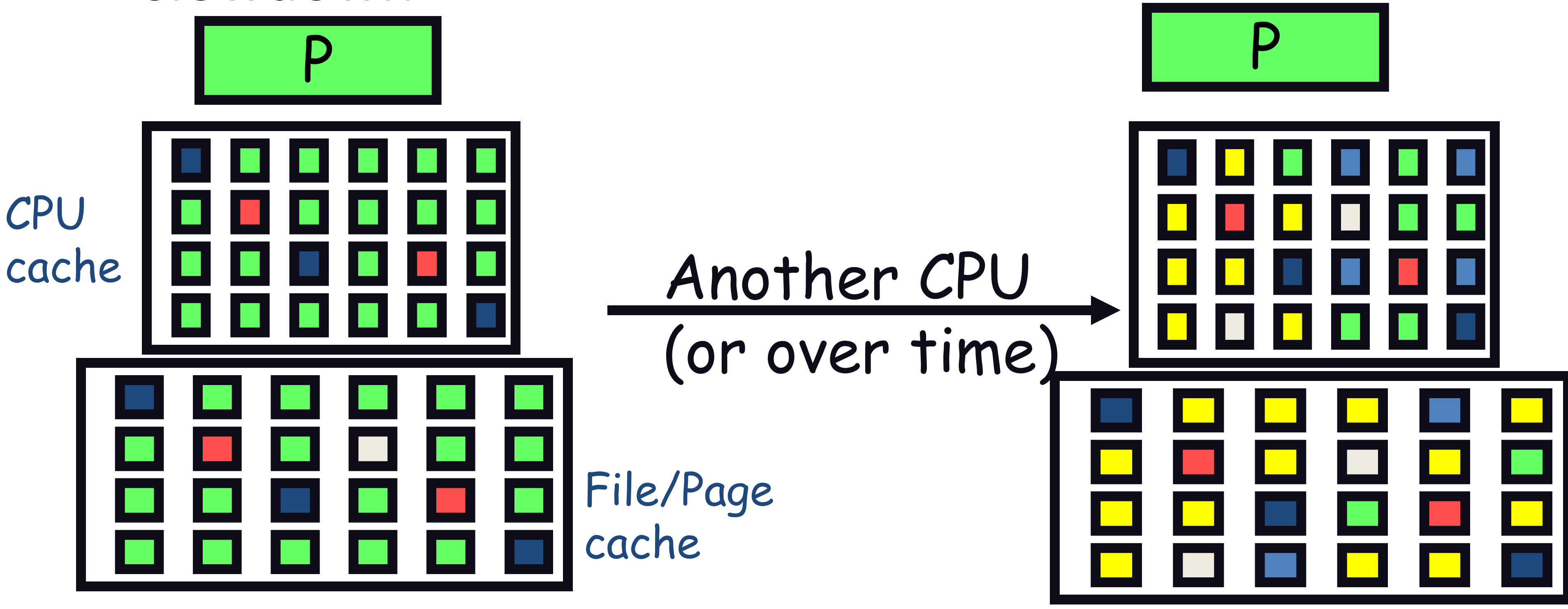
Changing Assumptions

- 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

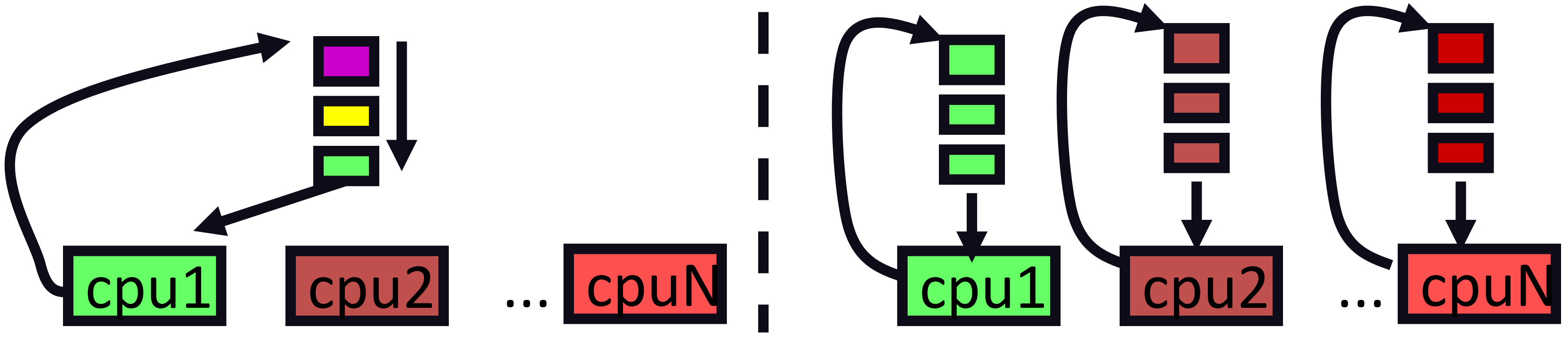
- Brute cswitch cost:

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



Context switch cost aware scheduling

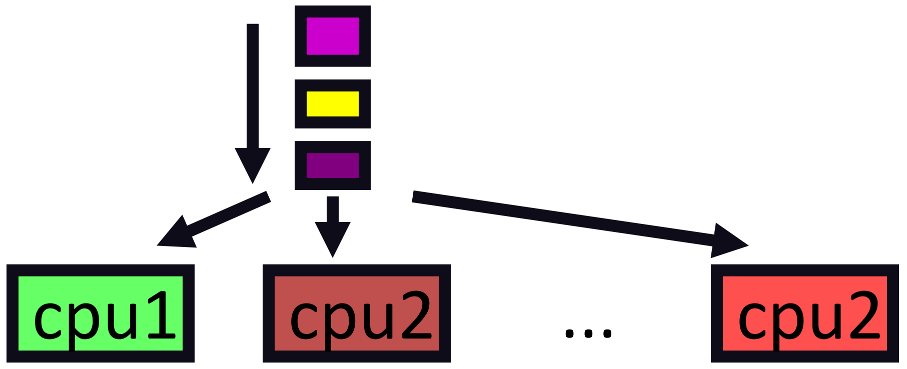
- Two level scheduling:
  - If process swapped out to disk, then "context-switching" very very expensive: must fault in many pages from disk.
  - One disk access costs ~10ms. On 500Mhz machine, 10ms = 5 million cycles!
  - So run in core subset for "a while", then move some between disk and memory. (How 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

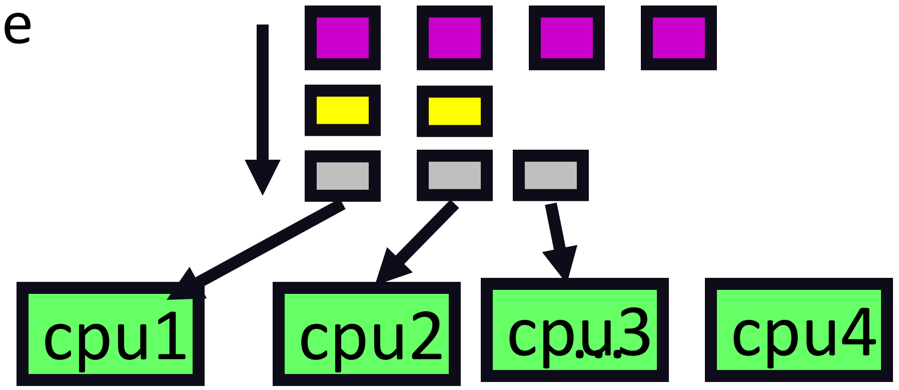
  as unit

  can be much faster

  Share caches

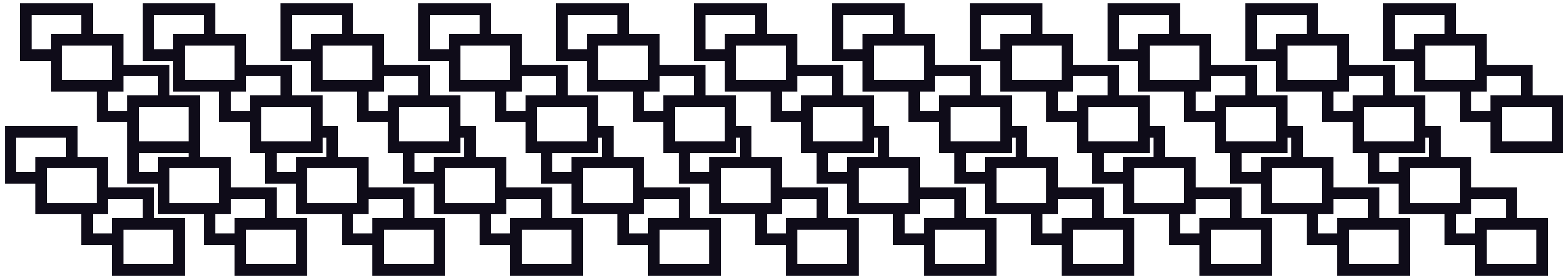
  No context switching to

  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 more!

  Send job to shortest run queue

  Result? Really close to optimal (with 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 away from least-recently-used
  - Fair: equal share of memory
- Printer: what job to print?
  - People = fairness paramount: uses FIFO rather than SJF. "admission control" to combat long jobs

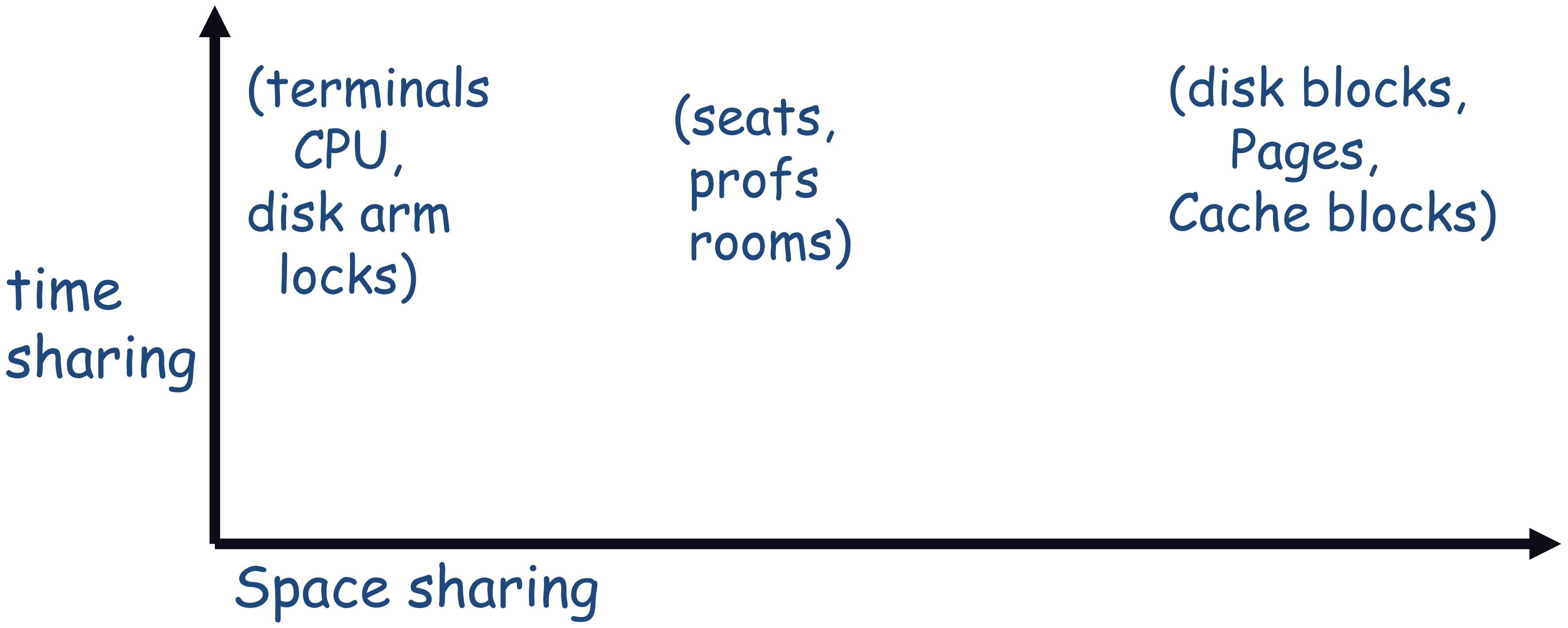
Unfair = speed

- Unfair in scheduling
  - STCF: *must* be unfair to be optimal
  - Favor I/O jobs? Penalize CPU. Other way, devices idle
  - Favor recent jobs over older ones = better cache performance
- LRU cache replacement
  - Rather than give each process 1/nth of cache, evict most useless entries.
- Traffic light:
  - Rather than RR at the car granularity (fair), coarsen, and schedule at the lane granularity. Reduces “context switch” overhead, but may make recent cars wait

Adaptation = speed

- Multilevel feedback is an “adaptive” technique
  - system feedback = better decisions at static policies
  - Why? more information = better decisions = speed
- Lock acquisition:
  - using dynamic information lets us get within 2x of optimal
- Ethernet transmission
  - Sending a message on a busy ethernet causes collision
  - Rather than wait fixed amount, backoff randomly and retry. If collide again, backoff further, repeat
- Buffer cache replacement:
  - eject least-recently-used (LRU) memory page rather than give each process 1/n of memory

How to allocate resources?


(terminals CPU, disk arm locks)
(seats, profs rooms)
(disk blocks, Pages, Cache blocks)
time sharing
Space sharing

- Space sharing (sometimes): split up. When to stop?
- Time-sharing (always): how long do you give out piece?

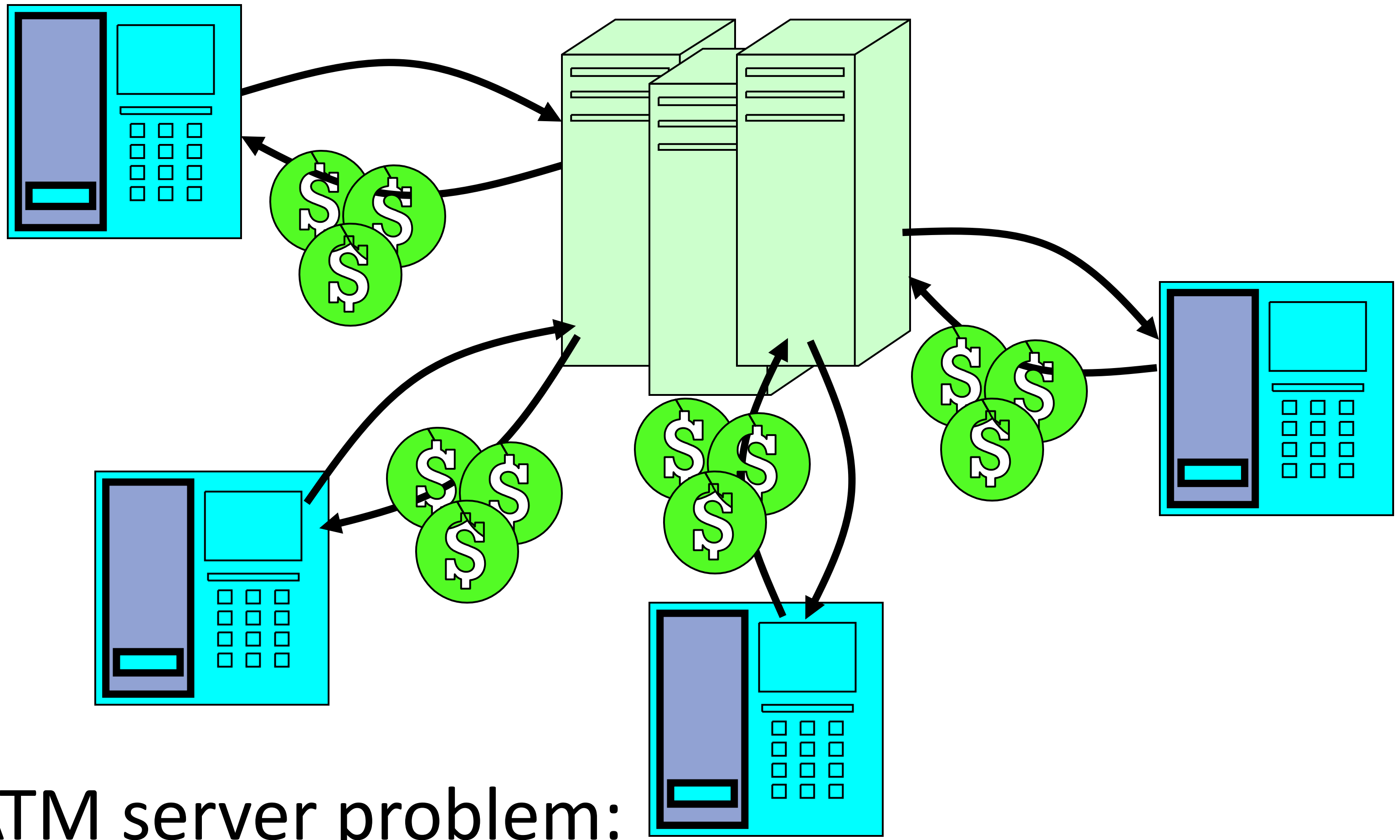
  Pre-emptible (CPU, memory) vs non-preemptible (locks, files, terminals)

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.
  - Scheduling has gotten *increasingly* ad hoc over the years. 1960s papers very math heavy, now mostly “tweak and see”

Event-Driven vs. Shared Memory
-or-
Multiprocessing vs. Multithreading

ATM Bank Server



- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don’t hand out too much money

ATM bank server example

- Suppose we wanted to implement a server process to handle requests from an ATM network:

```
BankServer() {
   while (TRUE) {
      ReceiveRequest(&op, &acctId, &amount);
      ProcessRequest(op, acctId, amount);
   }
}
ProcessRequest(op, acctId, amount) {
   if (op == deposit) Deposit(acctId, amount);
   else if …
}
Deposit(acctId, amount) {
   acct = GetAccount(acctId); /* may use disk I/O */
   acct->balance += amount;
   StoreAccount(acct); /* Involves disk I/O */
}
```

- How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-proc, or overlap comp and I/O)

Event Driven Version of ATM server

- Suppose we only had one CPU
  - Still like to overlap I/O with computation
  - Without threads, we would have to rewrite in event-driven style
- Example

```
BankServer() {
   while(TRUE) {
      event = WaitForNextEvent();
      if (event == ATMRequest)
         StartOnRequest();
      else if (event == AcctAvail)
         ContinueRequest();
      else if (event == AcctStored)
         FinishRequest();
   }
}
```

  - What if we missed a blocking I/O step?
  - What if we have to split code into hundreds of pieces which could be blocking?
  - This technique is used for graphical programming

Can Threads Make This Easier?

- Threads yield overlapped I/O and computation without “deconstructing” code into non-blocking fragments
  - One thread per request
- Requests proceeds to completion, blocking as required:

```
Deposit(acctId, amount) {
  acct = GetAccount(actId);    /* May use disk I/O */
  acct->balance += amount;
  StoreAccount(acct);             /* Involves disk I/O */
}
```

- Unfortunately, shared state is complicated too!

```
              Thread 1                              Thread 2
  load r1, acct->balance
                                     load r1, acct->balance
                                     add r1, amount2
                                     store r1, acct->balance
add r1, amount1
store r1, acct->balance
```