10/28

Scheduling

CPU scheduling
→ threads / tasks, fixed # of CPUs, which thread do we choose to run?

✡ Resource Allocation
→ limited resources, many more requests.

Examples: attu (schedule students onto limited machines)
AWS (scheduling new VMs onto physical machines)
Grocery (limited cashiers, many more customers)
Homework (one student, many homework :'( )

[Different situations require different metrics for evaluating decisions]

Metrics For CPU Scheduling

A thread can perform one to many tasks. For example, a file encryption thread:
① read file (I/O)
② encrypt file (CPU)
③ write file (I/O)

Definitions

- **Task/Job**
  - User request: e.g., mouse click, web request, shell command, …
- **Latency/response time** (Job completion time)
  - How long does a task take to complete? → time btwn when user first issues the task to when it completes
- Throughput
  - How many tasks can be done per unit of time?
- Overhead
  - How much extra work is done by the scheduler?
- Fairness
  - How equal is the performance received by different users?
- Strategy-proof
  - Can a user manipulate the system to gain more than their fair share?
- Predictability
  - How consistent is a user's performance over time?

- Ostep (textbook) refers to this as "turnaround time"
- != Ostep response time

Why care about this? interactivity!

Scheduling Mechanism

→ When does the scheduler run? How do we switch from one task to another?

starts:

kernel/main.c

```
35  // Common CPU setup code.
36  static void mpmain(void) {
37    cprintf("cpu%d: starting\n", cpunum());
38    idtinit(); // load idt register
39    // xchg(&cpu->started, 1); // tell startothers() we're up
40    scheduler(); // start running processes
41  }
```

timer interrupt:

kernel/trap.c

```
45    switch (tf->trapno) {
46    case TRAP_IRQ0 + IRQ_TIMER:
47      if (cpunum() == 0) {
48        acquire(&tickslock);
49        ticks++;
50        wakeup(&ticks);
51        release(&tickslock);
52      }
53      lapiceoi();
54      break;
```

→ acknowledge the interrupt (more can come in now)

```
104   // Force process to give up CPU on clock tick.
105   // If interrupts were on while locks held, would need to check nlock.
106   if (myproc() && myproc()->state == RUNNING &&
107       tf->trapno == TRAP_IRQ0 + IRQ_TIMER)
108     yield();
```

↳ calls sched() → switch to scheduler()

- exit

wait for events:

```
237 void sleep(void *chan, struct spinlock *lk) {
238   if (myproc() == 0)
239     panic("sleep");
240
241   if (lk == 0)
242     panic("sleep without lk");
243
244   // Must acquire ptable.lock in order to
245   // change p->state and then call sched.
246   // Once we hold ptable.lock, we can be
247   // guaranteed that we won't miss any wakeup
248   // (wakeup runs with ptable.lock locked),
249   // so it's okay to release lk.
250   if (lk != &ptable.lock) { // DOC: sleeplock0
251     acquire(&ptable.lock);  // DOC: sleeplock1
252     release(lk);
253   }
254
255   // Go to sleep.
256   myproc()->chan = chan;
257   myproc()->state = SLEEPING;
258   sched();
```

Scheduling Policies

① First in First out (FIFO)

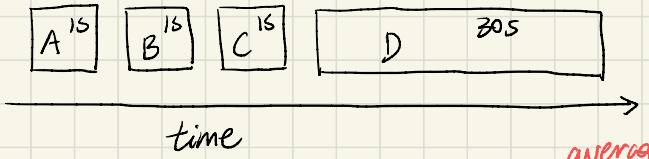
→ scheduling tasks in the order they arrive, each task runs to completion

→ wait in line in grocery store

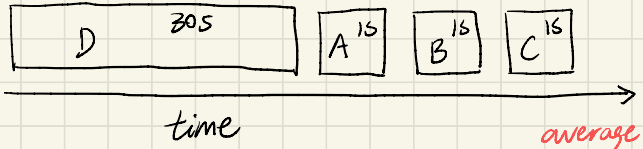
assume that jobs arrive at approximately the same time in the following order

both execution & wait time are included

$T_{response\ time} = T_{completion} - T_{arrival}$



average response time = $\frac{(1+2+3+33)}{4}$



average response time = $\frac{(30+31+32+33)}{4}$

depends on the order of arrival

Pros: simple, minimum switching btwn threads

Cons: varying response time

② Shortest Job First (SJF)

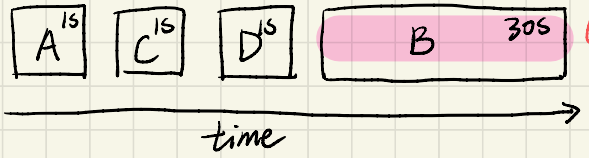
→ also called Shortest Remaining Time First (SRTF)

→ complete the short task first, if shorter task arrives, preempt the current task, switch to the shorter task

→ similar to express lanes in grocery stores

How would we know if a task is long or short?

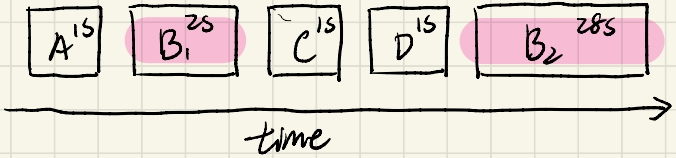
assume that jobs arrive at approximately the same time in alphabetical order



(though B arrived second, it's at the end of the queue)

if more small jobs keep arriving
B can be starved (never get a chance to run)

assume that jobs arrive at different times in alphabetical order



→ B gets preempted when a smaller task arrives

Pros: optimal average response time

(intuition: long tasks take a long time, so making it wait a little doesn't affect its response time as much)

Cons: Starvation, can result in more context switches if we keep preempting longer tasks

③ Round Robin (RR)

→ FIFO but with fixed time for each task ← time slice / time quantum

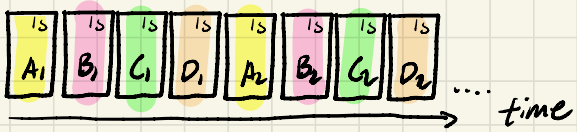
→ no starvation!

I/O bound vs. CPU bound jobs

↳ relies on I/O, doesn't accomplish more if you give it more CPU (eg. file read)

↳ relies on CPU, accomplish more if you give it more CPU (eg. encryption job)

assume that jobs arrive at approximately the same time in alphabetical order



(Subscript = # of times scheduled)

impact on average response time

[if each task takes 2 seconds to finish w/ 1s time slice]

$SJF = \frac{(2+4+6+8)}{4}$ $RR = \frac{(1+2+3+4+5+6+7+8)}{4}$

How to decide on the time quantum?

→ too large? similar to FIFO

→ too small? lots of context switch overhead

→ typically 10-100ms

assume 10ms time quantum

A: I/O bound (runs for 1ms, blocks for 5ms, runs for 2ms)

B, C: CPU bound (needs 20ms total to finish the task)

blocks before time quantum expires

A waits a long time before getting scheduled again, "fixed time"

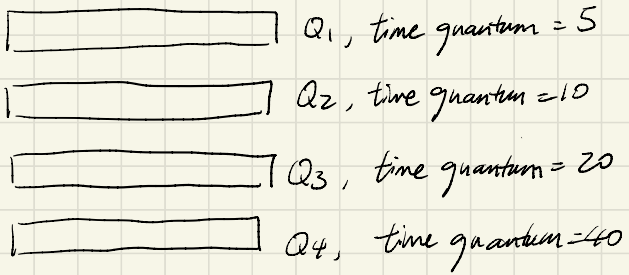
impacts I/O bound & CPU bound jobs differently

☠ bad for I/O task response time

④ Multilevel Feedback Queue (MLFQ)

→ RR but multiple queues with increasing time quantum

→ wants both good response time & no starvation



- scheduler starts with the top queue & work its way down when queue gets empty
- tasks within a queue is ran in RR fashion
- all tasks start at the top queue, if a task uses up its time quantum, it moves down a queue, otherwise stay the same or moves up a queue
- periodically move all tasks to top queue again