9. Scheduling

Schedulers can work on multiple levels:

- long-term scheduler
  - which processes should be admitted to OS?
    - look at return value of fork()
    - if fork() < 0, denied by OS
- medium-term scheduler
  - which processes reside in RAM?
- short-term scheduler
  - which threads get to run on the limited number of CPUs

A process can be in one of three states:

a. **Running** ≔ executing instructions on a processor
b. **Ready** ≔ ready to run but waiting on OS
c. **Blocked** ≔ waiting on another event

In addition to these three, there are the edge cases of:

a. **Initial** ≔ just created, environment not set up
b. **Final/Zombie** ≔ process is complete but hasn't been cleaned up yet

Moving from ready to running is called being **scheduled**
Moving from running to ready is called being **de-scheduled**

The act of **scheduling** is assigning CPUs to threads

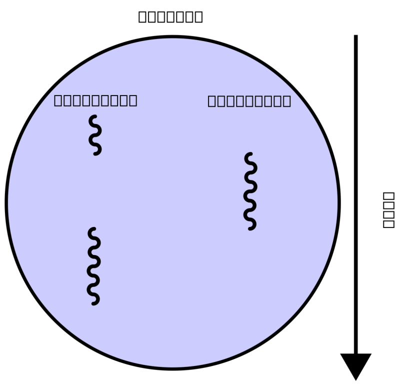
- each instruction pointer requires a CPU to run
- this requires a harmony of hardware and software

Determining how to schedule is easy when we have enough CPUs, but what do we do if there are more threads than CPUs?

We need a few things to answer this
Theory: scheduling policy
Practice: Scheduling and Dispatch Mechanisms

A basic two-thread execution may look like this:

Context switches can be timed in two broad ways:



*tiny gaps where neither thread is executing are called **context switches***

1. Cooperative Scheduling
   - a thread “volunteers” to give up its CPU with syscalls
     - close: SYSENTER → kernel → SYSEXIT
2. Preemptive Scheduling
   - the OS preempts threads every time slice
     - this is cheap and easier on the OS
     - this is common in IOT/embedded systems
     - short slice = less efficient
     - long slice = long wait

How do threads “volunteer”?

```
#include <sched.h>
int sched_yield(void);
```

We can use this to make waiting on a device more efficient:

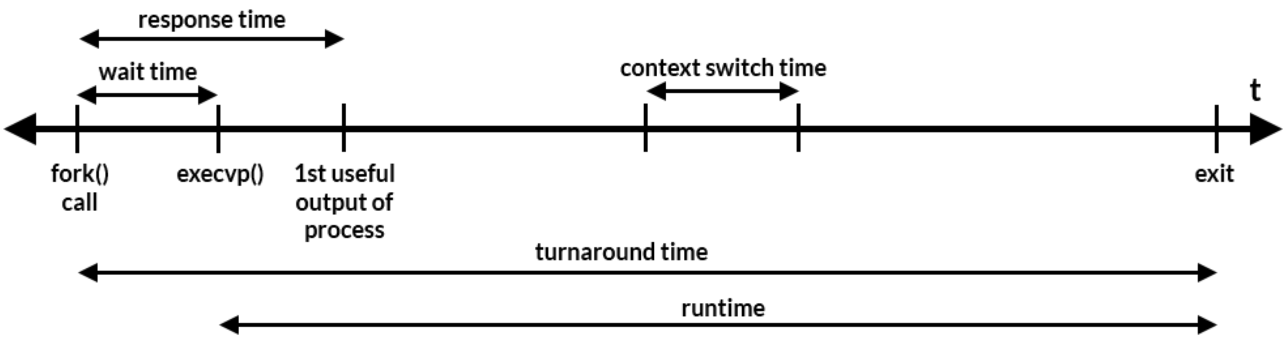
```
// we can bust wait
while(isbusy(device)) continue;
// we can poll, which is still ineffecient (but better)
while(isbusy(device)) sched_yield()
// or we can block, telling kernel to wake it when the conditi
on is met
while(isbusy(device)) wait_until_ready()
```

Linux scheduler algorithm:

```
for (;;) {
  choose an unblocked thread
  load it into a CPU
  run it
  yield
  store state
  for each thread that has become ready
    unblock
```

```
}
```

This scheduler is completely terrible! People have measured! But what did they measure?



In addition, SEASNET screws us students over with a priority queue:
- root
- operations staff
- students/faculty

And each of these contains its sub-scheduler & algorithm

Usually, priority 1 is the highest priority, but SEASNET uses the ides of niceness
  - if p1 has niceness x and p2 has niceness y > x, p2 will defer
  - users can raise a program's niceness,

We can't be too harsh though... there is a lot of complexity involved in **real-time systems**

Real-Time Scheduling

This contains two types of deadlines:

HARD:
- deadlines CANNOT be missed, → performance = correctness
- predictability > performance, → caches are the enemy
- use polling instead of interrupts, since polling controls test duration

SOFT:
- a missed deadline is not necessarily a failure
- 2 scheduling options:

    - rate-monotonic scheduling:
        - give a % usage to job data streams
    - earliest deadline first
        - can drop late requests when inundated
        - one stream can monopolize
- ex) Video Playback
    - each frame is treated as its own request
    - when the connection is slow, the scheduler periodically drops frames
        - this is often imperceptible to humans

Most real-life systems have both hard and soft real-time scheduling; this introduces a lot of complexity.