Schedulers can work on multiple levels:

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

A process can be in one of three states:

1. *Running* := executing instructions on a processor
2. *Ready* := ready to run but waiting on OS
3. *Blocked* := waiting on another event

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

1. *Initial* := just created, environment not set up
2. *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

Tiny gaps where neither thread is executing are called *context switches*, and are done by the OS. These can be timed in two broad ways:

1. Cooperative Scheduling :=a thread "volunteers" to give up its CPU with syscalls
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"?

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

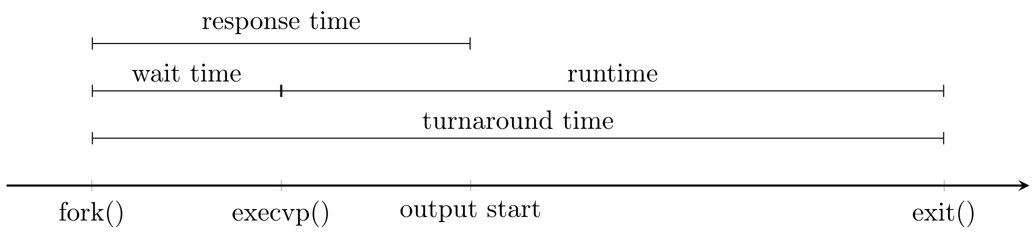
We can use this to make waiting on a device more efficient:

```c
// 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 condition is met
while(isbusy(device)) wait_until_ready()
```

The Linux scheduler runs the following 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 with priorities:

1. root
2. operations staff
3. students/faculty

And each of these contains its own sub-scheduler and 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, but cannot lower it.

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

0.1 Real-Time Scheduling

This contains two types of deadlines:

- HARD:
  - deadlines CANNOT be missed, $\implies$ performance = correctness
  - predictability $>$ performance, $\implies$ 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:
    1. rate-monotonic scheduling $:=$ give a % usage to job data streams
    2. earliest deadline first $:=$ can drop late requests when inundated. This allows one stream to monopolize the CPU.
       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.