```
* survey quiz questions

1. elevator algorithm?

optimizing IO in traditional hard disk (mechanical) devices.  faster to
read/write blocks along the way, and not randomly move around.  Just like a
real elevator.

modern I/O devices include Flash based Solid State disks (SSDs) that don't
have moving parts, but require a different I/O scheduling alg. b/c of their
own differences in behavior.

2. what resources does a process in READY state (or "RUNNABLE") waiting for?

relates to the scheduler.  a process in READY state is waiting for the CPU
(or a free CPU core).

Other scheduler states?

1. RUNNING: when a process actually executes on a CPU/core

2. READY/RUNNABLE: waiting for the CPU

3. WAITING (BLOCKED, SLEEPING): waiting on I/O.  A separate queue b/c I/O is
   much much slower than the CPU, and it's easier to manage the two waiting
   states separately.

How do you move between states?

(a) READY -> RUNNING: A scheduler process (or kernel thread) runs, pick one
or more ready processes to run, and load them into a free CPU (instead of
the currently running scheduler code itself).

Common method to pick a process to run is "round robin" (a circular queue).
But can also give priority to different processes based on the resources
they need, the kind of work they need (e.g., networking vs. disk), user
priorities, etc.

(b) RUNNING -> READY: processes don't get to run as long as they wanted.  It
won't be fair.  An OS has to ensure "fairness".

At boot time OS, programs an interrupt controller for all sorts of devices,
each with its own interrupt handler routine.  One of these routines wakes
up the scheduler code, to determine whom to run next, store the previously
run process at the end of the (round robin) queue, etc.  Note: a h/w
interrupt forces the CPU to preserve the current running program's state
(CPU registers, SP, CP, memory maps, etc.).

The scheduler is woken up frequently enough, to ensure that a running process
doesn't exceed its time slice (or quanta).  When it happens many times a
second, it appears as if all processes are running at the same time.

Q: does scheduler have a priority?

A: it depends on the priority of the class of interrupts in question.  For
example, clock interrupts to update the computer time, are higher priority
than network interrupts (worst case -- drop a packet and wait for it to be
resent).

Q: who monitors the scheduler?

A: no one really.  OS developers have to ensure that it's small and runs
quickly.
```

Another way a process can stop executing on a CPU: Maybe process is done. A process can also explicitly "yield" to the scheduler. (e.g., in linux you can call schedule() or yield()).

(c) RUNNING -> WAITING?

Happens if a process is running and then issues some request to access a resource that's not available, and requires an I/O request to be issued (e.g., networking, storage I/O). Then separately, and asynchronous request will be issued to get that I/O processed. A "long" time later, the I/O request is done (e.g., reading a file).

(d) Taking a process off of the WAITING state?

When an I/O is finally done (e.g., a disk has the data requested), the I/O device can interrupt the CPU (can also happen through DMA). Then often another piece of software, like a "dispatcher" (or "soft IRQs" in Linux), is woken up to figure out who was the data for. This dispatcher (often a kthread), wakes up to look at recently arrived data items, identifies for whom this data was for, and the takes that process off of the WAITING state.

Key: a process can get itself into WAIT state, but cannot get out on its won. The scheduler doesn't either: it deals with RUNNING/READY transitions. So only some OTHER piece of code can get a process off of WAITING state -- a dispatcher (or even an interrupt handler in the old days).

A process moves from WAITING state to READY state, not typically RUNNING state. The process isn't running yet. The scheduler has to pick it at some future time, and actually let it run.

A final word: scheduling is complex. Have be careful not to unfairly treat high vs. low priority tasks (inversion), or not provide enough service to any one or more tasks (starvation).

Modern schedulers have multiple queues, try to find short vs. long-term processes, and try to schedule the short term ones more quickly. They also try to detect interactive vs. non-interactive processes. Detect user interaction by detecting if accessing "interactive" devices like keyboard, mouse, etc. Such processes spend most of their life WAITING, they tend to run for a long time, they most get interrupts from keyboard/mouse. Important to quickly process interrupts of interactive processes, because users expect a smooth response.