SYSTEMS ENGINEERING · WEEK 7 · STATION III

Processes

Fork, wait, exit. A process is a virtual machine with a name.

DEFINITION

A process is a machine context that believes it owns the machine, plus one door.

ITS OWN MACHINE

Registers, pc, a 4 GB address space translated by its own page table, a slice of time. Weeks 2 to 5 built each of these. The process sees the machine of week 2 and nothing else.

ONE DOOR

`ecall`: exit, read, write, open, brk, and from this week fork and wait. Everything a process can do to the world goes through the door, and the kernel decides on the other side. Isolation is the door being the only opening.

A PARENT

Every process but the first was created by another one, and remains its child until it exits and is waited for. The tree of processes is the kernel's only record of who is responsible for whom.

Which is the same thing as a virtual machine, at a different granularity: a process shares the file system and the kernel with its siblings; a virtual machine shares only the hardware. The book calls both machine contexts because they are.

FORK

One call, two returns. The copy is the whole machine.

```
pid = fork();
if (pid == 0) {
// child: pid is 0
exit(42);
} else {
// parent: pid is the child's
wait(&status); // blocks until the child exits
}
```

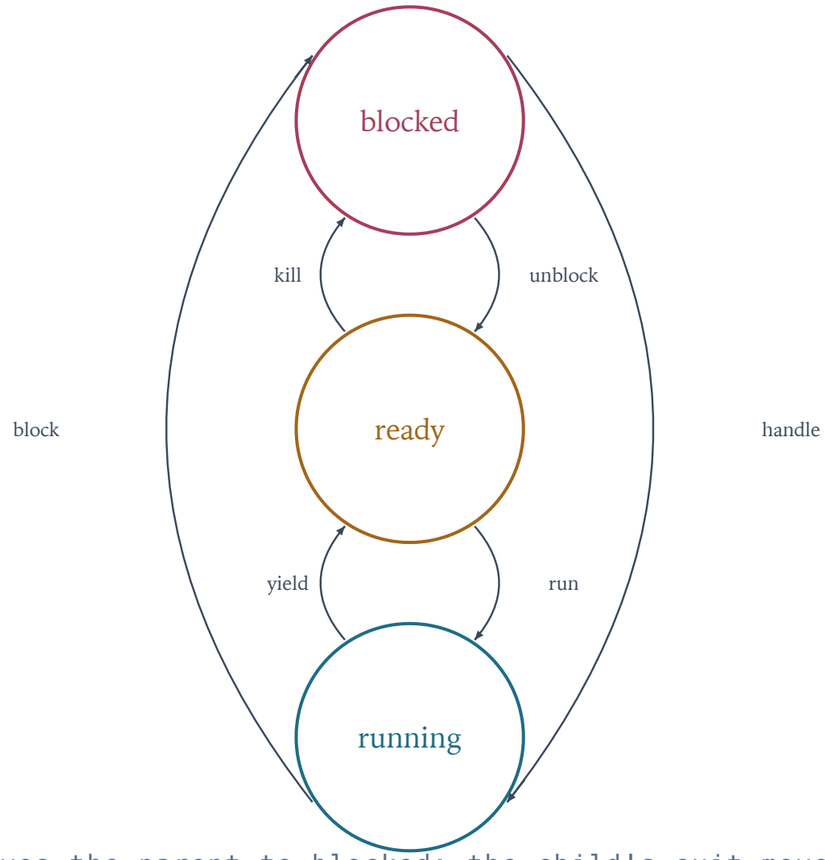
`fork` creates a new context that is a copy of the caller: same code, same data, same heap and stack, same pc. It returns in both. The only difference between them is the return value, and everything a parent and child do differently descends from that one bit.

In selfie: create a context, copy the page table's mapped pages into fresh frames, copy the registers, set `a0` to 0 in the child and to the child's id in the parent, put both on the ready queue. Real kernels copy lazily, on the first write, which is another notation for the same meaning.

A virtual machine snapshot, at process granularity. The compiler class's fixed point was one program producing itself; fork is one machine producing itself. Self-reference again, harmless here because the copy is isolated.

WAIT AND EXIT

`wait` blocks. Blocked is a state the kernel can see.



wait moves the parent to blocked; the child's exit moves it back to ready

`exit` ends a context and records its status; `wait` blocks the caller until some child has exited, then returns that status. The parent is not polling; the kernel moves it to blocked, gives its slices to others, and moves it back when the child's exit arrives.

The kernel cannot know whether a program will exit. It can know, exactly, whether a program has called `exit`: the door records it. Decidable facts about processes are facts about the door, and only those. Rice, drawn as a line around the interface.

A parent waiting on a child that waits on the parent: both blocked, forever, and the kernel cannot see it unless it looks for cycles. Deadlock, week 8's problem, first met here.

UNDER THE HOOD

Copying a process costs pages. Creating one costs nothing.

A machine context is a few kilobytes. Creating an empty one is instantaneous, and paging fills it on demand. So a process is cheap until it touches memory, which is why real systems can run thousands.

Fork copies mapped pages: the cost is proportional to what the parent touched, not to its address space. On real hardware copy-on-write defers even that, and the child pays only for the pages it changes.

MEASURE IT

In your `fork-wait` implementation, count the pages copied per fork, and compare against the profile's page-fault count. Which pages of selfie itself does a fork of selfie copy? The Cost week will want the number.

ASSIGNMENT

`fork-wait-exit`: finish the process tree.

1. Complete `fork` from last week: a child context with copied pages and registers, both on the ready queue, return values 0 and the child's id.
2. `wait` blocks the parent until a child exits, then returns the child's exit status through the pointer argument. Design what happens if there is no child.
3. `exit` records the status, frees the context, and wakes a blocked parent if there is one. What if the parent exits first? Decide, and document the decision.
4. Test with the book's fork examples; `./grader/self.py fork-wait` and `./grader/self.py fork-wait-exit`.

LOOK AHEAD

Keep the blocked state explicit and separate from ready; week 8 reuses it for locks, and week 9 asks what memory a process that exited leaves behind. Orphans and zombies are not exotic: they are what a tree of contexts does when the door records exits and nothing else.