Process: address space, execution states, OS resources
Metadata: process control block (PCB)
   \( \rightarrow \) pid, address space info, kstack

```c
// Per-process state
struct proc {
    struct vspace vspace; // Virtual address space descriptor
    char* kstack; // Kernel stack
    enum procstate state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    struct trap_frame *tf; // Trap frame for current syscall
    struct context *context; // swtch() here to run process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    char name[16]; // Process name (debugging)
};
```
Process APIs: Fork

→ creates a new process that’s an exact copy of the calling process

→ Address space

parent VAS

map to child VAS

physical memory

(simplified mapping)

→ Execution states

→ registers, SP, PC

→ where are the parent’s execution states?

*trapframe! parent is executing a system call.

child’s trapframe is a copy of the parent’s

different rax (return value)

parent gets the child’s pid.

child gets 0.

Independent VAS!

Update in one doesn’t affect the other
fork();

one parent

one child

pid = fork();

if (pid == 0) {
    fork();

    \[\]

    one parent

    one child

    \[\]

    one grandchild.

fork();

one parent

one child

another child (2nd fork)

one grandchild (2nd fork)
Process APIs: `exec`

- Loads a new program into the current process (replaces the current program!)

\[ \text{exec(} \text{"B")} \]

Same process, different address space, different execution states.
\[ \text{\texttt{rip} = program B's entry point} \]
\[ \text{\texttt{rsp} = program B's args} \]

\( (\text{pid}10) \) process VAS according to program A
\( (\text{pid}10) \) process VAS according to program B

\( \rightarrow \) set up a new address space,
Switch to it, frees the old address space.
Process APIs: exec

- exec also lets you pass arguments to the newly loaded program.
  - `int main (int argc, char** argv)` is an array of `char*` (string), null terminated array.
  - `argc` & `argv` is set up by the kernel
  - `argv` is on the user stack
  - `argc` is the number of arguments

- `argv[0]`: address of "a"
- `argv[1]` = 0

```
argv array
```

```
| "a" |
| 0  |
| a's addr |

user stack page

set up argument - "a"
Fork exec combo

- Simple semantics
- Easy to support redirect

Example: ls > output

```c
pid = fork();
if (pid == 0) {
    fd = open("output");
    close(stdout);
    dup(fd);  // stdout now points to output file
    execl("ls");
}
```

Alternative APIs

- Spawn (windows)
- Clone (unix, select which resource gets copied)

→ But also has large performance cost

- Fork allocates physical memory, copies over parent's memory just to throw away everything on exec!

→ Copy-on-write (cow)

- Share the same phys. memory for as long as possible (until a write)
- Upon write, makes a copy so the write can be carried out independently

→ How do we catch it?

- By mapping shared pages read only, catch all writes via page fault
- Upon cow, allocate physical memory
  - Copy the data over, remap to new 7 w/ read write perm

- Kernel must track which pages are cow vs. actual read only