10/6/23

Process: address space, execution states, OS resources

Metadata: process control block (PCB)

→ pid, address space info, kstack

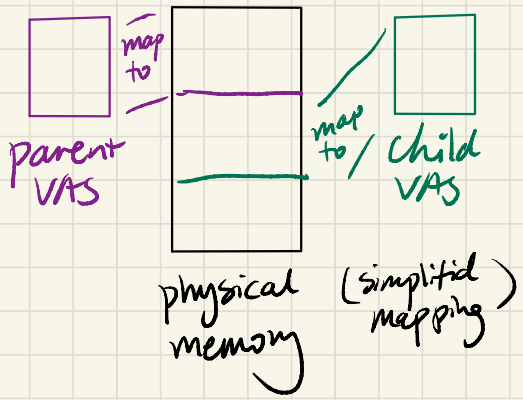
```
// Per-process state
struct proc {
  struct vspace vspace;        // Virtual address space descriptor
  char* kstack;                // Kernel stack
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct 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)
};
```

xk PCB

Process APIs: Fork

→ creates a new process (child) that's an exact copy of the calling process (parent)

→ Address space



Independent VAS!

Update in one doesn't affect the other

→ 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.

man 2 fork (manpage)

How many processes?

```
fork();
```

one parent
one child

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

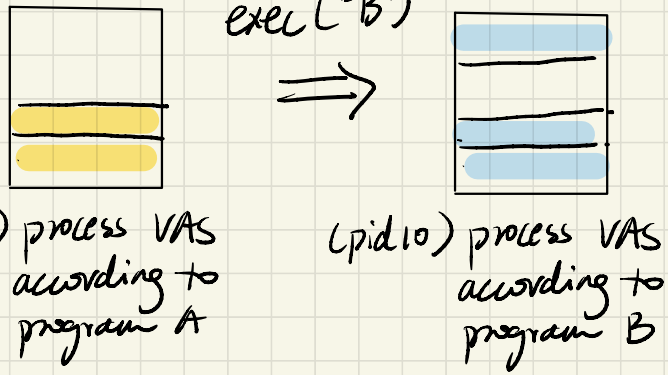
one parent
one child
one grand child

```
fork();
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!)



☆ same process, different address space, different execution states.

(rip = program B's entry point)

(rsp = program B's args)

→ set up a new address space,
switch to it, frees the old address space.

Process APIs : exec

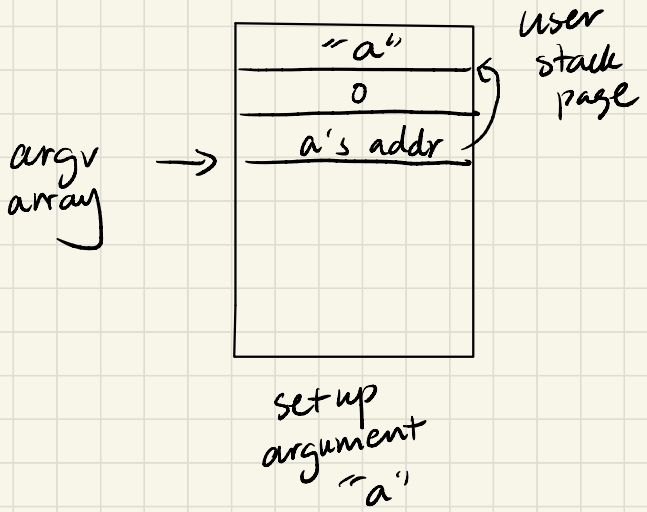
→ exec also lets you pass arguments to the newly loaded program.

→ `int main ( int argc, char** argv )` → an array of char* (string), null terminated array.

argc & argv is set up by the kernel

rdi → argc rsi → argv

the strings & argv array is on the user stack



argv[0] = address of "a"

argv[1] = 0

Fork exec combo

→ simple semantics

→ easy to support redirect

```
example: ls > output
  pid = fork();
  if (pid==0){
    fd = open("output");
    close(stdout);
    dup(fd); // stdout now points to output file
    exec("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 ↑ w/ read write perm

* kernel must track which pages are cow vs. actual read only