10/4/23

On boot, OS sets up the IDT

Mode Transfer (user → kernel → user)

HW → switch into kernel mode, switch to kernel stack (kstack), save user's PC & SP onto the stack, set PC to kernel handler (specified by IDT)

OS → kernel handler pushes rest of the regs onto kstack, execute more logic

OS → kernel handler pops saved regs.

HW → pops PC & SP (switch back to user stack), switch to user mode (reflected in %cs value)

user process → resume execution

How does the first process start in user mode?

XK userinit:

sets up the process's trap-frame, follow the second half of mode transfer flow to return to userspace

```
// Set up first user process.
void userinit(void) {
  struct proc *p;
  extern char _binary_out_initcode_start[], _binary_out_initcode_size[];

  p = allocproc();

  initproc = p;
  assertm(vspaceinit(&p->vspace) == 0, "error initializing process's virtual address descriptor");
  vspaceinitcode(&p->vspace, _binary_out_initcode_start, (int64_t)_binary_out_initcode_size);
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ss = (SEG_UDATA << 3) | DPL_USER;
  p->tf->rflags = FLAGS_IF;
  p->tf->rip = VRBOT(&p->vspace.regions[VR_CODE]);  // beginning of initcode.S
  p->tf->rsp = VRTOP(&p->vspace.regions[VR_USTACK]);

  safestrcpy(p->name, "initcode", sizeof(p->name));

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);
  p->state = RUNNABLE;
  release(&ptable.lock);
}
```

System Call Arguments & Validation

- → protected procedure call
- → x86_64 calling convention = first 8 args in registers (rdi, rsi, rdx, rcx ...) rest on the stack.
- → where are the syscall args? trapframe! (kernel stack)
- → argument validation
  - → string args (null terminated, need to validate memory address is valid before accessing it)
  - → void * & size
  - → out of range fd.

Process Abstraction

→ running instance of a program

→ isolation & protection boundary

  → failure isolation

  → no visibility into other processes unless explicitly granted

→ consists of

1). execution stream (thread)

  → code stream, execution states (PC, SP, regs)

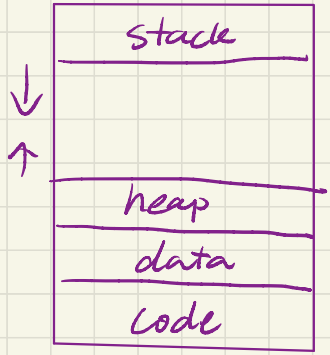
2). address space

3). client data from using other OS abstractions

  (open files, sockets pipes)

(virtual address space)

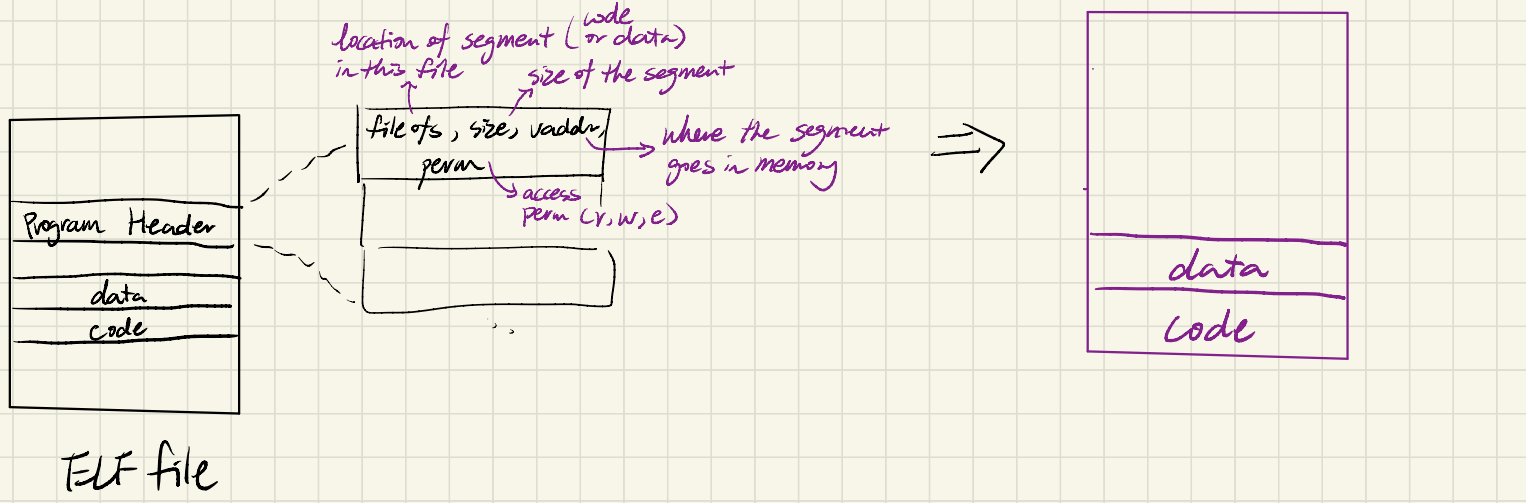
process's VAS



Process Implementation

→ program to process?

→ loads from ELF file to process's VAS.



→ How do processes share a CPU?

→ scheduling (OS policy)

→ Round Robin

→ time slice/time quantum (10-100 ms)

→ Process's States / Process Life Cycle