4/1/24

Mode Transfers Wrap up & Processes

Mode Transfer (user → kernel → user)

HW → switches into kernel mode, saves user's PC & SP, switches to kernel stack, pushes saved regs on to kstack, sets PC to kernel handler (specified by IDT)

OS → pushes rest of the regs onto kstack

OS → executes handler logic

OS → kernel handler pops saved regs.

HW → pops PC & SP (switch back to user stack), switch to user mode

user process → resumes execution

xk code flow

vectors.S 20.31 KiB

```
.globl alltraps
.globl vector0
vector0:
  push $0
  push $0
  jmp alltraps
.globl vector1
vector1:
  push $0
  push $1
  jmp alltraps
.globl vector2
vector2:
  push $0
  push $2
  jmp alltraps
.globl vector3
vector3:
  push $0
  push $3
  jmp alltraps
```

```
vector10:
  push $10
  jmp alltraps
.globl vector11
vector11:
  push $11
  jmp alltraps
.globl vector12
vector12:
  push $12
  jmp alltraps
.globl vector13
vector13:
  push $13
  jmp alltraps
.globl vector14
vector14:
  push $14
  jmp alltraps
.globl vector15
```

handlers in IDT entries

trapasm.S 447 B

```
.globl alltraps
alltraps:
  push %r15
  push %r14
  push %r13
  push %r12
  push %r11
  push %r10
  push %r9
  push %r8
  push %rdi
  push %rsi
  push %rbp
  push %rdx
  push %rcx
  push %rbx
  push %rax

  mov %rsp, %rdi
  call trap
```

set up trap frame

kernel saving register state

```
void trap(struct trap_frame *tf) {
  uint64_t addr;

  if (tf->trapno == TRAP_SYSCALL) {
    if (myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if (myproc()->killed)
      exit();
    return;
  }

  switch (tf->trapno) {
  case TRAP_IRQ0 + IRQ_TIMER:
    if (cpunum() == 0) {
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case TRAP_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case TRAP_IRQ0 + IRQ_IDE + 1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case TRAP_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
```

actual handler logic

```
struct trap_frame {
  uint64_t rax; // rax
  uint64_t rbx;
  uint64_t rcx;
  uint64_t rdx;
  uint64_t rbp;
  uint64_t rsi;
  uint64_t rdi;
  uint64_t r8;
  uint64_t r9;
  uint64_t r10;
  uint64_t r11;
  uint64_t r12;
  uint64_t r13;
  uint64_t r14;
  uint64_t r15;
  uint64_t trapno;
  /* error code, pushed by hardware or 0 by software */
  uint64_t err;
  uint64_t rip;
  uint64_t cs;
  uint64_t rflags;
  /* ss:rsp is always pushed in long mode */
  uint64_t rsp;
  uint64_t ss;
} __packed;
```

inc/trap.h

System Call Arguments & Validation

→ 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)

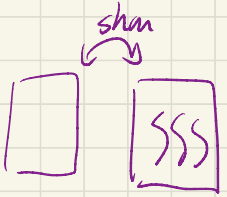
→ argument validation

→ string args (null terminated, need to validate memory address before accessing it)

→ void * & size

→ out of range fd.

→ return value? %rax in the trapframe!



multithreaded process or process w/ shared memory may change the string post validation

→ needs to make a copy of str args.

Process Abstraction

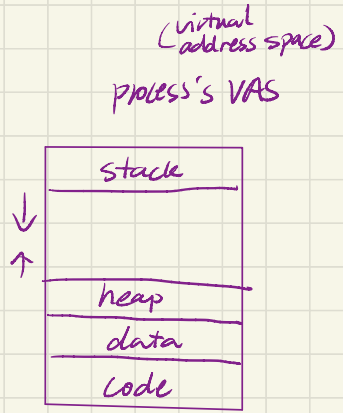
→ running instance of a program

→ consists of

1). execution stream (thread)
   → rip, stack, regs.

2). virtual address space

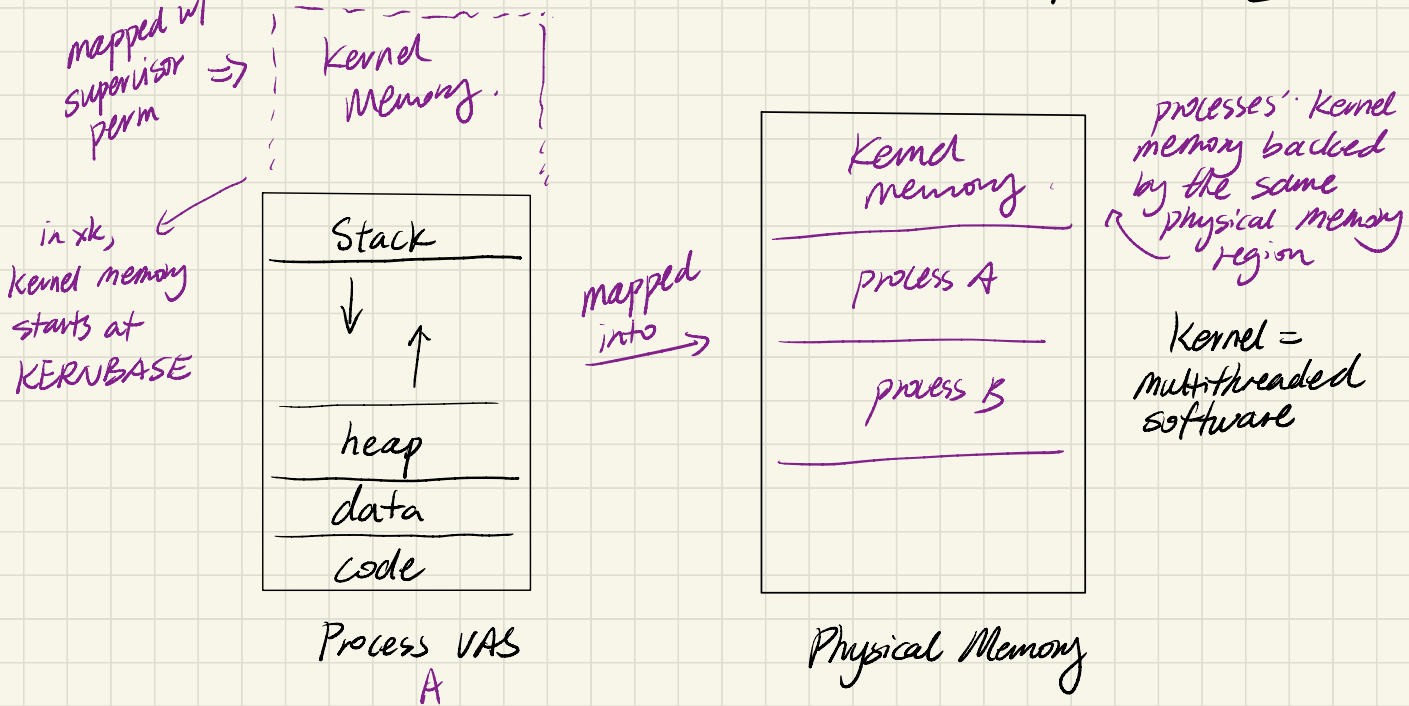
3). process metadata (Process Control Block) → struct proc in xk.
   → process id, kernel stack, OS abstractions (file descriptors, open files)



→ isolation & protection boundary
   → failure isolation
   → no visibility into other processes unless explicitly granted

loading translation table is expensive

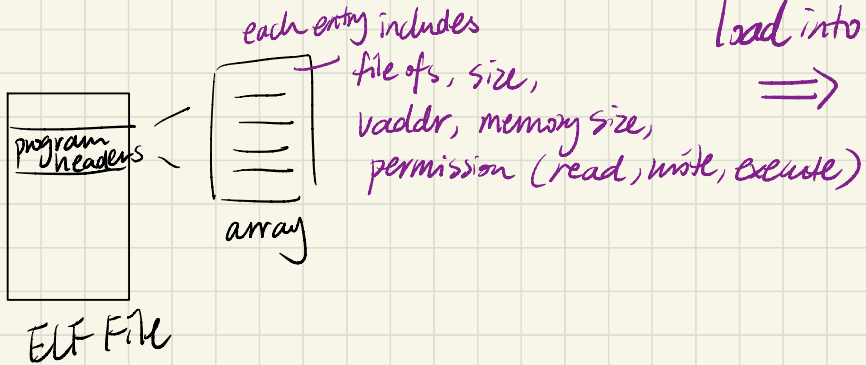
clearing TLB (caches address translation) also expensive (causes more miss)



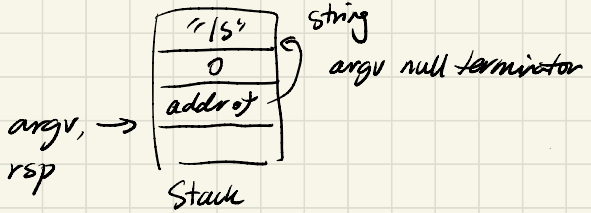
Process Implementation

→ program to process?

→ ELF tells the kernel the entry point of the program & how to set up the VAS



Done setting up process's view



→ Initialize PCB (kernel stack)

→ assign pid, allocate kstack, initialize file descriptors

→ how does the process start execution?

→ starts in the kernel, follows the return path of mode transfer

→ kernel needs to set up the trapframe to reflect the starting state of the process

→ How do processes share a CPU?

→ scheduling: OS policy on who should run on the CPU?

→ let processes take turn using the CPU for a small amt of time

→ Round Robin

→ time slice/time quantum (10-100 ms)

→ FIFO order

[context switch: switching btwn processes/threads, typically take place in kernel mode]