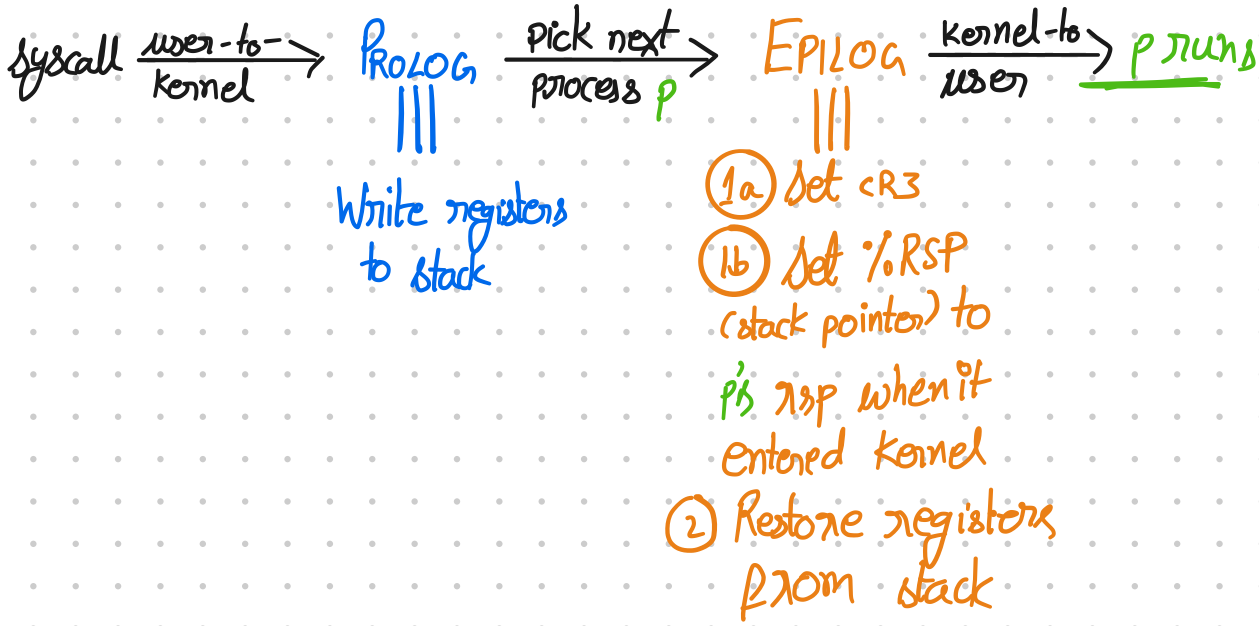
CS202 (S-002): Lecture 15: Context Switches

- Last class: Context switch

Core idea



PROLOG

```
134 generic_exception_handler:
135         pushq %gs
136         pushq %fs
137         pushq %r15
138         pushq %r14
139         pushq %r13
140         pushq %r12
141         pushq %r11
142         pushq %r10
143         pushq %r9
144         pushq %r8
145         pushq %rdi
146         pushq %rsi
147         pushq %rbp
148         pushq %rbx
149         pushq %rdx
150         pushq %rcx
151         pushq %rax
152         movq %rsp, %rdi
153         call exception
154         # `exception` should never return.
```

← first arg into C code

Pick Next

```
268 // schedule
269 //    Pick the next process to run and then run it.
270 //    If there are no runnable processes, spins forever.
271 //
272 void schedule(void) {
273     pid_t pid = current->p_pid;
274     while (1) {
275         pid = (pid + 1) % NPROC;
276         if (processes[pid].p_state == P_RUNNABLE) {
277             run(&processes[pid]);
278         }
279         // If Control-C was typed, exit the virtual machine.
280         check_keyboard();
281     }
282 }
```

```
284
285 // run(p)
286 //    Run process `p`. This means reloading all the registers from
287 //    `p->p_registers` using the `popal`, `popl`, and `iret` instruct
288 //
289 //    As a side effect, sets `current = p`.
290 //
291 void run(proc* p) {
292     assert(p->p_state == P_RUNNABLE);
293     current = p;
294
295     // Load the process's current pagetable.
296     set_pagetable(p->p_pagetable);
297
298     // This function is defined in k-exception.S. It restores the pro
299     // registers then jumps back to user mode.
300     exception_return(&p->p_registers);
301
302 spinloop: goto spinloop;                // should never get here
303 }
304
```

← %rdi — calling convention

EPILOG

```
157         .globl exception_return
158 exception_return:
159         movq %rdi, %rsp
160         popq %rax
161         popq %rcx
162         popq %rdx
163         popq %rbx
164         popq %rbp
165         popq %rsi
166         popq %rdi
167         popq %r8
168         popq %r9
169         popq %r10
170         popq %r11
171         popq %r12
172         popq %r13
173         popq %r14
174         popq %r15
175         popq %fs
176         popq %gs
177         addq $16, %rsp
178         iretq
```

Remember — CORE IDEA

Manipulating the stack pointer allows one to switch threads

1. For call/syscall/int/... information on the stack determines where control is transferred (i.e., TO WHAT CODE) when RET/IRET are called.
2. Content of registers + memory determine state seen by code
   - On return, RESTORED BY EPILOG ⟷ Reads from stack
3. %CR3 dictates address space
   - Must change when switching b/w processes.

Very little of the context switch logic depends on being in kernel

Userspace CANNOT change %CR3

But OK if we are switching b/w threads

Why?

Userspace threads (HANDOUT)

```
typedef struct tcb {
  unsigned long saved_rsp;    /* Stack pointer of thread */
  char *t_stack;              /* Bottom of thread's stack */
  /* ... */
  };
```

← Where the stack begins

current → saved_rsp ↦ %RSP

```
void swtch(tcb *current, tcb *next);
```

%RDI %RSI

```
# Save call-preserved (aka "callee-saved") regs of 'current'
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15

# store old stack pointer, for when we swtch() back to "current" later
movq %rsp, (%rdi)                    # %rdi->saved_rsp = %rsp
```

next → saved_rsp

} PROLOG

} Change stack

```
41          movq (%rsi), %rsp               # %rsp = %rsi->saved_rsp
42
43          # Restore call-preserved (aka "callee-saved") regs of 'next'
44          popq %r15
45          popq %r14
46          popq %r13
47          popq %r12
48          popq %rbx
49          popq %rbp
50
51          # Resume execution, from where "next" was when it last entered swtch()
52          ret
53
```

EPILOG & Return.

mmap(2)

- Bridge virtual memory & file system

So far

- Virtual memory – Page tables
  - ↳ Manipulated by the kernel

- mmap(2) syscall to manipulate page tables & address space from userspace

Many uses

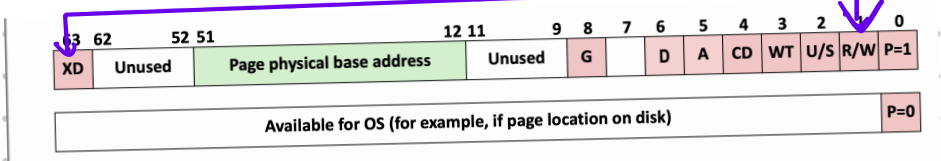
- allocators (the thing implementing malloc/free)
  - Growing & Shrinking heap
- treating files like memory
- JIT, etc.


```
SYNOPSIS
       #include <sys/mman.h>

       void *mmap(void *addr, size_t len, int prot, int flags,
           int fildes, off_t off);
```

Virtual Address

Read/Write/Execute



- fildes
  - ↳ File descriptor
    - → source of data
    - → Can be -1 to indicate no file should

Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for this page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

be accessed

- off: where to map file from

- Flags
  - Lots of options

```
SYNOPSIS
       #include <sys/mman.h>

       int munmap(void *addr, size_t len);
```

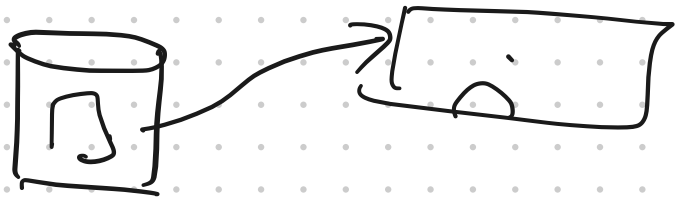
- Remove mapping

```
SYNOPSIS
       #include <sys/mman.h>

       int mprotect(void *addr, size_t len, int prot);
```

- Change protection

- Can manipulate VM using syscalls.
- Files can be a source of memory contents
  - ↳ Saw this previously too - when?



```
void f() {
    g(5);
}
```

```
void g(int a) {
    printf("%d\n", a+5);
}
```