OS’25 Project

PART I: **PREREQUISITES**

Agenda

- PART 0: ROADMAP
- PART I: PREREQUISITES
- PART II: GROUP MODULES
- PART III: INDIVIDUAL MODULES
- PART IV: OVERALL TESTING & BONUSES

Agenda

- **PART I: PREREQUISITES**
  - Pointers
  - Lists
  - System Calls
  - Spin Locks

Pointers

PART I: PREREQUISITES

Memory and Pointers

a. Why we need pointers?!
b. Pointers vs. Variables...
    i. Definition
    ii. Data type & size
    iii. Setting values
    iv. Incrementing…
    v. Structure and accessing its members

Pointer vs Variables: Definition

Pointers	Variables
`char *ptr;`	`char x;`

Pointer vs Variables:
Data type & Size

Pointers	Variables
`char *ptr;`	`char x;`

Pointers

`char *ptr;`

Data type:

- Data type to point into it

Size of `ptr` (address)**:**

- 4 Byte
- Size of address bus of CPU (protected → 32-bit)

Variables

`char x;`

Data type:

- Data type of variable itself

Size of `x`**:**

- 1 Byte → `sizeof(char)`

Pointer vs Variables: Assigning value

Pointers	Variables

Pointers

Assigning value: Addr.

```
char *c_ptr = 0x100;
//changes the pointed address
c_ptr = 0x50;
c_ptr = &x;

//changes  the value within the pointed address
*c_ptr = 'A';
```

Variables

Assigning value:

```
char x = 10;
x = 50;
x = 'A';
```

Pointer vs Variables: Incrementing

Pointers	Variables

increment:

```
char *c_ptr = 0x100;
c_ptr++; // ptr=0x101

int *i_ptr = 0x100;
i_ptr++; // ptr=0x104
```

Increases by the size of its type

increment:

```
char x = 10;
x++; // x=11

int x = 10;
x++; // x=11
```

Increases by 1

Pointer vs Variables:
Structure & its members

```
struct MyStruct {
   int x, y;
   char c;
   char* c_ptr;
}
```

Pointers

Init. & Assign.:

```
Struct MyStruct *my_struct;
my_struct->x = 5;
my_struct->c = 'A';
(*my_struct).y = 10
```

Variables

Init. & Assign.:

```
Struct MyStruct my_struct;
my_struct.x = 5;
my_struct.c = 'A';
```

Pointer vs Variables:
Structure & its members

```
struct MyStruct {
   int x, y;  8 B
   char c;   +1 B
   char* c_ptr; +4B
};
```

Pointers	Variables
Init. & Assign.:	**Init. & Assign.:**

`Struct MyStruct *my_struct;`
`my_struct = 100;`
`my_struct++;   // ptr=113 increases by size of struct`
	`Struct MyStruct my_struct;`

LISTs in FOS

PART I: PREREQUISITES

How to define LISTS in FOS?

To define a LIST that points to objects of type **struct my_struct**:

1. Create a list **head** that holds info about the list (size, head, tail).
   - `LIST_HEAD([LIST_TYPE_DEF], [STRUCT NAME THAT WILL POINTS TO]);`
   - **Ex:**

```
LIST_HEAD(MY_LIST_TYPE, my_struct);
```

2. Add **next** and **previous** pointers to the **struct**
   - `LIST_ENTRY([STRUCT NAME]) prev_next_info;`
   - **Ex:**

```
struct my_struct
{
    int x, y;
    LIST_ENTRY(my_struct) prev_next_info;
};
```

3. Define your list
   - `struct [LIST_TYPE_DEF] my_list`
   - **Ex:**

```
struct MY_LIST_TYPE my_list;
```

How to use LISTS?

Set of helper ready made functions are available in [Appendix](#)

- `LIST_INIT(…)`
- `LIST_INSERT_HEAD(…)`
- `LIST_SIZE(…)`
- `LIST_REMOVE(…)`
- …

IMPORTANT: you should **pass** the list to any of these functions by **reference**

(i.e. Put **&** before the name of the list)

System Calls

PART I: PREREQUISITES

System Calls – Idea

It's OS procedure that executes privileged instructions (e.g., I/O); (API exported by kernel)

Causes a **trap**, which

1. Switch to the kernel mode
2. Look in Interrupt Descriptor Table (IDT)
3. Jumps to the **syscall handler** in the kernel.

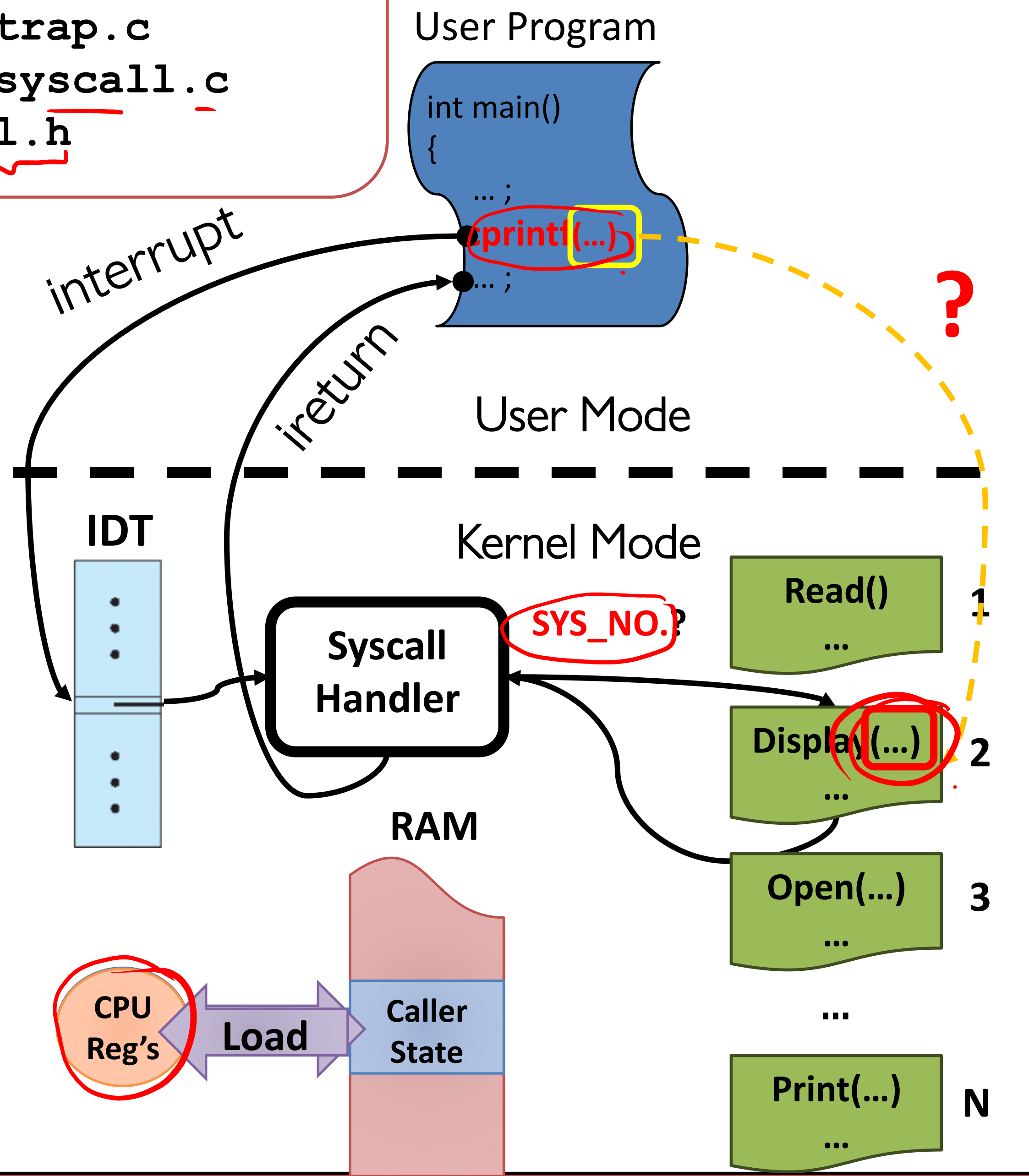
How to pass params/return value from/to user to kernel or vice versa?

OS should verify the caller's parameters.

3. Call **associated function** that serve the given system call and pass to it the caller's **parameters**
4. After finish, **restore** caller's state (CPU Reg's)
5. use **iret** instruction to **return** to user mode.

Where in Code?

```
lib/syscall.c
kern/trap/trap.c
kern/trap/syscall.c
inc/syscall.h
```



System Calls – Params Validation

At **kernel side**: need to validate any **address (or range)** that is passed from user to kernel to ensure that:

1. NOT **null** pointers,
2. NOT **illegal** pointers (e.g. pointing to kernel memory) (i.e. outside User Area)
3. NOT **invalid** pointers (e.g. pointing to unmapped memory),

First TWO cases should be checked in the **system call**. If violated, kernel can **terminate** the **user program** by calling: `env_exit();`

Third case cause a "page fault" that can be handled in the **page_fault()** handler.

This technique is normally faster because it takes advantage of the processor's MMU, so it tends to be used in real kernels (including Linux).

Spin Locks

PART I: PREREQUISITES

Locks: **Definition**

Locks provide two **atomic** operations:

- Lock.acquire() – **wait** until lock is **free**; then **mark** it as **busy**
  - After this returns, we say the calling thread ***holds*** the lock
- Lock.release() – **mark** lock as **free**
  - Should only be called by a thread that currently holds the lock
  - After this returns, the calling thread no longer holds the lock

Negatives of interrupt-based implementation:

- **Can't** give lock implementation to **users**
- **Doesn't** work well on **multiprocessor**
- Lock entire Building!!

Locks: **Implementation**

Exchange Instruction

- Exchanges the contents of a register with a memory location.
- Available in Intel IA-32 (Pentium) and IA-64 (Itanium)
- The entire function is carried out **atomically**; By H/W
  - not subject to interruption (i.e. ***indivisible***).

(Swap)

```
void exchange (int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

Locks: **Implementation (SpinLock)**

Simple lock that doesn't require entry into the kernel:

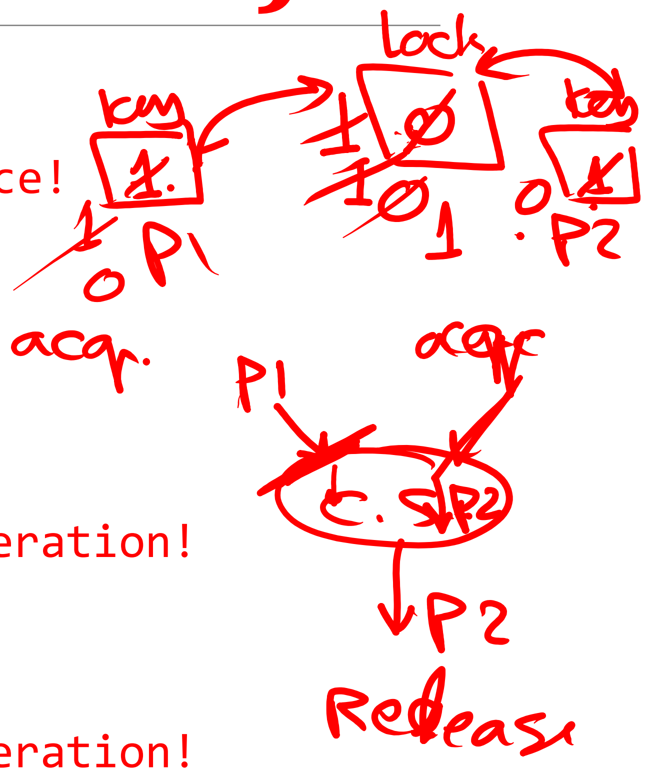
```
// (Free) Can access this memory location from user space!
int lock = 0;             // Interface: acquire(&lock);
                          //            release(&lock);

acquire(int *thelock) {
   int mykey = 1;
   while (xchg(thelock, mykey) != 0);      // Atomic operation!
}

release(int *thelock) {
   *thelock = 0;                            // Atomic operation!
}
```

spin

P1



Busy-Waiting: thread consumes cycles while waiting

Locks: **Implementation (SpinLock)**

Positives

- Machine can receive interrupts
- User code can use this lock
- Works on a multiprocessor
- No System Calls at all

Usually used for
short-time critical section

Negatives

- This is very **inefficient** as thread will consume cycles waiting (**busy-waiting**)
- Cache coherence issue in multi-cores
  - always read-write while waiting ➔ high traffic on cache buses to ensure data consistency
- **Priority Inversion**: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
  - **Solution: disable** the interrupt in acquire and **enable** it in release

High Prio. Low Prio

SpinLock: **In Code (KERNEL)**

```
struct kspinlock {
  uint32 locked;          // Is the lock held?
```

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void acquire_kspinlock(struct kspinlock *lk)
{
  if(holding_kspinlock(lk))
      panic("acquire_spinlock: lock \"%s\" is already

  pushcli();  /*disable interrupts to avoid deadlock (


  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0) ;
```

```
// Release the lock.
void release_kspinlock(struct kspinlock *lk)
{
    if(!holding_kspinlock(lk))
    {
        printcallstack(lk);
        panic("release: lock \"%s\" is either not held or
    }
    // Release the lock, equivalent to lk->locked = 0.
    // This code can't use a C assignment, since it might
    // not be atomic. A real OS would use C atomics here.
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();     //enable the interrupt
}
```

SpinLock: **In Code (USER)**

```
struct uspinlock {
  uint32 locked;          // Is the lock held?
```

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void acquire_uspinlock(struct uspinlock *lk)
{
    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0) ;
}
```

```
// Release the lock.
void release_uspinlock(struct uspinlock *lk)
{
    if(!(lk->locked))
    {
        panic("release: lock \"%s\" is not held!", lk->name);
    }
    // Release the lock, equivalent to lk->locked = 0.
    // This code can't use a C assignment, since it might
    // not be atomic. A real OS would use C atomics here.
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );
}
```

NO INTERRUPT ACCESS @USER SIDE

NOW, LET'S START CODING **TOGETHER...**

☺ Enjoy **developing** your **own OS** ☺