CS310 Operating Systems

Lecture 9: Syscall exec*()
Dual Mode of Operation – Part 1

Ravi Mittal

IIT Goa

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - Class presentation: University of California, Berkeley: David Culler, Anthony D. Joseph, John Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner
  - Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau,
    - Chapter 5: Process APIs
    - Programs are taken from this chapter
  - CS 423 Operating System Design, Uinv of Illinois, Prof Fagen-Ullmschneider
  - CS351 University of Washington

Read the following:

- Book: Operating Systems: Principles and Practice (2nd Edition) Anderson and Dahlin
  - Volume 1, Kernel and Processes
    - Chapter 2.2: Dual Mode of Operation
- Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau,
  - Chapter 5: Process APIs

We will study..

- Process – Last class, `exec*()` syscall example
- Dual Mode of Operation
- Hardware support for dual mode operation
- Mode Transfer: User to kernel - Introduction

Read the following:

- Operating Systems: Principles and Practice (2nd Edition) Anderson and Dahlin
  - Volume 1, Concurrency
    - Chapter 2: Kernel Abstraction

Previous Classes..

Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Fully describes program state

Done

- **Address space**
  - Set of memory addresses accessible to program (for read or write)
- **Process: an instance of a running program**
  - Protected Address Space + One or more Threads
- **Dual mode operation / Protection**
  - Only the "system" has the ability to access certain resources

Last Class: Process Syscalls: `fork()` `exit`, `wait`, `exec*()`

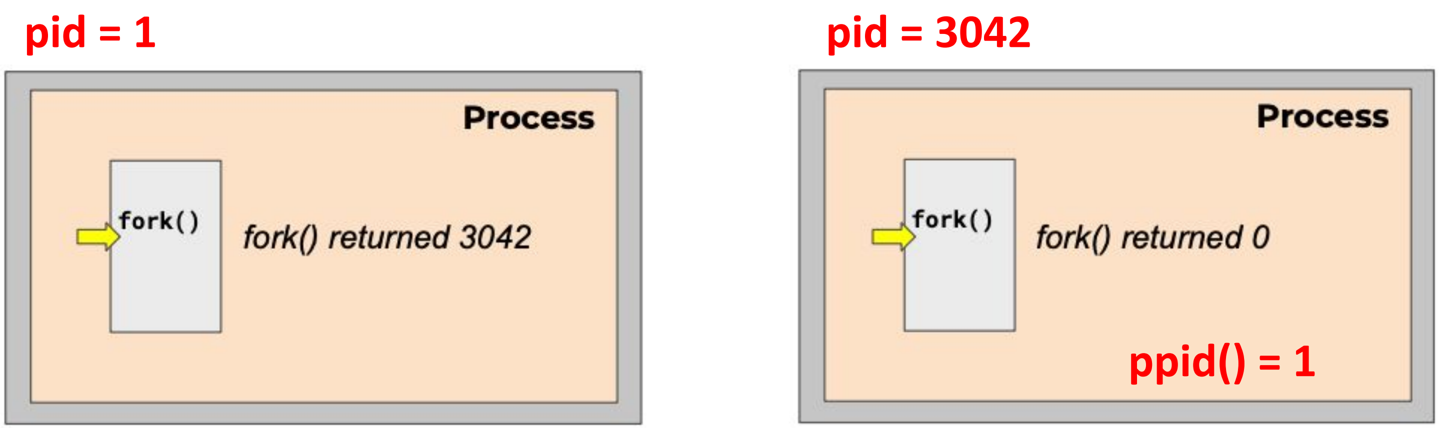
Process Related System Calls (in Unix)

Last class

- fork() creates a new child process
  - All processes are created by forking from a parent
  - The *init* process is ancestor of all processes
- exec() makes a process execute a given executable
- exit() terminates a process
- wait() causes a parent to block until child terminates

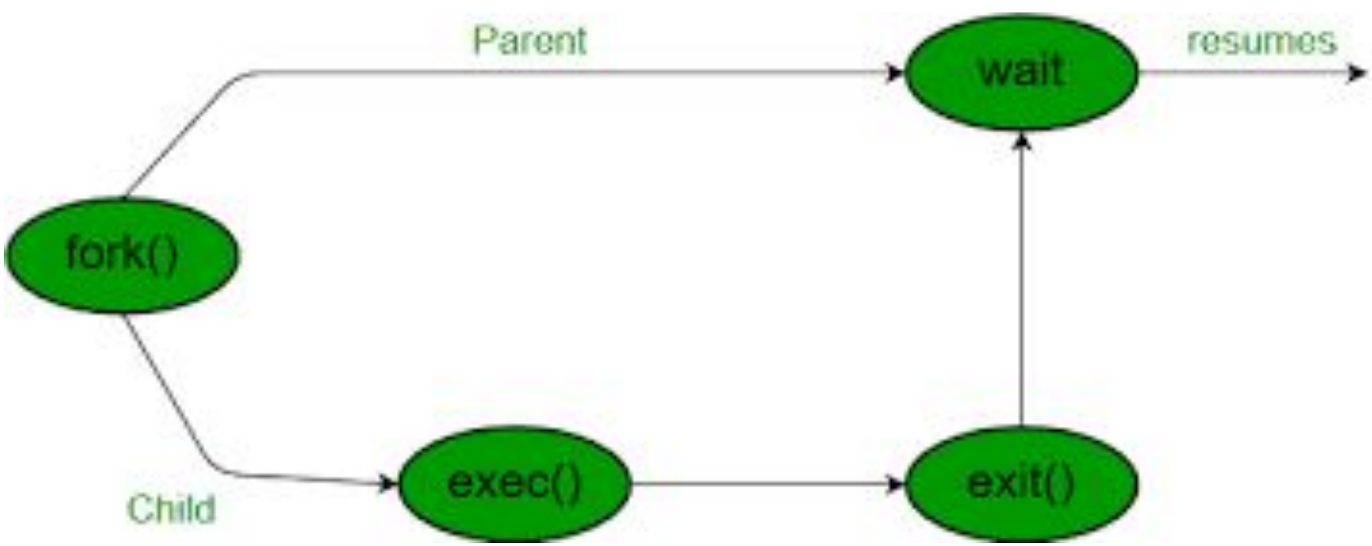
Creating a process

- Just one process – Parent process
- Initially there is one process – `init` with id = 1



Waiting for children to die with `wait()`

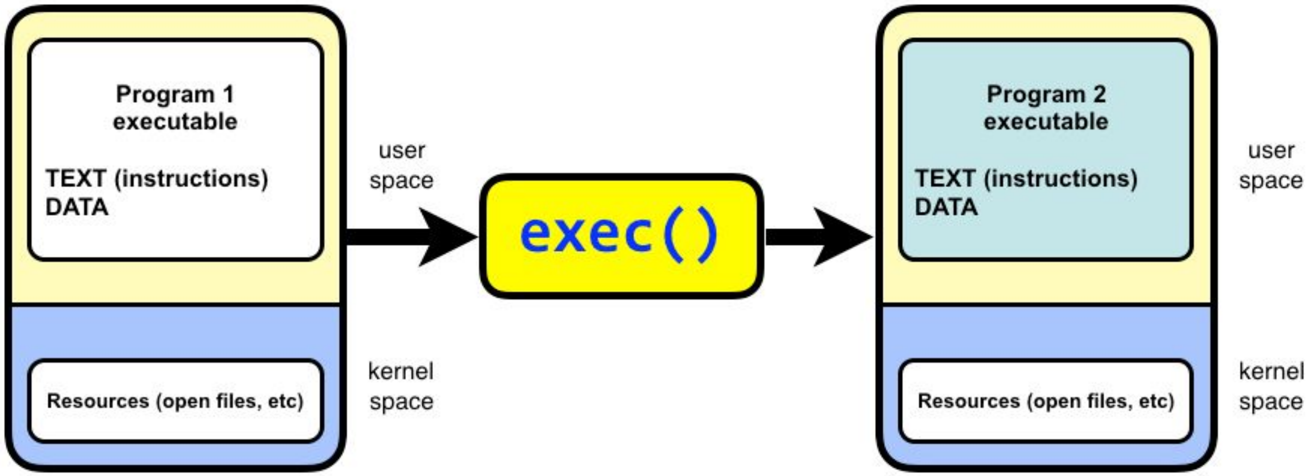
- The parent can wait for the child to die by executing the `wait` system call
- It is quite useful for a parent to wait for a child process to finish what it has been doing
  - on success, **returns** the process ID of the terminated child; on error, -1 is **returned**.



`Wait()`: Synchronizing with Children

- **`int wait(int *child_status)`**
  - Suspends current process (*i.e.* the parent) until one of its children terminates
  - Return value is the PID of the child process that terminated
    - *On successful return, the child process is reaped*
  - If `child_status != NULL`, then the `*child_status` value indicates why the child process terminated
    - Special macros for interpreting this status – see **`man wait(2)`**
    - When a child process terminates, `wait` stores the termination status of the terminated child (the value returned by main) into variable status, and returns the process number of the terminated child process
    - **`Null`**: Status value will not be stored

exec( ) system call



- The exec family of system calls replaces the program executed by a process
- When a process calls exec, all code (text) and data in the process is lost and replaced with the executable of the new program
- All open file descriptors remains open after calling exec
  - unless explicitly set to close-on-exec

p3.c

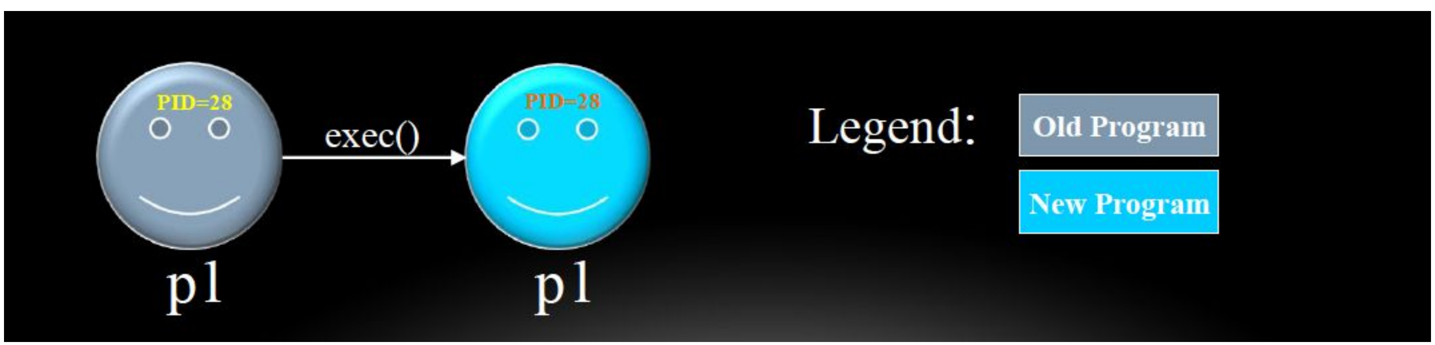
```c
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

Program Output

```
hello world (pid:25155)
hello, I am child (pid:25156)
      32     123     966 p3.c
hello, I am parent of 25156 (wc:25156) (pid:25155)
```

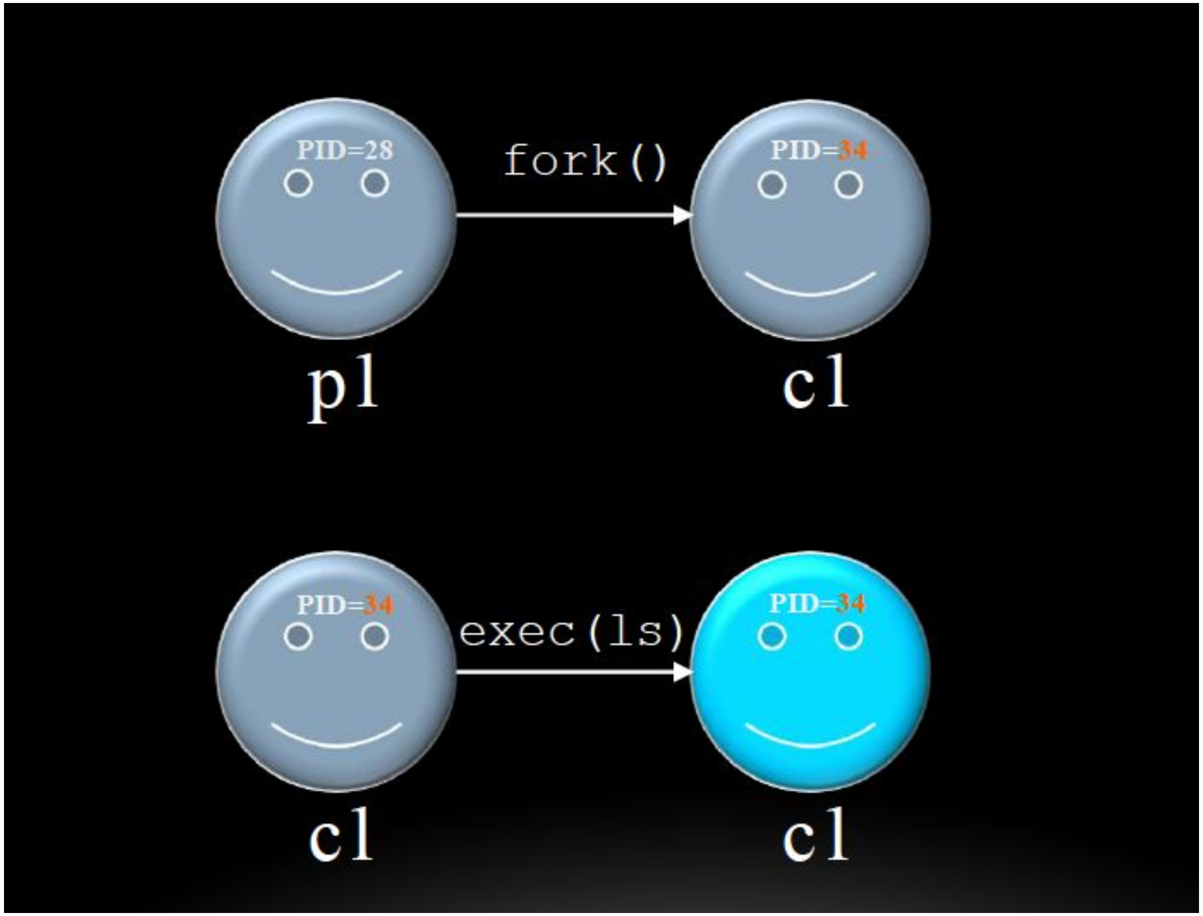
exec() – more information

- Upon success, exec() never returns to the caller
    - A successful exec replaces the current process image, so it cannot return anything to the program that made the call
- If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions
- As a new process is not created, the process identifier (PID) does not change
    - However, machine code, data, heap, and stack of the process are replaced by those of the new program



fork() and exec() combined

- Often after doing fork() we want to load a new program into the child



Zombies

- A terminated process still consumes system resources
  - Various tables maintained by OS
  - Called a "zombie" (a living corpse, half alive and half dead)
- *Reaping* is performed by parent on terminated child
  - Parent is given exit status information and kernel then deletes zombie child process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid of 1)
    - **Note:** on recent Linux systems, `init` has been renamed `systemd`
  - In long-running processes (*e.g.* shells, servers) we need *explicit* reaping

Zombie State

- Why keep process descriptor around?
  - Parent may be waiting for child to terminate
    - via the `wait()` system call
  - Parent needs to get the exit code of the child
  - If descriptor was destroyed immediately, this information could not be gotten
  - After getting this information, the process descriptor (or PCB) can be removed
    - no more remnants of the process

Process Summary

- `fork` makes two copies of the same process (parent & child)
  - Returns different values to the two processes
- `exec*` replaces current process from file (new program)
  - Runs two different programs:
    - First `fork()`
    - **if** (pid == 0) { `execv(...)` } **else** { */* parent code */* }

- `wait` or `waitpid` used to synchronize parent/child execution and to reap child process

Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Fully describes program state

Done

- **Address space**
  - Set of memory addresses accessible to program (for read or write)
- **Process: an instance of a running program**
  - Protected Address Space + One or more Threads

Next

- **Dual mode operation / Protection**
  - Only the "system" has the ability to access certain resources

BREAK (10 MINS)

Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Fully describes program state

Done

- **Address space**
  - Set of memory addresses accessible to program (for read or write)
- **Process: an instance of a running program**
  - ~~Protected Address Space + One or more Threads~~
- **Dual mode operation / Protection**
  - Only the "system" has the ability to access certain resources

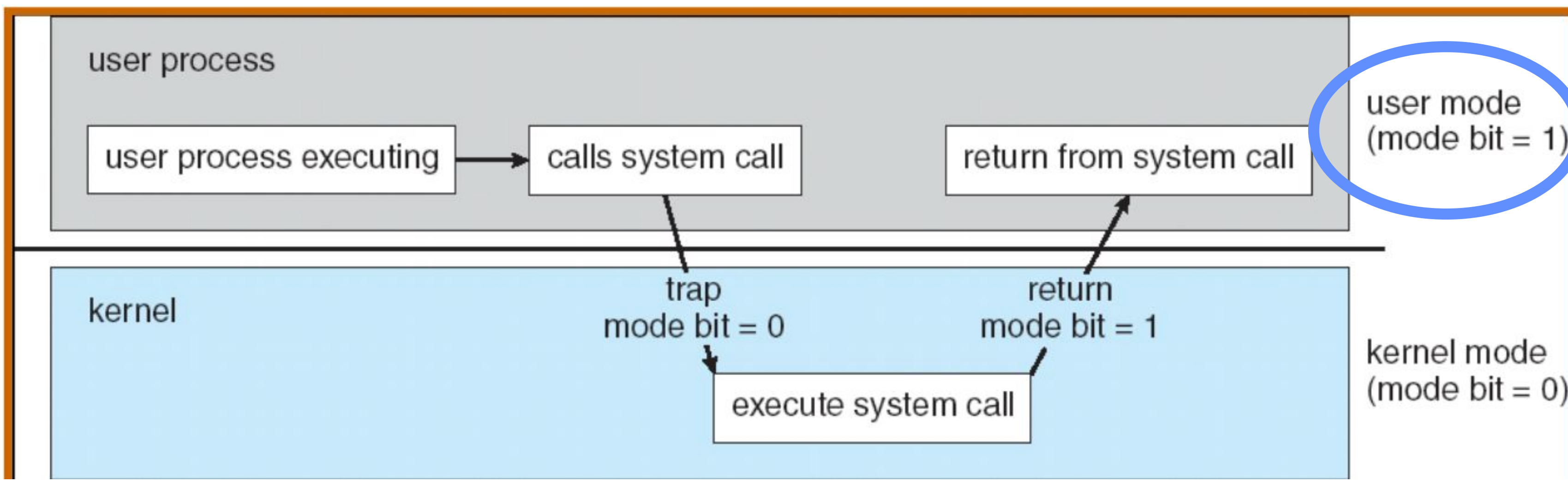
Dual Mode of Operation

Fourth OS Concept: Dual Mode Operation

- Hardware provides at least two modes:
  - Kernel mode (or supervisor/protected)
  - User mode: Normal programs executed
- Kernel mode
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
- User mode
  - Limited privileges

Dual Mode Operation

- Certain operations are prohibited when running in user mode
- Transitions between user mode and kernel mode are carefully controlled
- Mode bit in processor determines if system is in User mode or Kernel mode



Book: The Operating System Concepts: Silberschatz, Galvin, Gagne

Dual Mode Operation

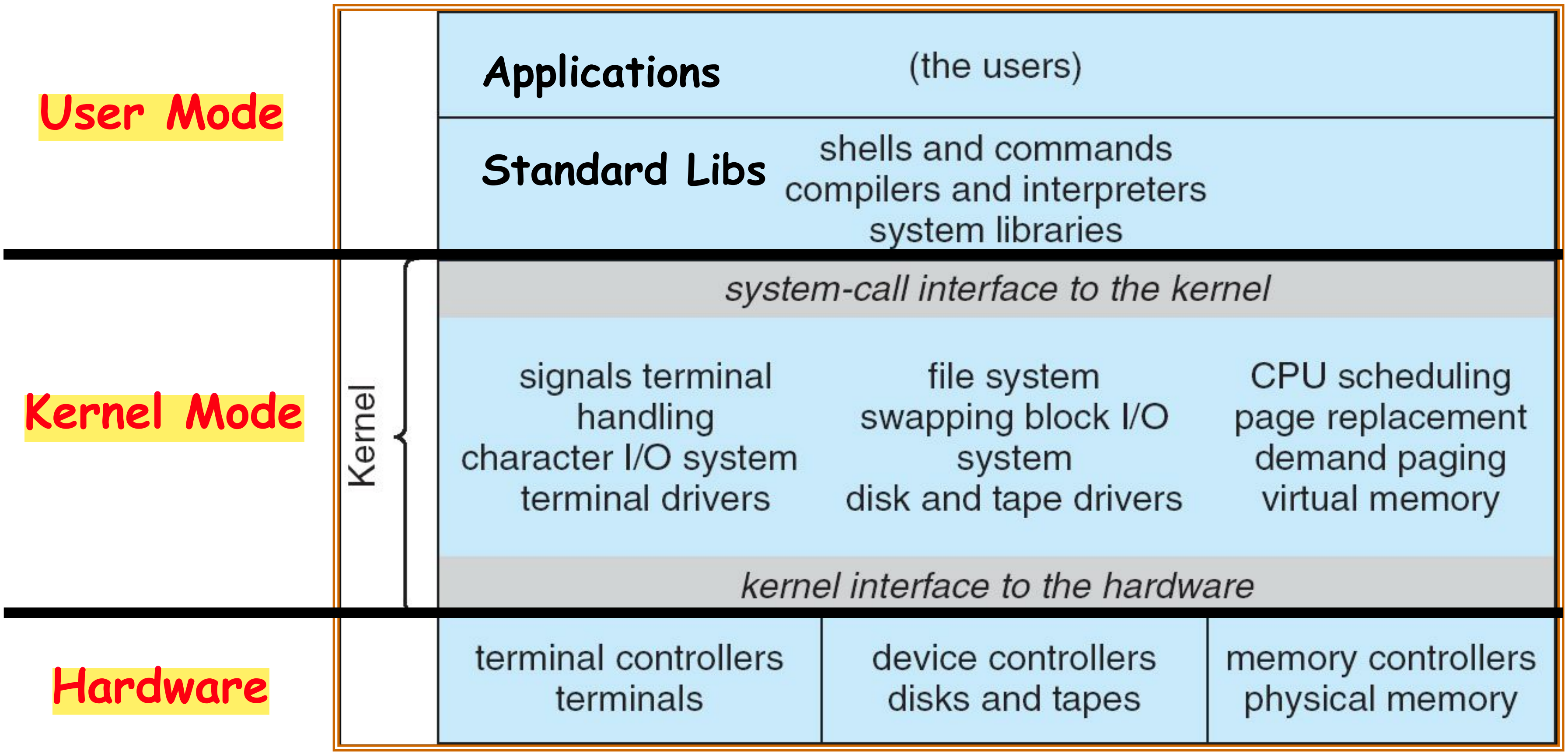
- In user mode, the processor checks every instruction before executing it to verify that it is permitted by the process to be performed
- What is needed in the hardware to support dual mode operation?
  - A bit of state (user/system mode bit)
  - Certain operations / actions only permitted in system/kernel mode
    - In user mode they fail or trap
  - User □ Kernel transition *sets* system/kernel mode AND saves the user PC
    - Operating system code carefully puts aside user state then performs the necessary operations
  - Kernel □ User transition *clears* system/kernel mode AND restores appropriate user PC
    - Example: return-from-interrupt

Aside: In RISC-V, where is this user/system mode bit ?

Mode bit

- Mode bit : one bit register
  - Kernel mode: Mode bit = 1
    - Processor is in kernel mode and it can do anything
  - User mode: Mode bit = 0
    - Processor is in user mode and it is restricted
- Where is this bit in the processor?
  - Processor Status Register (x86)
    - PSR contains flags that control processor's operation
    - Flags are set and reset as a by-product of executing instructions
    - Not accessible to applications
  - CPSR: Current Program Status Register (ARM Architecture)

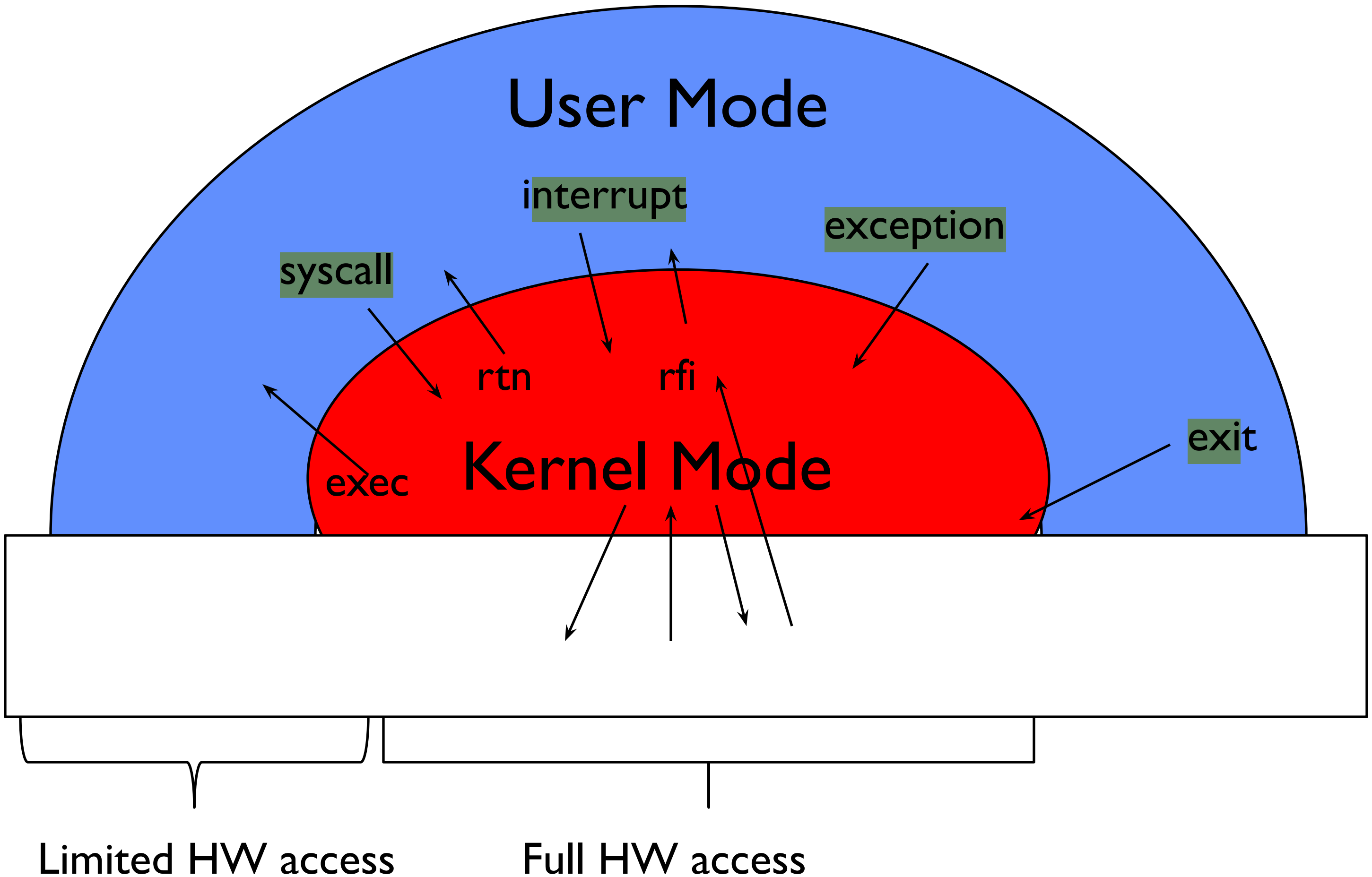
For example: UNIX System Structure



Monolithic Architecture

Book: The Operating System Concepts: Silberschatz, Galvin, Gagne

User/Kernel (Privileged) Mode



rfi : return from Interrupt

Hardware Support for Dual Mode Operation

Hardware Support: Dual-Mode Operation

- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - In user mode, all memory accesses outside of a process's valid memory region must be prohibited
  - Prevent user code from overwriting the kernel
- Timer Interrupts
  - Processor must have a way to regain control from a user program in a loop

Hardware Support: Dual-Mode Operation

- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - In user mode, all memory accesses outside of a process's valid memory region must be prohibited
  - Prevent user code from overwriting the kernel
- Timer Interrupts
  - Processor must have a way to regain control from a user program in a loop

Privileged instructions

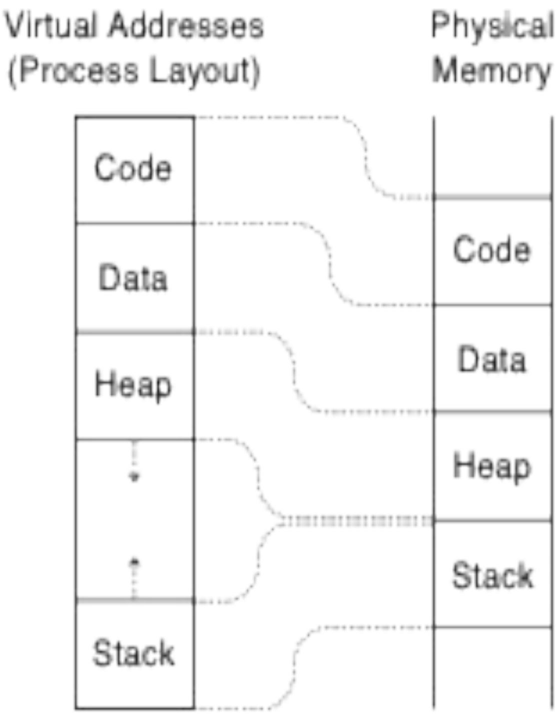
- Application programs can use only a subset of the full instruction set
- The operating system executes in kernel mode with the full power of the hardware
- Privileged Instructions: Instructions available in kernel mode, but not in user mode
  - I/O instructions and Halt instructions
  - Turn off all Interrupts
  - Set the Timer
  - Context Switching
  - Clear the Memory or Remove a process from the Memory
  - Modify entries in the Device-status table
- If an application in user mode attempts to access restricted memory, processor exception occurs □ Kernel mode □

Hardware Support: Dual-Mode Operation

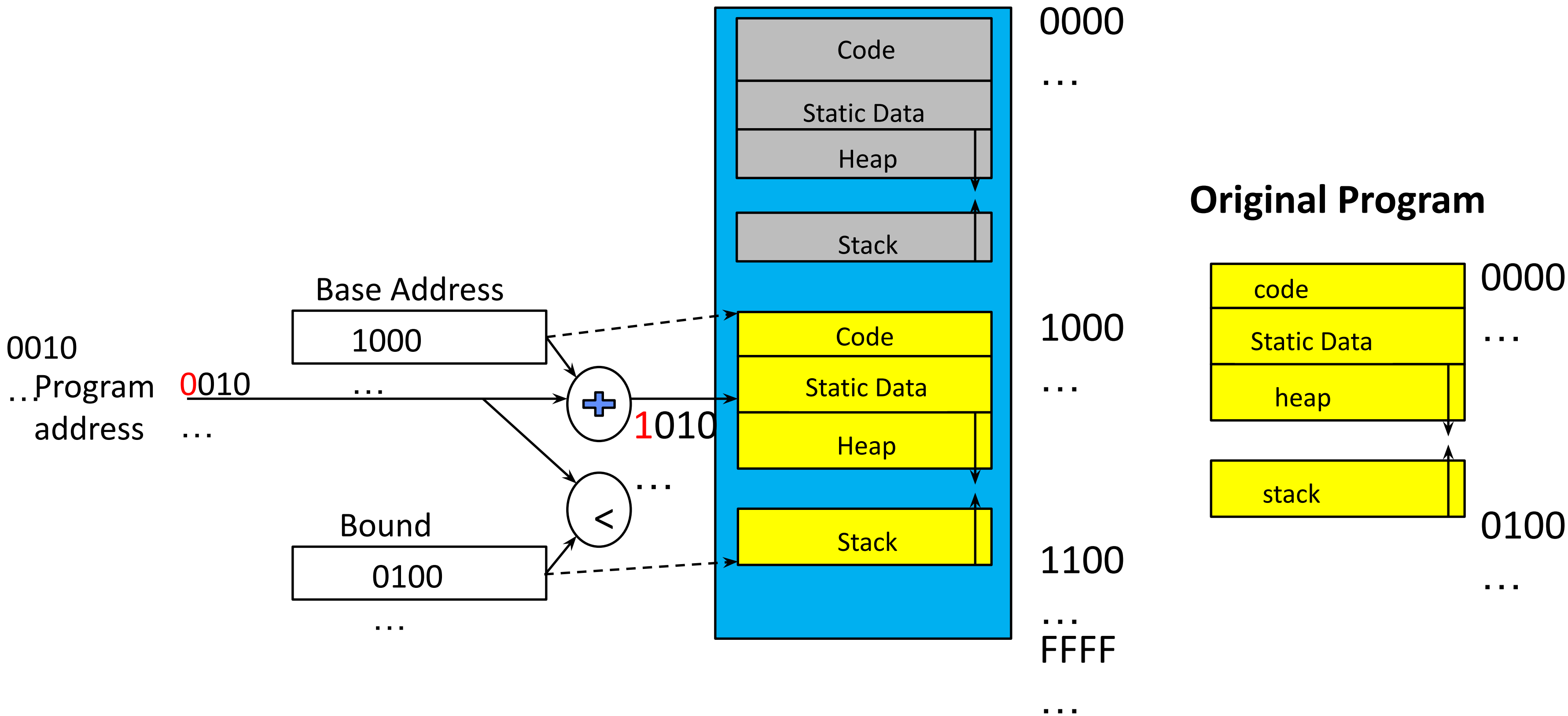
- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses (Memory Protection)
  - In user mode, all memory accesses outside of a process's valid memory region must be prohibited
  - Prevent user code from overwriting the kernel
- Timer Interrupts
  - Processor must have a way to regain control from a user program in a loop

Memory Protection

- OS must configure the hardware so that each application process can read and write only its own memory
  - Not the memory of other App or OS
- Approaches
  - Base and Bound
  - Virtual Address – Address space
- Virtual Address
  - Translation done in hardware, using a table
  - Table set up by operating system kernel



Base and Bound



Base and Bound

- Two extra registers: base and bound
  - Can be set only by OS
- Base register
  - Specifies the start of the process's memory region in physical memory
- Bound Register
  - Gives the end point of of the process's memory region in physical memory

Base and Bound - Limitations

- Two extra comparison for each instruction
- Contiguous memory location is needed to keep process image
- Difficult to manage growth of Heap and Stack (in opposite direction)
- No possible to share memory between two processes
- Memory fragmentation
- Relocation is hard – Entire memory image is to be located

Virtual Memory – Paging

- We have studied Paged Virtual Memory in detail – CA course
- All modern OS use this mechanism
- Paged Virtual Memory has many advantages
  - Instructions operate on virtual addresses
  - Translated at runtime to physical addresses via a page table
    - This allows relocation of pages at physical memory – Easily
  - Enormous amount of flexibility to manage physical memory
  - It allows the heap and the stack start at separate ends of the virtual address space
    - they can grow according to program needs
  - Higher Efficiency with the use of Page Tables, TLBs etc
  - Protection and Security
  - Sharing possible

Hardware Support: Dual-Mode Operation

- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - In user mode, all memory accesses outside of a process's valid memory region must be prohibited
  - Prevent user code from overwriting the kernel
- Timer Interrupts
  - Processor must have a way to regain control from a user program in a loop

Hardware Timer

- All computer systems include a device – Hardware timer
- Hardware timer periodically interrupts the processor
  - Returns control to the kernel handler
  - Interrupt frequency set by the kernel
    - Not by user code!
  - Interrupts can be temporarily deferred
    - Not by user code!
    - Interrupt deferral crucial for implementing mutual exclusion
      - Will study it later (Mutual Exclusion implementation topic)
- Each processor has a separate timer

Timer based mode switch

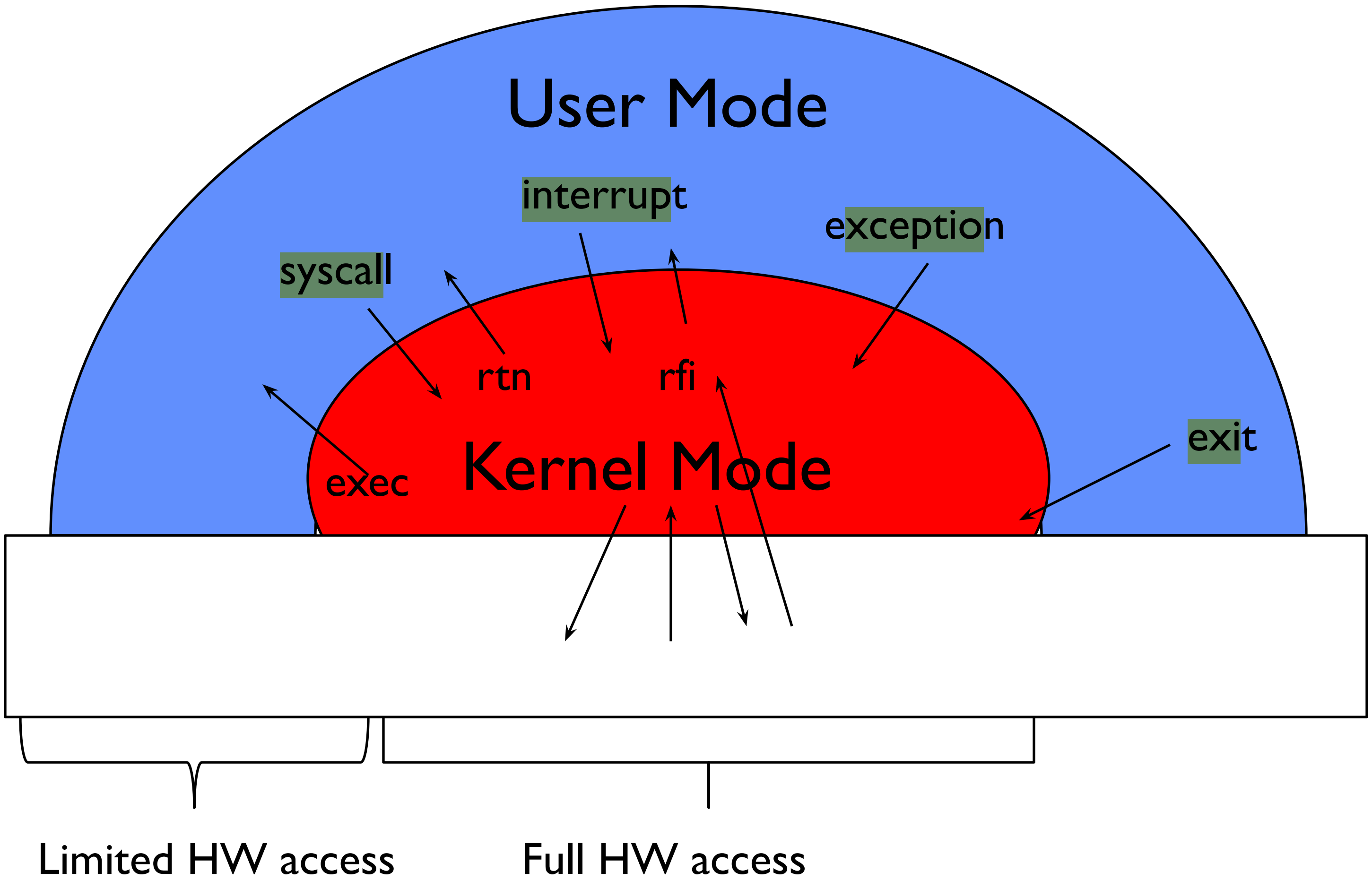
- For giving chance to multiple processes to run on CPU
  - Hardware must provide a way for the OS kernel to periodically regain control of the processor
- Example:
  - OS starts a user level process
  - User process is free to run any user level instructions it wants, call any function in process's memory region, and load/store any value in it's memory
  - Does User process have complete control of the hardware forever?
    - To break infinite loop of the application, OS must take control

Helping Kernel to gain control

- How does the kernel know if an application is in an infinite loop?
  - Kernel doesn’t know it
- Then would OS kill process after gaining control ?
  - No
  - It doesn’t know ; May be this is intended functionality
- Then.. What ?
  - It terminates the process when requested by the user or system administrator

Mode Transfer: User to Kernel

User to Kernel mode Transfer



User Kernel Mode Transfer

- System Call (syscalls)
  - User Process requests a system service
    - Open or delete a file, read/write data into files, create a new user process, establish a connection to web server etc
- Interrupt
  - External asynchronous event, independent of the process
  - e.g., Timer, I/O device
- Processor Exception (trap)
  - Hardware event caused by user program behavior that causes context switch
  - E.g., Divide by zero, bad memory access (segmentation fault)

Dual Mode: Summary

- Most of OS kernels support dual mode of operation
  - User Mode
  - Kernel Mode
- A single bit called Mode bit is set by hardware that defines if the mode is user or kernel
- Dual mode operation is implemented with the support of
  - Privileged Instructions – that Kernel executes
  - Limits on Memory Protection
  - Use of Timer Interrupts

Backup

Mode Transfer: User to Kernel -

Synchronous Exception

System Calls

- Each x86-64 system call has a unique ID number
- Examples:

Number	*Name*	*Description*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

Mode Transfer: User to Kernel -

Interrupt (asynchronous Exception)

Interrupts (Asynchronous Exception)

- An asynchronous signal to the processor
  - Some external event requires processor's attention
  - Indicated by setting the processor's *interrupt pin*
  - Processor stalls or completes existing instruction that is in progress; Saves current execution state; Starts execution of specially designated interrupt handler in the kernel
- Each different type of interrupt requires its own handler
- Example:
  - Timer Interrupt: Every few ms; an external timer chip triggers an interrupt ; Timer handler can switch execution to different process
  - I/O interrupt eg from mouse, keyboard, disk, Ethernet, WiFi, Flash drive, Inter-processor interrupts, DMA, arrival of a packet from a disk
  - Inter-processor Interrupt: For inter processor communication

Processor Exceptions

- A hardware event caused by user program behavior
- Causes transfer of control to the Kernel
- Processor saves the current execution state and runs specially designated exception handler in the kernel
- Example:
  - User process attempts to execute a privileged instruction
  - User process tries to access memory out of it’s own memory region
  - Process divides an integer by zero
  - Process attempts to write to read-only memory
  - A benign event: setting up a breakpoint
- Processor exceptions are used effectively to emulate VMs

Lecture Summary

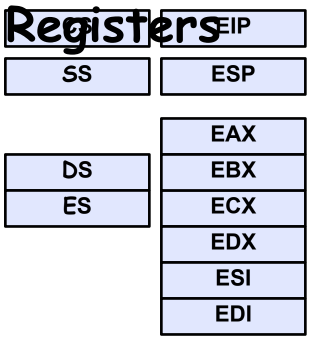
- The four important concepts of Operating Systems include
  - Thread
  - Address Space
  - Process
  - Dual mode of operation / Protection

Exceptions cause Mode Transfer

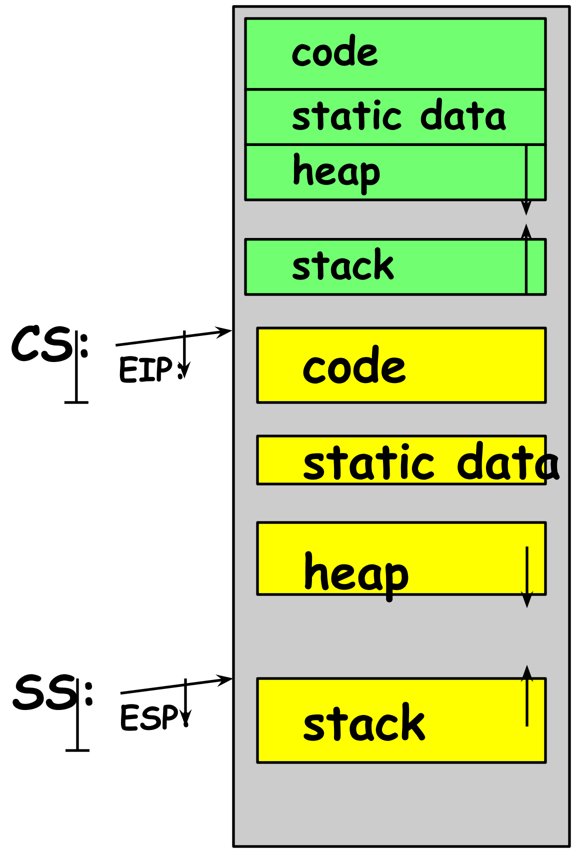
- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but “outside” the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- Trap or Exception
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …
- Interrupt
  - External asynchronous event triggers context switch
  - e. g., Timer, I/O device
  - Independent of user process

x86 – segments and stacks

Processor Registers



Start address, length and access rights associated with each segment register



- How does the OS kernel prevent a process from harming another process ?
- When there are multiple programs in Main Memory
  - What prevents a process from overwriting another process's data structures, or
  - Overwriting the OS image stored on disk?
- Recall RISC-V instructions
  - Most instructions such add, sub etc are perfectly safe
  - How can we allow them to execute directly on hardware?
- We implement as simple check in hardware called dual-mode operation
  - Represented by a single bit in the **processor status register** that signifies the current mode of the processor