RUTGERS UNIVERSITY
Computer Sciences Department

CS 416 Operating Systems Design

Sudarsun Kannan

Administrative

Announcements

- Midterm – 03/06 (in class)
- 1st Project to released this weekend
- 1st Homework (Sakai Quiz) early next week
- Attend recitations and ask questions!

RUTGERS UNIVERSITY
Computer Sciences Department

CS 416 Operating Systems Design

Sudarsun Kannan

Virtualization: The CPU

Questions answered in this lecture:

- What is a process?
- Why is limited direct execution a good approach for virtualizing the CPU?
- What execution state must be saved for a process?
- What 3 modes could a process in?

What is a Process?

Process: An execution stream in the context of a process state

What is an execution stream?

- Stream of executing instructions
- Running piece of code
- "thread of control"

What is process state?

- Everything that the running code can affect or be affected by
- Registers
  - General purpose, floating point, status, program counter, stack pointer
- Address space
  - Heap, stack, and code
- Open files

Processes vs. Programs

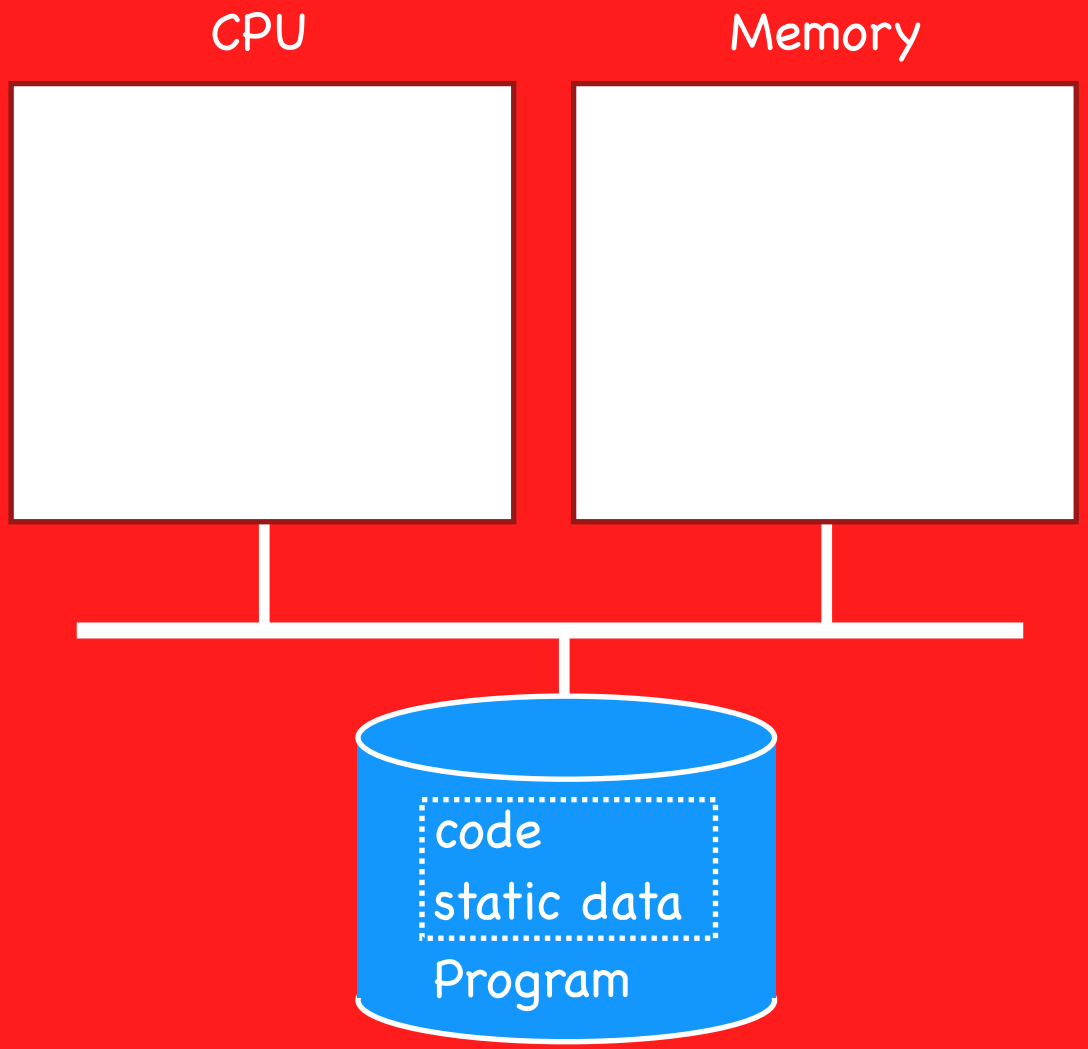
A process is different than a program

- Program: Static code and static data
- Process: Dynamic instance of code and data

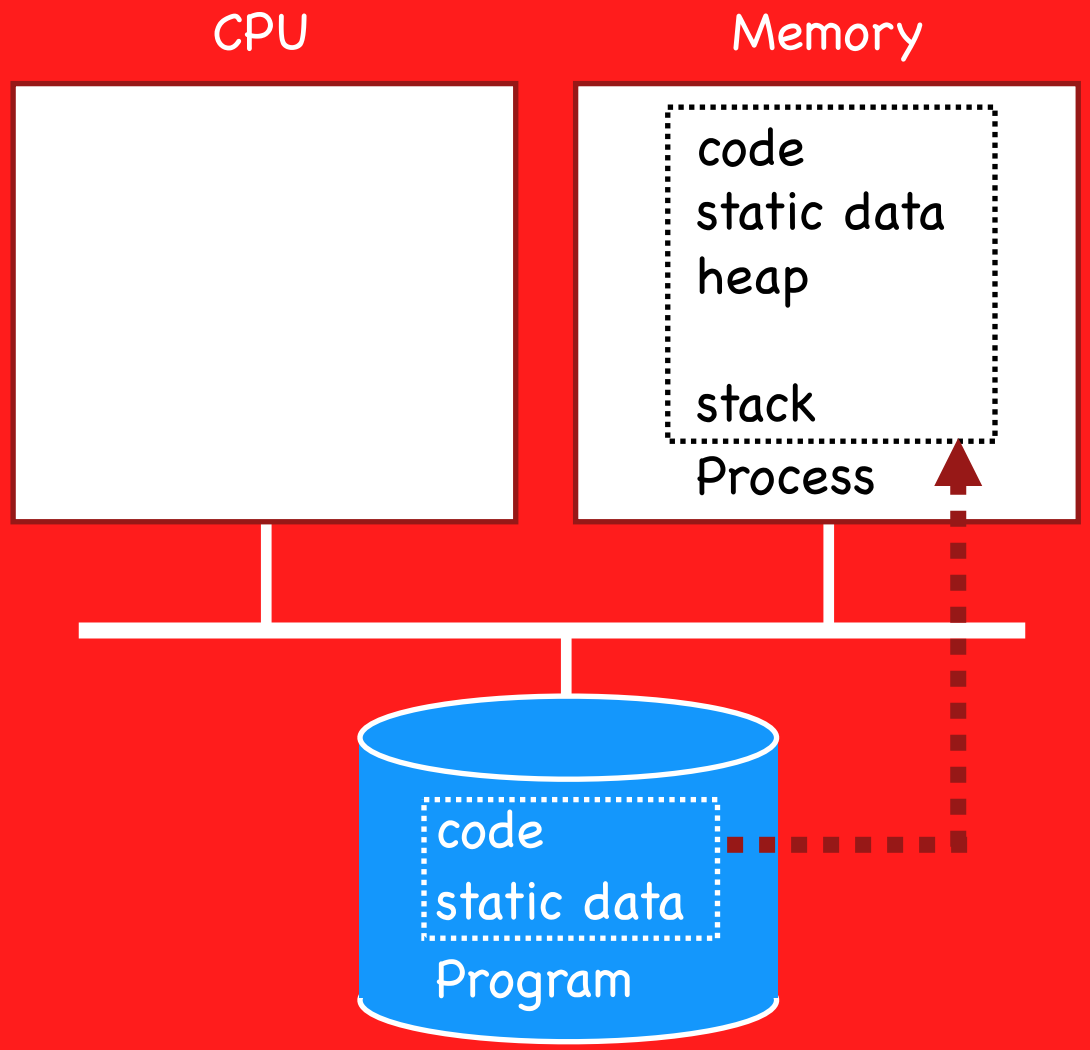
Can have multiple process instances of same program

- Can have multiple processes of the same program
  Example: many users can run "ls" at the same time

Process Creation



Process Creation

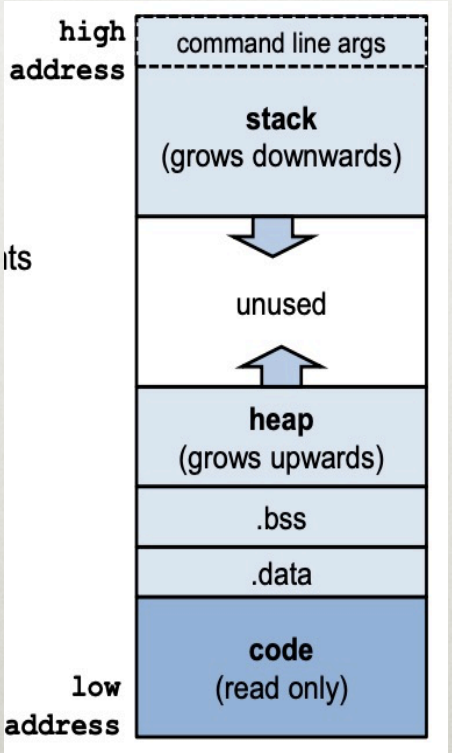


Processes vs. Threads

- A process is different than a thread
- Thread: "Lightweight process" (LWP)
  - An execution stream that shares an address space
  - Multiple threads within a single process
- Example:
  - Two **processes** examining same memory address 0xffe84264 see **different** values (I.e., different contents)
  - Two **threads** examining memory address 0xffe84264 see **same** value (I.e., same contents)

Process Memory Segments

- The OS allocates memory for each process - ie. a running program – for data and code
- This memory consists of different segments
- Stack - for local variables – incl. command line arguments and environment variables
- Heap - for dynamic memory
- Data segment for – global uninitialised variables (.bss) – global initialised variables (.data)
- Code segment typically read-only



Virtualizing the CPU

Goal:
Give each process impression it alone is actively using CPU

Resources can be shared in **time** and **space**

Assume single uniprocessor
- Time-sharing (multi-processors: advanced issue)

Memory?
- Space-sharing (later)

Disk?
- Space-sharing (later)

How to Provide Good CPU Performance?

Direct execution

- Allow user process to run directly on hardware
- OS creates process and transfers control to starting point (i.e., main())

Problems with direct execution?

1. Process could do something restricted
   Could read/write other process data (disk or memory)
2. Process could run forever (slow, buggy, or malicious)
   OS needs to be able to switch between processes
3. Process could do something slow (like I/O)
   OS wants to use resources efficiently and switch CPU to other process

Solution:

Limited direct execution – OS and hardware maintain some control

Problem 1: Restricted OPS

How can we ensure user process can't harm others?

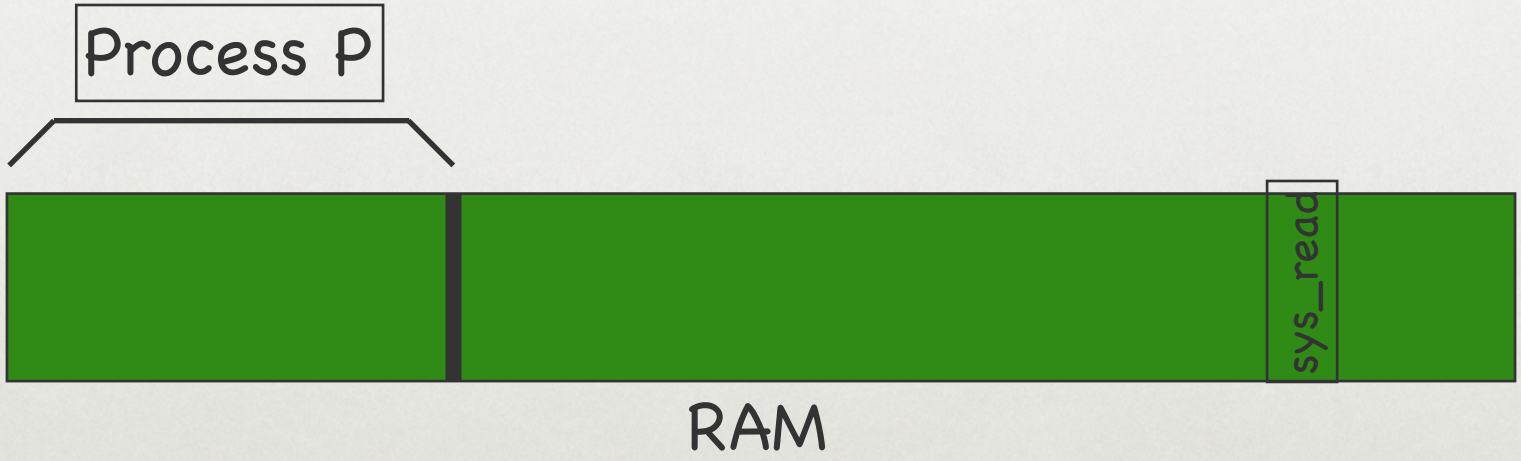
Solution: privilege levels supported by hardware (bit of status)

- User processes run in user mode (restricted mode)
- OS runs in kernel mode (not restricted)
  - Instructions for interacting with devices
  - Could have many privilege levels (advanced topic)

How can process access device?

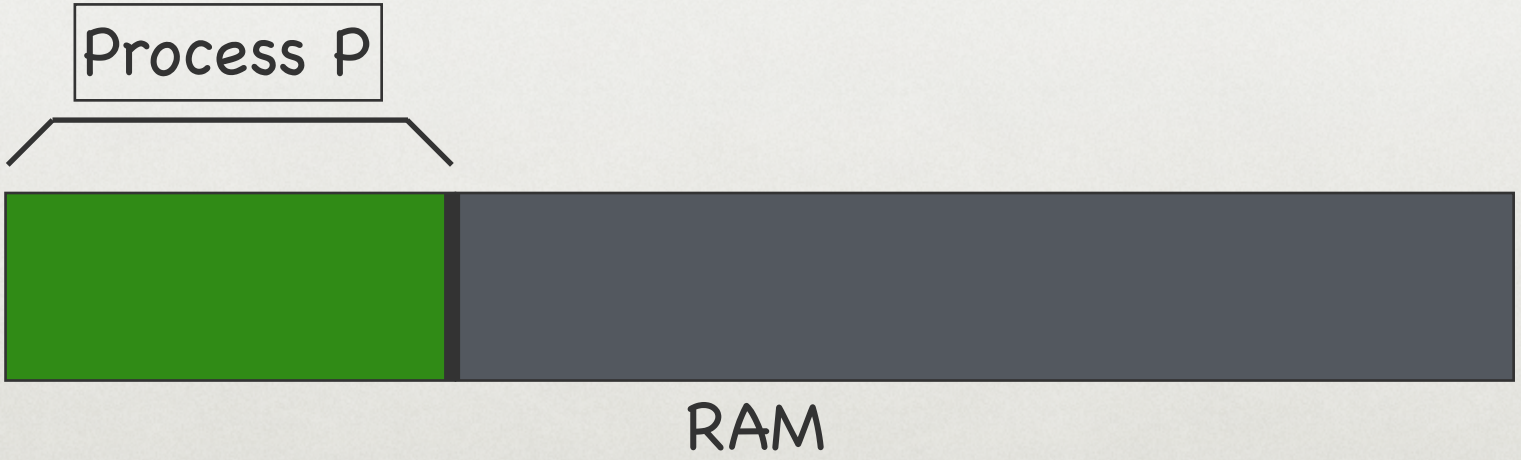
- System calls (function call implemented by OS)
- Change privilege level through system call (trap)

System Call



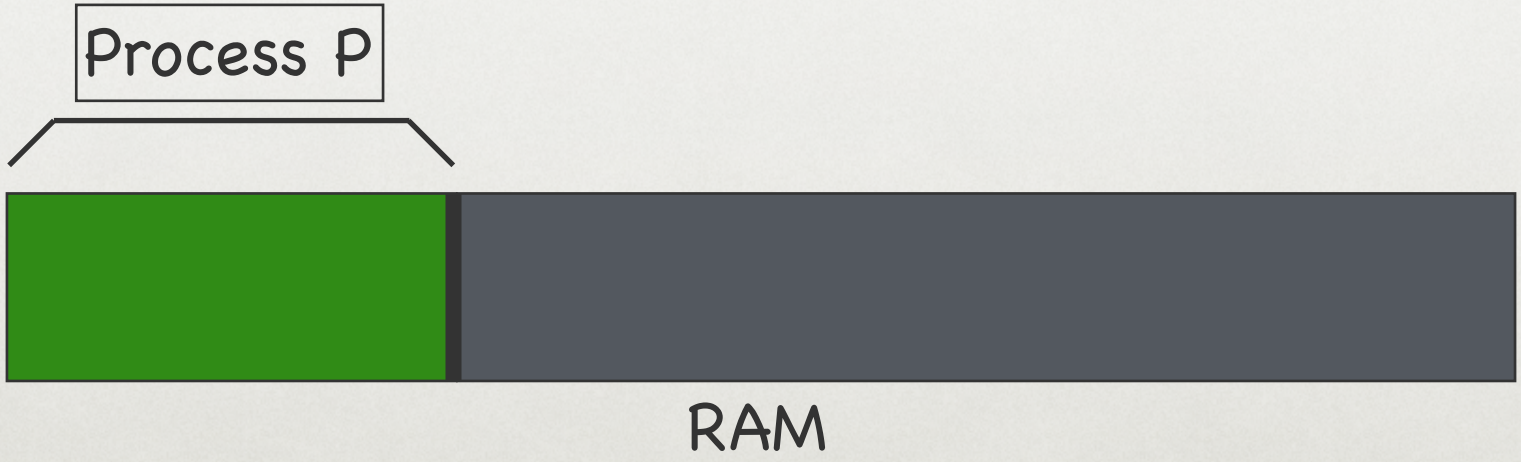
P wants to call read()

System Call



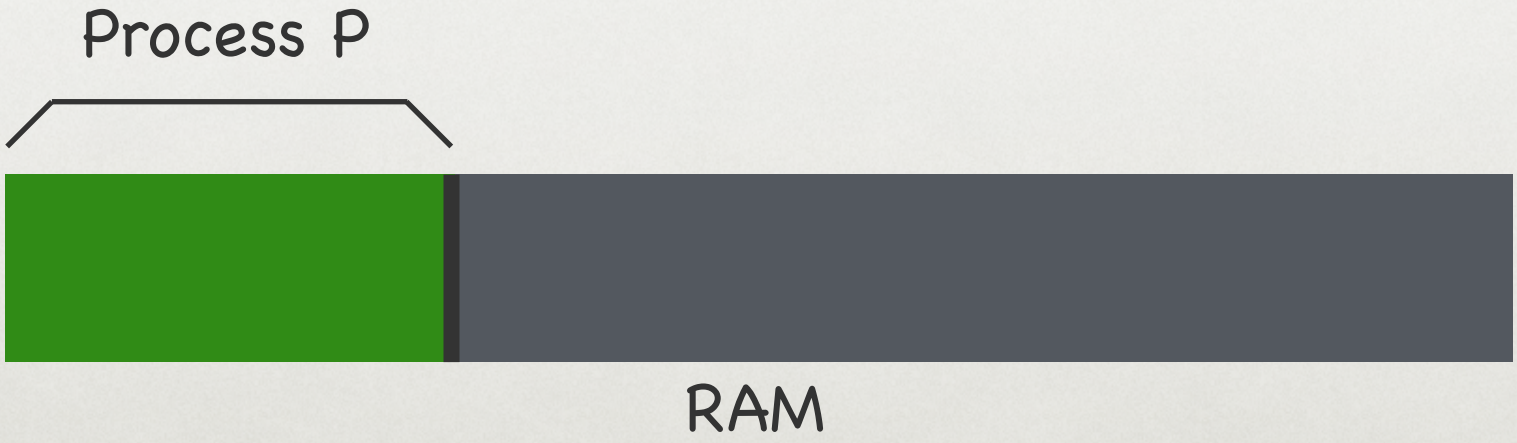
P can only see its own memory because of **user mode** (other areas, including kernel, are hidden)

System Call



P wants to call read() but no way to call it directly

System Call



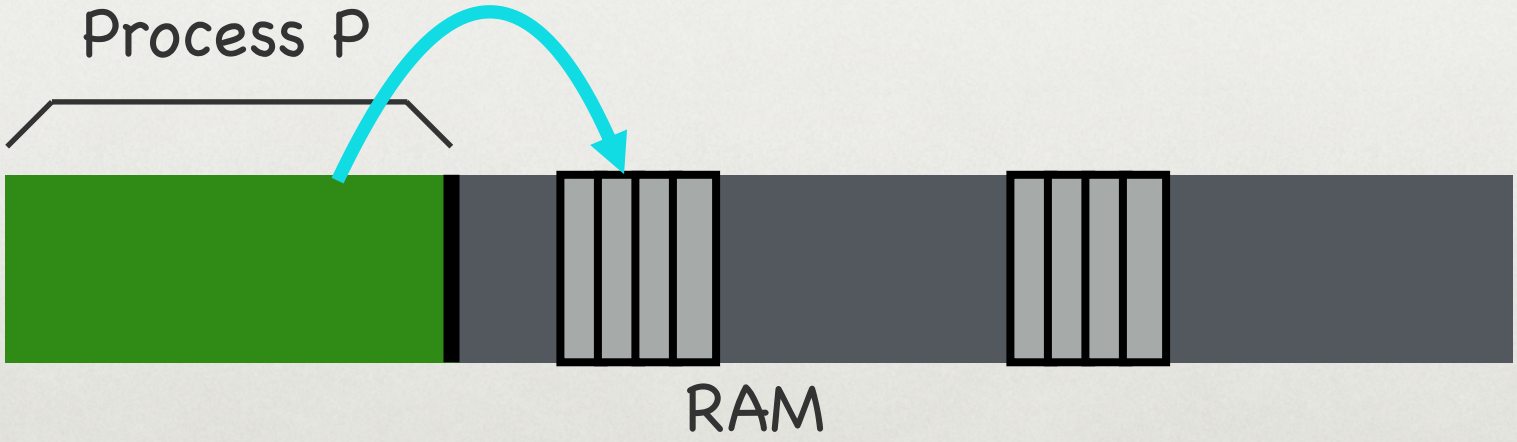
read():

```
movl $6, %eax;    int $64
```

movl %eax, …

- CPU uses contents of EAX register as source operand

System Call

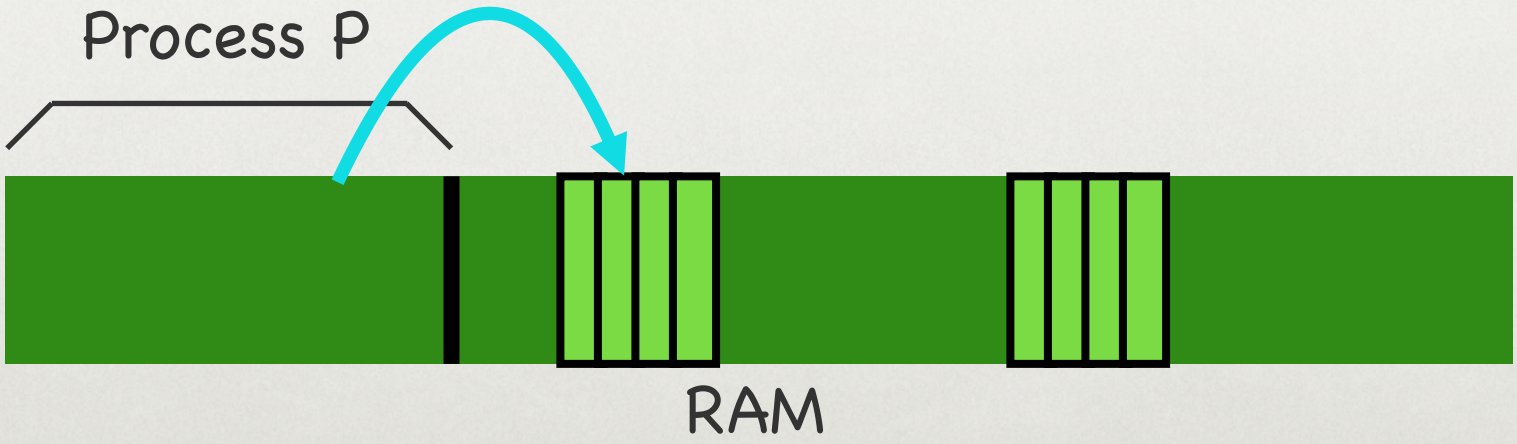



```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

SYSTEM CALL



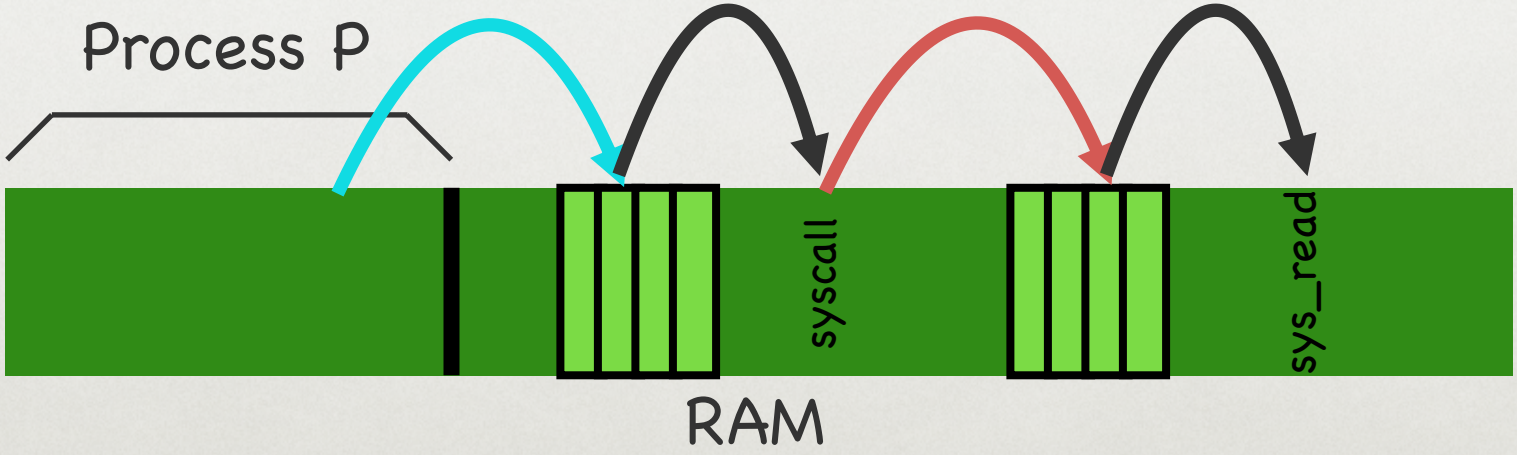

```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

Kernel mode: we can do anything!

System Call



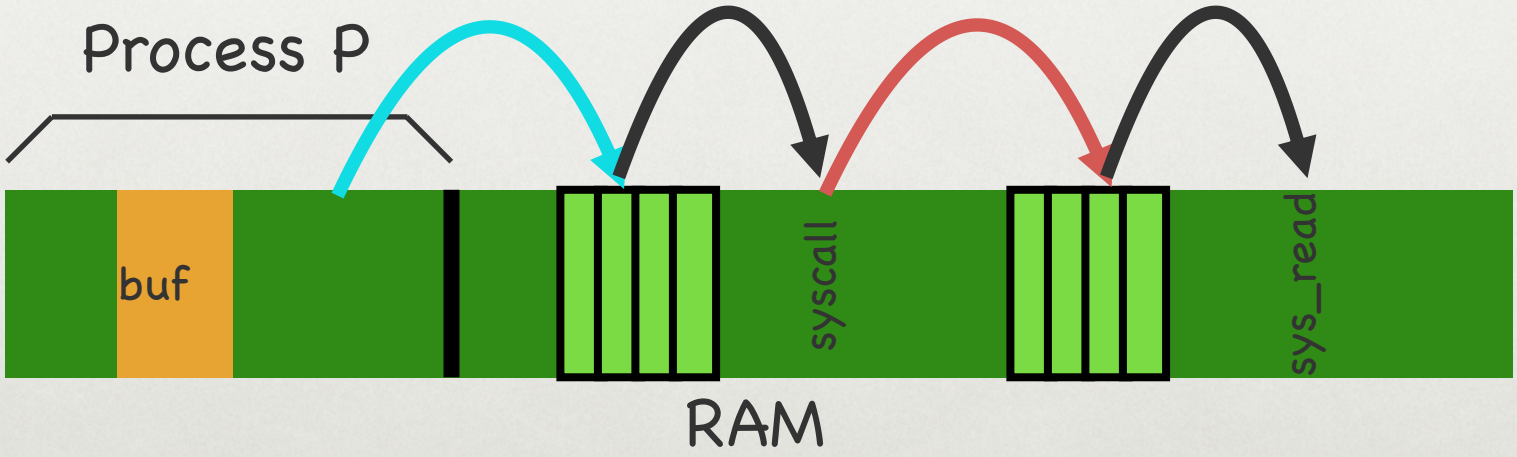

```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

Follow entries to correct system call code

System Call




```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

Kernel can access user memory to fill in user buffer
return-from-trap at end to return to Process P

System Call

App

System Call `movl $6, %eax;` `int $64`

H/W-level Trap Table

$63	illegal access
$64	system call
$65	Device Interrupt

OS

```
Syscall() {
  sysnum = %eax
  sys_handle= get_fn_table(sysnum)
  sys_handle ();
}
```

Syscall table

Num	Function
6	sys_read
7	sys_write

What to limit?

User processes are not allowed to perform:

- General memory access
- Disk I/O
- Special x86 instructions like `lidt`

What if process tries to do something restricted?

Problem 2: How to take CPU AWAY?

OS requirements for **multiprogramming** (or multitasking)

- Mechanism
  - To switch between processes
- Policy
  - To decide which process to schedule when

Separation of policy and mechanism

- Reoccuring theme in OS
- Policy: Decision-maker to optimize some workload performance metric
  - Which process when?
  - Process **Scheduler**: Future lecture
- Mechanism: Low-level code that implements the decision
  - How?
  - Process **Dispatcher**: Today's lecture

Dispatch Mechanism

OS runs dispatch loop

```
while (1) {
        run process A for some time-slice
        stop process A and save its context
        load context of another process B
}
```

Question 1: How does dispatcher gain control?

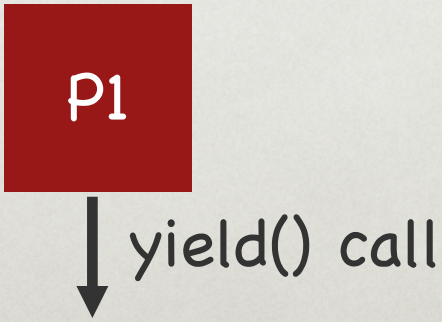
Question 2: What execution context must be saved and restored?

Q1: How does Dispatcher get CONTROL?

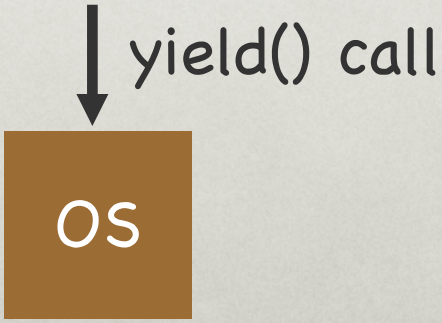
Option 1: Cooperative Multi-tasking

- Trust process to relinquish CPU to OS through traps
  - Examples: System call, page fault (access page not in main memory), or error (illegal instruction or divide by zero)
  - Provide special `yield()` system call

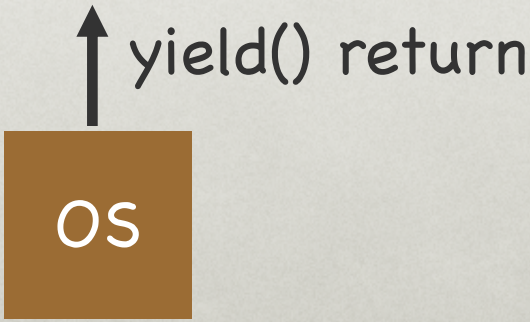
Cooperative Approach



Cooperative Approach

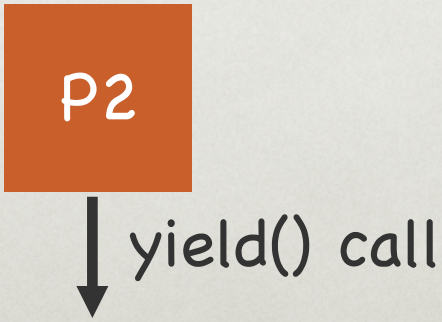


Cooperative Approach



Cooperative Approach

Cooperative Approach



Q1: How Does Dispatcher Run?

- Problem with cooperative approach?
- Disadvantages: Processes can misbehave
  - By avoiding all traps and performing no I/O, can take over entire machine
  - Only solution: Reboot!
- Not performed in modern operating systems

Q1: How does Dispatcher run?

Option 2: True Multi-tasking

- Guarantee OS can obtain control periodically
- Enter OS by enabling periodic alarm clock
  - Hardware generates timer interrupt (CPU or separate chip)
  - Example: Every 10ms
- User must not be able to mask timer interrupt
- Dispatcher counts interrupts between context switches
  - Example: Waiting 20 timer ticks gives 200 ms time slice
  - Common time slices range from 10 ms to 200 ms

Q2: What Context must be Saved?

Dispatcher must track context of process when not running

- Save context in process control block (PCB) (or, process descriptor)
- PCB is a structure maintained for each process in the OS

What information is stored in PCB?

- PID
- Process state (I.e., running, ready, or blocked)
- Execution state (all registers, PC, stack ptr) -- context
- Scheduling priority
- Accounting information (parent and child processes)
- Credentials (which resources can be accessed, owner)
- Pointers to other allocated resources (e.g., open files)

Requires special hardware support

- Hardware saves process PC and PSR on interrupts

Q2: What Context must be Saved?

Dispatcher must track context of process when not running

- Save context in process control block (PCB) (or, process descriptor)
- PCB is a structure maintained for each process in the OS

What information is stored in PCB?

- PID
- Process state (I.e., running, ready, or blocked)
- **Execution state (all registers, PC, stack ptr) -- context**
- Scheduling priority
- Accounting information (parent and child processes)
- Credentials (which resources can be accessed, owner)
- Pointers to other allocated resources (e.g., open files)

Requires special hardware support

- Hardware saves process PC and PSR on interrupts

Q3: What's inside a PCB?

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                  // Start of process memory
    uint sz;                    // Size of process memory
    char *kstack;               // Bottom of kernel stack
                                // for this process
    enum proc_state state;      // Process state
    int pid;                    // Process ID
    struct proc *parent;        // Parent process
    int killed;                 // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;          // Current directory
    struct context context;     // Switch here to run process
    struct trapframe *tf;       // Trap frame for the
                                // current interrupt
};
```

Q3: What's inside a PCB?

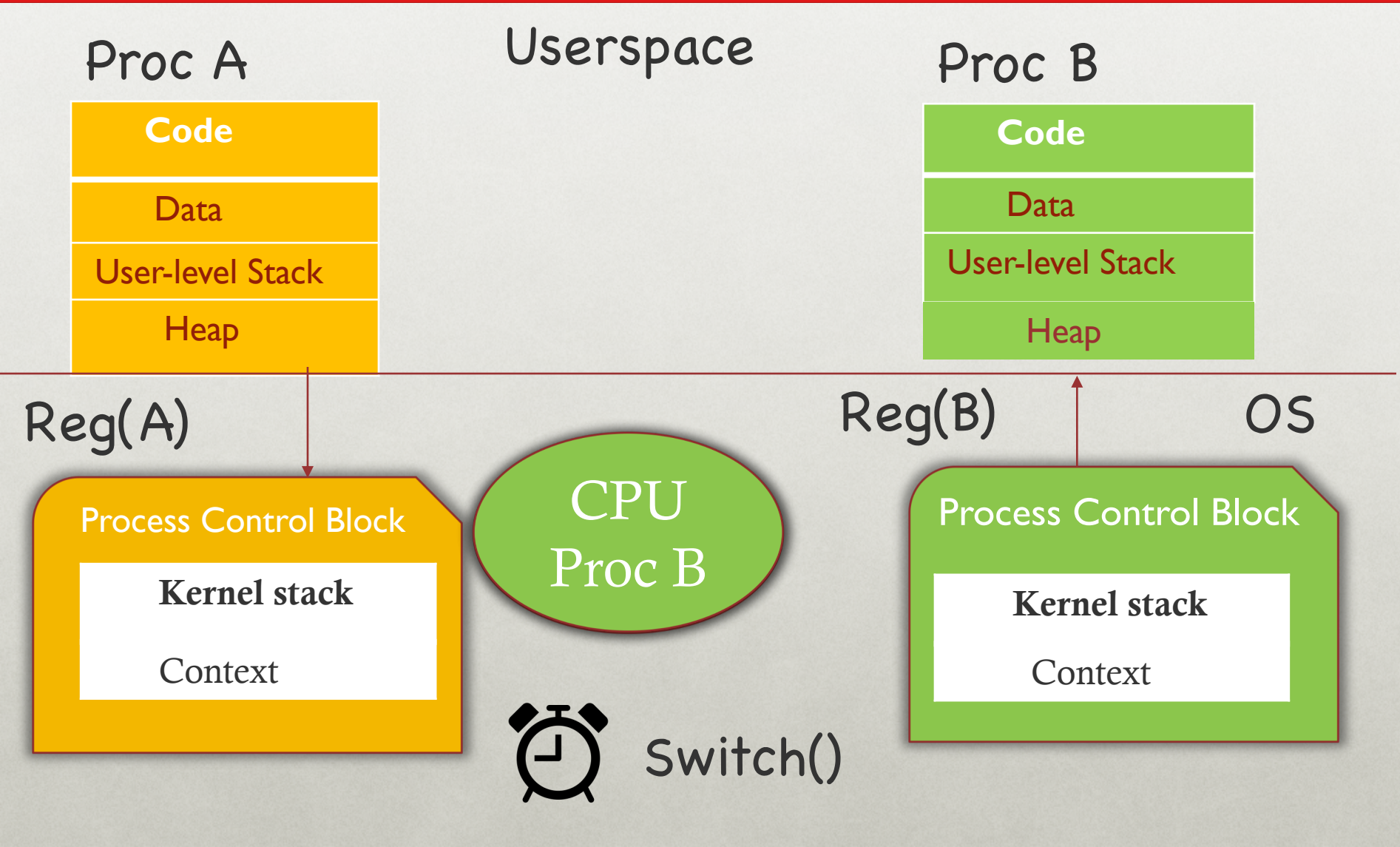
```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                  // Start of process memory
    uint sz;                    // Size of process memory
    char *kstack;               // Bottom of kernel stack
                                // for this process
    enum proc_state state;      // Process state
    int pid;                    // Process ID
    struct proc *parent;        // Parent process
    int killed;                 // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;          // Current directory
    struct context context;     // Switch here to run process
    struct trapframe *tf;       // Trap frame for the
                                // current interrupt
};
```

Q3: How Context is Saved?

Example:

- Process A has moved from user to kernel mode, OS decides it must switch from A to B
- Save context (PC, registers, kernel stack pointer) of A on kernel stack
- Switch SP to kernel stack of B
- Restore context from B's kernel stack
- Who has saved registers on B's kernel stack?
- – OS did, when it switched out B in the past
- Now, CPU is running B in kernel mode, return-from-trap to switch to user mode of B

Q3: How Context is Saved



Q4: What Context must be Saved?

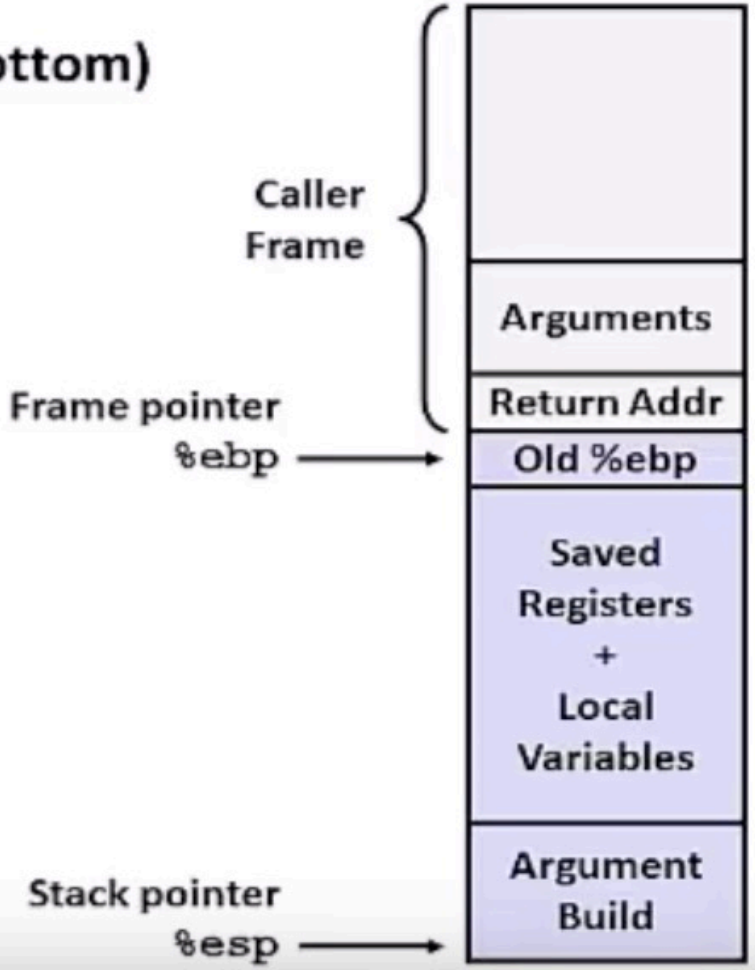
```
// the registers will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;   // Index pointer register
    int esp;   // Stack pointer register
    int ebx;   // Called the base register
    int ecx;   // Called the counter register
    int edx;   // Called the data register
    int esi;   // Source index register
    int edi;   // Destination index register
    int ebp;   // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

Stack Detour

IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build" area (parameters for function about to be called)
  - Local variables (if can't be kept in registers)
  - Saved register context (when reusing registers)
  - Old frame pointer (for caller)
- **Caller's Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



Stack Detour

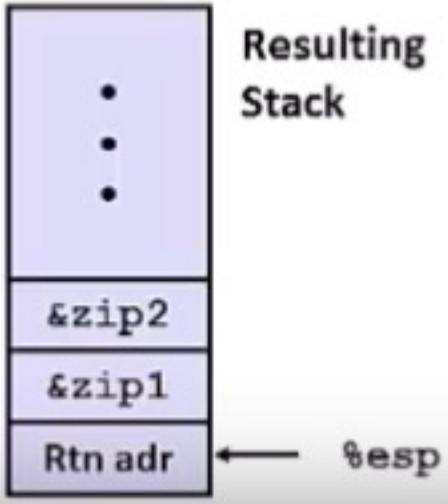
```
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Calling swap from call_swap

```
call_swap:
    • • •
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    • • •
```



Stack Detour

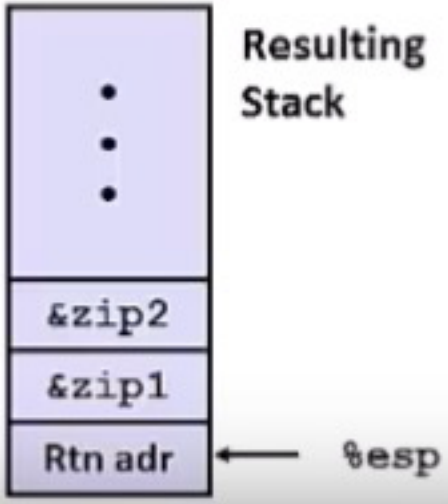
```
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Calling swap from call_swap

```
call_swap:
    • • •
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    • • •
```



Stack Detour

Disassembled swap

```
080483a4 <swap>:
 80483a4:    55             push   %ebp
 80483a5:    89 e5          mov    %esp,%ebp
 80483a7:    53             push   %ebx
 80483a8:    8b 55 08       mov    0x8(%ebp),%edx
 80483ab:    8b 4d 0c       mov    0xc(%ebp),%ecx
 80483ae:    8b 1a          mov    (%edx),%ebx
 80483b0:    8b 01          mov    (%ecx),%eax
 80483b2:    89 02          mov    %eax,(%edx)
 80483b4:    89 19          mov    %ebx,(%ecx)
 80483b6:    5b             pop    %ebx
 80483b7:    c9             leave
 80483b8:    c3             ret
```

Calling Code

```
 8048409:    e8 96 ff ff ff    call 80483a4 <swap>
 804840e:    8b 45 f8          mov  0xfffffff8(%ebp),%eax
```

Stack Detour

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

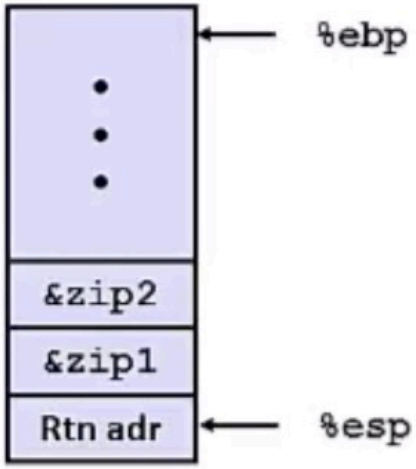
Body

Finish

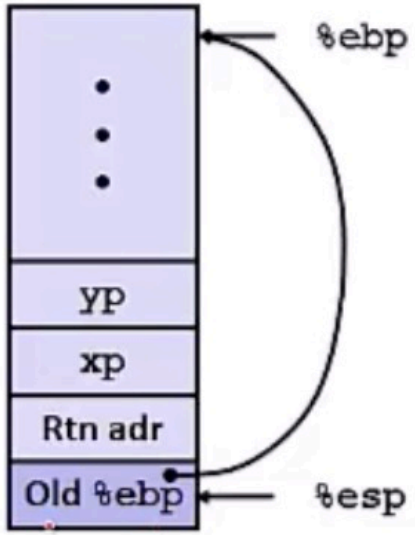
Stack Detour

swap Setup #1

Entering Stack



Resulting Stack

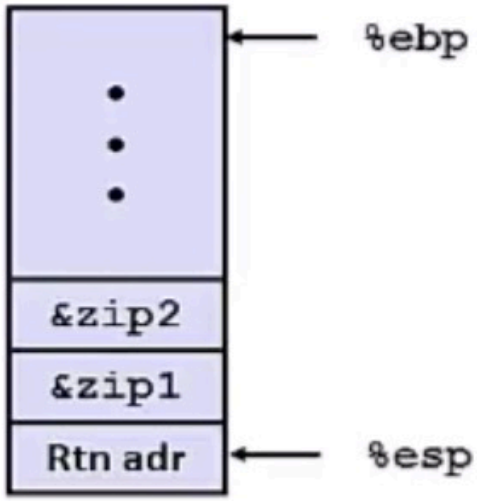



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

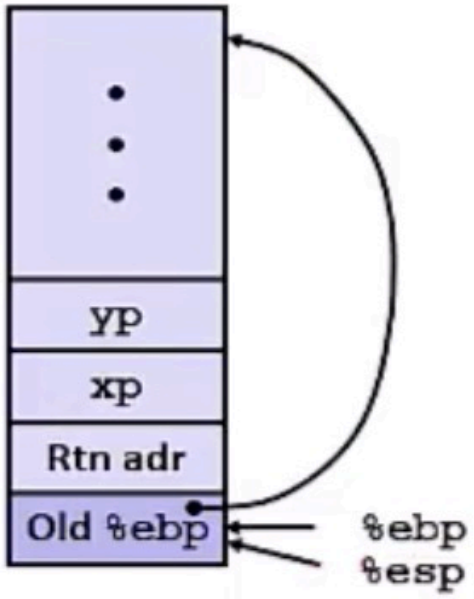
Stack Detour

swap Setup #2

Entering Stack



Resulting Stack

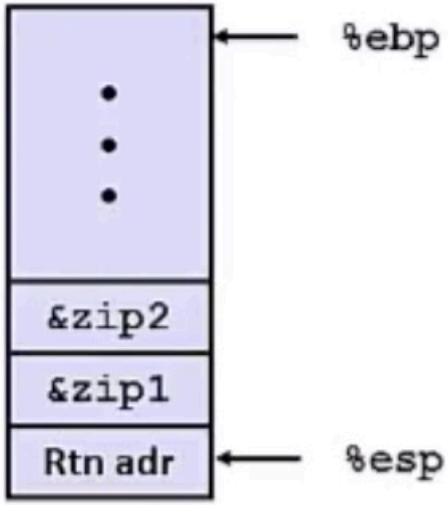



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

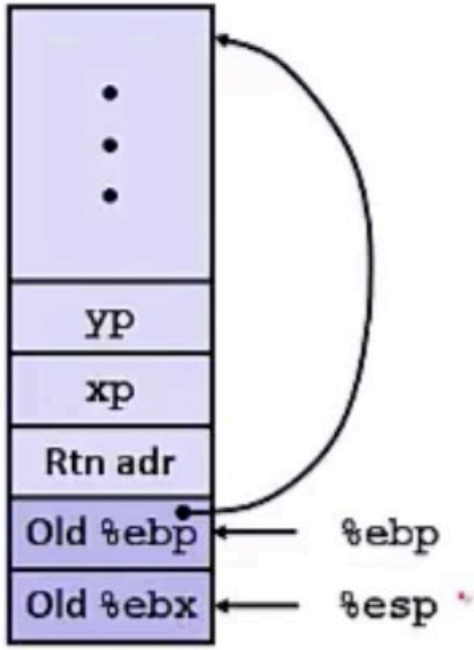
Stack Detour

swap Setup #3

Entering Stack



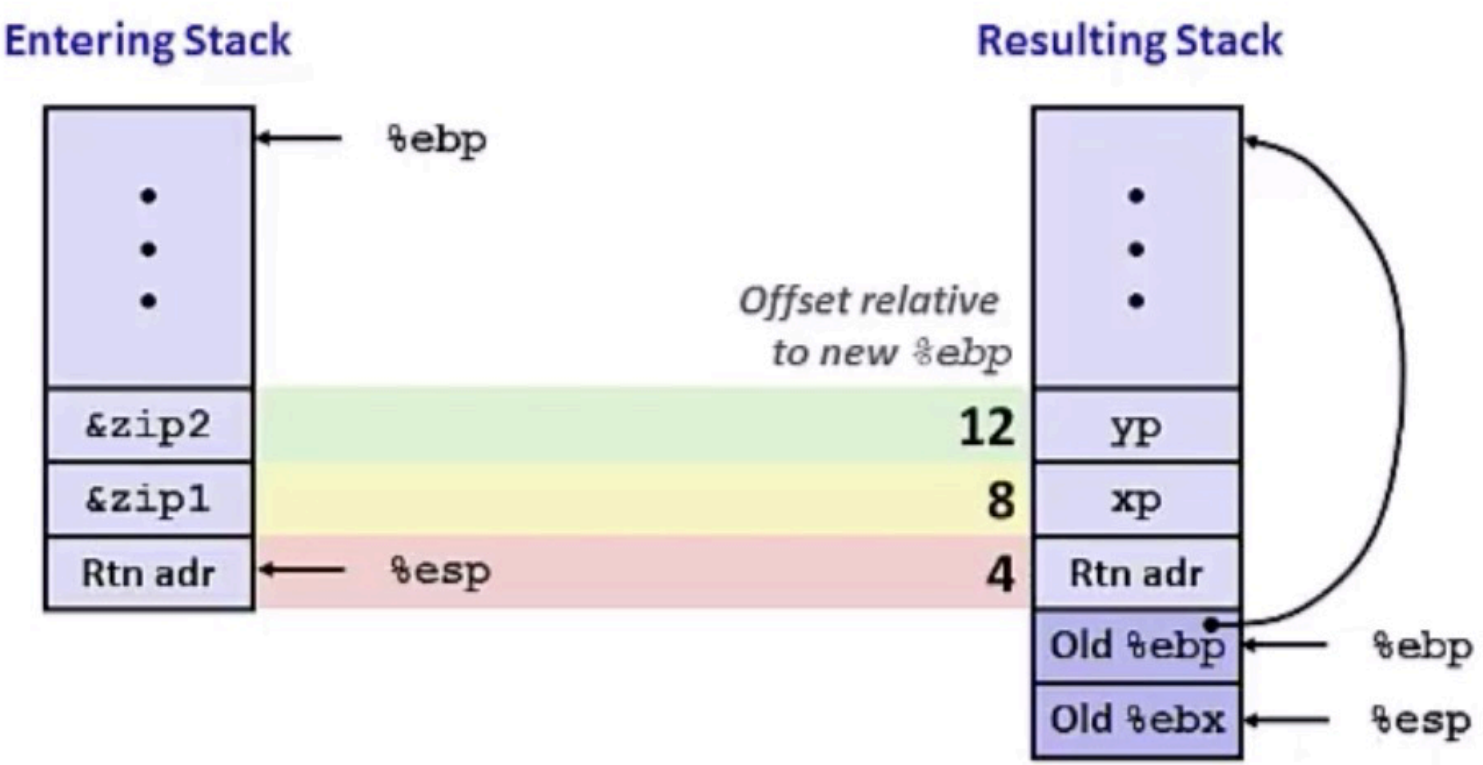
Resulting Stack




```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

Stack Detour

swap Body

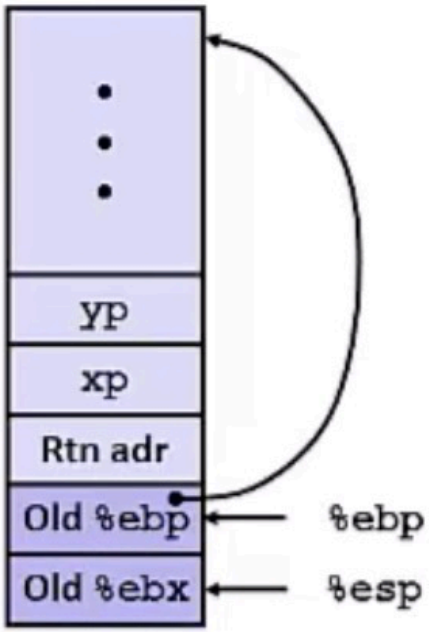



```
movl 12(%ebp),%ecx   # get yp
movl  8(%ebp),%edx   # get xp
. . .
```

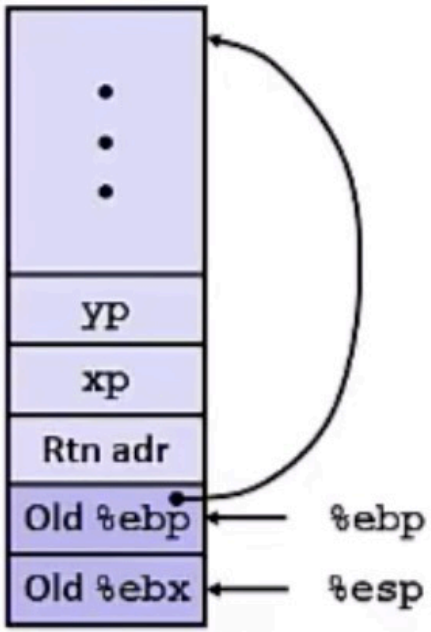
Stack Detour

swap Finish #1

swap's Stack



Resulting Stack



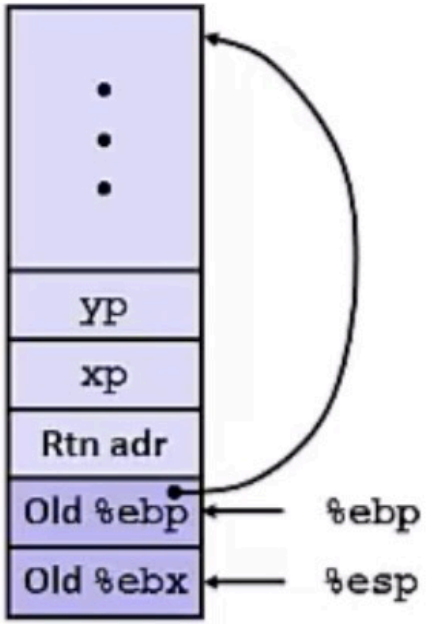

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation: Saved and restored register %ebx

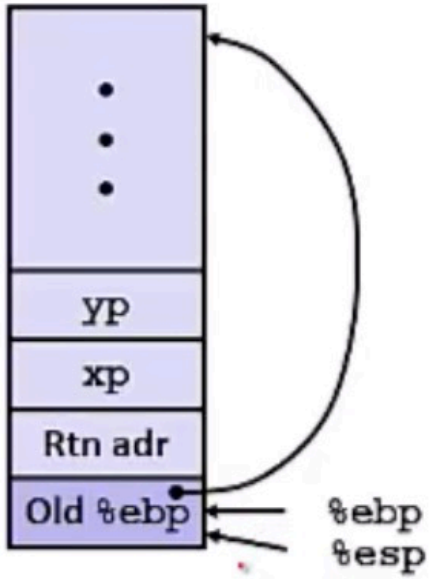
Stack Detour

swap Finish #2

swap's Stack



Resulting Stack

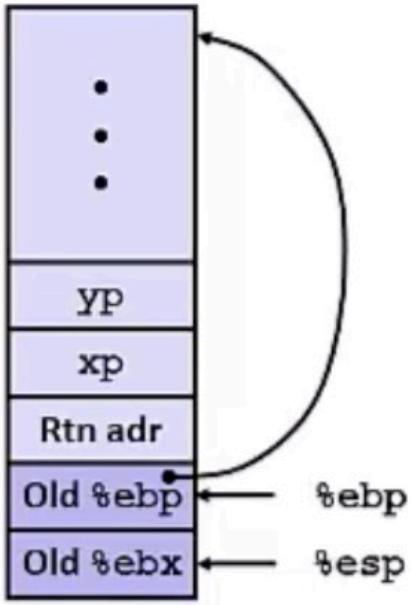



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

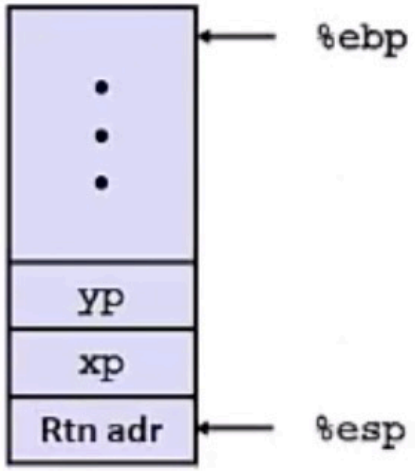
Stack Detour

swap Finish #3

swap's Stack



Resulting Stack




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Q4: What Context must be Saved?

```
// the registers will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;   // Index pointer register
    int esp;   // Stack pointer register
    int ebx;   // Called the base register
    int ecx;   // Called the counter register
    int edx;   // Called the data register
    int esi;   // Source index register
    int edi;   // Destination index register
    int ebp;   // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

Q4: What Context must be Saved?

```
1   # void swtch(struct context **old, struct context *new);
2   #
3   # Save current register context in old
4   # and then load register context from new.
5   .globl swtch
6   swtch:
7     # Save old registers
8     movl 4(%esp), %eax  # put old ptr into eax
9     popl 0(%eax)        # save the old IP
10    movl %esp, 4(%eax)  # and stack
11    movl %ebx, 8(%eax)  # and other registers
12    movl %ecx, 12(%eax)
13    movl %edx, 16(%eax)
14    movl %esi, 20(%eax)
15    movl %edi, 24(%eax)
16    movl %ebp, 28(%eax)
```

Q4: What Context must be Saved?

```
1   # void swtch(struct context **old, struct context *new);
2   #
3   # Save current register context in old
4   # and then load register context from new.
5   .globl swtch
6   swtch:
7     # Save old registers
8     movl 4(%esp), %eax  # put old ptr into eax
9     popl 0(%eax)        # save the old IP
10    movl %esp, 4(%eax)  # and stack
11    movl %ebx, 8(%eax)  # and other registers
12    movl %ecx, 12(%eax)
13    movl %edx, 16(%eax)
14    movl %esi, 20(%eax)
15    movl %edi, 24(%eax)
16    movl %ebp, 28(%eax)
17
18    # Load new registers
19    movl 4(%esp), %eax  # put new ptr into eax
20    movl 28(%eax), %ebp # restore other registers
21    movl 24(%eax), %edi
22    movl 20(%eax), %esi
23    movl 16(%eax), %edx
24    movl 12(%eax), %ecx
25    movl 8(%eax), %ebx
26    movl 4(%eax), %esp  # stack is switched here
27    pushl 0(%eax)       # return addr put in place
28    ret                 # finally return into new ctxt
```

Operating System	Hardware	Program
		Process A ...

Operating System	Hardware	Program
		Process A ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

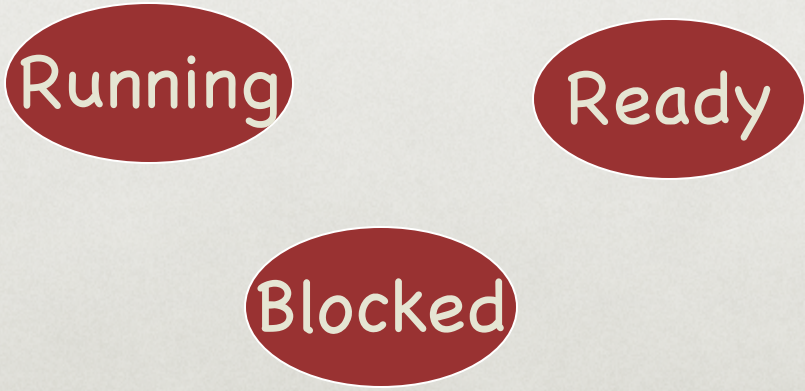
Operating System	Hardware	Program
		Process A
		...
	timer interrupt	
	save regs(A) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call **switch()** routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)		
return-from-trap (into B)		

Operating System	Hardware	Program
		Process A
		...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call **switch()** routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B's IP	

Operating System	Hardware	Program
		Process A
		...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call **switch()** routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B's IP	
		Process B
		...

Problem 3: Slow Ops such as I/O?

When running process performs op that does not use CPU, OS switches to process that needs CPU (policy issues)



OS must track mode of each process:

- Running:
  - On the CPU (only one on a uniprocessor)
- Ready:
  - Waiting for the CPU
- Blocked
  - Asleep: Waiting for I/O or synchronization to complete

Problem 3: Slow OPS SUCH as I/O?

OS must track every process in system

- Each process identified by unique Process ID (PID)

OS maintains queues of all processes

- Ready queue: Contains all ready processes
- Event queue: One logical queue per event
  - e.g., disk I/O and locks
  - Contains all processes waiting for that event to complete

Next Topic: Policy for determining which **ready** process to run

Summary

Virtualization:
Context switching gives each process impression it has its own CPU

Direct execution makes processes fast

Limited execution at key points to ensure OS retains control

Hardware provides a lot of OS support

- user vs kernel mode
- timer interrupts
- automatic register saving

RUTGERS UNIVERSITY
Computer Sciences Department

CS 416 Operating Systems Design

Sudarsun Kannan

Virtualization: The CPU

Process Creation

Two ways to create a process

- Build a new empty process from scratch
- Copy an existing process and change it appropriately

Option 1: New process from scratch

- Steps
  - Load specified code and data into memory;
    Create empty call stack
  - Create and initialize PCB (make look like context-switch)
  - Put process on ready list
- Advantages: No wasted work
- Disadvantages: Difficult to setup process correctly and to express all possible options
  - Process permissions, where to write I/O, environment variables
  - Example: WindowsNT has call with 10 arguments

Process Creation

Option 2: Clone existing process and change

- Example: Unix fork() and exec()
  - Fork(): Clones calling process
  - Exec(char *file): Overlays file image on calling process
- Fork()
  - Stop current process and save its state
  - Make copy of code, data, stack, and PCB
  - Add new PCB to ready list
  - Any changes needed to child process?
- Exec(char *file)
  - Replace current data and code segments with those in specified file
- Advantages: Flexible, clean, simple
- Disadvantages: Wasteful to perform copy and then overwrite of memory

Unix Process Creation

How are Unix shells implemented?

```
While (1) {
  Char *cmd = getcmd();
  Int retval = fork();
  If (retval == 0) {
       // This is the child process
       // Setup the child's process environment here
       // E.g., where is standard I/O, how to handle signals?
       exec(cmd);
       // exec does not return if it succeeds
       printf("ERROR: Could not execute %s\n", cmd);
       exit(1);
  } else {
       // This is the parent process; Wait for child to finish
       int pid = retval;
       wait(pid);
  }
}
```