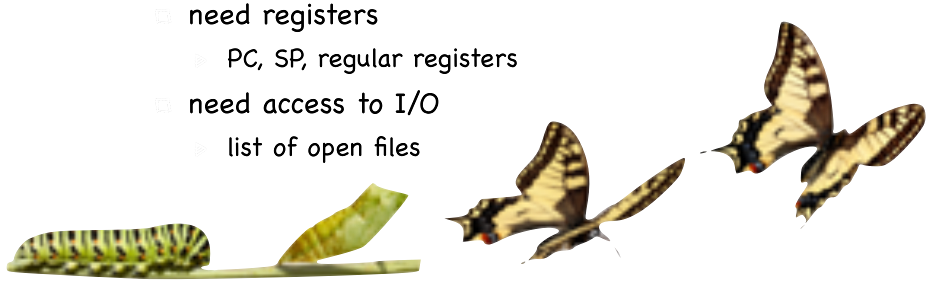
The Process

A running program

From Program to Process

- To make the program's code and data come alive
  - need a CPU
  - need memory — the process' address space
    - for data, code, stack, heap
  - need registers
    - PC, SP, regular registers
  - need access to I/O
    - list of open files



A First Cut at the API

- Create
  - causes the OS to create a new process
- Destroy
  - forcefully terminates a process
- Wait (for the process to end)
- Other controls
  - e.g. to suspend or resume the process
- Status
  - running? suspended? blocked? for how long?

How the OS Keeps Track of a Process

- A process has code
  - OS must track program counter
- A process has a stack
  - OS must track stack pointer
- OS stores state of process in Process Control Block (PCB)
  - Data (program instructions, stack & heap) resides in memory, metadata is in PCB

Process Control Block

PC
Stack Ptr
Registers
PID
UID
Priority
List of open files
Process status
Kernel stack ptr
…

You'll Never Walk Alone

- Machines run (and thus OS must manage) multiple processes
  - how should the machine's resources be mapped to these processes?
- OS as a referee…

You'll Never Walk Alone

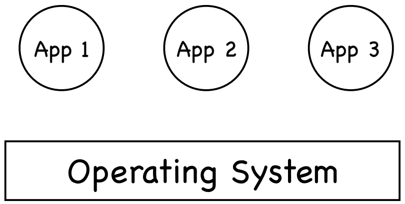
- Machines run (and thus OS must manage) multiple processes
  - how should the machine's resources be mapped to these processes?
- Enter the illusionist!

  - give every process the illusion of running on a private CPU
    - which appears slower than the machine's
  - give every process the illusion of running on a private memory
    - which may appear larger(??) than the machine's

Virtualize the CPU

Virtualize memory

Isolating Applications



Reading and writing memory, managing resources, accessing I/O…

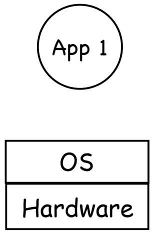
- Buggy apps can crash other apps
- Buggy apps can crash OS
- Buggy apps can hog all resources
- Malicious apps can violate privacy of other apps
- Malicious apps can change the OS

Mechanism and Policy

- Mechanism
  - what the system can do
- Policy
  - what the system should do

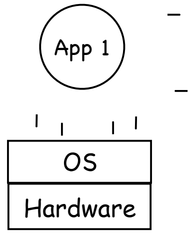
Mechanisms should not determine policies!

The Process, Refined



- An abstraction for isolation
  - the execution of an application program with restricted rights
- The enforcing mechanism must not hinder functionality
  - still efficient use of hardware
  - enable safe communication

The Process, Refined



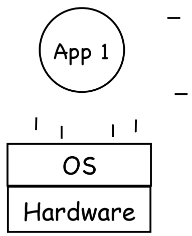
- An abstraction for isolation
  - the execution of an application program with restricted rights
- The enforcing mechanism must not hinder functionality
  - still efficient use of hardware
  - enable safe communication

Special

- The process abstraction is enforced by the kernel
  - all kernel is in the OS
  - not all the OS is in the kernel
    - (why not? robustness)
    - widgets libraries, window managers etc

How can the OS Enforce Restricted Rights?

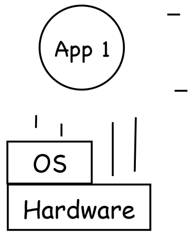
- Easy: kernel interprets each instruction!



- slow
- many instructions are safe: do we really need to involve the OS?

How can the OS enforce restricted rights?

Mechanism: Dual Mode Operation



- hardware to the rescue: use a mode bit
  - in user mode, processor checks every instruction
  - in kernel mode, unrestricted rights
- hardware to the rescue (again) to make checks efficient

Amongst our weaponry are such diverse elements as…

- Privileged instructions
  - in user mode, no way to execute potentially unsafe instructions
- Memory isolation
  - in user mode, memory accesses outside a process' memory region are prohibited
- Timer interrupts
  - kernel must be able to periodically regain control from running process

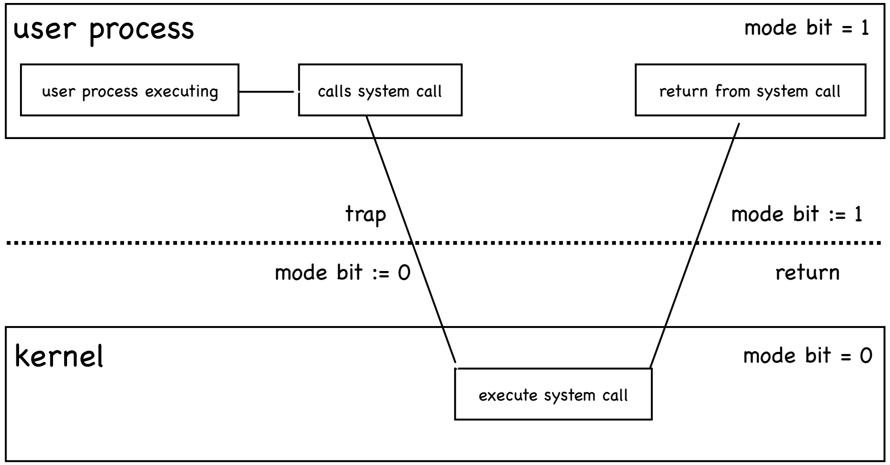
I. Privileged instructions

- Set mode bit
- I/O ops
- Memory management ops
- Disable interrupts
- Set timers
- Halt the processor

I. Privileged instructions

- But how can an app do I/O then?
  - system calls achieve access to kernel mode only at specific locations specified by OS
- Executing a privileged instruction while in user mode (naughty naughty…) causes a processor exception….
  - …which passes control to the kernel

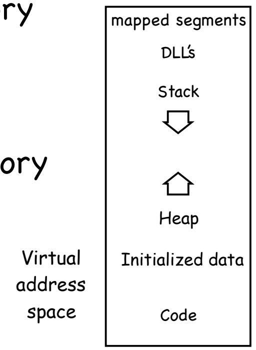
Crossing the line



II. Memory Protection

Step 1: Virtualize Memory

- Virtual address space: set of memory addresses that process can "touch"
  - CPU works with virtual addresses
- Physical address space: set of memory addresses supported by hardware

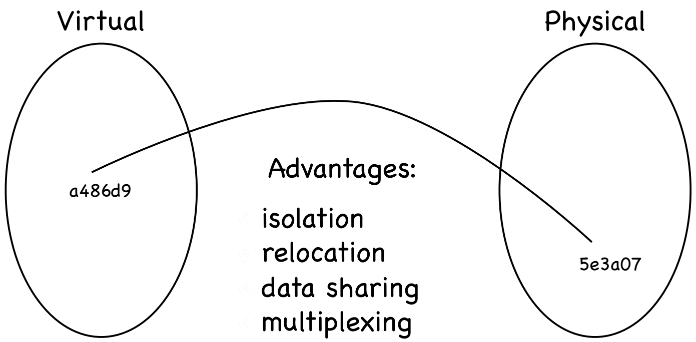


II. Memory Isolation

Step 2: Address Translation

- Implement a function mapping

into

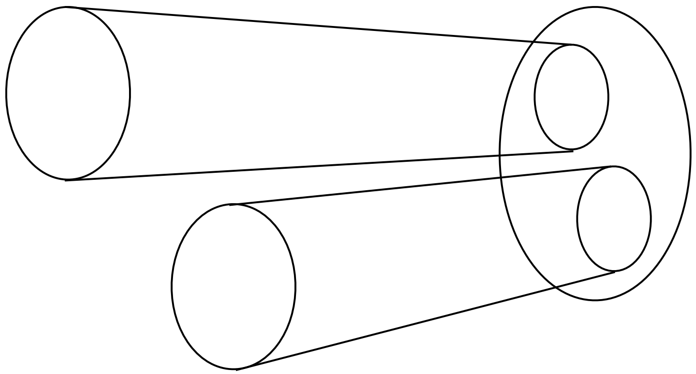


Advantages:

- isolation
- relocation
- data sharing
- multiplexing

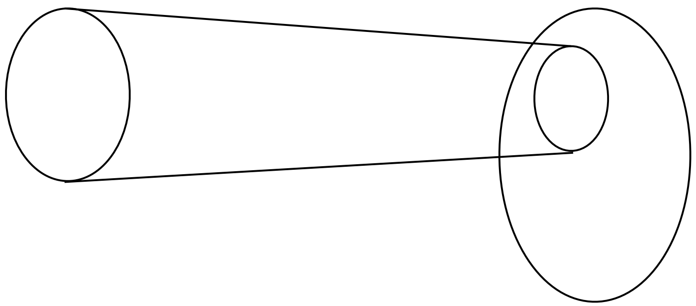
Isolation

- At all times, functions used by different processes map to disjoint ranges — aka "Stay in your room!"



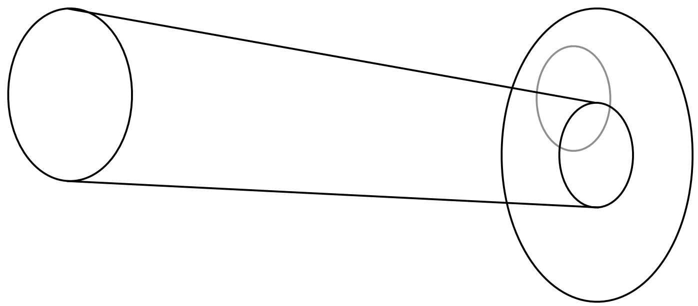
Relocation

- The range of the function used by a process can change over time



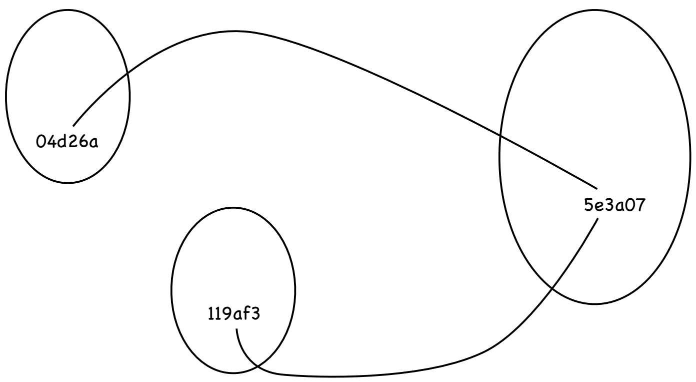
Relocation

- The range of the function used by a process can change over time — "Move to a new room!"



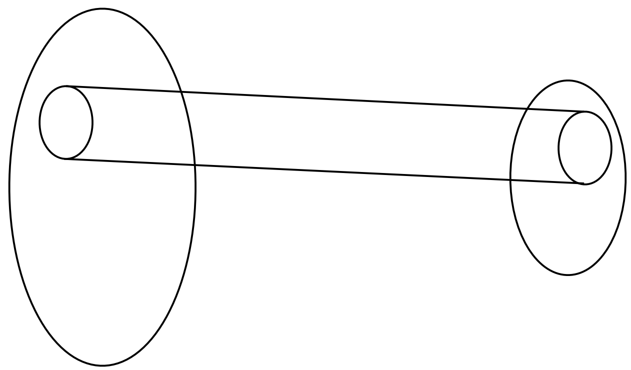
Data Sharing

- Map different virtual addresses of distinct processes to the same physical address — "Share the kitchen!"



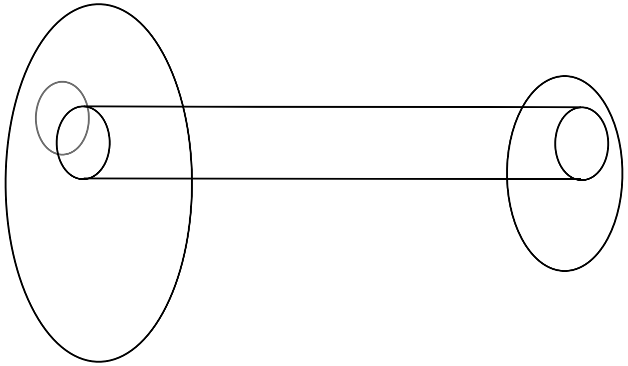
Multiplexing

- Create illusion of almost infinite memory by changing domain (set of virtual addresses) that maps to a given range of physical addresses — ever lived in a studio?



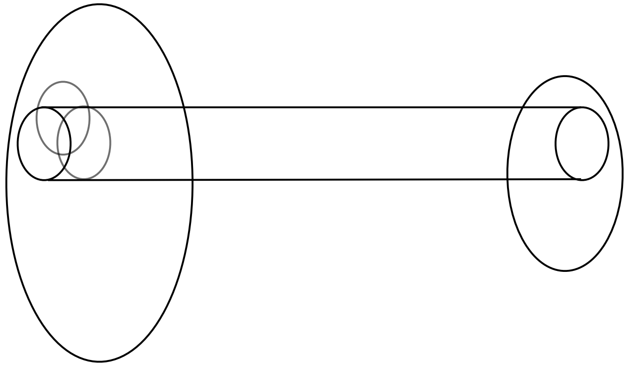
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



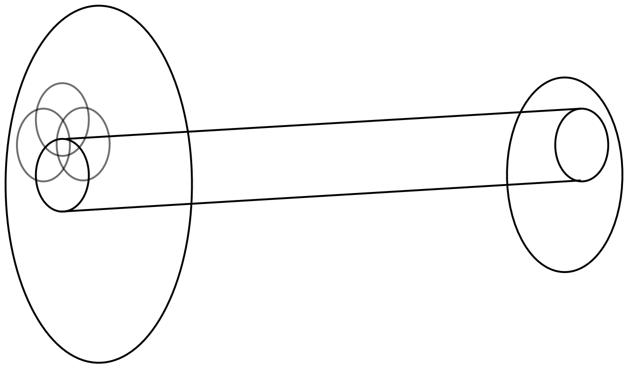
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



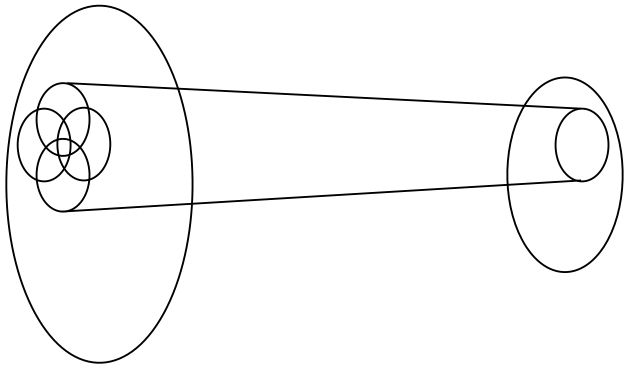
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



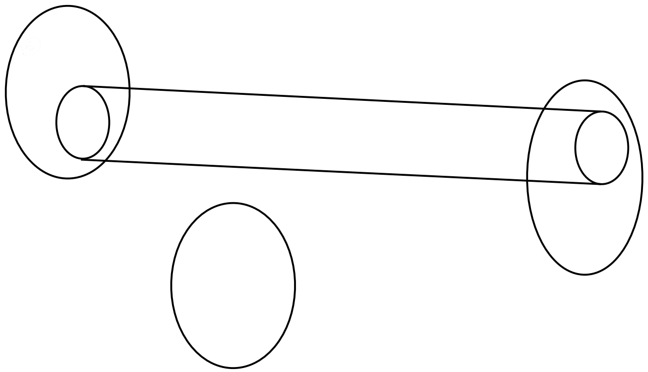
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



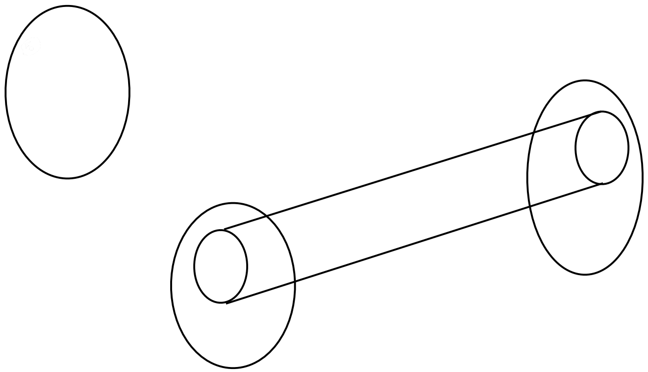
More Multiplexing

- At different times, different processes can map part of their virtual address space into the same physical memory — change tenants!

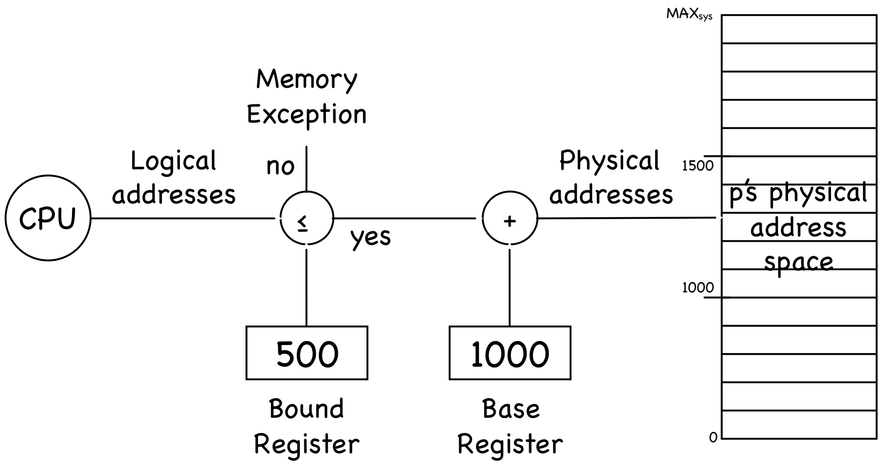


More Multiplexing

- At different times, different processes can map part of their virtual address space into the same physical memory — change tenants!



A simple mapping mechanism: Base & Bound



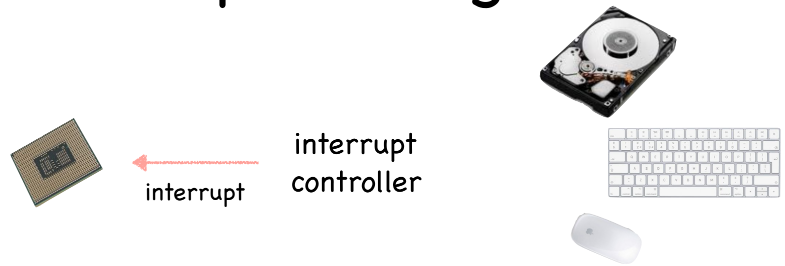
On Base & Limit

- Contiguous Allocation: contiguous virtual addresses are mapped to contiguous physical addresses
- Isolation is easy, but sharing is hard
  - Two copies of emacs: want to share code, but have heap and stack distinct...
- And there is more...
  - Hard to relocate
  - Hard to account for dynamic changes in both heap and stack

III. Timer Interrupts

- Hardware timer
  - can be set to expire after specified delay (time or instructions)
  - when it does, control is passed back to the kernel
- Other interrupts (e.g. I/O completion) also give control to kernel

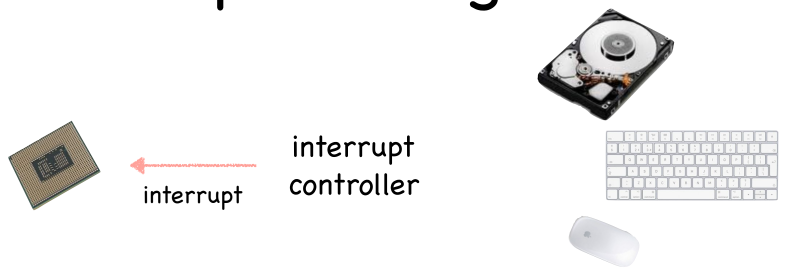
Interrupt Management



Interrupt controllers implements interrupt priorities:

- Interrupts include descriptor of interrupting device
- Priority selector circuit examines all interrupting devices, reports highest level to the CPU
- Controller can also buffer interrupts coming from different devices
  - more on this later...

Interrupt Management



Maskable interrupts

- can be turned off by the CPU for critical processing

Nonmaskable interrupts

- indicate serious errors (power out warning, unrecoverable memory error, etc.)

Types of Interrupts

Exceptions

- process missteps (e.g. division by zero)
- attempt to perform a privileged instruction
  - sometime on purpose! (breakpoints)
- synchronous/non-maskable

System calls/traps

- user program requests OS service
- synchronous/non-maskable

Interrupts

- HW device requires OS service
  - timer, I/O device, interprocessor
- asynchronous/maskable

Interrupt Handling

- Two objectives
  - handle the interrupt and remove the cause
  - restore what was running before the interrupt
    - state may have been modified on purpose
- Two "actors" in handling the interrupt
  - the hardware goes first
  - the kernel code takes control by running the interrupt handler

A Tale of Two Stack Pointers

- Interrupt is a program: it needs a stack!
  - so, each process has two stacks pointers:
    - one when running in kernel mode
    - another when running in user mode
- Why not using the user-level stack pointer?
  - user SP may be badly aligned or pointing to non writable memory
  - user stack may not be large enough, and may spill to overwrite important data
  - security:
    - kernel could leave sensitive data on stack
    - pointing SP to kernel address could corrupt kernel

Handling Interrupts: HW

- On interrupt, hardware:
  - sets supervisor mode (if not set already)
  - disable (masks) interrupts (partially privileged)
  - pushes PC, SP, and PSW (kernel mode bit, interrupts enabled bit, Condition codes) of user program on interrupt stack
  - sets PC to point to the first instruction of the appropriate interrupt handler
    - depends on interrupt type
    - interrupt handler specified in interrupt vector loaded at boot time

Interrupt Vector
I/O interrupt handler
System Call handler
Page fault handler
...

Handling Interrupts: SW

- We are now running the interrupt handler!
  - IH first pushes the registers' contents on the interrupt stack (part of the PCB)
    - need registers to run the IH
    - only saves necessary registers (that's why done in SW, not HW)

Typical Interrupt Handler Code

```
HandleInterruptX:
  PUSH %Rn
     ...            only need to save registers not
  PUSH %R1          saved by the handler function

  CALL _handleX

  POP %R1
     ...            restore the registers saved above
  POP %Rn

  RETURN_FROM_INTERRUPT
```

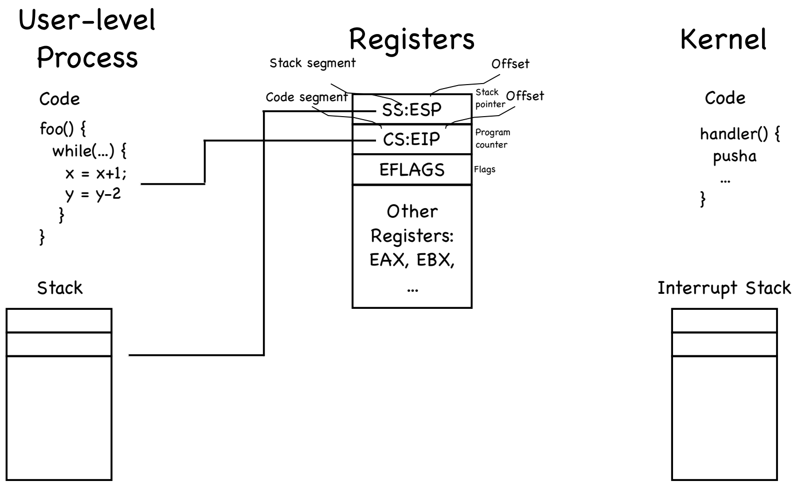
Returning from an Interrupt

- Hardware pops PC, SP, PSW
- Depending on content of PSW
  - switch to user mode
  - enable interrupts
- From exception and system call, increment PC on return (we don't want to execute again the same instruction)
  - on exception, handler changes PC at the base of the stack
  - on system call, increment is done by hw when saving user level state

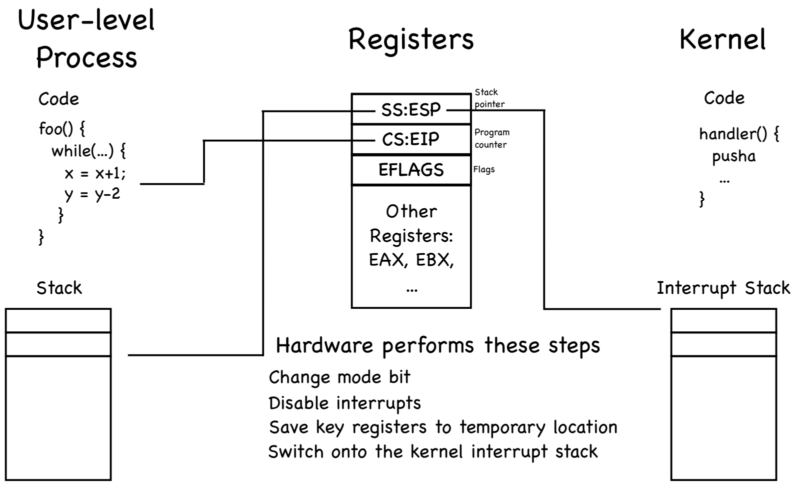
Starting a new process: the recipe

1. Allocate & initialize PCB
2. Setup initial page table (to initialize a new address space)
3. Load program intro address space
4. Allocate user-level and kernel-level stacks.
5. Copy arguments (if any) to the base of the user-level stack
6. Simulate an interrupt
   a) push initial PC, user SP
   b) push PSW (supervisor mode off, interrupts enabled)
7. Clear all other registers
8. RETURN_FROM_INTERRUPT

Interrupt Handling on x86



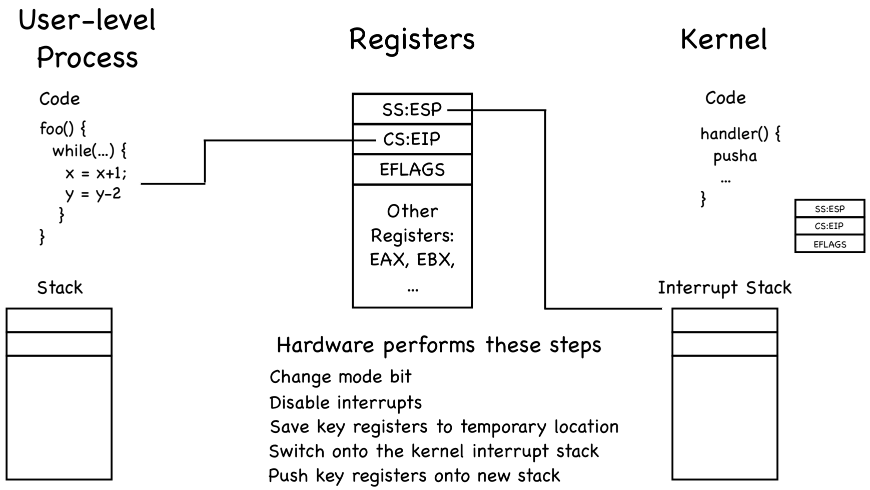
Interrupt Handling on x86



Hardware performs these steps

Change mode bit
Disable interrupts
Save key registers to temporary location
Switch onto the kernel interrupt stack

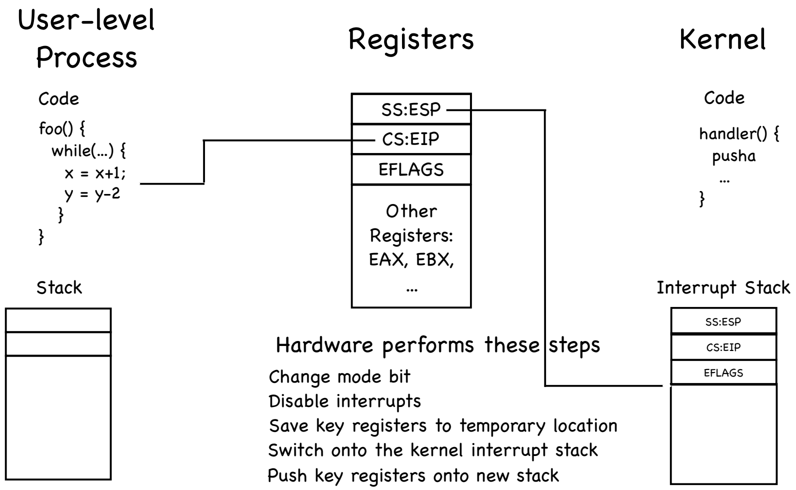
Interrupt Handling on x86



Hardware performs these steps

Change mode bit
Disable interrupts
Save key registers to temporary location
Switch onto the kernel interrupt stack
Push key registers onto new stack

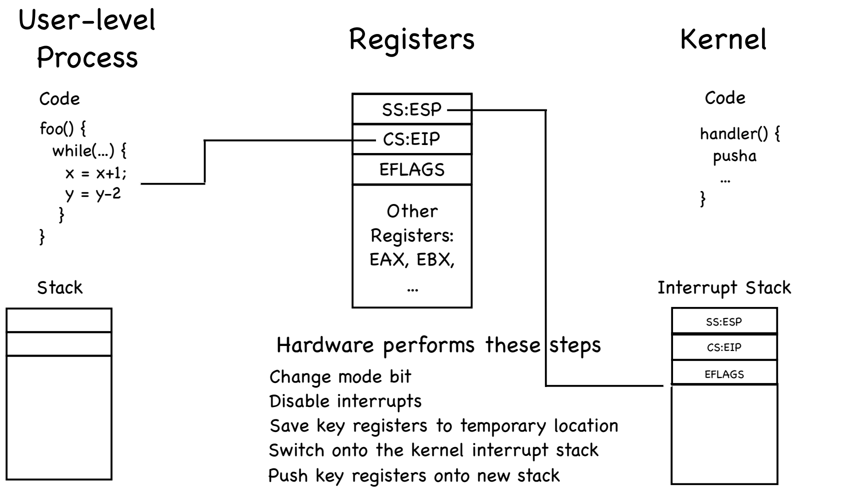
Interrupt Handling on x86



Hardware performs these steps

Change mode bit
Disable interrupts
Save key registers to temporary location
Switch onto the kernel interrupt stack
Push key registers onto new stack

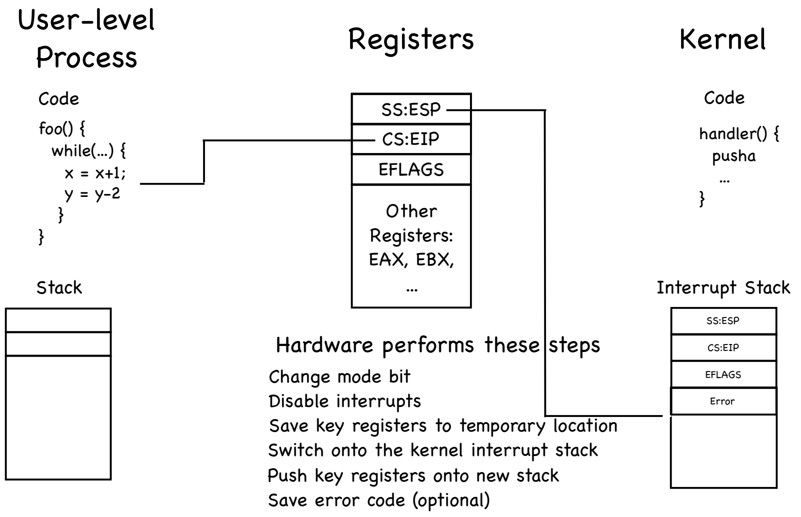
Interrupt Handling on x86



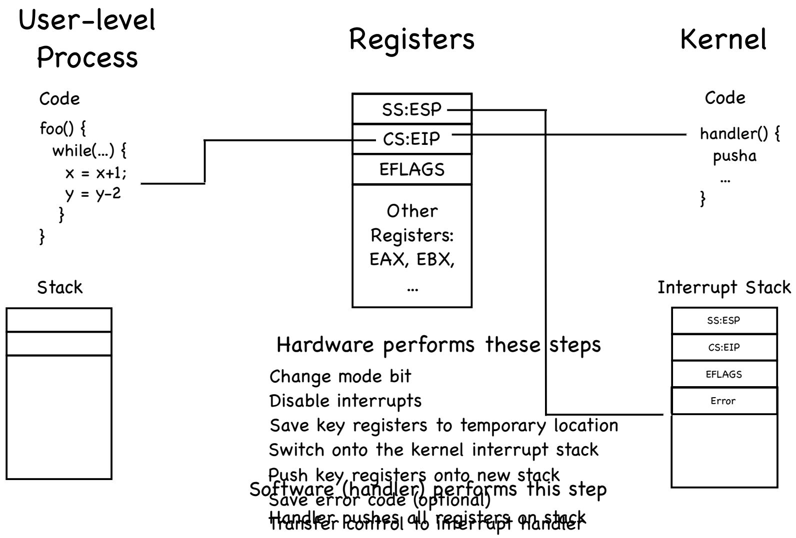
Hardware performs these steps

Change mode bit
Disable interrupts
Save key registers to temporary location
Switch onto the kernel interrupt stack
Push key registers onto new stack
Save error code (optional)

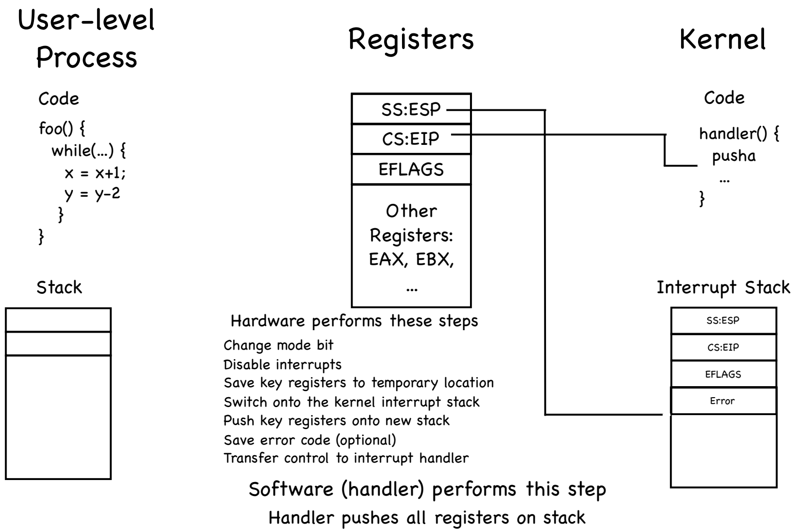
Interrupt Handling on x86



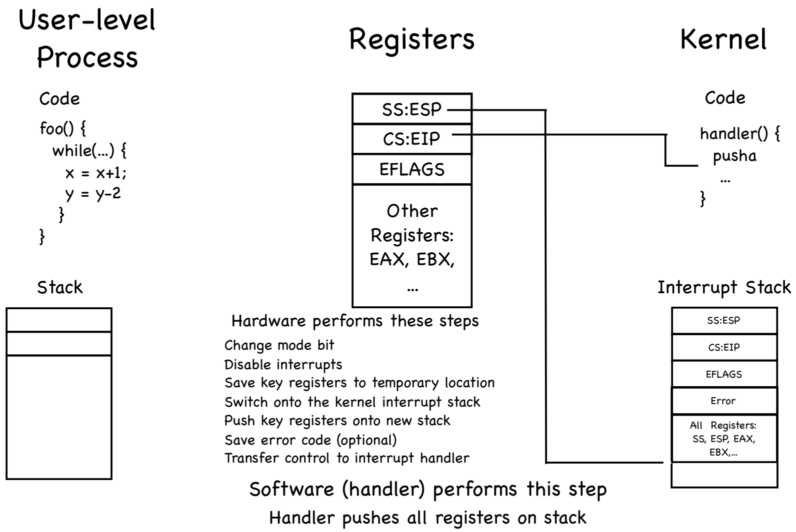
Interrupt Handling on x86



Interrupt Handling on x86



Interrupt Handling on x86



Interrupt Safety

- Kernel should disable device interrupts as little as possible
  - interrupts are best serviced quickly
- Thus, device interrupts are often disabled selectively
  - e.g., clock interrupts enabled during disk interrupt handling
- This leads to potential "race conditions"
  - system's behavior depends on timing of uncontrollable events

Interrupt Race Example

- Disk interrupt handler enqueues a task to be executed after a particular time
  - while clock interrupts are enabled
- Clock interrupt handler checks queue for tasks to be executed
  - may remove tasks from the queue
- Clock interrupt may happen during enqueue

Concurrent access to a shared data structure (the queue!)

Making code interrupt-safe

- Make sure interrupts are disabled while accessing mutable data!
- But don't we have locks?
  - Consider

```
void function ()
{
  lock(mtx);
  /* code */
  unlock(mtx);
}
```

Is `function` thread-safe?

Operates correctly when accessed simultaneously by multiple threads

To make it so, grab a lock

Is function interrupt-safe?

Operates correctly when called again (re-entered) before it completes

To make it so, disable interrupts

Example of Interrupt-Safe Code

```
void enqueue(struct task *task) {
  int level = interrupt_disable();
  /* update queue */
  interrupt_restore(level);
}
```

- Why not simply re-enable interrupts?
  - Say we did. What if then we call enqueue from code that expects interrupts to be disabled?
    - Oops...
  - Instead, remember interrupt level at time of call; when done, restore that level

Many Standard C Functions are not Interrupt-Safe

- Pure system calls are interrupt-safe
  - e.g., read(), write(), etc.
- Functions that don't use global data are interrupt-safe
  - e.g., strlen(), strcpy(), etc.
- malloc(), free (), and printf() are not interrupt-safe
  - must disable interrupts before using it in an interrupt handler
  - and you may not want to anyway (printf() is huge!)

But they are all thread-safe!

System calls

- Programming interface to the services the OS provides:
  - read input/write to screen
  - create/read/write/delete files
  - create new processes
  - send/receive network packets
  - get the time / set alarms
  - terminate current process
  - ...

The Skinny

- Simple and powerful interface allows separation of concern
  - Eases innovation in user space and HW
- "Narrow waist" makes it
  - highly portable
  - robust (small attack surface)
- Internet IP layer also offers skinny interface

Web Servers
Compilers
Databases
Word Processing
Web Browsers
Email

Portable OS Library

System call interface

Portable OS Kernel

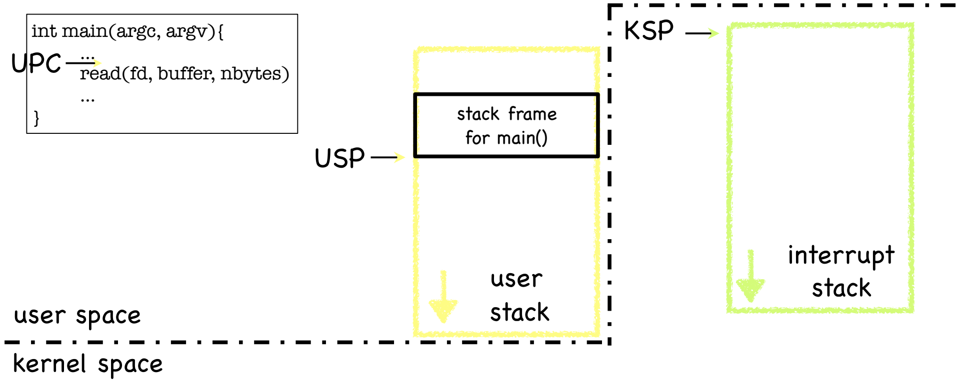
x86
ARM
PowerPC
10Mbps/100Mbps/1Gbps Ethernet
1802.11 a/b/g/n
SCSI
Graphics accellerators
LCD Screens

- Much care spent in keeping interface secure
  - e.g., parameters first copied to kernel space, then checked
    - to prevent them from being changed after they are checked!

Executing a System Call

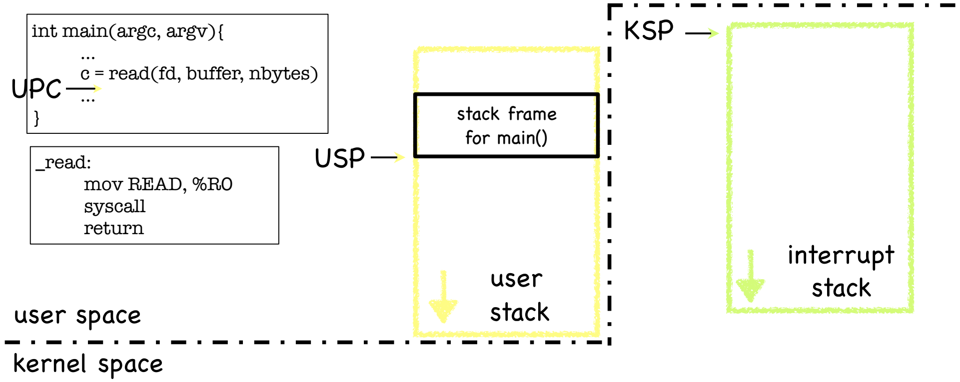
- Process:
  - Calls system call function in library
  - Places arguments in registers and/or pushes them onto user stack
  - Places syscall type in a dedicated register
  - Executes syscall machine instruction
- Kernel
  - Executes syscall interrupt handler
  - Places result in dedicated register
  - Executes RETURN_FROM_INTERRUPT
- Process:
  - Executes RETURN_FROM_FUNCTION

Executing read System Call



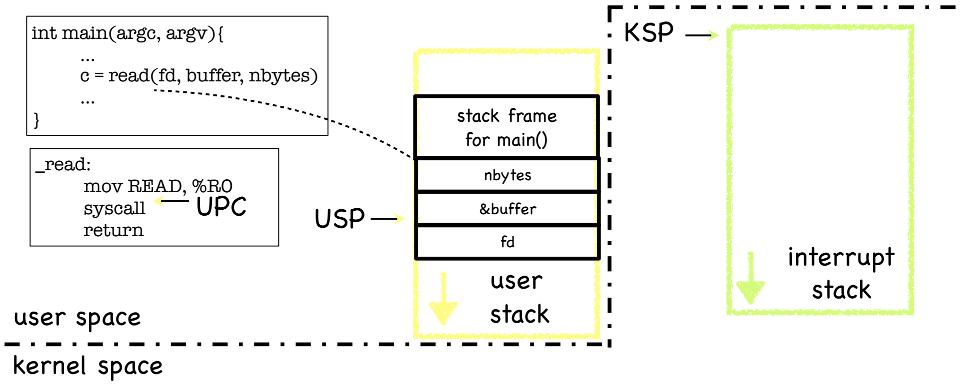
UPC: user program counter
USP: user stack pointer
KSP: kernel stack pointer
note: interrupt stack is empty while process running

Executing read System Call



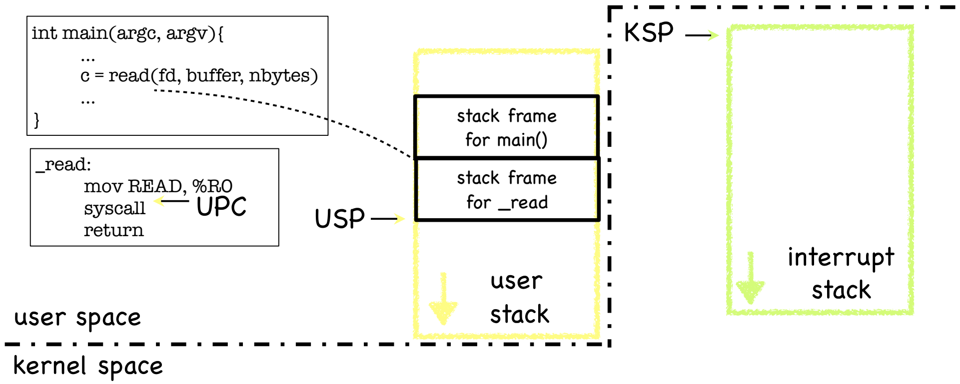
UPC: user program counter
USP: user stack pointer
KSP: kernel stack pointer
note: interrupt stack is empty while process running

Executing read System Call



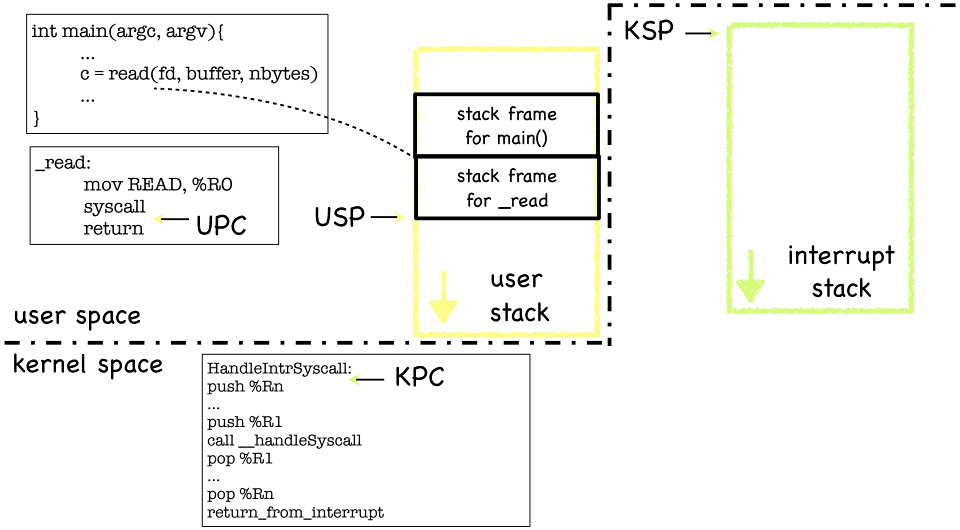
UPC: user program counter
USP: user stack pointer
KSP: kernel stack pointer
note: interrupt stack is empty while process running

Executing read System Call

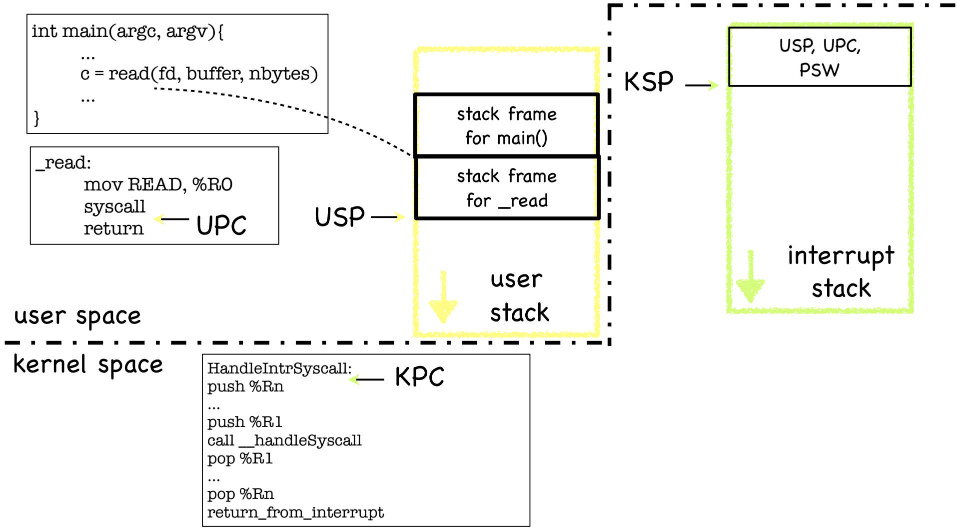


UPC: user program counter
USP: user stack pointer
KSP: kernel stack pointer
note: interrupt stack is empty while process running

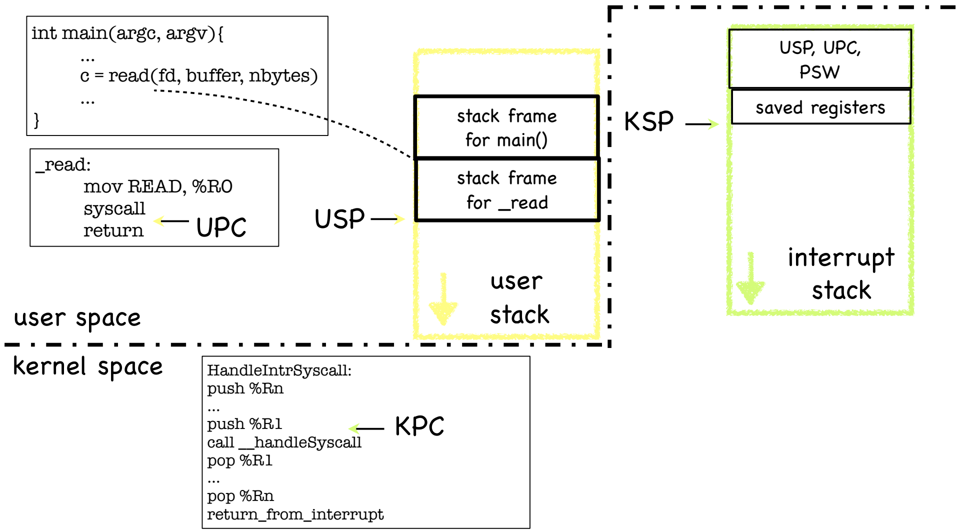
Executing read System Call



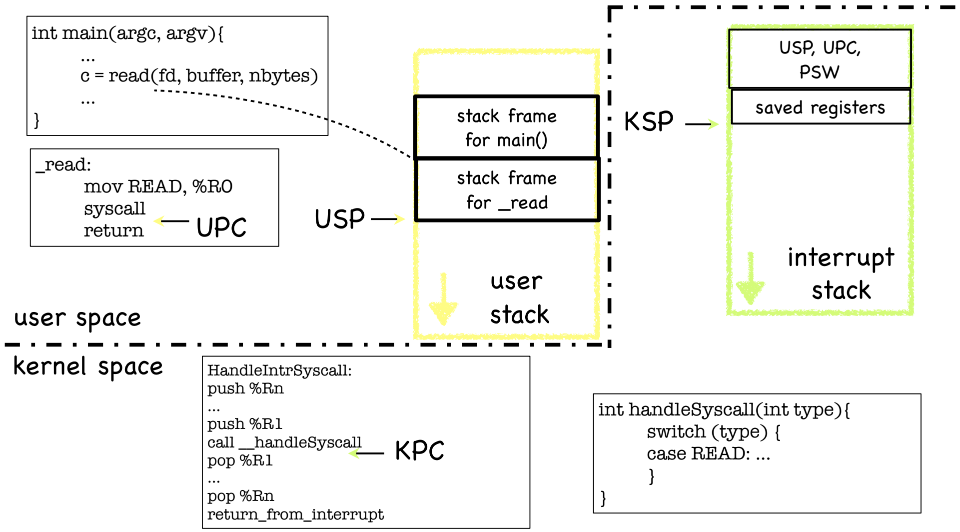
Executing read System Call



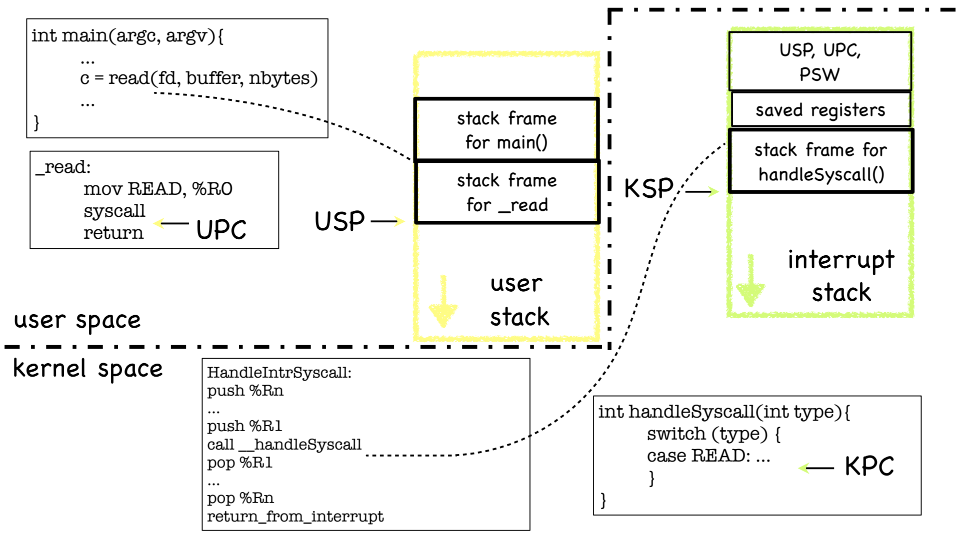
Executing read System Call



Executing read System Call



Executing read System Call



What if read needs to block?

- read may need to block if
  - It reads from a terminal
  - It reads from disk, and block is not in cache
  - It reads from a remote file server

We should run another process!

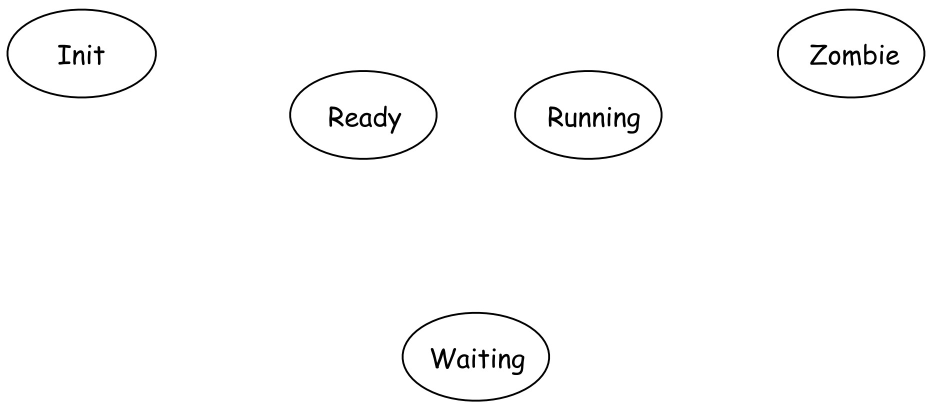
Virtualizing the CPU

- OS keeps a PCB for each process
- It has space to hold a "frozen" version of the state process's state
  - Program counter
  - Process status (ready, running, etc)
  - CPU registers
  - CPU scheduling info
  - Memory management info
  - Account info
  - I/O status info
- to be saved when the process relinquishes the CPU
- and reloaded when the process reacquires the CPU

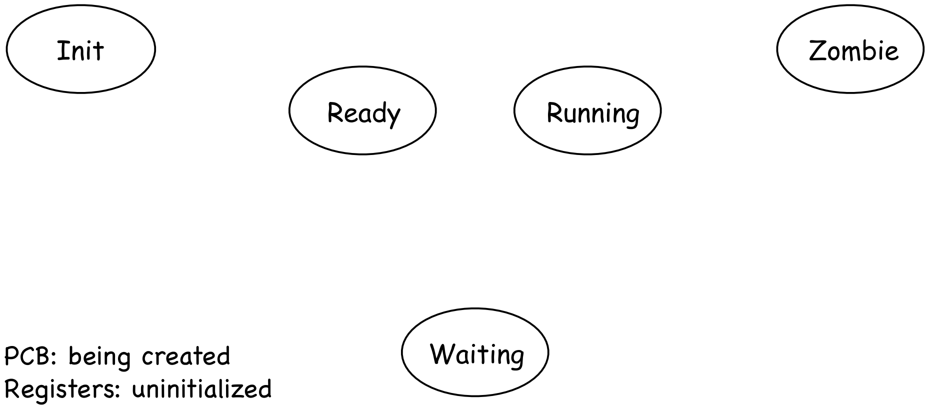
Process Control Block

PC
Stack Ptr
Registers
PID
UID
Priority
List of open files
Process status
Kernel stack ptr
Location in Memory
Location of executable on disk
...

Process Life Cycle

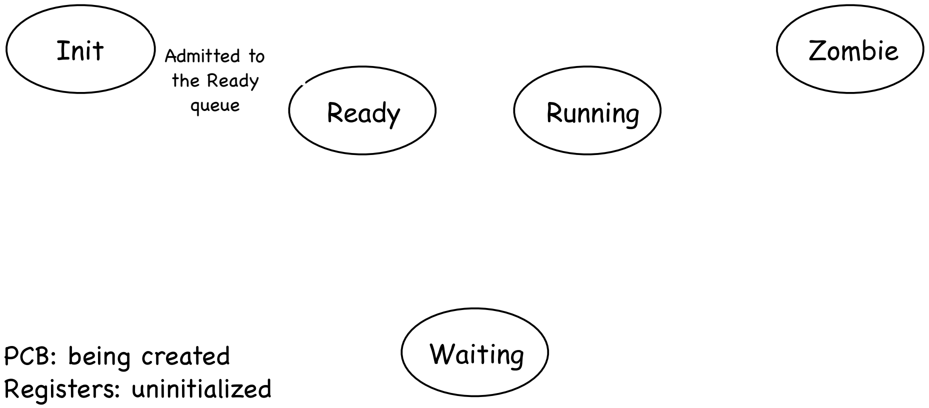


Process Life Cycle



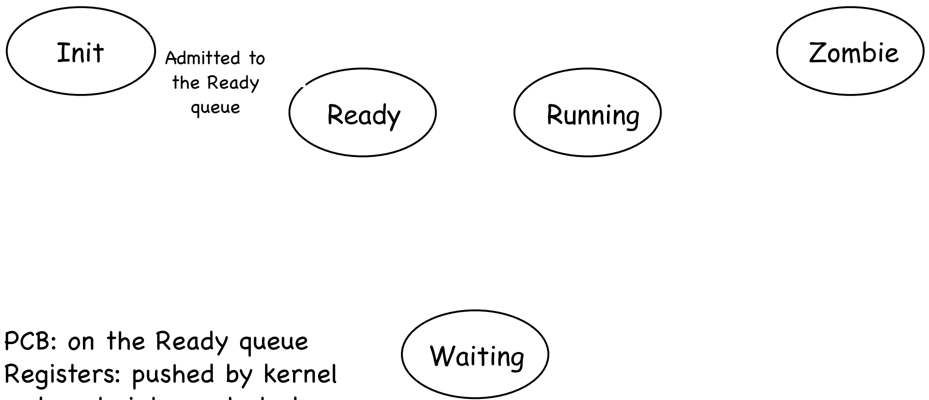
PCB: being created
Registers: uninitialized

Process Life Cycle



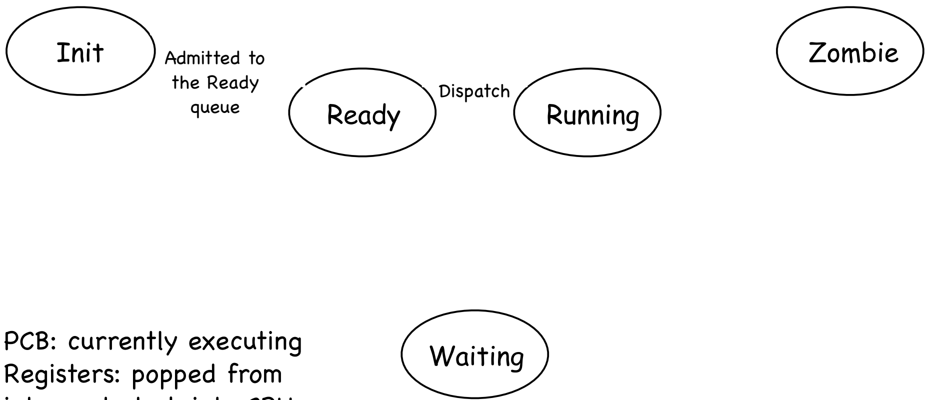
PCB: being created
Registers: uninitialized

Process Life Cycle



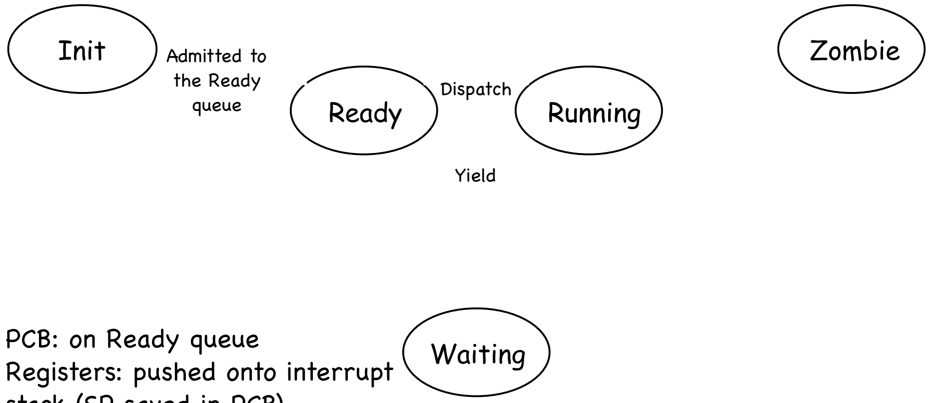
PCB: on the Ready queue
Registers: pushed by kernel code onto interrupt stack

Process Life Cycle



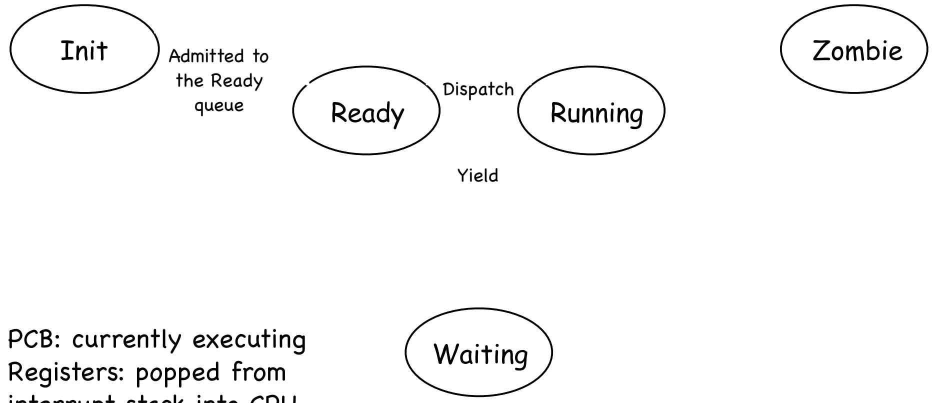
PCB: currently executing
Registers: popped from interrupt stack into CPU

Process Life Cycle



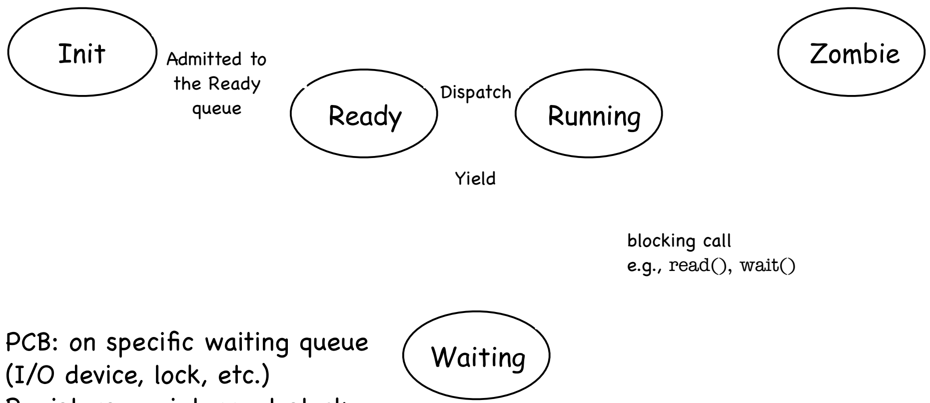
PCB: on Ready queue
Registers: pushed onto interrupt stack (SP saved in PCB)

Process Life Cycle



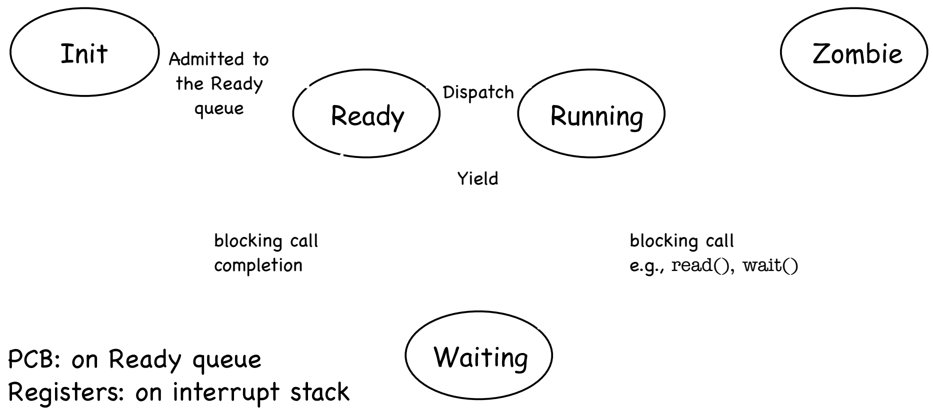
PCB: currently executing
Registers: popped from interrupt stack into CPU

Process Life Cycle



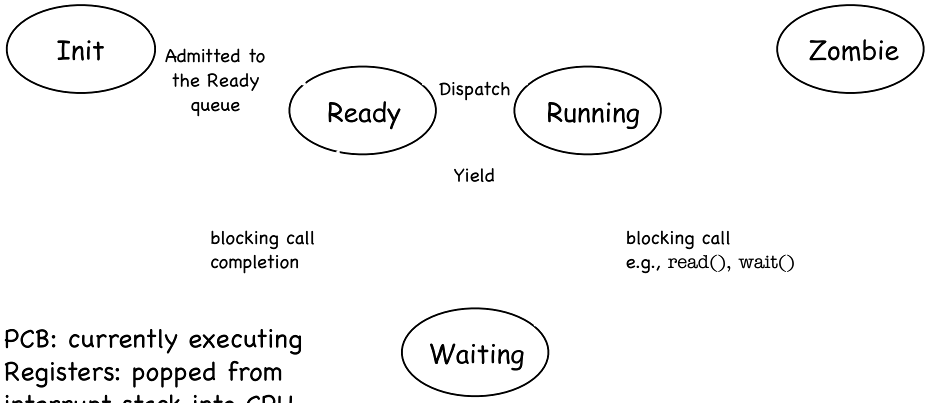
PCB: on specific waiting queue (I/O device, lock, etc.)
Registers: on interrupt stack

Process Life Cycle



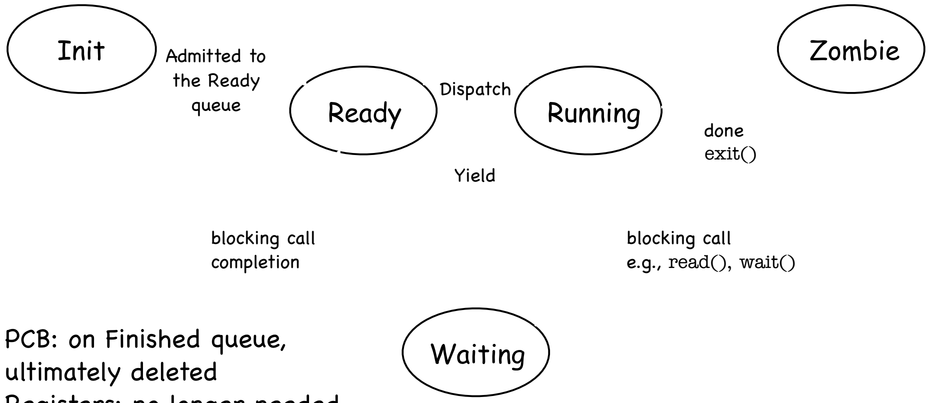
PCB: on Ready queue
Registers: on interrupt stack

Process Life Cycle



PCB: currently executing
Registers: popped from interrupt stack into CPU

Process Life Cycle



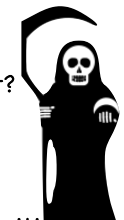
PCB: on Finished queue, ultimately deleted
Registers: no longer needed

Invariants to keep in mind

- At most one process/core running at any time
- When CPU in user mode, current process is RUNNING and its interrupt stack is empty
- If process is RUNNING
  - its PCB not on any queue
  - it is not necessarily in USER mode
- If process is RUNNABLE or WAITING
  - its registers are saved at the top of its interrupt stack
  - its PCB is either
    - on the READY queue (if RUNNABLE)
    - on some WAIT queue (if WAITING)
- If process is a ZOMBIE
  - its PCB is on FINISHED queue

Cleaning up Zombies

- Process cannot clean up itself (why?)
- Process can be cleaned up
  - by some other process, checking for zombies before returning to RUNNING state
  - or by parent which waits for it
    - but what if parent turns into a zombie first?
  - or by a dedicated "reaper" process



- Linux uses a combination
  - if alive, parent cleans up child that it is waiting for
  - if parent is dead, child process is inherited by the initial process, which is continually waiting

How to Yield/Wait?

- Must switch from executing the current process to executing some other READY process
  - Current process: RUNNING   READY
  - Next process: READY   RUNNING

  Save kernel registers of Current on its interrupt stack

  Save kernel SP of Current in its PCB

  Restore kernel SP of Next from its PCB

  Restore kernel registers of Next from its interrupt stack

Yielding

```
ctx_switch: //ip already pushed
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    movq    %rsp, (%rdi)
    movq    %rsi, %rsp
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    retq
```

```
struct pcb *current, *next;

void yield(){
  assert(current->state == RUNNING);
  current->state = RUNNABLE;
  runQueue.add(current);
  next = scheduler();
  next->state = RUNNING;
  ctx_switch(&current->sp, next->sp)
  current = next;
}
```

Starting a New Process

```
ctx_start:
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    movq    %rsp, (%rdi)
    movq    %rsi, %rsp
    callq   ctx_entry
```

```
void createProcess( func ){
  current->state = READY;
  readyQueue.add(current);
  struct pcb *next = malloc(...);
  next->func = func;
  next->state = RUNNING;
  ctx_start(&current->sp, next->top_of_stack)
  current = next;
}

void ctx_entry(){
  current = next;
  (*current->func)();
  current->state = ZOMBIE;
  finishedQueue.add(current);
  next = scheduler();
  next->state = RUNNING;
  ctx_switch(&current->sp, next->sp)
  // this location cannot be reached
}
```

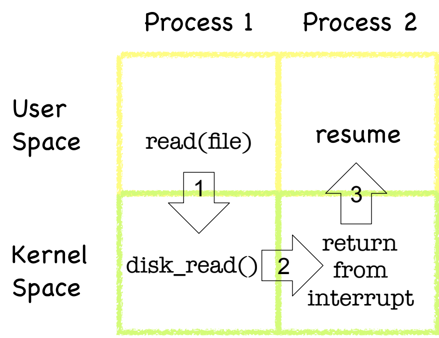
Anybody there?

- What if no process is READY?
  - scheduler() would return NULL — aargh!
- To avoid armageddon
  - OS always runs a low priority process, in an infinite loop executing the HLT instruction
    - halts CPU until next interrupt
  - Interrupt handler executes yield() if some other process is put on the Ready queue

Three Flavors of Context Switching

- Interrupt: from user to kernel space
  - on system call, exception, or interrupt
  - Px user stack   Px interrupt stack
- Yield: between two processes, inside kernel
  - from one PCB/interrupt stack to another
  - Px interrupt stack   Py interrupt stack
- Return from interrupt: from kernel to user space
  - with the homonymous instruction
  - Px interrupt stack   Px user stack

Switching between Processes



Save Process 1 user registers

Save Process 1 kernel registers and restore Process 2 kernel registers

Restore Process 2 user registers

System Calls to Create a New Process

- Windows
  - CreateProcess(...);
- Unix (Linux)
  - fork() + exec(...)

CreateProcess (Simplified)

```
if (!CreateProcess(
    NULL,          // No module name (use command line)
    argv[1],       // Command line
    NULL,          // Process handle not inheritable
    NULL,          // Thread handle not inheritable
    FALSE,         // Set handle inheritance to FALSE
    0,             // No creation flags
    NULL,          // Use parent's environment block
    NULL,          // Use parent's starting directory
    &si,           // Pointer to STARTUPINFO structure
    &pi )          // Ptr to PROCESS_INFORMATION structure
)
```

[Windows]

fork (actual form)

process identifier

```
int pid = fork();
```

..but needs exec(...)

[Unix]

Kernel Actions to Create a Process

- fork()
  - allocate ProcessID
  - initialize PCB
  - create and initialize new address space
  - inform scheduler new process is READY
- exec(program, arguments)
  - load program into address space
  - copy arguments into address space's memory
  - initialize h/w context to start execution at ``start"
- CreateProcess(...) does both

Creating and managing processes

Syscall	Description
fork()	Create a child process as a clone of the current process. Return to both parent and child. Return child's pid to parent process; return 0 to child
exec (prog, args)	Run application prog in the current process with the specified args (replacing any code and data that was present in process)
wait (&status)	Pause until a child process has exited
exit (status)	Tell kernel current process is complete and its data structures (stack, heap, code) should be garbage collected. May keep PCB.
kill (pid, type)	Send an interrupt of a specified type to a process (a bit of an overdramatic misnomer...)

[Unix]

In action

Process 13
Program A

PC

pid
?

```
pid = fork();
if (pid==0)
  exec(B);
else
wait(&status);
```

In action

Process 13
Program A

PC →
```
pid = fork();
if (pid==0)
  exec(B);
else
wait(&status);
```
pid ?

Process 13
Program A

PC →
```
pid = fork();
if (pid==0)
  exec(B);
else
wait(&status);
```
pid 14

Process 14
Program A/B

PC →
```
pid = fork();
if (pid==0)
  exec(B);
else
wait(&status);
main() {
  ...
  exit(3);
}
```
pid 0

In action

Process 13
Program A

PC →
```
pid = fork();
if (pid==0)
  exec(B);
else
wait(&status);
```
pid ?

Process 13
Program A

PC
```
pid = fork();
if (pid==0)
  exec(B);
else
wait(&status);
```
pid 14 → wait(&status);

Status 3

Process 14
Program B

PC →
```
main() {
  ...
  exit(3);
}
```

What is a shell?

Job control system

- Runs programs on behalf of the user
- Allows programmer to create/manage set of programs
  - sh — Original Unix shell (Bourne, 1977)
  - csh — BSD Unix C shell (tcsh enhances it)
  - bash — "Bourne again" shell
- Every command typed in the shell starts a child process of the shell
- Runs at user-level. Uses syscalls: fork, exec, etc.

The Unix shell (simplified)

```
while(! EOF)
read input
handle regular expressions
int pid = fork()    // create child
if (pid == 0) { // child here
  exec("program", argc, argv0,...);
}
else {  // parent here
...
}
```

Signals (Virtualized Interrupts)

Asynchronous notifications in user space

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., CTRL-C from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGSTP	Stop until SIGCONT	Stop signal from terminal (e.g., CTRL-Z from keyboard)

Signal Example

```
int main() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++) // N forks
        if ((pid[i] = fork()) == 0) {
            while(1); // child infinite loop
        }
    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {      // parent continues executing
        printf("Killing proc. %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) // parent checks for each child's exit
            printf("Child %d terminated w/exit status %d\n", wpid,
                   WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
    exit(0);
}
```

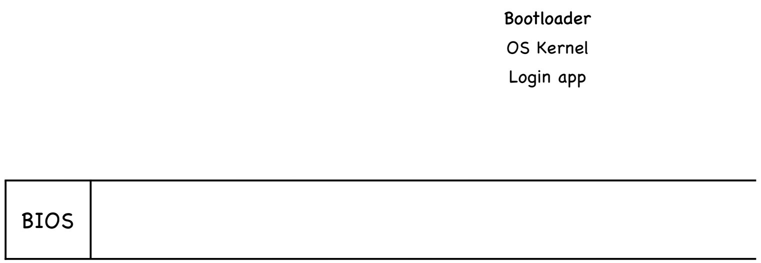
Handler Example

```
void int_handler(int sig) {
    printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}
int main() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler) // register handler for SIGINT
    for (i = 0; i < N; i++) // N forks
        if ((pid[i] = fork()) == 0) {
            while(1); // child infinite loop
        }
    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {      // parent continues executing
        printf("Killing proc. %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) // parent checks for each child's exit
            printf("Child %d terminated w/exit status %d\n", wpid,
                   WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
    exit(0);
}
```

Kernel Operation (conceptual, simplified)

```
Initialize devices
Initialize "first process"
while (TRUE) {
    while device interrupts pending
      - handle device interrupts
    while system calls pending
      - handle system calls
    if run queue is non-empty          <- CPU Scheduling
      - select a runnable process and switch to it
    otherwise
      - wait for device interrupt
}
```

Booting an OS Kernel

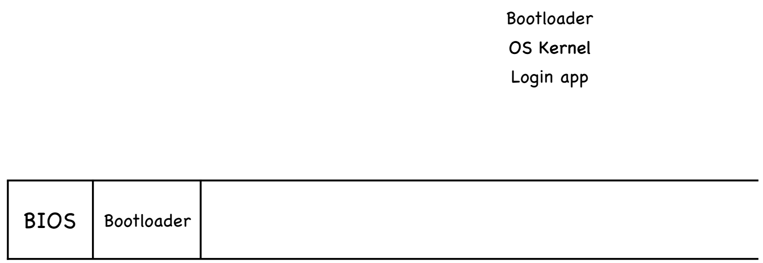


Basic Input/Output System

- In ROM; includes the first instructions fetched and executed

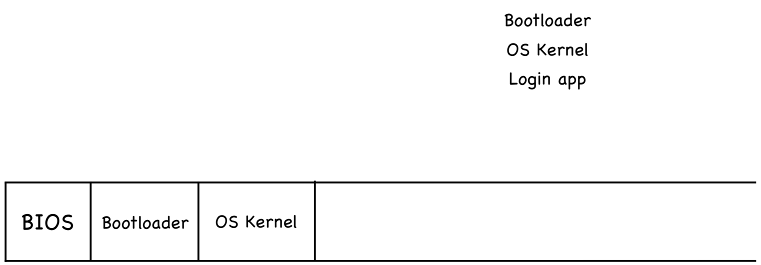
(1) BIOS copies Bootloader, checking its cryptographic hash to make sure it has not been tampered with

Booting an OS Kernel



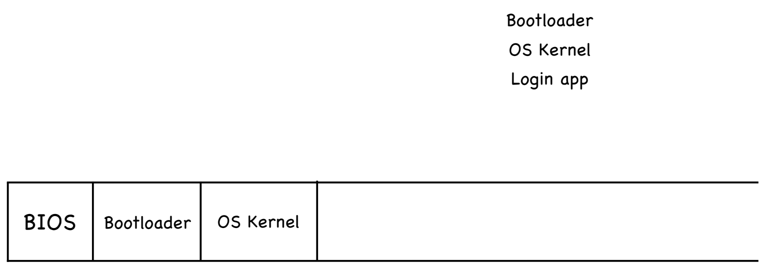
(2) Bootloader copies OS Kernel, checking its cryptographic hash

Booting an OS Kernel



(2) Bootloader copies OS Kernel, checking its cryptographic hash

Booting an OS Kernel



(3) Kernel initializes its data structures (devices, interrupt vector table, etc)

Booting an OS Kernel

Bootloader
OS Kernel
Login app

④ Kernel: Copies first process from disk

Booting an OS Kernel

Bootloader
OS Kernel
Login app

④ Kernel: Copies first process from disk
Changes PC and sets mode bit to 1

And the dance begins!