

CSE 3320

Chapter 1: Getting Started

Trevor Bakker

The University of Texas at Arlington

What is an Operating System?

A collection of one or more software modules that manages and controls the resources of a computer or other computing device or electronic device and gives users and programs an interface to utilize these resources

Terminology

Services - functions the OS kernel provides to users, mostly through APIs via OS calls. These can be grouped into categories by functionality, such as file manipulation services (create, read, copy), memory allocation (get, free) or miscellaneous services (get system time)

Terminology

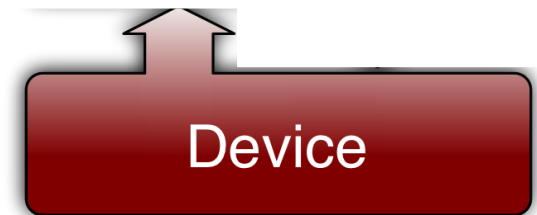
Utility - Programs no part of the OS kernel, but work closely with the kernel to provide was of use or access to system information.

Example: A shell such as bash, csh, or ksh provides a user interface to system services and can call other utilities such as ls.

Terminology

Device - A piece of hardware connected to the main computer system hardware.

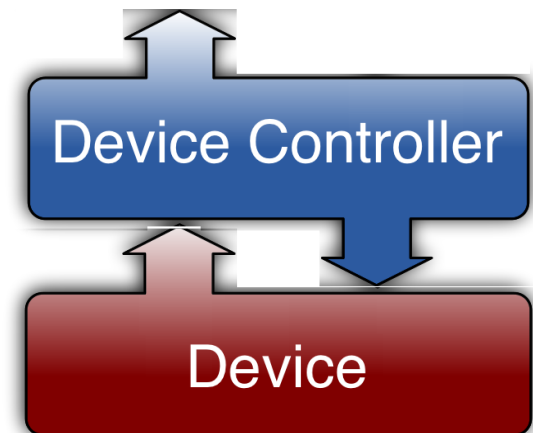
Example: hard drives, video card, mouse



Terminology

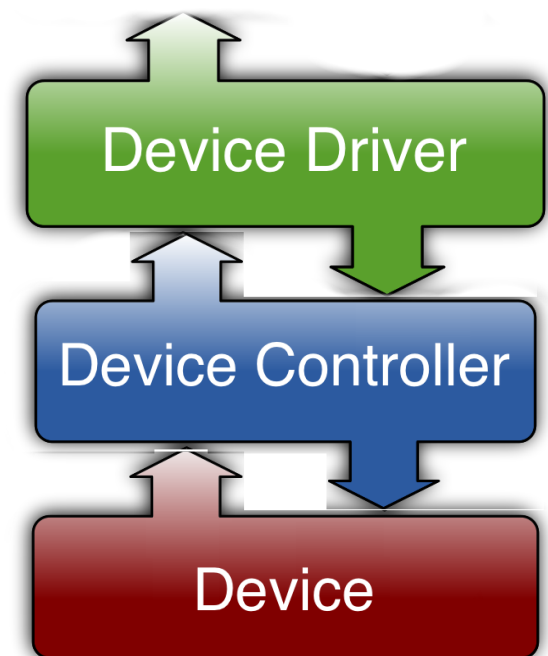
Device Controller - Hardware interface which helps connect a device or a group of similar devices to a computer system

Example: disk controller, USB controller



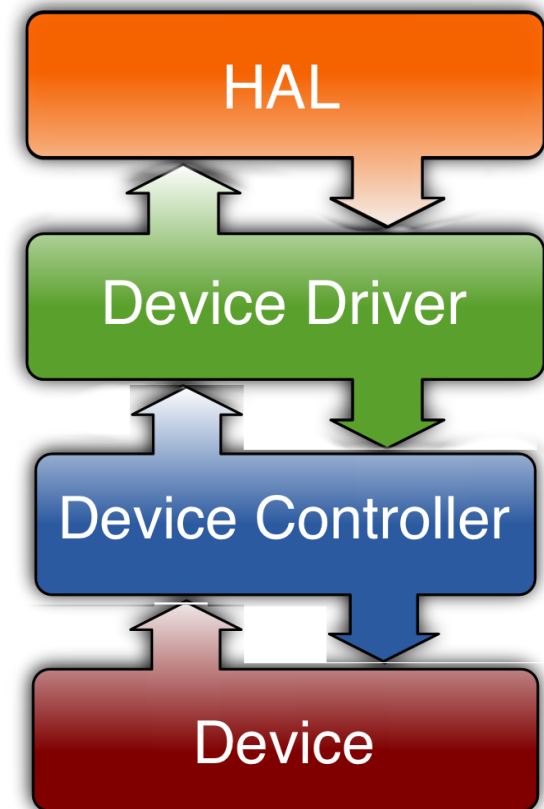
Terminology

Device Driver - A software routine that is part of the OS, and is used to communicate with and control a device through its device controller.



Terminology

Hardware Abstraction Layer - are software routines or modules that provide a device independent layer through which to communicate with hardware.



BIOS History

- Early days of personal computers were DIY
- Assembling hardware and programming simple programs were the norm (Assembly Language)



Altair image © Michael Holley and placed in Public Domain

© 2020 Trevor Bakker and The University of Texas at Arlington

Assembly Language

- Tedious
- Large complex programs are difficult and time-consuming to develop
- Users began to demand more functionality in programs

```
SUBTTL DSKREAD -- PHYSICAL DISK READ
PAGE
        procedure   DskRead,NEAR
ASSUME DS:NOTHING,ES:NOTHING

; Inputs:
;     DS:BX = Transfer addr
;     CX = Number of sectors
;     DX = Absolute record number
;     ES:BP = Base of drive parameters
; Function:
;     Call BIOS to perform disk read
; Outputs:
;     DI = CX on entry
;     CX = Number of sectors unsuccessfully transferred
;     AX = Status word as returned by BIOS (error code in AL if error)
;     Zero set if OK (from BIOS)
;     Zero clear if error
; SI Destroyed, others preserved

        PUSH    CX
        MOV     AH,ES:[BP.dpb_media]
        MOV     AL,ES:[BP.dpb_UNIT]
        PUSH    BX
        PUSH    ES
        invoke  SETREAD
        JMP     DODSKOP

SUBTTL DWRITE -- SEE ABOUT WRITING
PAGE
        entry    DWRITE
ASSUME DS:NOTHING,ES:NOTHING
```

Wild west of PC hardware

- 1970's and 1980's saw the commoditization of PCs
- By 1981 there were 200,000 microcomputers running CP/M, in more than 3000 different hardware configurations
- Operating systems need a better way to interface to a wide variety of hardware

Needed a better way

- Needed to be able to add new devices without writing directly to each new piece of hardware.
- Writing assembly to address hardware is fun but it's no way to build a complex and widely used operating system.
- Programmers needed to be able to write programs that could work on different systems with little changes.

BIOS developed

- BIOS (Basic Input/Output System) was invented by Gary Kildall
- First appeared in the CP/M operating system in 1975
- Stored on ROM on the motherboard
- Replaced later with firmware BIOS that can be flashed and updated without removing the chip.

Terminology

BIOS (Basic Input/Output System) -Software that abstracts the common device hardware, such as keyboards, basic video, and system clock.



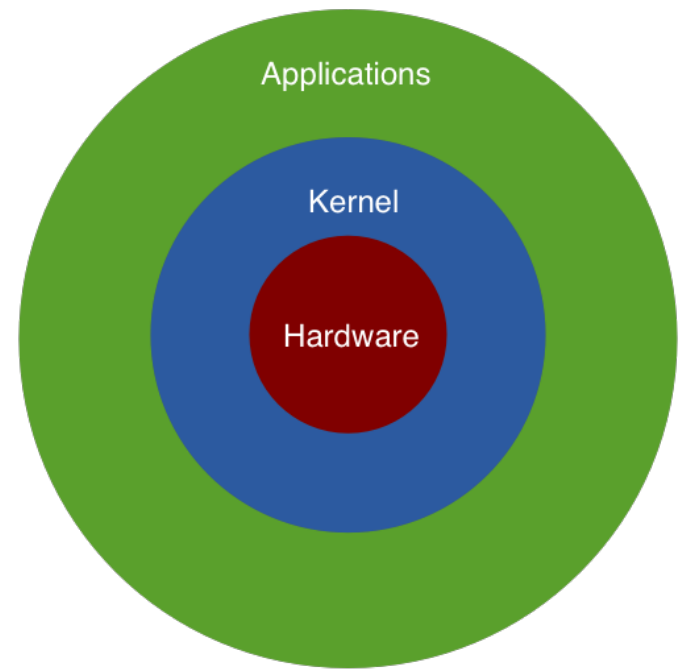
Image © Timothy Vollmer licensed
under CC BY-NC-SA 2.0c

Why BIOS and abstraction layers?

- Abstracts the hardware
- The OS does can deal with all common devices the same.
- Don't need to rev your OS to handle a 112 key keyboard instead of an 88 key keyboard

Terminology

Kernel - The part of the OS that implements basic functionality and is always resident in memory.

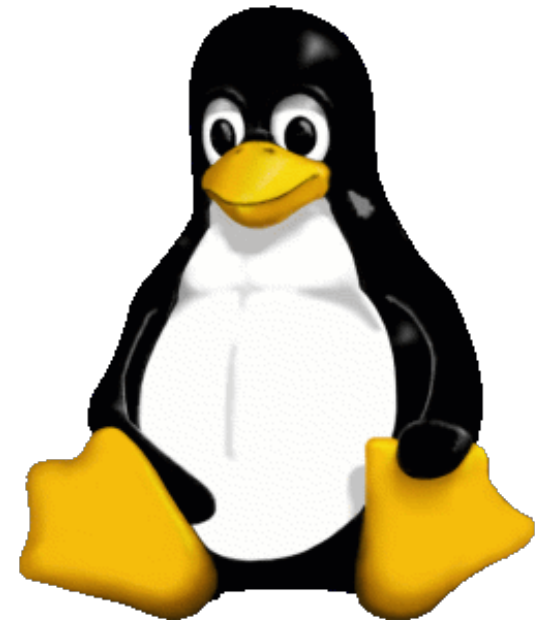


Monolithic Kernel

- The first operating systems were written as a single program.
- Core functionality such as memory allocation and scheduling, as well as services such as device drivers and the file system exist in the same space.
- Problems with bloat. OS occupies more and more memory
- A bug in a device driver can bring down the entire system

Linux

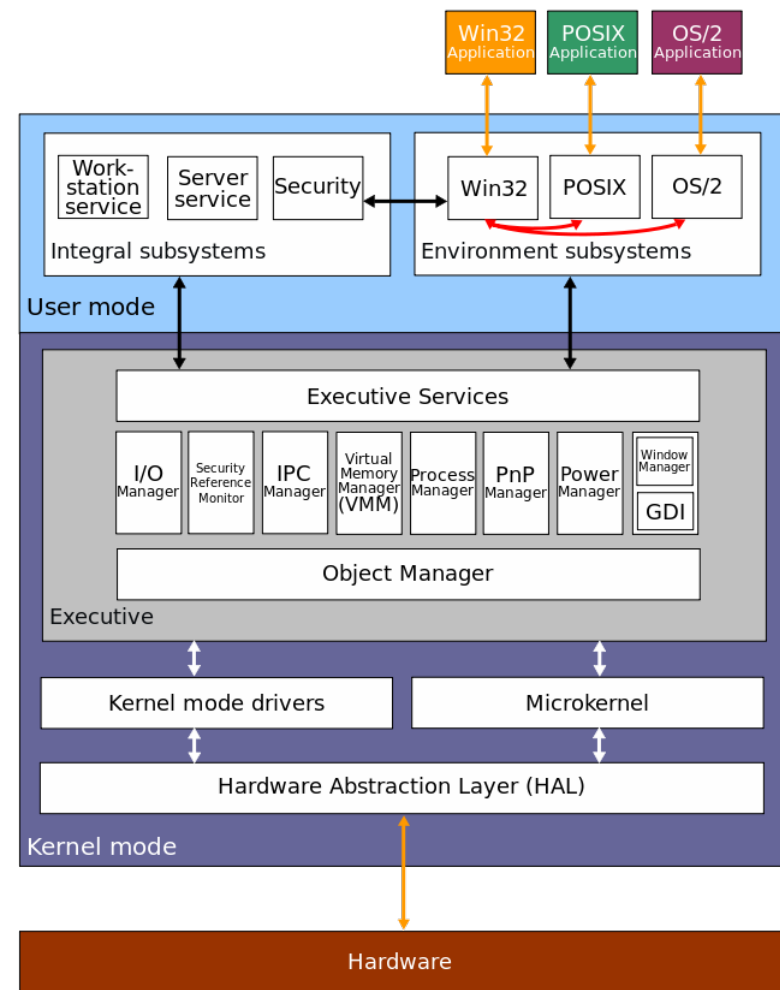
- Linux is a monolithic kernel
- Monolithic does not mean it isn't modular
- Linux supports dynamic loadable modules
- Chapter 6 we will discuss Linux



Tux image © 1995 by lewing@isc.tamu.edu

Layered Architecture

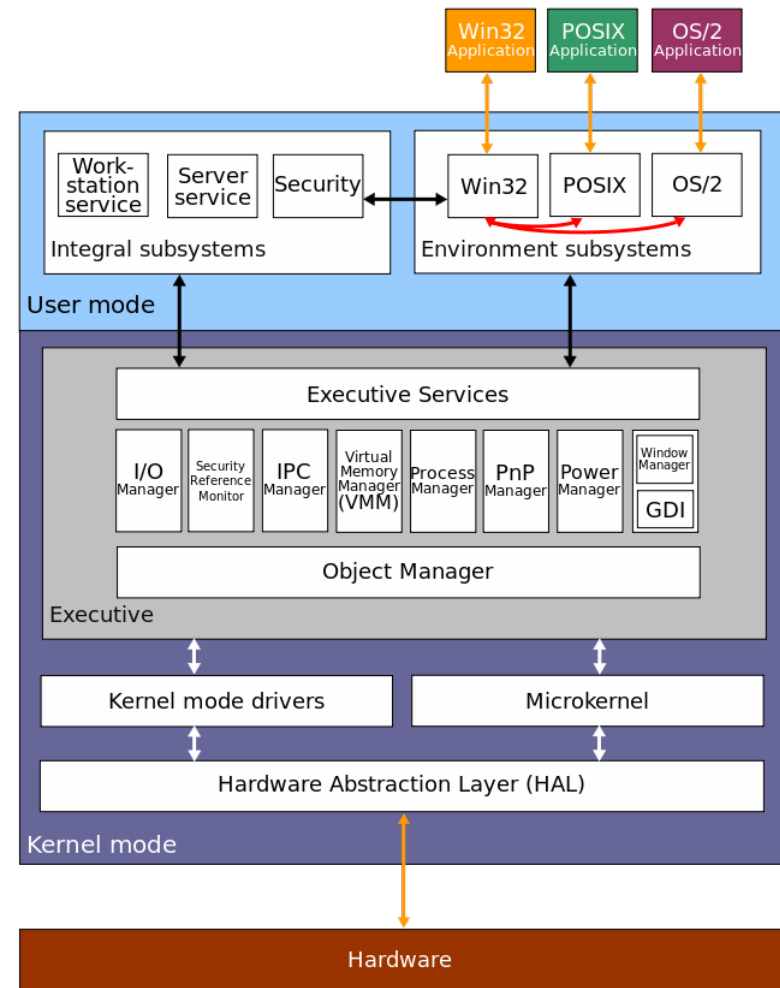
- Modules at one level call functions provided by modules at the same or lower level.
- Example: Windows NT Architecture (Windows 2000, Windows XP, Vista, Windows 7, Windows 8)



Windows NT Architecture © 2005 Grm Wnr licensed under GNU Free Documentation License.

Layered Architecture

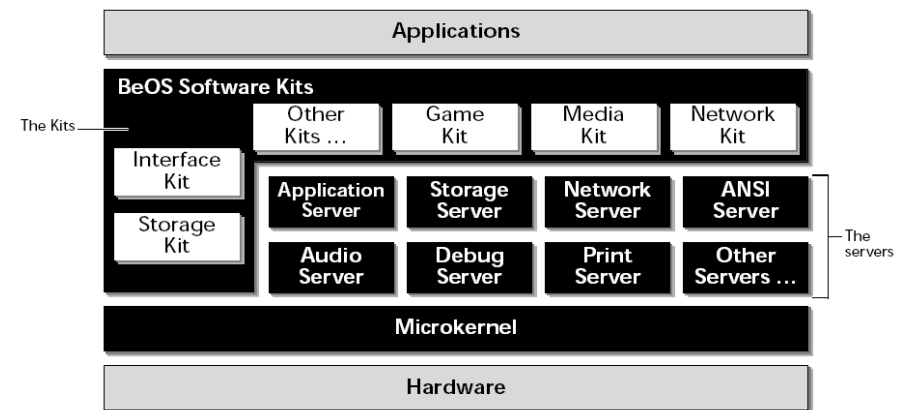
- Each layer provides a more abstract view than the layer below
- Usually only 2 or 3 layers are used because it's difficult to separate complex functionality into multiple clean layers



Windows NT Architecture © 2005 Grm Wnr licensed under GNU Free Documentation License.

Object Oriented Architecture

- Each O/S module is implemented as an object and provides services
- Any object can invoke the services of another
- Example: BeOS



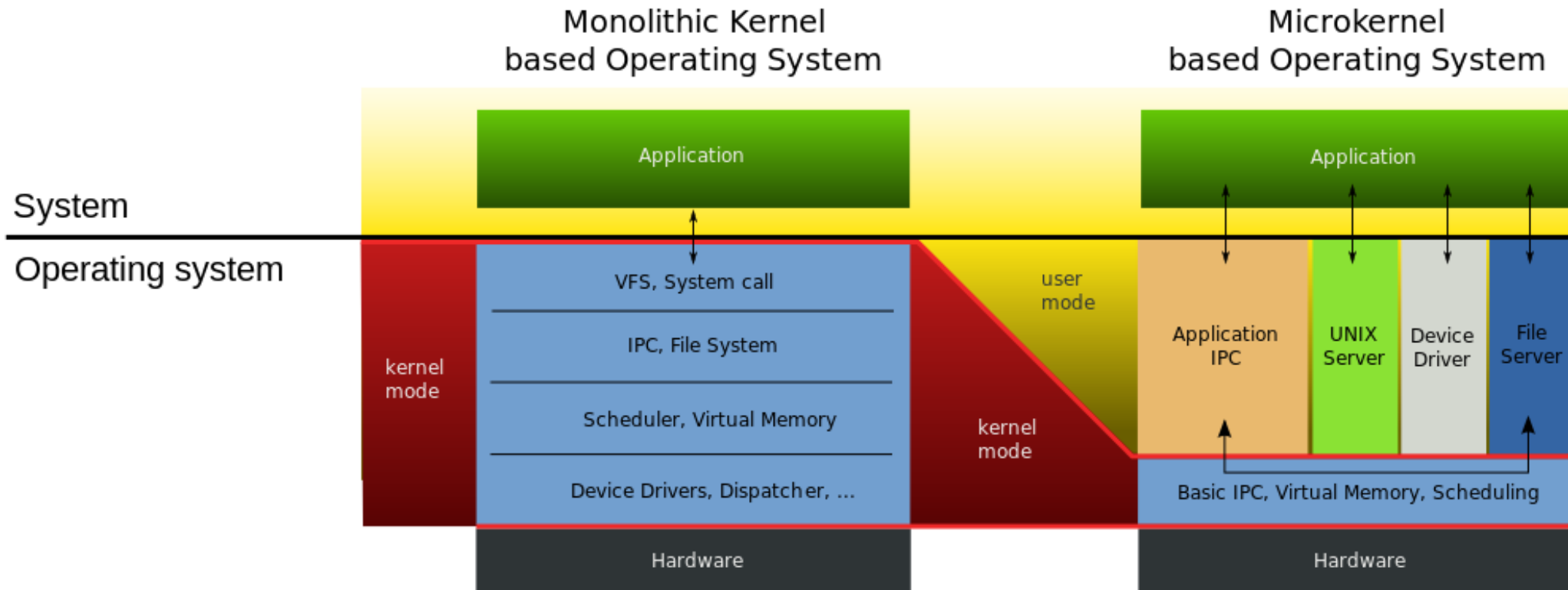
Microkernel

- Only basic functionality is included in the kernel
 - What is basic? Only code that must run in supervisor mode because it must use privileged resources such as protected instructions
- Everything else runs in user space.

Microkernel

- Theoretically more robust since limiting the amount of code that runs in protected mode limits the number of catastrophic crashes
- Easier to inspect for flaws since a smaller portion of code exists
- May run slower since there are more interrupts from user space to kernel
- Example: Minix

Micro v. Monolithic



"OS-structure2" by Golftheman - <http://en.wikipedia.org/wiki/Image:OS-structure.svg>. Licensed under Public domain via Wikimedia Commons - <http://commons.wikimedia.org/wiki/File:OS-structure2.svg#mediaviewer/File:OS-structure2.svg>

Process Management

- When I type `vi proc.c` and press enter the OS creates a process
 - A process is a program in execution
 - Processes are also called task or job
 - In particular, you will see the term task used with Linux.

Pseudo parallelism

- » In a single CPU system the CPU is only running one process at a time
 - » Process are switched over the course of time giving the illusion of of parallelism.
- » Contrast with a multiprocessor system in which multiple CPU share the same memory and execute processes in parallelism.

Process Model

- » All running software on the computer is organized into a number of sequential processes.
- » Conceptually each process has its own virtual CPU.
- » In reality, the CPU is shared by the processes.
 - » Multiprogramming: The rapid switching back and forth

The Process Model (2)

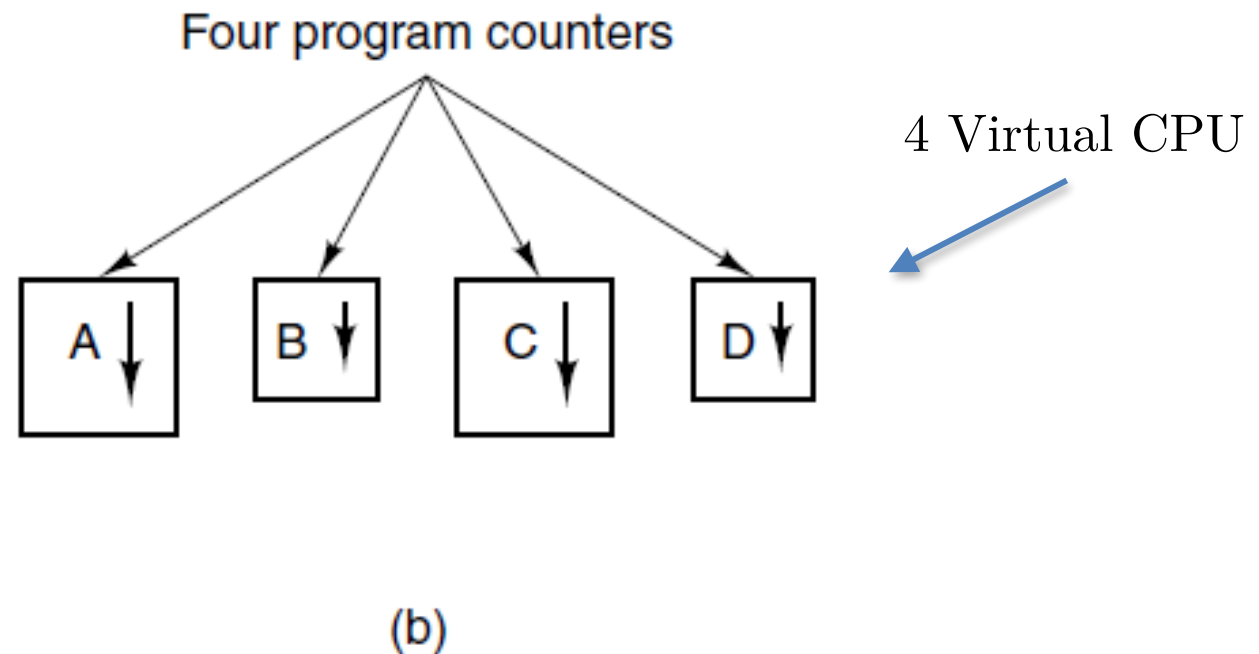


Figure 2-1. (b) Conceptual model of four independent, sequential processes.

The Process Model (1)

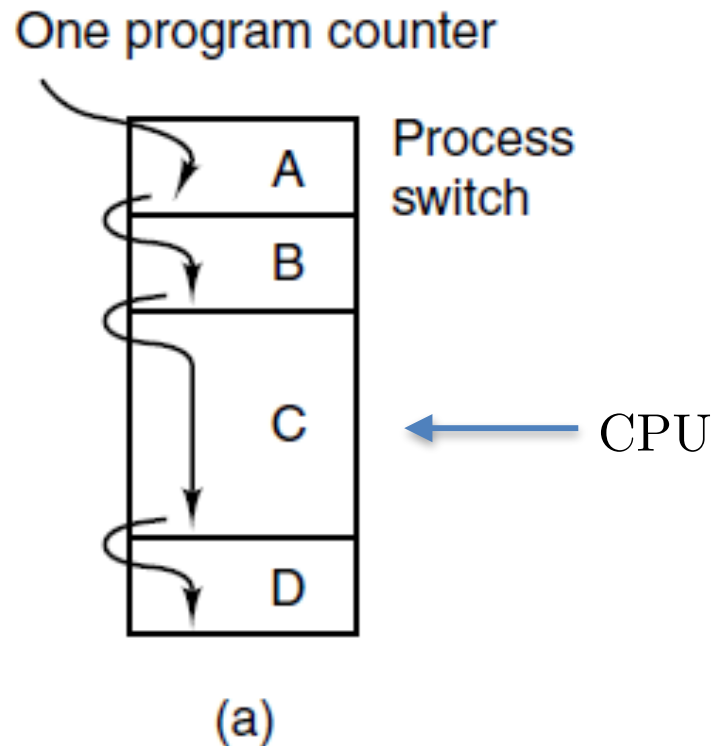


Figure 2-1. (a) Multiprogramming of four programs.

The Process Model (3)

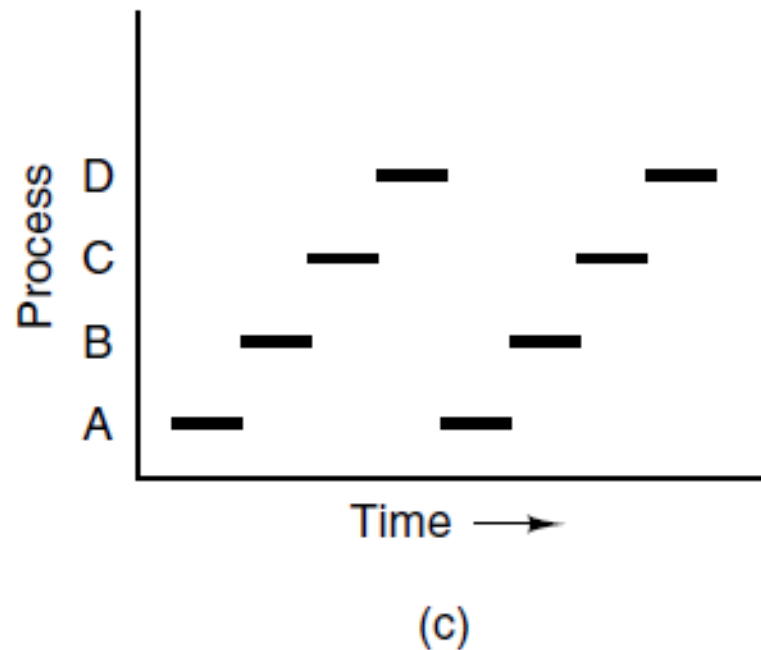
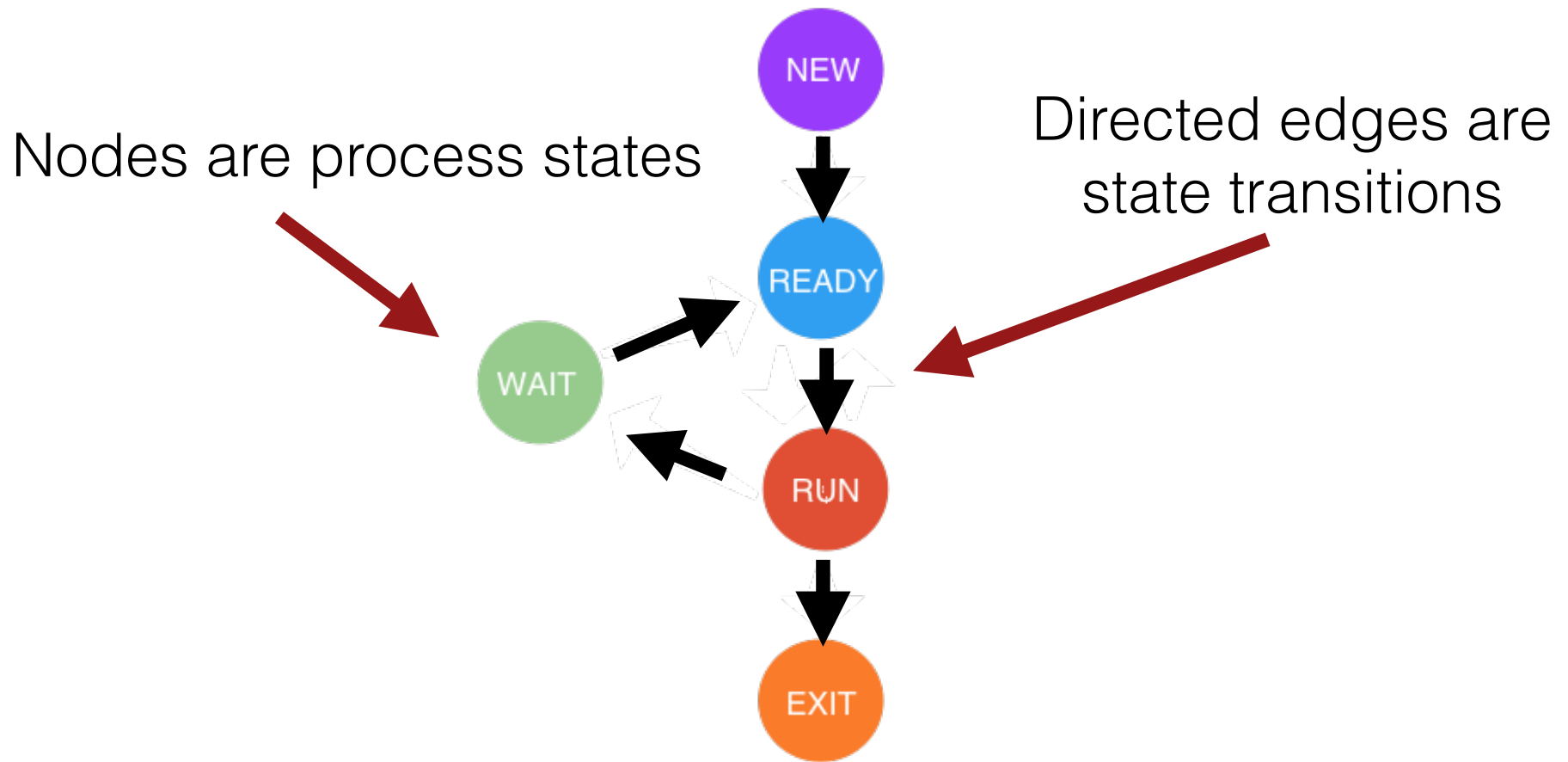


Figure 2-1. (c) Only one program is active at once.

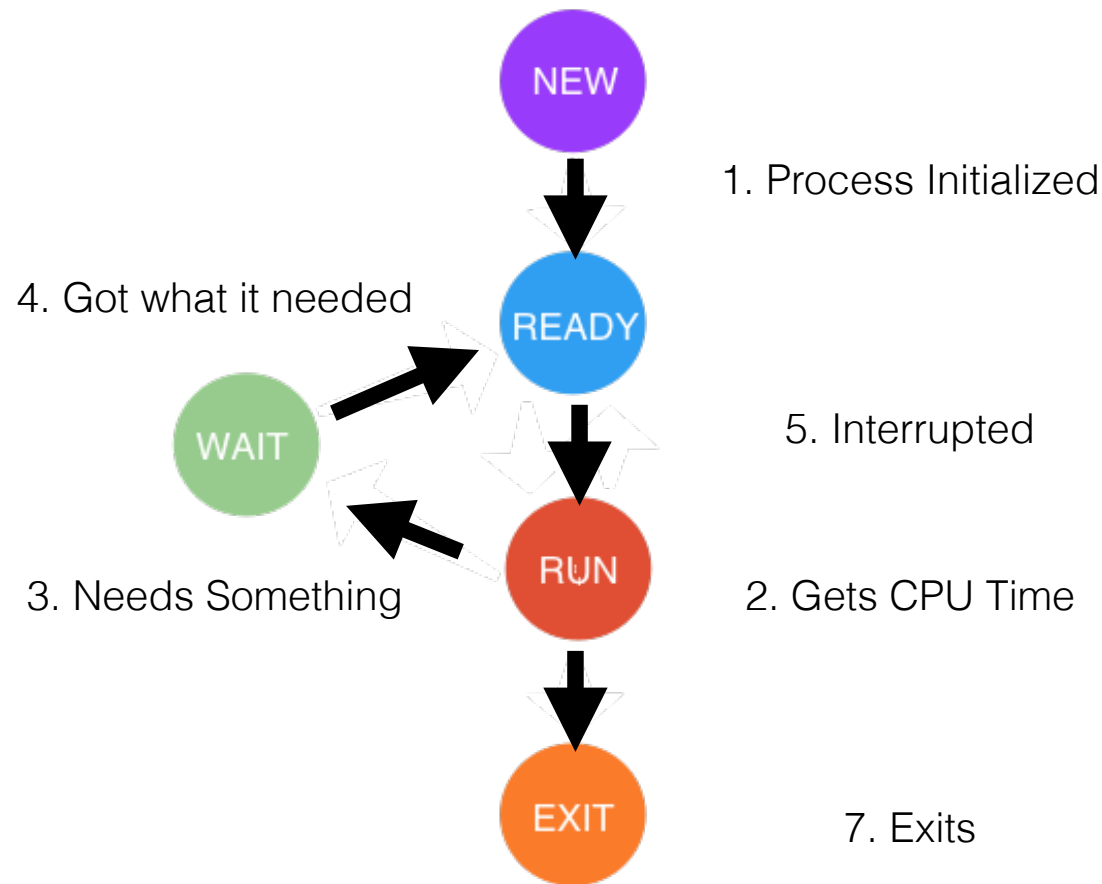
Programs v. Processes

- Program - a sequence of instructions written to perform a specified task.
- Process - an instance of a program in execution.
- A computer program is a passive collection of instructions; a process is the actual execution of those instructions.

Process State Diagram



Process Lifecycle



Types of Processes

- User Processes - Applications executing on behalf of a user.
 - Example: World of Warcraft
- System Program Processes - Programs that perform a common system service instead of a specific end-user service.
 - Example: gcc
- OS Processes - Also known as daemons. These are processes that execute OS functions.
 - Example: network services

Process Execution Modes

- Privileged - OS kernel processes which can execute all types of hardware operations and access all memory
- User Mode - Can not execute low-level I/O. Memory protection keeps these processes from trashing memory owned by the OS or other processes.

Kernel Space v. User Space

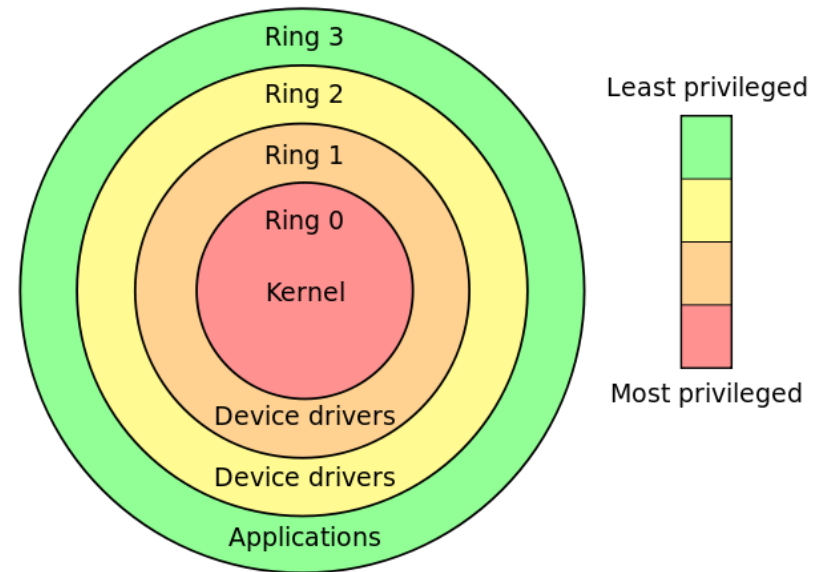
- A process is executing either in user space, or in kernel space. Depending on which privileges, address space a process is executing in, we say that it is either in user space, or kernel space.
- When executing in user space, a process has normal privileges and can and can't do certain things. When executing in kernel space, a process has every privilege, and can do anything.
- Processes switch between user space and kernel space using system calls.

Kernel Space v. User Space

- These two modes aren't just labels; they're enforced by the CPU hardware.
- If code executing in User mode attempts to do something outside its purview such as accessing a privileged CPU instruction or modifying memory that it has no access to:
 - Trappable exception is thrown. Instead of your entire system crashing, only that particular application crashes.

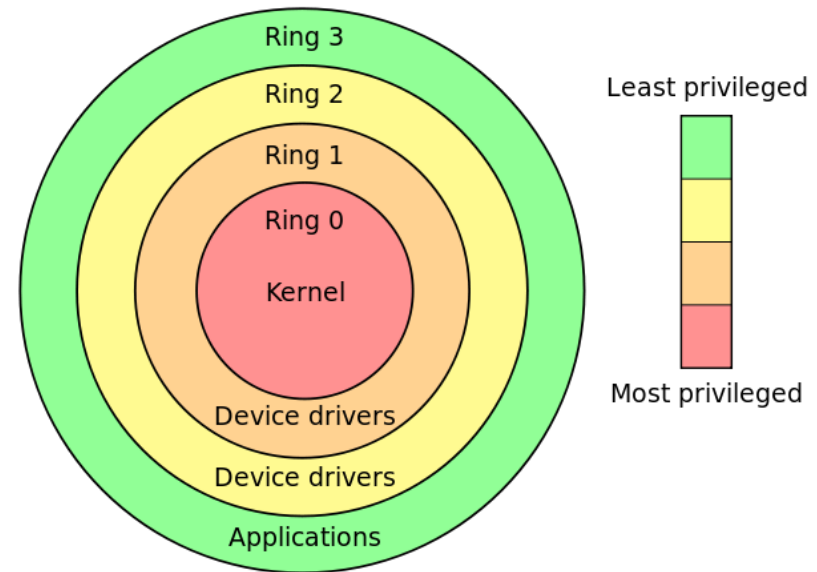
x86 Protection Rings

- Four privilege levels or rings, numbered from 0 to 3, with ring 0 being the most privileged and 3 being the least.
- Rings 1 and 2 aren't used in practice.



x86 Protection Rings

- Programs that run in Ring 0 can do anything with the system.
- Code that runs in Ring 3 should be able to fail at any time without impact to the rest of the computer system.



CPU Rings and Privilege

- CPU privilege level has nothing to do with operating system users.
- Whether you're root, Administrator, guest, or a regular user, it does not matter.
- All user code runs in ring 3 and all kernel code runs in ring 0, regardless of the OS user on whose behalf the code operates.

CPU Rings and Privilege

- Due to restricted access to memory and I/O ports, user mode can do almost nothing to the outside world without calling on the kernel.
- It can't open files, send network packets, print to the screen, or allocate memory.
- User processes run in a severely limited sandbox set up by ring zero.

CPU Rings and Privilege

- That's why it's impossible, by design, for a process to leak memory beyond its existence or leave open files after it exits. All of the data structures that control such things – memory, open files, etc – cannot be touched directly by user code; once a process finishes, the sandbox is torn down by the kernel.

How?

- The switch consists of three steps:
 1. Change the processor to kernel mode;
 2. Save and reload the MMU to switch to the kernel address space;
 3. Save the program counter and reload it with the kernel entry point.

How, in more details

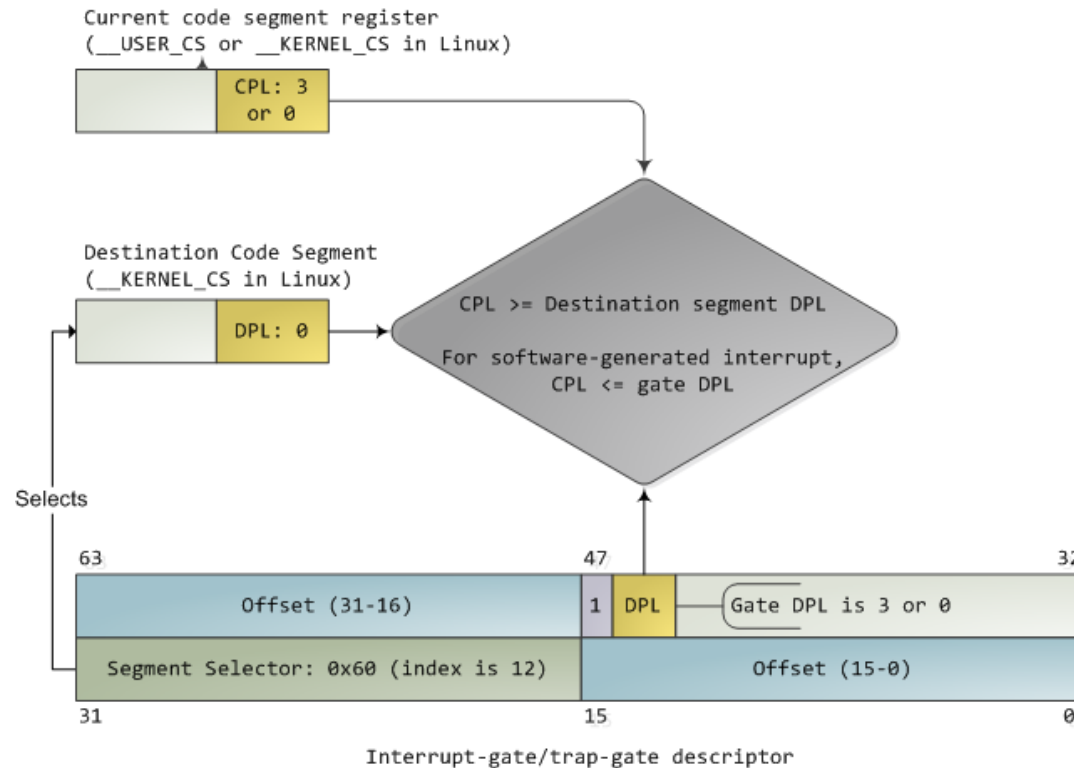
- Accomplished via gate descriptors and via the sysenter instruction.
- A gate descriptor is a segment descriptor and comes in four sub-types:
 1. call-gate descriptor.
 2. interrupt-gate descriptor.
 3. trap-gate descriptor.
 4. task-gate descriptor.

How does it know?



- The CPU keeps track of the current and requested privilege levels via the data segment selector and the code segment selector.

How, in more details



- DPL - Desired privilege level
- CPL - Current privilege level

CPU States

- Running a user process (ring 3, your code)
- Running a syscall (ring 0, kernel code)
- Running a interrupt handler (ring 0, kernel code)
- Running a kernel thread (ring 0, kernel code)

x86 Privileged Instructions

Privileged Level (Ring 0) Instructions

Instruction	Description
LGDT	Loads an address of a GDT into GDTR
LLDT	Loads an address of a LDT into LDTR
LTR	Loads a Task Register into TR
MOV <i>Control Register</i>	Copy data and store in Control Registers
LMSW	Load a new Machine Status WORD
CLTS	Clear Task Switch Flag in Control Register CR0
MOV <i>Debug Register</i>	Copy data and store in debug registers
INVD	Invalidate Cache without writeback
INVLPG	Invalidate TLB Entry
WBINVD	Invalidate Cache with writeback
HLT	Halt Processor
RDMSR	Read Model Specific Registers (MSR)
WRMSR	Write Model Specific Registers (MSR)
RDPMC	Read Performance Monitoring Counter
RDTSC	Read time Stamp Counter

Returning back to user space

- Finally, when it's time to return to ring 3, the kernel issues an iret or sysexit instruction to return from interrupts and system calls, respectively, thus leaving ring 0 and resuming execution of user code with a CPL of 3.

Cost of Promotion

- Promoting from user to kernel space is expensive.
 - 1000 - 1500 cycles.
- This mechanism consists of three steps:
 1. Change the processor to kernel mode.
 2. Save and reload the MMU to switch to the kernel address space.
 3. Save the program counter and reload it with the kernel entry point.

Cost is worth it

- The CPU's strict segregation of code between User and Kernel mode is completely transparent to users, but it is the difference between a computer that crashes all the time (applications) and a computer that crashes catastrophically all the time (entire OS, think blue screen of death).

How does the OS track a process?

- Each process has a unique process identifier, or PID
- The POSIX standard guarantees a PID as a signed integral datatype.
- The datatype is an opaque type called `pid_t`

Process Control Block

- The kernel maintains a data structure to keep track of all the process information called the process control block or (PCB).
- The PCB also includes pointers to other data structures describing resources used by the process such as files (open files table) and memory (page tables).
- Maintains the state of the process
 - Over 170+ fields

Some Process Control Block Fields

Memory

Open streams/files

Devices, including abstract ones like windows

Links to condition handlers (signals)

Processor registers (single thread)

Process identification

Process state

Priority

Owner

Which processor

Links to other processes (parent, children)

Process group

Resource limits/usage

Access rights

Process Control Block

- Every task also needs its own stack
- So every task, in addition to having its own code and data, will also have a stack-area that is located in user-space, plus another stack-area that is located in kernel-space
- Each task also has a process-descriptor which is accessible only in kernel-space

Process Control Block

- Different information is required at different times
- UNIX for example has two separate places in memory with the process control block and process stack. One of them is in the kernel the other is in user space.
- Why? User land data is only required when the process is running.

Why a kernel stack?

- Kernels can't trust addresses provided by user
- Address may point to kernel memory that is not accessible to user processes
- Address may not be mapped
- Memory region may be swapped out from physical RAM
- Leftover data from kernel ops could be read by process
 - Kernel-level heartbleed bug

Process Tables

- The OS holds the process control blocks in the process table
- Usually implemented as an array of pointers to process control block structures
- Linux calls the PCB `task_struct`

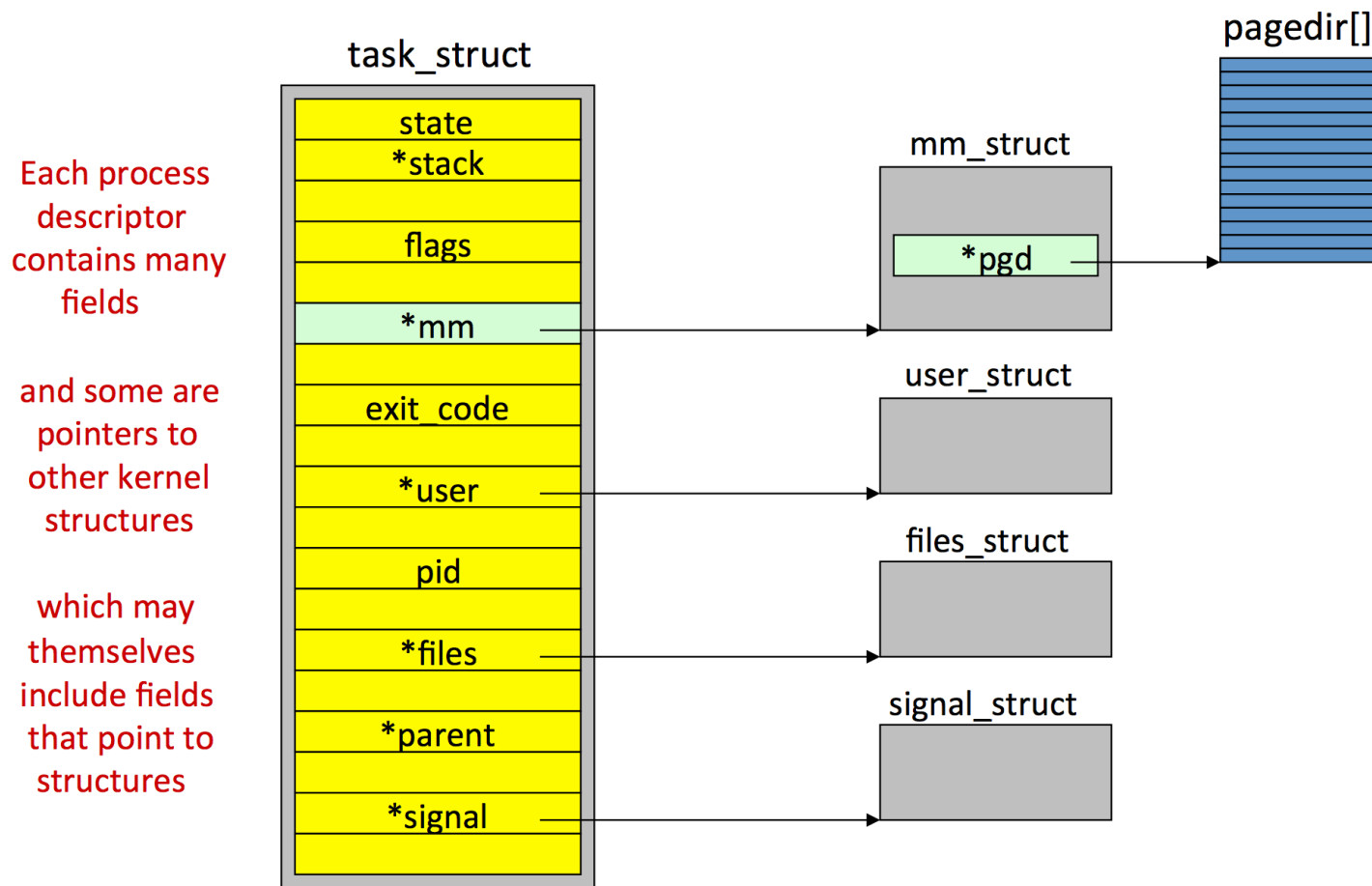
Linux PCB Data Structure

- /include/linux/sched.h

```
1166 struct task_struct {
1167     volatile long state;      /* -1 unrunnable, 0 runnable, >0 stopped */
1168     void *stack;
1169     atomic_t usage;
1170     unsigned int flags;      /* per process flags, defined below */
1171     unsigned int ptrace;
1172
1173     int lock_depth;          /* BKL lock depth */
1174
1175 #ifdef CONFIG_SMP
1176 #ifdef __ARCH_WANT_UNLOCKED_CTXSW
1177     int oncpu;
1178 #endif
1179 #endif
1180
1181     int prio, static_prio, normal_prio;
1182     unsigned int rt_priority;
1183     const struct sched_class *sched_class;
1184     struct sched_entity se;
1185     struct sched_rt_entity rt;
1186
1187 #ifdef CONFIG_PREEMPT_NOTIFIERS
1188     /* list of struct preempt_notifier: */
1189     struct hlist_head preempt_notifiers;
1190 #endif
1191
1192     /*
1193      * fpu_counter contains the number of consecutive context switches
1194      * that the FPU is used. If this is over a threshold, the lazy fpu
1195      * saving becomes unlazy to save the trap. This is an unsigned char
1196      * so that after 256 times the counter wraps and the behavior turns
1197      * lazy again; this to deal with bursty apps that only use FPU for
1198      * a short time
1199      */
1200     unsigned char fpu_counter;
1201     s8 oomkilladj; /* OOM kill score adjustment (bit shift). */

```

Linux Task_Struct



Memory Management

- Before the process can start the binary executable must be brought into main memory .

1. Memory to hold the programs executable code must be allocated.
2. Memory for the program's data and temporary storage.

- Multiple processes can be resident in memory at the same time.

REGION TYPE	VIRTUAL	RESIDENT
=====	=====	=====
Kernel Alloc Once	4K	4K
MALLOC	36.2M	412K
MALLOC (admin)	24K	8K
MALLOC_LARGE (reserved)	128K	0K
STACK GUARD	56.0M	0K
Stack	8192K	20K
VM_ALLOCATE	8K	8K
__DATA	784K	536K
__LINKEDIT	66.2M	10.9M
__TEXT	7556K	5544K
shared memory	4K	4K
=====	=====	=====
TOTAL	174.6M	17.3M
TOTAL, minus reserved VM space	174.5M	17.3M

Physical v. Logical Addresses

- » Originally programs were compiled to reference addresses that corresponded one to one with physical memory addresses.
- » Unknowingly using concept of logical and physical memory
 - Logical addresses - Set of addresses the CPU generates as the program is executed.
 - Physical addresses - Set of addresses used to reference physical memory.

Binding

The process of determining where in the physical memory the subroutine should go and making the reference in the main routine point to the subroutine.

Binding Model

- Coding the program
- Translating into object module
 - Compiler, Assembler, Interpreter
- Linking with other modules
- Loading into primary memory
- Running the process

Binding at Coding Time

- In a hypothetical embedded system
- Could locate things by hand
 - Put main module at location 100
 - Put the subroutine at location 500
- ORG assembler directive
 - In main module make call to ORG 500 to point to subroutine code
 - In subroutine make call to ORG 100 to point to main module.
- Dedicated hardware locations

Binding at Linking Time

- Subroutines we really don't care where they are located.
 - Use symbolic names
- Assembler outputs object module that includes references that need to be fixed or linked.
- Linker loads our module, processes it and finds the items it needs to find. Once found it will go back and link the references in the main module to the addresses found.

Binding at Linking Time

- Benefits of binding at link time?
 - Flexibility - We don't change code
- Cons?
 - Objects are bigger. Carry references
 - Time

Compile Time Binding

```
int main()  
{  
    int x = 0xDEAD;  
    int y = 0xBEEF;  
    int z = x + y;  
    return z;  
}
```

Compile Time Binding

```
[tbakker@omega ~]$ objdump -d main
```

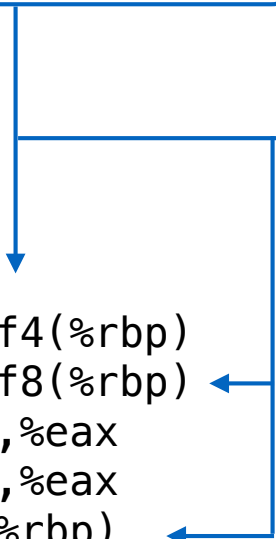
```
main:      file format elf64-x86-64
```

```
Disassembly of section .text:
```

```
00000000004000b0 <main>:
```

4000b0: 55	push	%rbp
4000b1: 48 89 e5	mov	%rsp,%rbp
4000b4: c7 45 f4 ad de 00 00	movl	\$0xdead,0xffffffffffff4(%rbp)
4000bb: c7 45 f8 ef be 00 00	movl	\$0xbeef,0xffffffffffff8(%rbp)
4000c2: 8b 45 f8	mov	0xffffffffffff8(%rbp),%eax
4000c5: 03 45 f4	add	0xffffffffffff4(%rbp),%eax
4000c8: 89 45 fc	mov	%eax,0xffffffffffffc(%rbp)
4000cb: 8b 45 fc	mov	0xffffffffffffc(%rbp),%eax
4000ce: c9	leaveq	
4000cf: c3	retq	

Variables x, y, and z
have an absolute
address



Link Time Binding

```
#include "header.h"

int main()
{
    int x = 0xDEAD;
    int y = 0xBEEF;
    int z = 0;

    z = add_numbers( x, y );

    return z;
}
```

Link Time Binding

main.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:

0:	55	push	%rbp
1:	48 89 e5	mov	%rsp,%rbp
4:	48 83 ec 10	sub	\$0x10,%rsp
8:	c7 45 f4 ad de 00 00	movl	\$0xdead,0xfffffffffffff4(%rbp)
f:	c7 45 f8 ef be 00 00	movl	\$0xbeef,0xfffffffffffff8(%rbp)
16:	c7 45 fc 00 00 00 00	movl	\$0x0,0xfffffffffffffc(%rbp)
1d:	8b 75 f8	mov	0xfffffffffffff8(%rbp),%esi
20:	8b 7d f4	mov	0xfffffffffffff4(%rbp),%edi
23:	e8 00 00 00 00	call	add_numbers ←
28:	89 45 fc	mov	%eax,0xfffffffffffffc(%rbp)
2b:	8b 45 fc	mov	0xfffffffffffffc(%rbp),%eax
2e:	c9	leaveq	
2f:	c3	retq	


add_numbers has not
been resolved to an
address

Link Time Binding

We can use nm to look at the symbols in our object file

U means the symbol is undefined in our object file

```
[tbakker@omega ~]$ nm main.o
00000000000000000000 U add_numbers
00000000000000000000 T main
```



Link Time Binding

```
[tbakker@omega ~]$ readelf --relocs main.o
```

```
Relocation section '.rela.text' at offset 0x540 contains 1 entries:
```

Offset	Info	Type	Sym. Value	Sym. Name + Addend
000000000024	000900000002	R_X86_64_PC32	0000000000000000	add_numbers + ffffffffffffffff



The compiler leaves behind a *relocation* (of type R_X86_64_PC32) which is saying "in the final binary, patch the value at offset 0x24 in this object file with the address of symbol add_numbers."

Runtime Binding

```
[tbakker@omega ~]$ gcc -shared header.c -fPIC -o libaddnumbers.so  
[tbakker@omega ~]$ gcc main.c -laddnumbers -L. -o main -nostdlib
```

Let's compile our example and tell the compiler we are going to use a shared library

Runtime Binding

```
[tbakker@omega ~]$ objdump -d main
```

```
main:      file format elf64-x86-64
```

```
Disassembly of section .plt:
```

```
00000000004002e0 <add_numbers@plt-0x10>:
```

```
4002e0: ff 35 b2 01 20 00    pushq 2097586(%rip)      # 600498 <_GLOBAL_OFFSET_TABLE_+0x8>
4002e6: ff 25 b4 01 20 00    jmpq  *2097588(%rip)     # 6004a0 <_GLOBAL_OFFSET_TABLE_+0x10>
4002ec: 0f 1f 40 00          nopl  0x0(%rax)
```

```
00000000004002f0 <add_numbers@plt>:
```

```
4002f0: ff 25 b2 01 20 00    jmpq  *2097586(%rip)     # 6004a8 <_GLOBAL_OFFSET_TABLE_+0x18>
4002f6: 68 00 00 00 00      pushq $0x0
4002fb: e9 e0 ff ff ff      jmpq  4002e0 <add_numbers@plt-0x10>
```

```
Disassembly of section .text:
```

```
0000000000400300 <main>:
```

```
400300: 55                  push  %rbp
400301: 48 89 e5            mov   %rsp,%rbp
400304: 48 83 ec 10         sub   $0x10,%rsp
400308: c7 45 f4 ad de 00 00 movl  $0xdead,0xffffffffffffff4(%rbp)
40030f: c7 45 f8 ef be 00 00 movl  $0xbeef,0xffffffffffffff8(%rbp)
400316: c7 45 fc 00 00 00 00 movl  $0x0,0xffffffffffffffc(%rbp)
40031d: 8b 75 f8            mov   0xffffffffffffff8(%rbp),%esi
400320: 8b 7d f4            mov   0xffffffffffffff4(%rbp),%edi
400323: e8 c8 ff ff ff     callq 4002f0 <add_numbers@plt>
400328: 89 45 fc            mov   %eax,0xffffffffffffffc(%rbp)
40032b: 8b 45 fc            mov   0xffffffffffffffc(%rbp),%eax
40032e: c9                  leaveq
40032f: c3                  retq
```

The compiler has told the linker and loader that `add_numbers` can be found using the procedure linkage table which will then point to the global offset table

Interrupts

- Mechanism used by the OS to signal the system that a high-priority event has occurred that requires immediate attention.
- I/O drives a lot of interrupts. Mouse movements, disk reads, etc
- The controller causing the interrupt places the interrupt number in an interrupt register. The OS must then take action

Interrupt Vector

- Normal technique for handling interrupts is a data structure called the interrupt vector.
- One entry for each interrupt.
- Each entry contains the address for the interrupt service routine.
- Some small hardware devices don't provide an interrupt system and instead uses an event loop. This is known as a status-driven system.

System Calls

- How do our programs utilize the resources controlled by the OS or communicate with other process?
- Because user mode software can not access hardware devices directly, they must notify the operating system in order to complete system tasks. This includes displaying text, obtaining input from user, printing a document, etc.

System Calls

- System calls - Function call provided by the operating system
- Different than a normal function call

System Calls

- Instead of directly calling a section of code the system call instruction issues an interrupt.
- By not allowing the application to execute code freely the operating system can verify that the application has appropriate privileges to call the function.
- glibc library