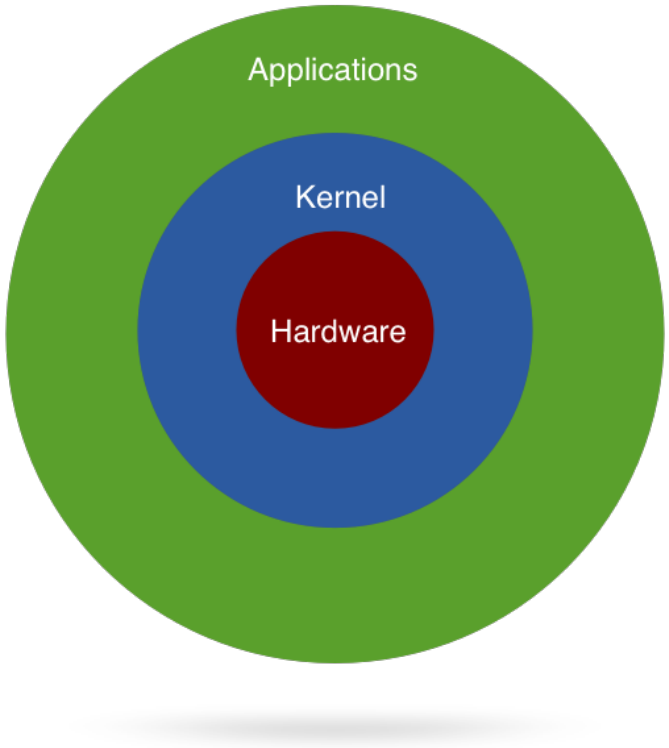
Introduction

Chapter 1

Components of a Modern Computer (1)

- One or more processors
- Main memory
- Disks
- Printers
- Keyboard
- Mouse
- Display
- Network interfaces
- I/O devices

Kernel



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

Components of a Modern Computer (2)

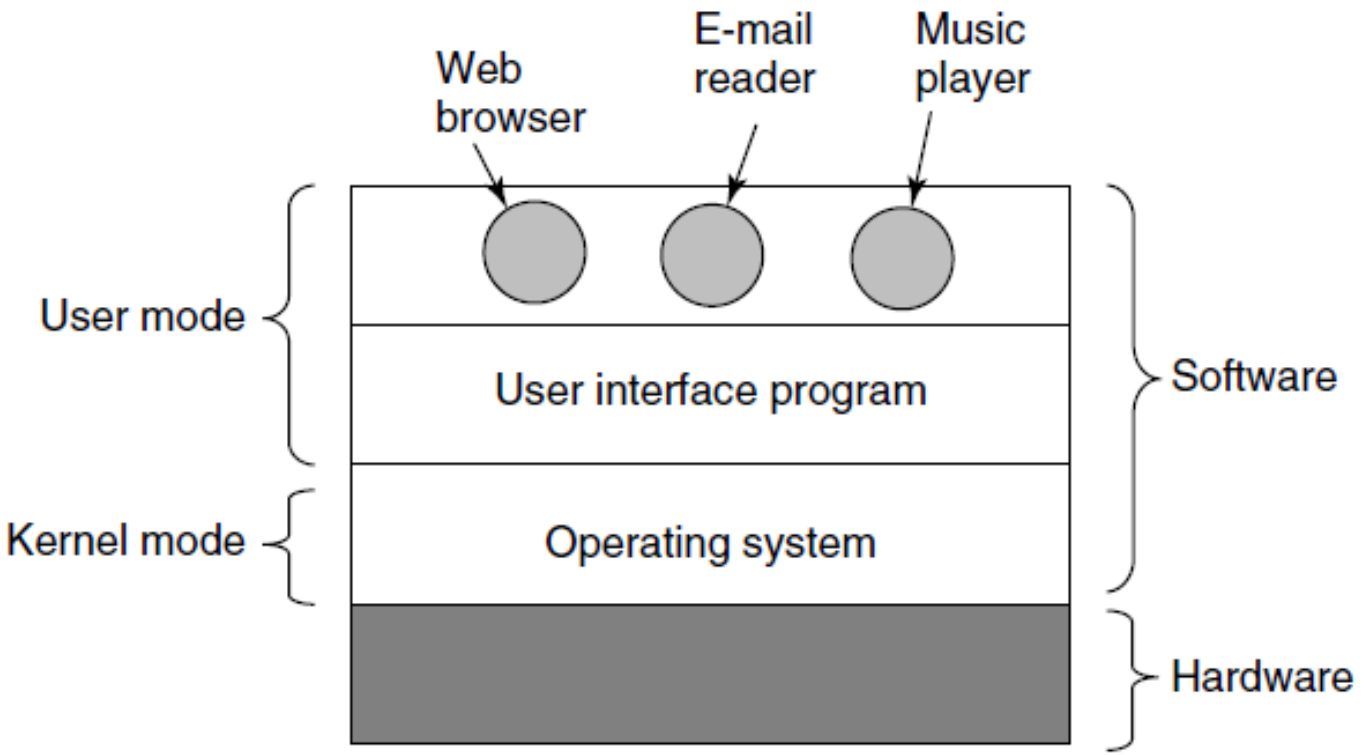


Figure 1-1. Where the operating system fits in.

The Operating System as an Extended Machine

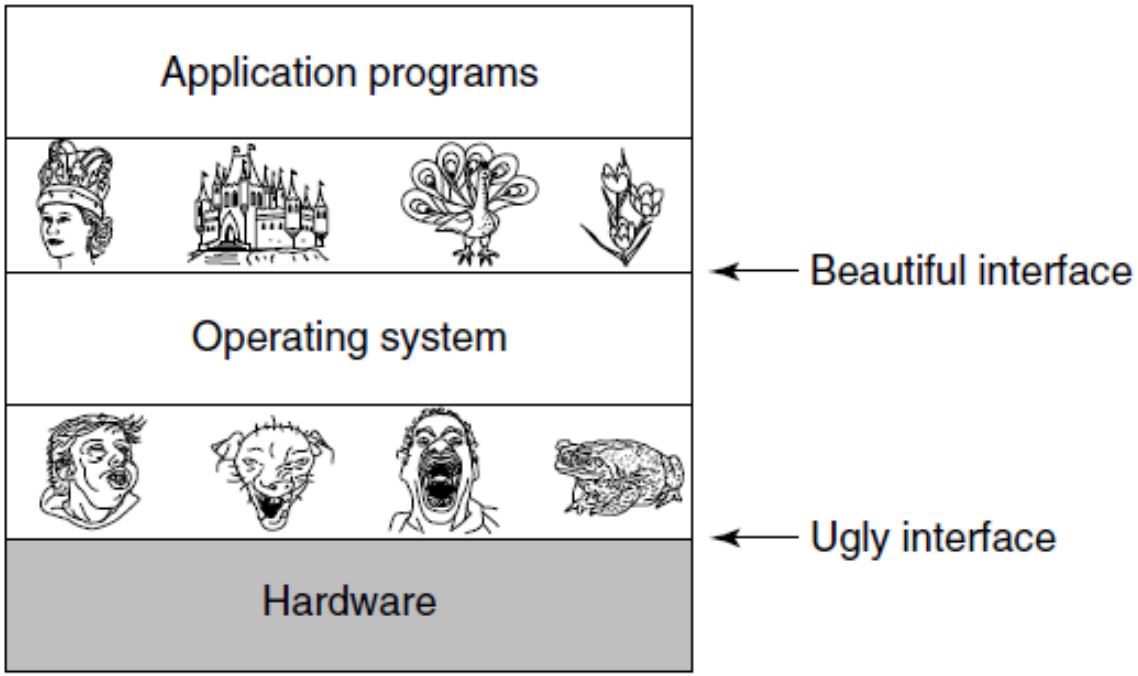


Figure 1-2. Operating systems turn ugly hardware into beautiful abstractions.

The Operating System as a Resource Manager

- Top down view
  - Provide abstractions to application programs
- Bottom up view
  - Manage pieces of complex system
- Alternative view
  - Provide orderly, controlled allocation of resources

The Operating System's Role

- Provide the user with a cleaner model of the computer
  - An abstracted interface to the resources
- » Manage the resources

Resource Management

- Multiplexing - Sharing resources in two different ways; time and space
  - Time: CPU
  - Space: Memory or disk

Processors (1)

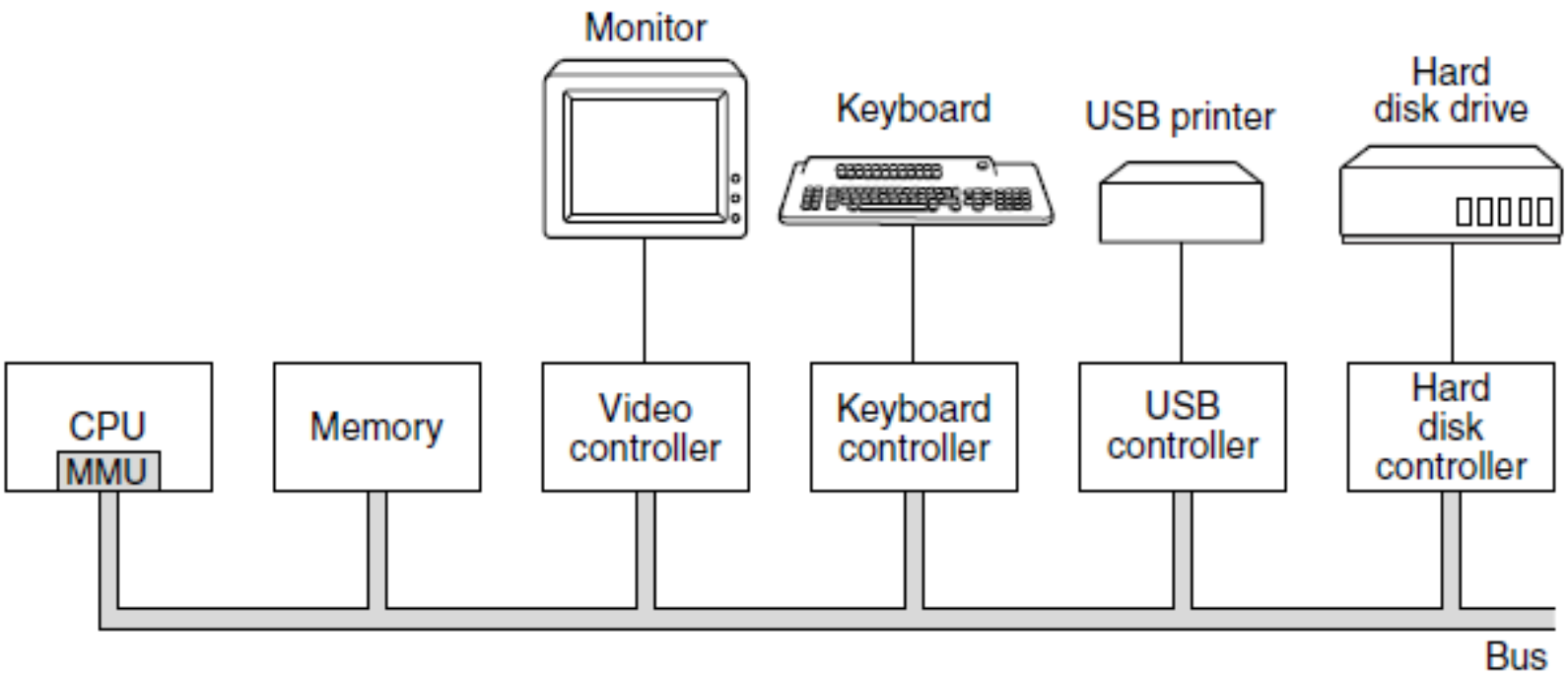
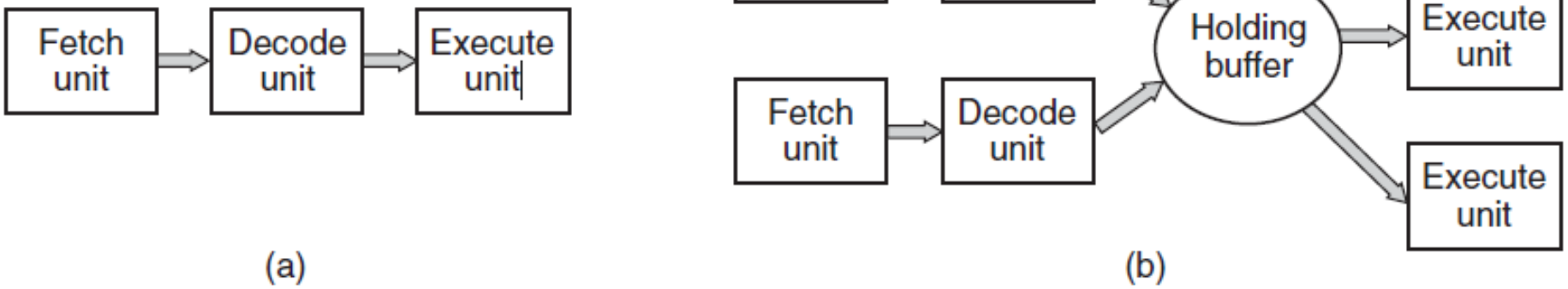


Figure 1-6. Some of the components of a simple personal computer.

Processors (2)



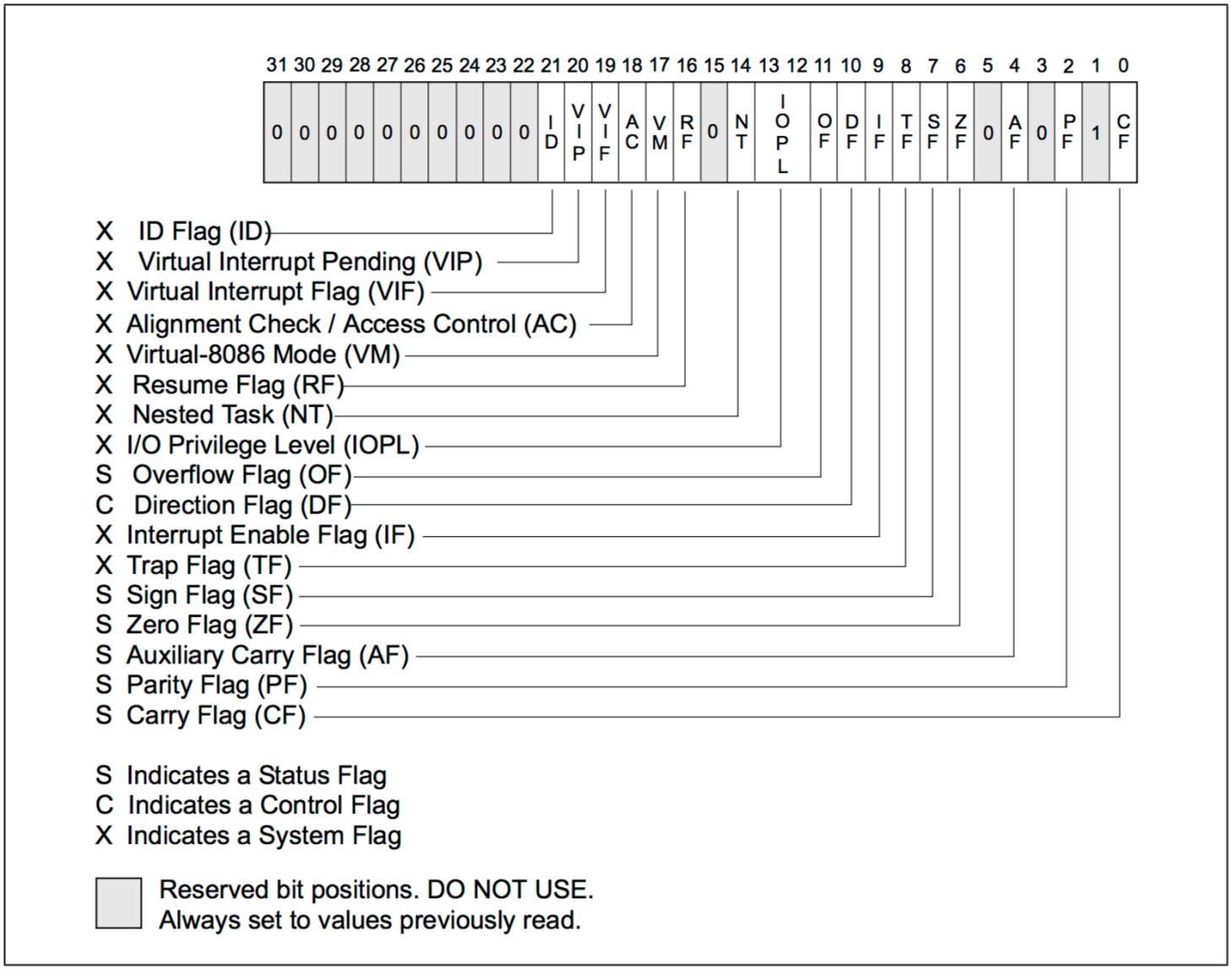
Registers

- x86_64
  - 16 general 64-bit registers
    - (rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15)
  - 16 128-bit XMM registers

Registers We'll Care About

- Program Counter (%rip) - Contains the memory address of the next instruction to be fetched
- Stack Pointer - Points to the top of the current stack in memory.
- Program Status Word (PSW) - Holds the condition code bits, the CPU priority, the mode (user or kernel), and other control bits
    - Called EFLAGS register on x86_64 architecture.

Program Status Word (PSW)



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.

Memory (1)

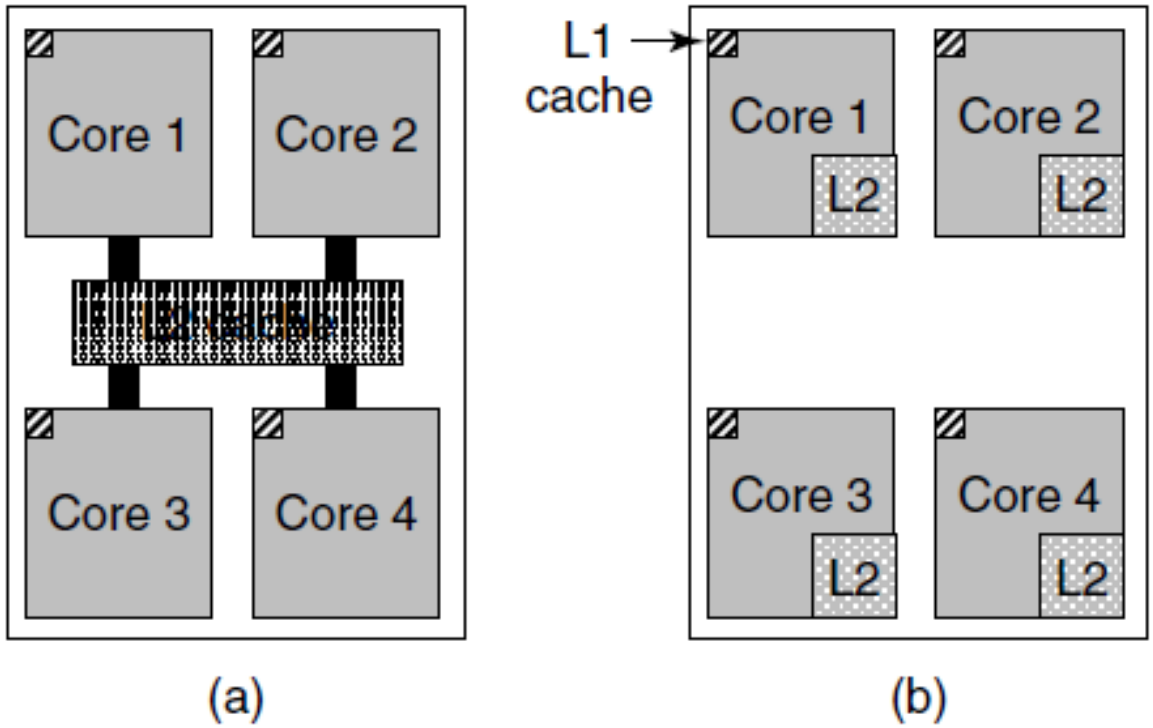


Figure 1-8. (a) A quad-core chip with a shared L2 cache. (b) A quad-core chip with separate L2 caches.

Memory (2)

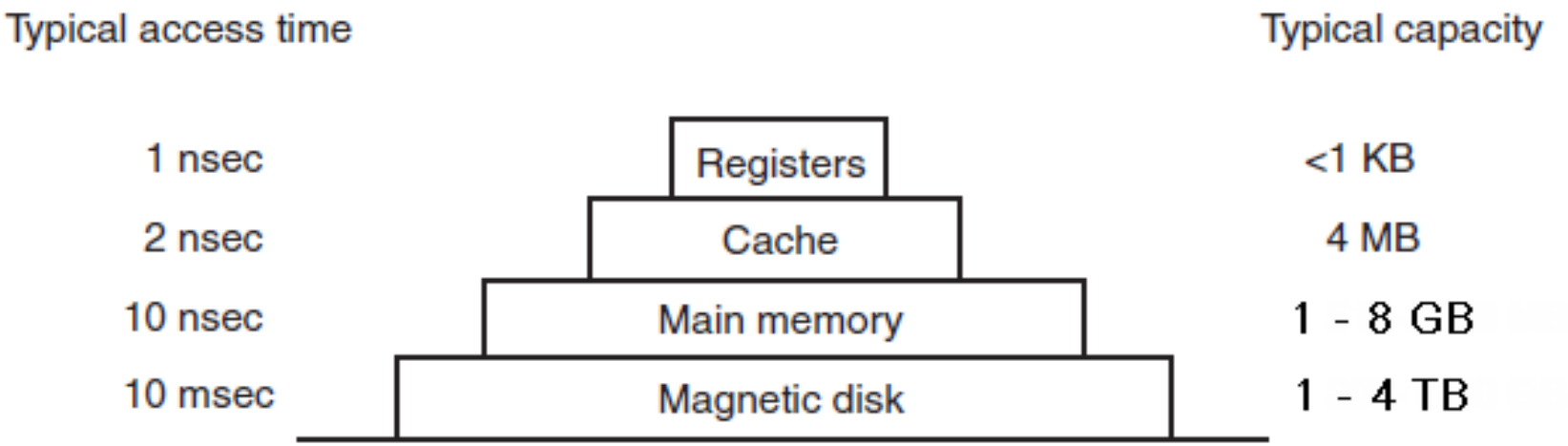
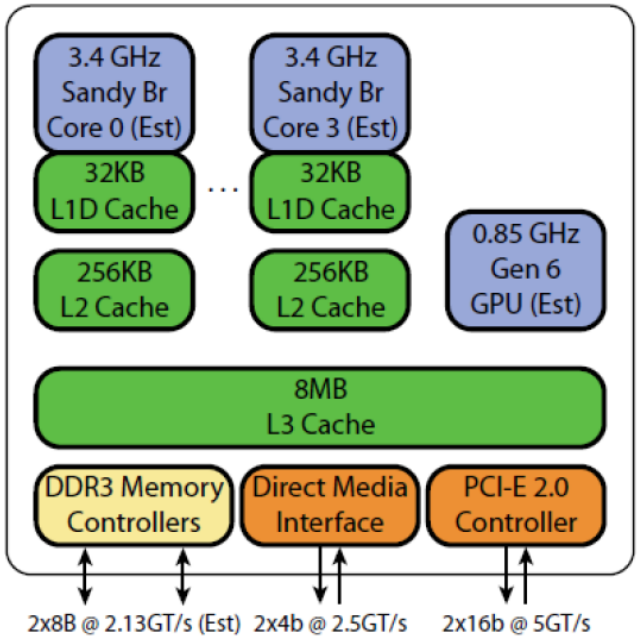


Figure 1-9. A typical memory hierarchy. The numbers are very rough approximations.

Memory (3)

- Intel Sandy Bridge Caches
  - L1 Data cache = 32 KB.
  - L1 Instruction cache = 32 KB
  - L2 Cache = 256 KB.
  - L3 Cache = 8 MB.



Memory (3)

- Intel Sandy Bridge Cache Latency
- L1 Data Cache Latency = 4 cycles for simple access via pointer
- L1 Data Cache Latency = 5 cycles for access with complex address calculation
- L2 Cache Latency = 12 cycles
- L3 Cache Latency = 27.85 cycles
- RAM Latency = 28 cycles + 56 ns

Memory (3)

- Main memory is divided up into cache lines
  - 64 bytes each
    - Cache line 0 - addresses 0 to 63
    - Cache line 1 - addresses 64 - 127
    - etc

Memory (4)

Caching system issues:

1. When to put a new item into the cache.

2. Which cache line to put the new item in.

3. Which item to remove from the cache when a slot is needed.

4. Where to put a newly evicted item in the larger memory.

Disks

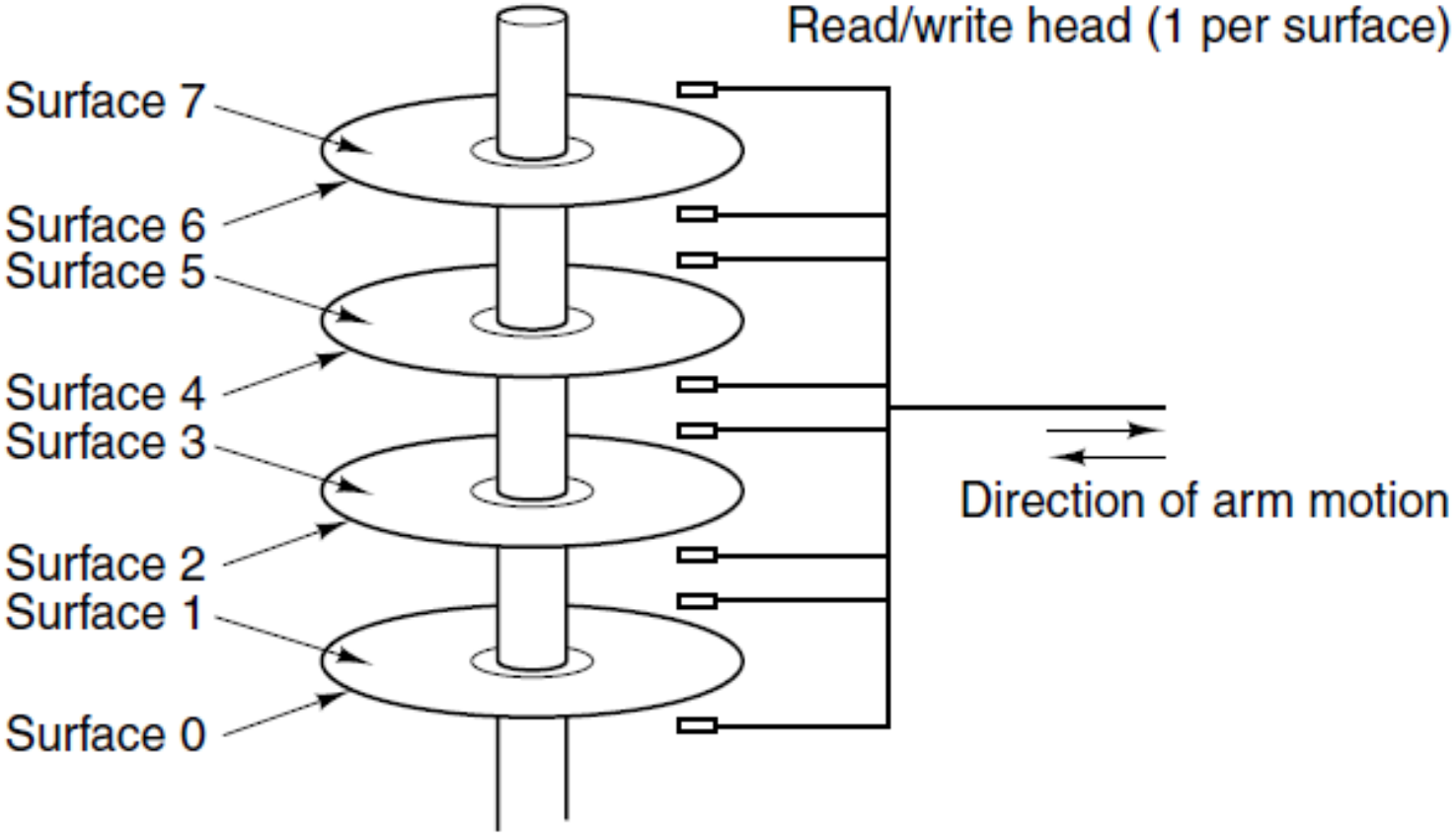
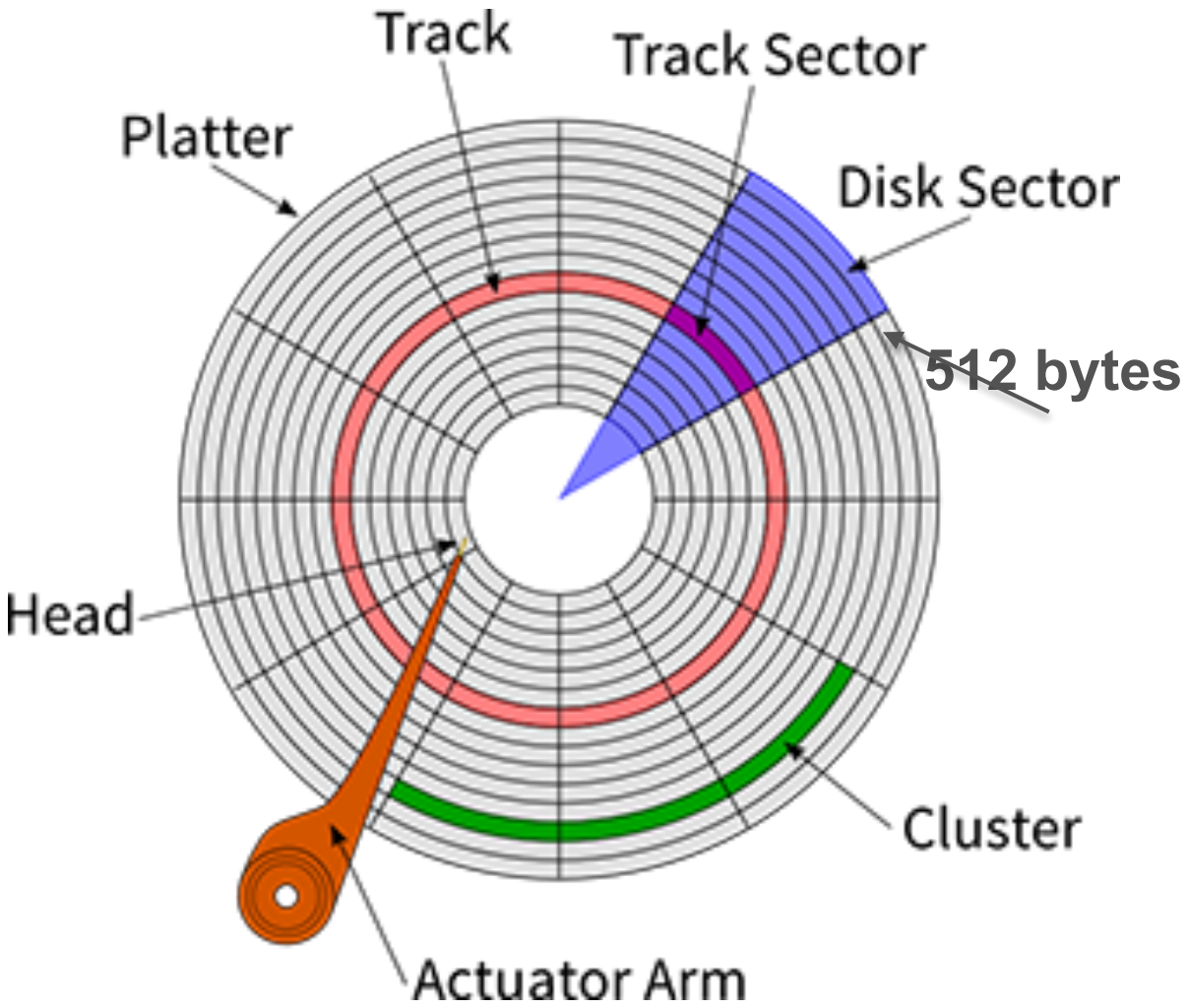


Figure 1-10. Structure of a disk drive.

Disks



I/O Devices

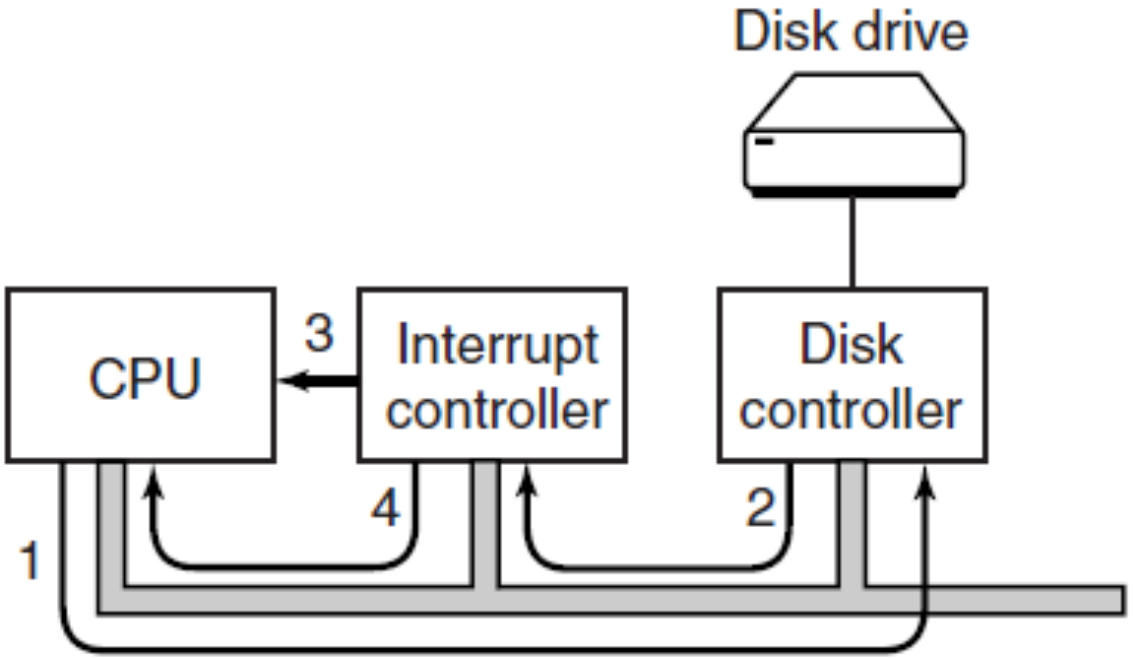


Figure 1-11. (a) The steps in starting an I/O device and getting an interrupt.

I/O Devices

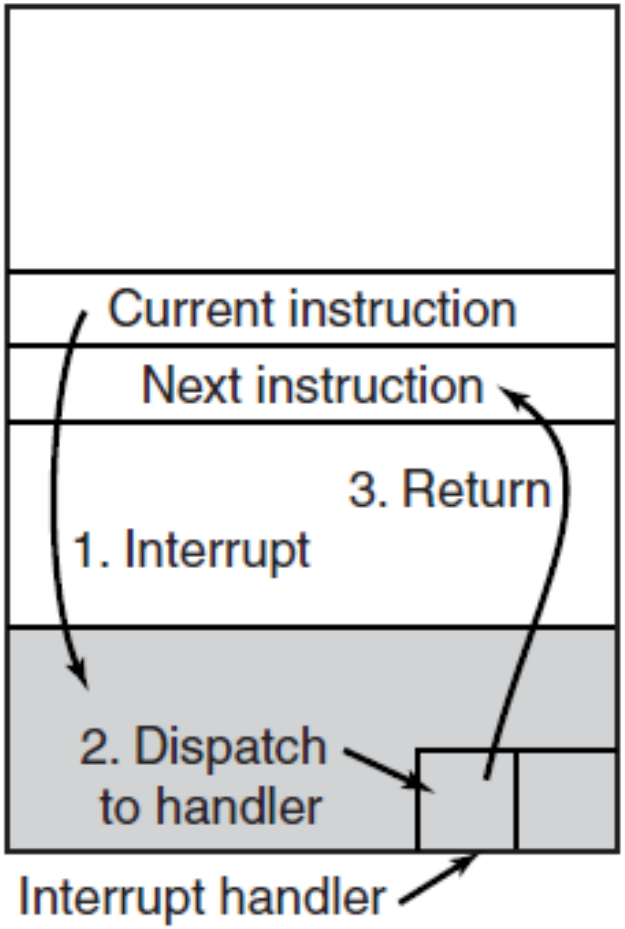


Figure 1-11. (b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program.

Buses

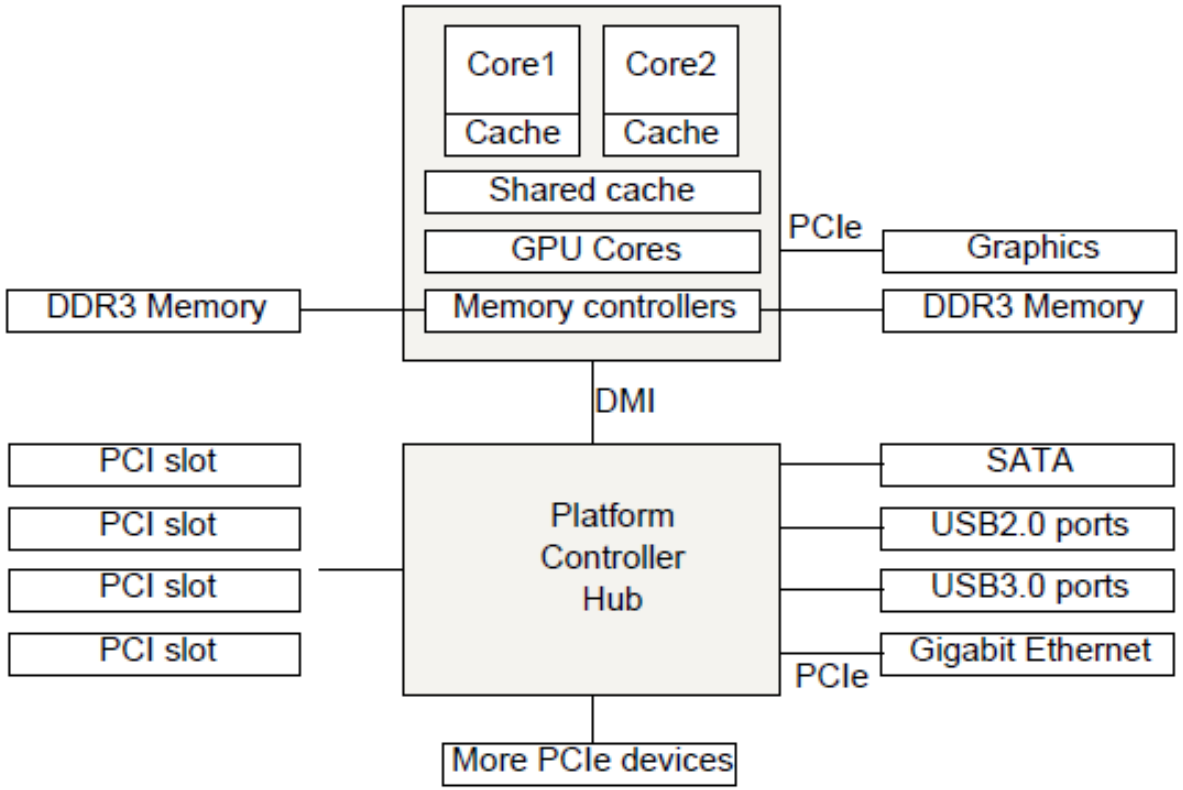


Figure 1-12. The structure of a large x86 system

The Operating System Zoo

- Mainframe Operating Systems
- Server Operating Systems
- Multiprocessor Operating Systems
- Personal Computer Operating Systems
- Handheld Computer Operating Systems
- Embedded Operating Systems
- Sensor Node Operating Systems
- Real-Time Operating Systems
- Smart Card Operating Systems

Programs v. Processes

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

© 2015 Trevor Bakker and The University of Texas at Arlington

Processes (2)

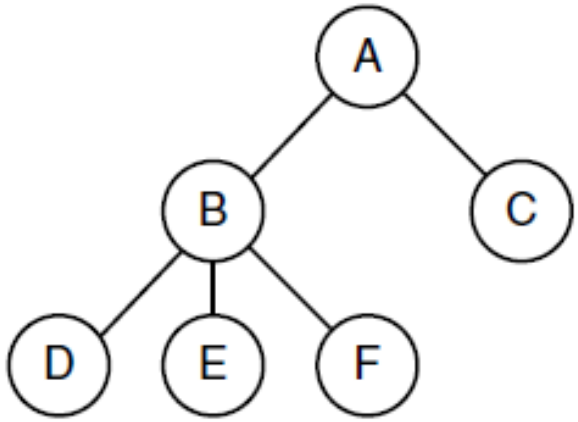
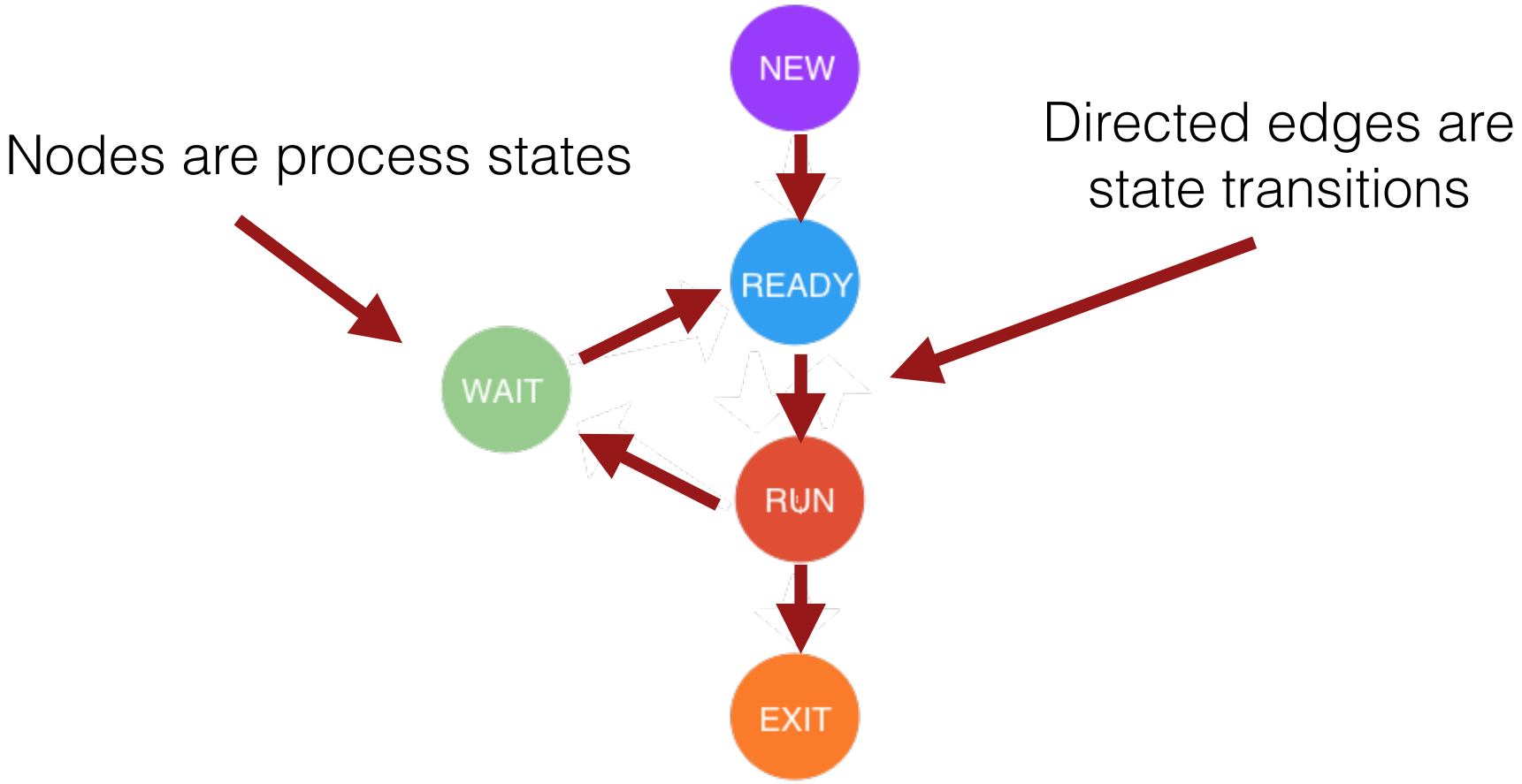


Figure 1-13. A process tree. Process A created two child processes, B and C. Process B created three child processes, D, E, and F.

Process State Diagram



Parent and children PIDs

- Why does fork return the child's PID to parent processes but it returns 0 to the children?
  - Provides a way to determine which process you are in.
  - Processes can only have one parent. To determine the parent PID you can call getppid().
  - Processes can have multiple children so there is no way for a function to give a parent its child PID

Process Creation

- After fork() both the parent and child continue executing with the instruction that follows the call to fork().
- The child process is a copy of the parent process. The child gets:
  - copy of the parent's data space
  - copy of the parent's heap
  - copy of the parent's stack
  - text segment if it's read-only

Copy-On-Write

- On Linux the parent process's pages are not copied for the child process.
- The pages are shared between the child and the parent process.
  - Pages are marked read-only
- When either process modifies a page, a page fault occurs and a separate copy of that particular page is made for that process which performed the modification.
- This process will then use the newly copied page rather than the shared one in all future references. The other process continues to use the original copy of the page

Inherited Properties

- user ID, group ID
- process group ID
- controlling terminal
- current working directory
- root directory
- file mode creation mask
- signal mask
- environment
- attached shared memory segments

Unique Properties

- the return value from fork()
- the process IDs
- the parent process IDs
- file locks
- pending alarms are cleared for the child
- the set of pending signals for the child is set to zero

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.

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

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

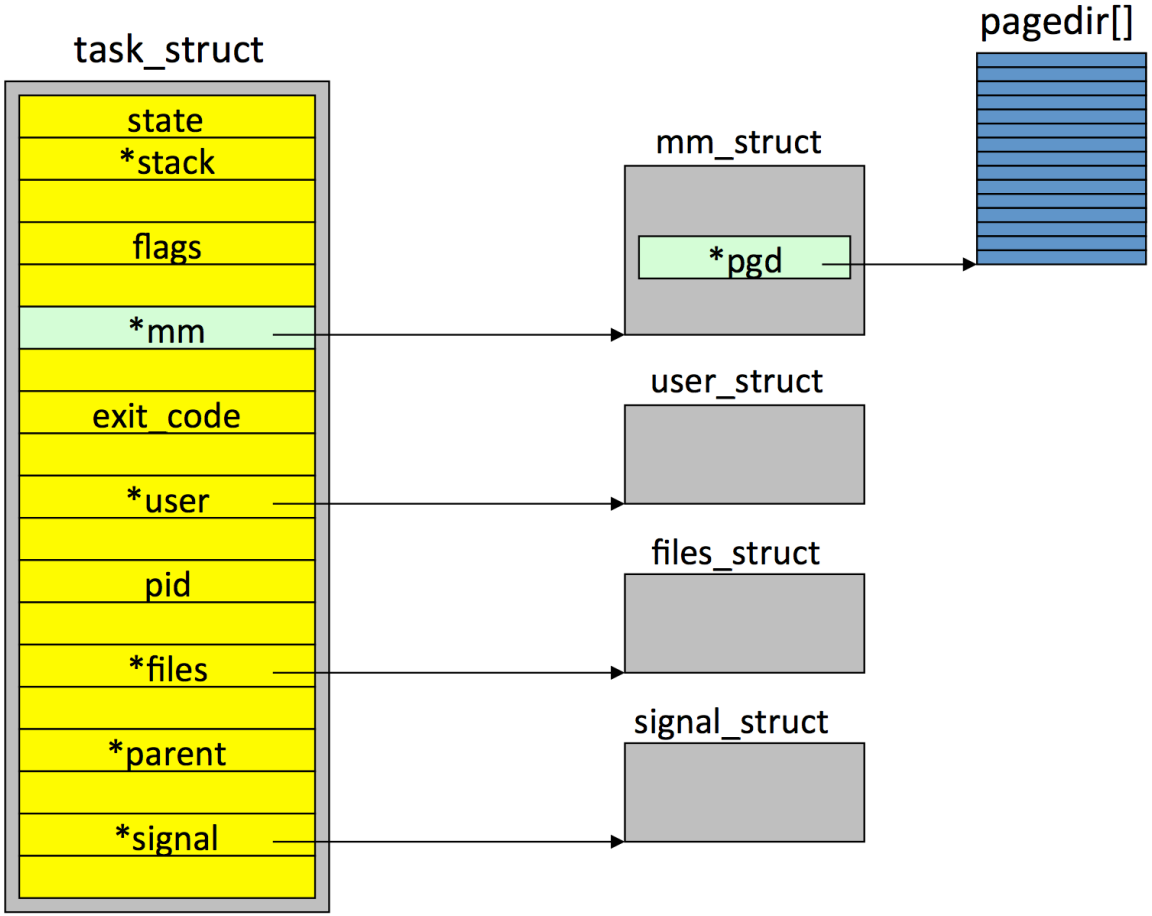
```
1166 struct task_struct {
1167         volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
1168         void *stack;
1169         atomic_t usage;
1170         unsigned int flags;     /* per process flags, defined below */
1171         unsigned int ptrace;
1172
1173         int lock_depth;         /* 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

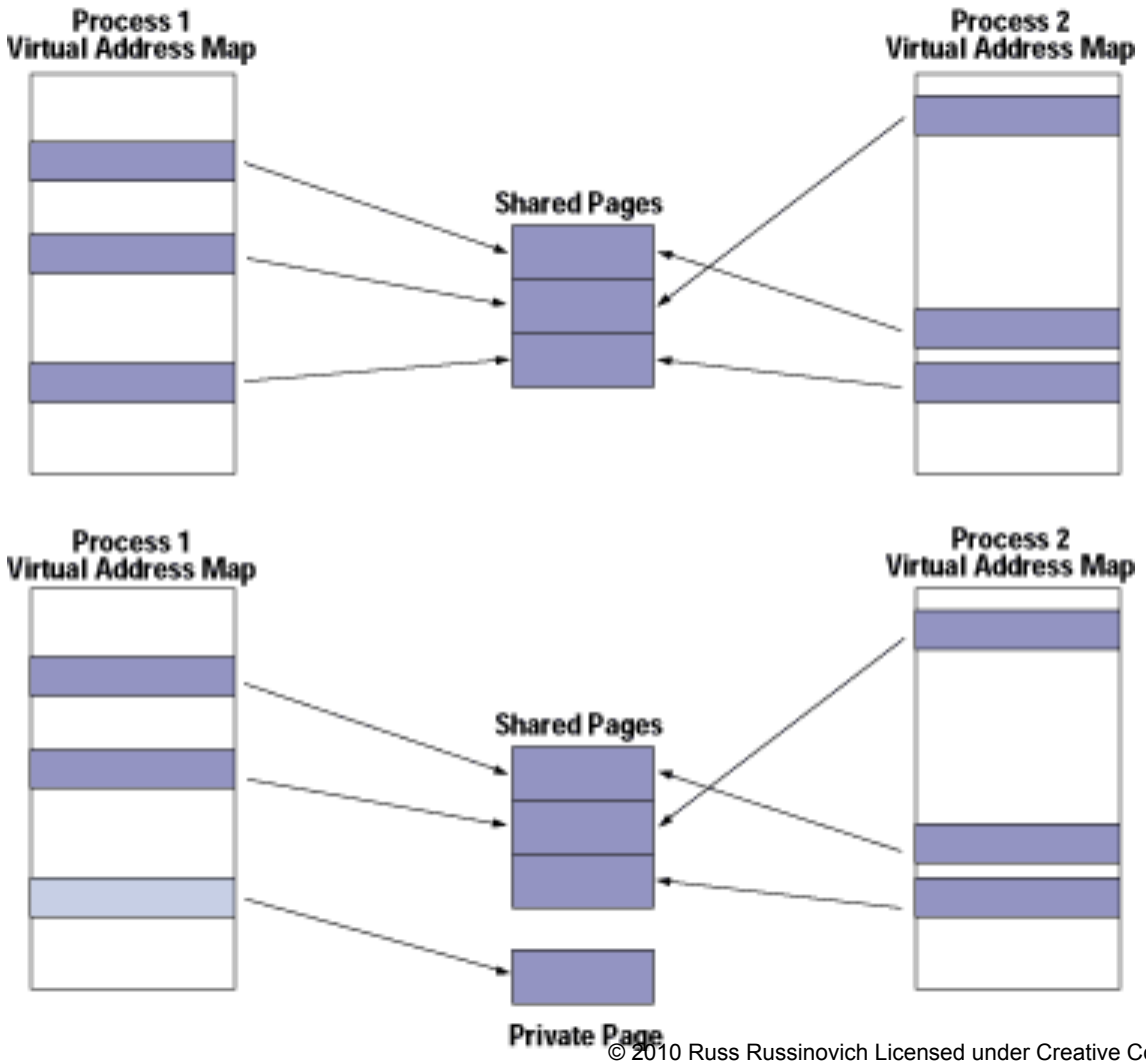
Each process descriptor contains many fields

and some are pointers to other kernel structures

which may themselves include fields that point to structures



Copy-On-Write



Process Creation

- Processes are created when an existing process calls the fork() function
- New process created by fork is called the *child process*
- fork() is a function that is called once but returns twice
  - Only difference in the return value. Returns 0 in the child process and the child's PID in the parent process

fork() man page

```
FORK(2)                    Linux Programmer's Manual                    FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a  child  process that differs from the parent process
       only in its PID and PPID, and in the fact  that  resource  utilizations
       are set to 0.  File locks and pending signals are not inherited.
```

fork() return values

RETURN VALUE

On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and errno will be set appropriately.

ERRORS

EAGAIN **fork**() cannot allocate sufficient memory to copy the parent's page tables and allocate a task structure for the child.

EAGAIN It was not possible to create a new process because the caller's **RLIMIT_NPROC** resource limit was encountered. To exceed this limit, the process must have either the **CAP_SYS_ADMIN** or the **CAP_SYS_RESOURCE** capability.

ENOMEM **fork**() failed to allocate the necessary kernel structures because memory is tight.

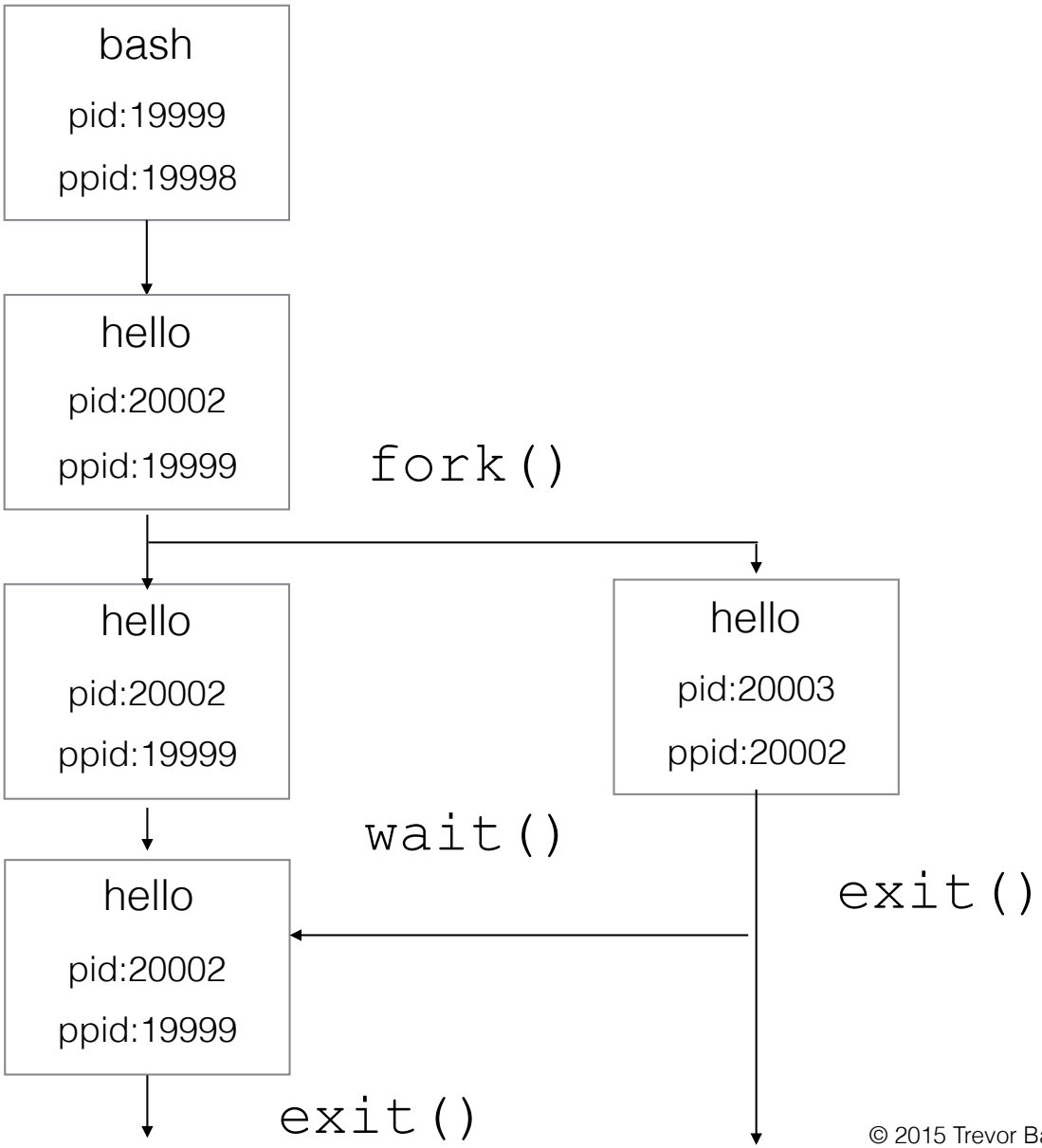
fork() code example

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  pid_t pid = fork();

  if (pid == -1) {
    // When fork() returns -1, an error happened.
    perror("fork failed");
    exit(EXIT_FAILURE);
  }
  else if (pid == 0) {
    // When fork() returns 0, we are in the child process.
    printf("Hello from the child process!\n");
    fflush(NULL);
    exit(EXIT_SUCCESS);
  }
  else {
    // When fork() returns a positive number, we are in the parent process
    // and the return value is the PID of the newly created child process.
    int status;
    (void)waitpid(pid, &status, 0);
    printf("Hello form the parent process!");
    fflush(NULL);
  }
  return EXIT_SUCCESS;
}
```

Hello World



fork() code example

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  pid_t pid = fork();

  if (pid == -1) {
    // When fork() returns -1, an error happened.
    perror("fork failed");
    exit(EXIT_FAILURE);
  }
  else if (pid == 0) {
    // When fork() returns 0, we are in the child process.
    printf("Hello from the child process!\n");
    fflush(NULL);
    exit(EXIT_SUCCESS);
  }
  else {
    // When fork() returns a positive number, we are in the parent process
    // and the return value is the PID of the newly created child process.
    int status;
    (void)waitpid(pid, &status, 0);
    printf("Hello form the parent process!");
    fflush(NULL);
  }
  return EXIT_SUCCESS;
}
```

I/O is buffered. Make sure to use `flush()`

fork() failures

- Too many processes already.
  - Usually a sign something else has gone wrong
  - cat /proc/sys/kernel/pid_max
- Too many processes for the current user
  - ulimit -a

Terminating a Process

- Three normal ways

1. Executing a return from the main function
2. Calling the exit function. C library function.
3. Calling the _exit function. System call.
    1. You should use _exit to kill the child program when the exec fails. Otherwise the child process may interfere with the parent process' external data by calling its signal handlers, or flushing buffers.
    2. You should also use _exit in any child process that does not do an exec, but those are rare.

Terminating a Process

- Two abnormal ways

1. Calling abort. Generates a SIGABORT signal.

2. The process received a signal

What if the parent terminates first?

- `exit` and `_exit` return the termination status of child processes to the parent. Who gets the status if the parent has already terminated?
- The `init` process becomes the parent of any process whose parent terminates.
  - The orphaned process is inherited by the `init` process
  - When a process is terminated the kernel iterates through all the active processes to see if the terminating process is the parent of any remaining processes. If so, it inherits that process

Zombie Process

- A process that has completed execution but still has an entry in the process table.
  - The entry is still needed to allow the parent process to read its child's exit status
- A child process always becomes a zombie before being removed from the resource table.
  - Normally, zombies are immediately waited on by their parent and then reaped by the system

wait() and waitpid()

- When a process terminates the parent is notified by the operating system sending a SIGCHLD signal.
  - Default handling is to ignore the signal
- Child termination is asynchronous
- POSIX provides two system calls wait and waitpid

wait()

```
WAIT(2)                    Linux Programmer's Manual                    WAIT(2)

NAME
       wait, waitpid - wait for process to change state

SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *status);
       pid_t waitpid(pid_t pid, int *status, int options);
       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- `wait` suspends the parents operation until the next child process terminates.
- If there is a child process already terminated and waiting for reaping `wait` will return immediately
- PID of the terminated child process is returned on success. -1 if failure.

wait() status parameter

WIFEXITED(status**)**
returns true if the child terminated normally, that is, by calling **exit**(3) or **_exit**(2), or by returning from main().

WEXITSTATUS(status**)**
returns the exit status of the child. This consists of the least significant 16–8 bits of the status argument that the child specified in a call to **exit**() or **_exit**() or as the argument for a return statement in main(). This macro should only be employed if **WIFEXITED** returned true.

WIFSIGNALED(status**)**
returns true if the child process was terminated by a signal.

WTERMSIG(status**)**
returns the number of the signal that caused the child process to terminate. This macro should only be employed if **WIFSIGNALED** returned true.

wait() status parameter

WCOREDUMP(status)
returns true if the child produced a core dump. This macro should only be employed if **WIFSIGNALED** returned true. This macro is not specified in POSIX.1-2001 and is not available on some Unix implementations (e.g., AIX, SunOS). Only use this enclosed in #ifdef WCOREDUMP ... #endif.

WIFSTOPPED(status)
returns true if the child process was stopped by delivery of a signal; this is only possible if the call was done using **WUNTRACED** or when the child is being traced (see **ptrace**(2)).

WSTOPSIG(status)
returns the number of the signal which caused the child to stop. This macro should only be employed if **WIFSTOPPED** returned true.

WIFCONTINUED(status)
(Since Linux 2.6.10) returns true if the child process was resumed by delivery of **SIGCONT**.

waitpid()

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- Allows the parent process to wait on a specific child process.

waitpid() code example

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

int main(void)
{
  pid_t child_pid = fork();
  int status;

  if (child_pid == 0)
  {
    // Sleep for a second
    sleep(1);

    // Intentionally SEGFAULT the child process
    int *p = NULL;
    *p = 1;

    exit(0);
  }

  // Wait for the child to exit
  waitpid( child_pid, &status, 0 );

  // See if the child was terminated by a signal
  if( WIFSIGNALED( status ) )
  {
    // Print the signal that the child terminated with
    printf("Child returned with status %d\n", WTERMSIG( status ) );
  }

  return 0;
}
```

Process Creation

`fork()` creates an exact copy of a process. How do we create a unique process?

By combining `fork` with an `exec` function

`exec` functions

- When exec is called the new program, specified by exec, completely replaces the running process.
  - text, data, heap and stack are all replaced
  - PID stays the same since it's not a new process

exec family

```
int execl(char const *path, char const *arg0, ...);
int execle(char const *path, char const *arg0, ..., char const *envp[]);
int execlp(char const *file, char const *arg0, ...);
int execv(char const *path, char const *argv[]);
int execve(char const *path, char const *argv[], char const *envp[]);
int execvp(char const *file, char const *argv[]);
```

Meaning of the letters after exec:

l – A list of command-line arguments are passedto the function.

e – An array of pointers to environment variables is passed to the new process image.

p – The PATH environment variable is used to find the file named in the path argument.

v – Command-line arguments are passed to the function as an array of pointers.

exec code example

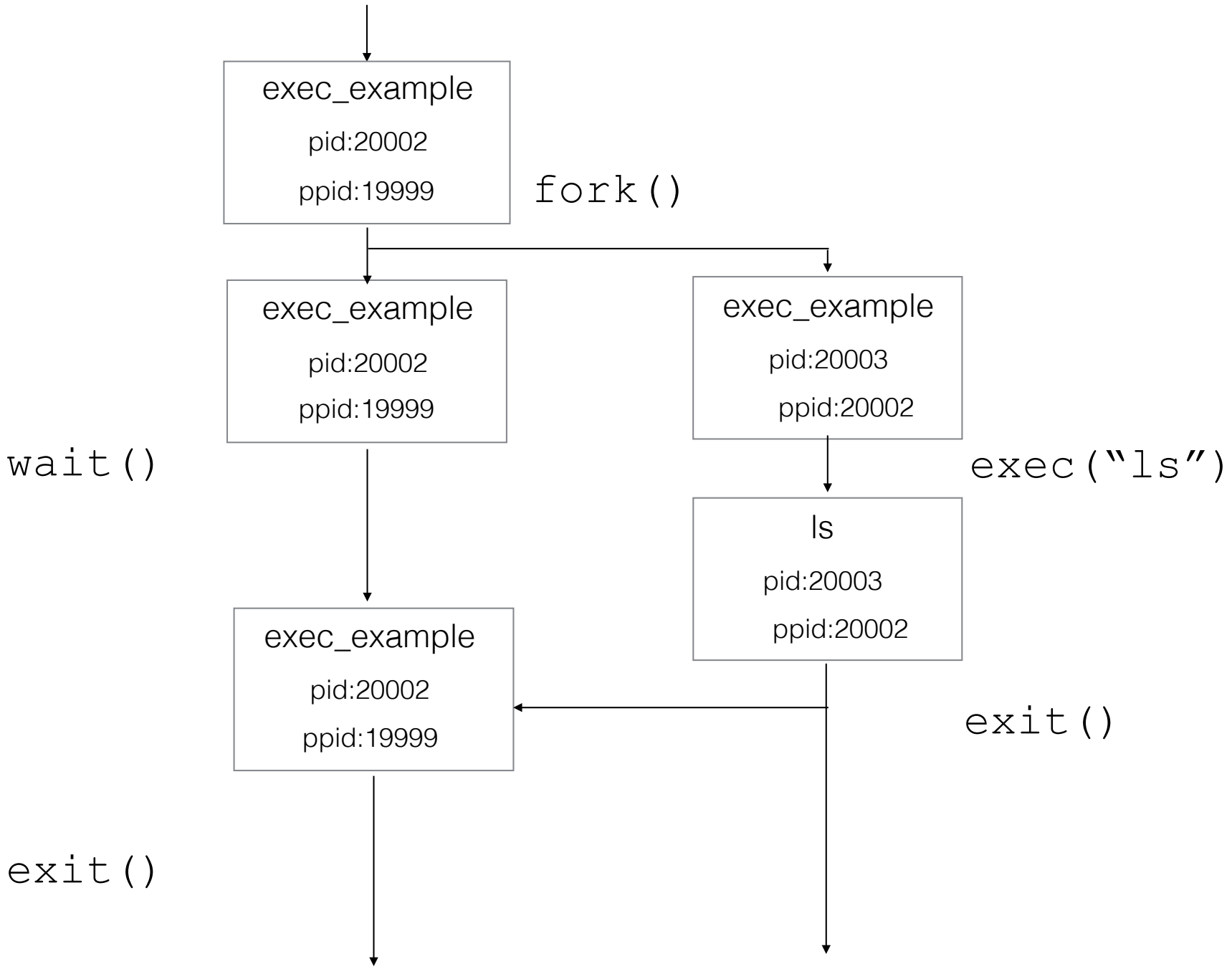
```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

int main(void)
{
  pid_t child_pid = fork();
  int status;

  if (child_pid == 0)
  {
    execl("/bin/ls", "ls", NULL );
    exit(0);
  }

  // Wait for the child to exit
  waitpid( child_pid, &status, 0 );

  return 0;
}
```



Kernel Mode v. User Mode

- A process is executing either in user mode, or in kernel mode. 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 mode, a process has normal privileges and can and can't do certain things. When executing in kernel mode, a process has every privilege, and can do anything.

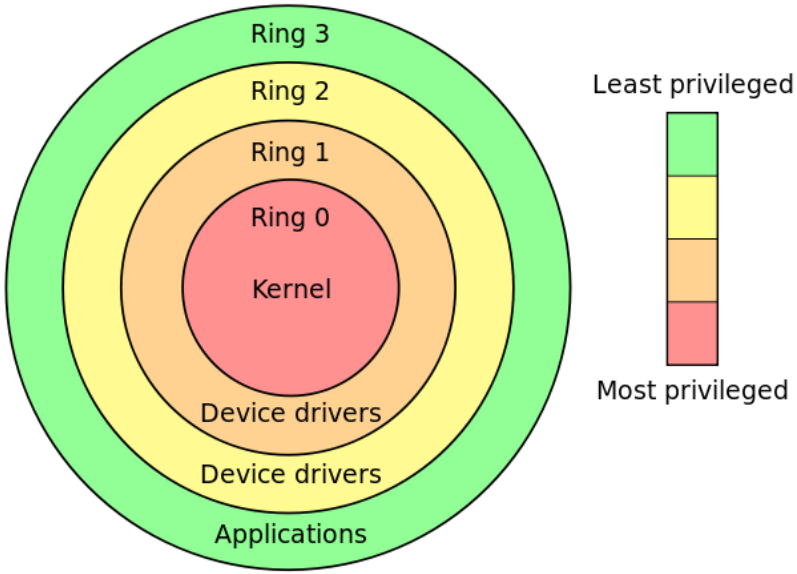
- Processes switch between user space and kernel space using system calls.

Kernel Mode v. User Mode

- 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

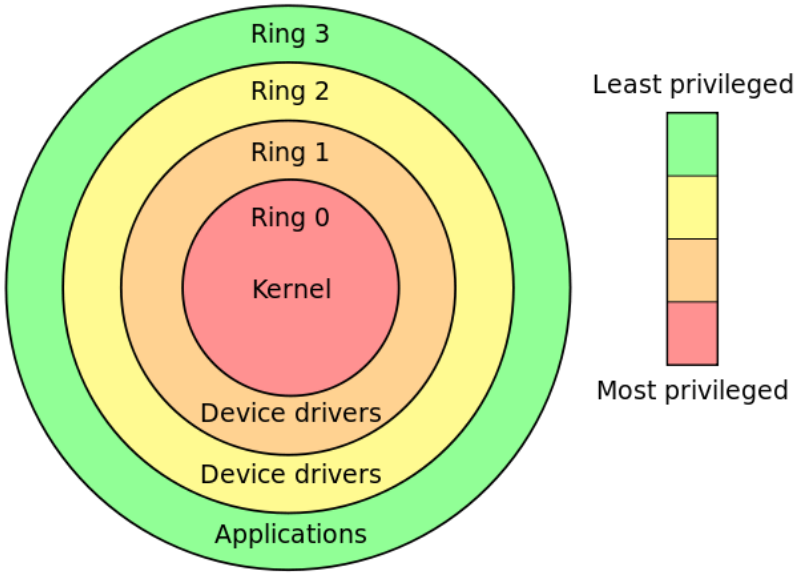
- Four privilege levels or rings, numbered from 0 to 3, with ring 0 being the most privileged and 3 being the least.
- Rings 1 and 2 weren't used in practice.
  - » VM's changed that



© 2015 Trevor Bakker and The University of Texas at Arlington

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.

© 2015 Trevor Bakker and The University of Texas at Arlington

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

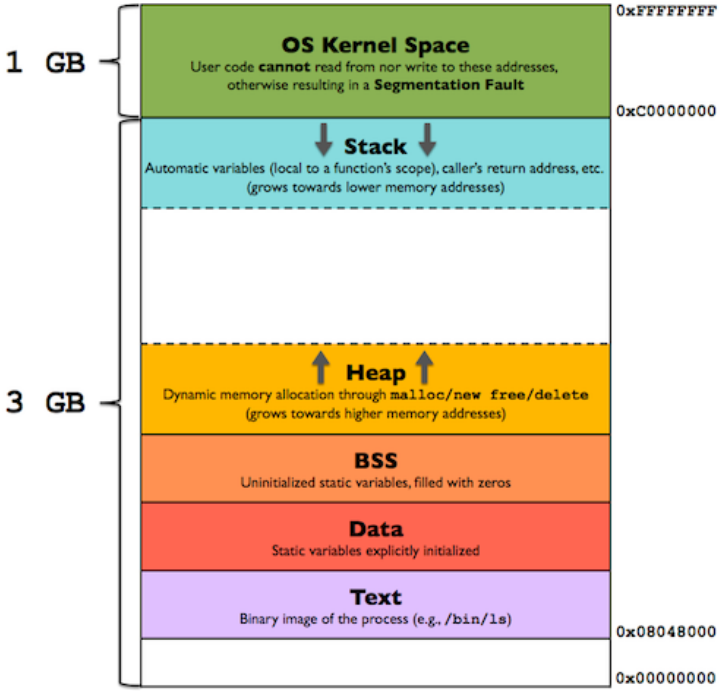
Instruction	Description
LGDT	Loads an address of a GDT into GDTR
LLDT	Loads an address of a LDT into LDTR
LTR	Loads a Task Register into TR
MOV *Control Register*	Copy data and store in Control Registers
LMSW	Load a new Machine Status WORD
CLTS	Clear Task Switch Flag in Control Register CR0
MOV *Debug Register*	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

© 2015 Trevor Bakker and The University of Texas at Arlington

Address Spaces

- An address space is the set of addresses in RAM that a process can use.
- Multiple programs in memory need OS to partition the available memory
  - Keep programs from interfering with each other and OS.

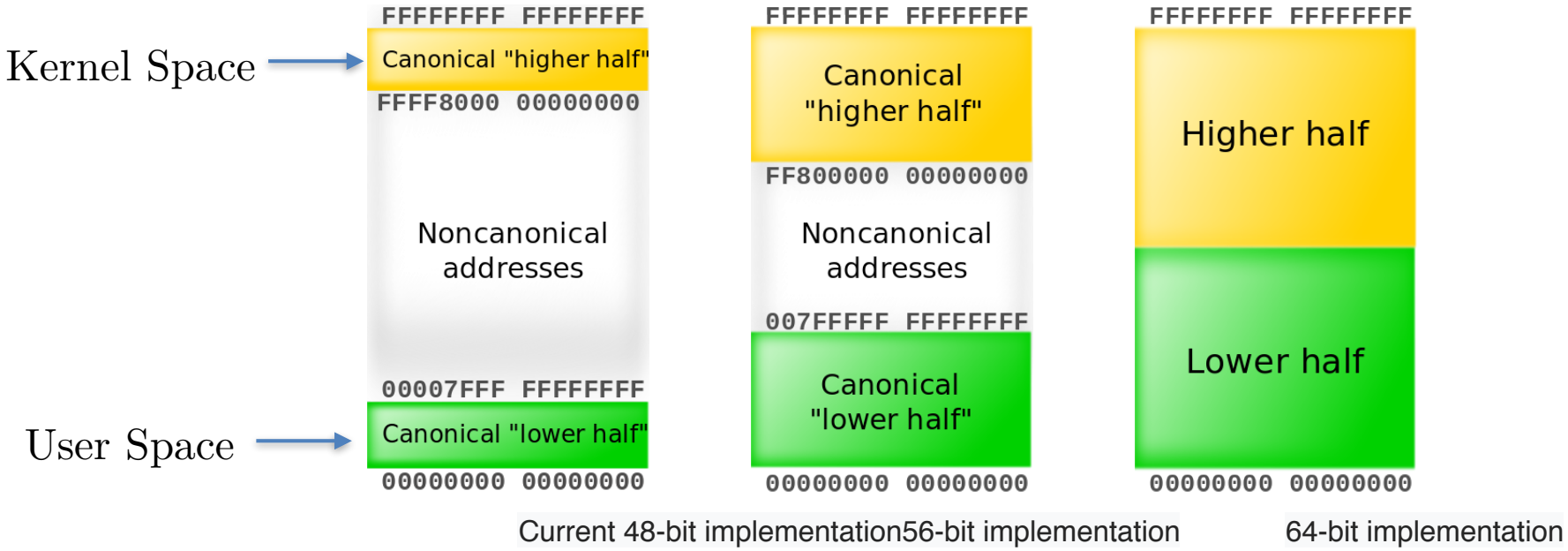
Address Spaces



32-bit address = 2^{32} = 4GB of address space

- Kernel gets upper 1 GB

Address Spaces



Only 48-bits implemented in current 64-bit architecture

Address Spaces

- What happens if the process has more address space than free main memory ?
  - Virtual memory
  - OS abstracts address space
    - Decouples programs address space from physical memory

Files (1)

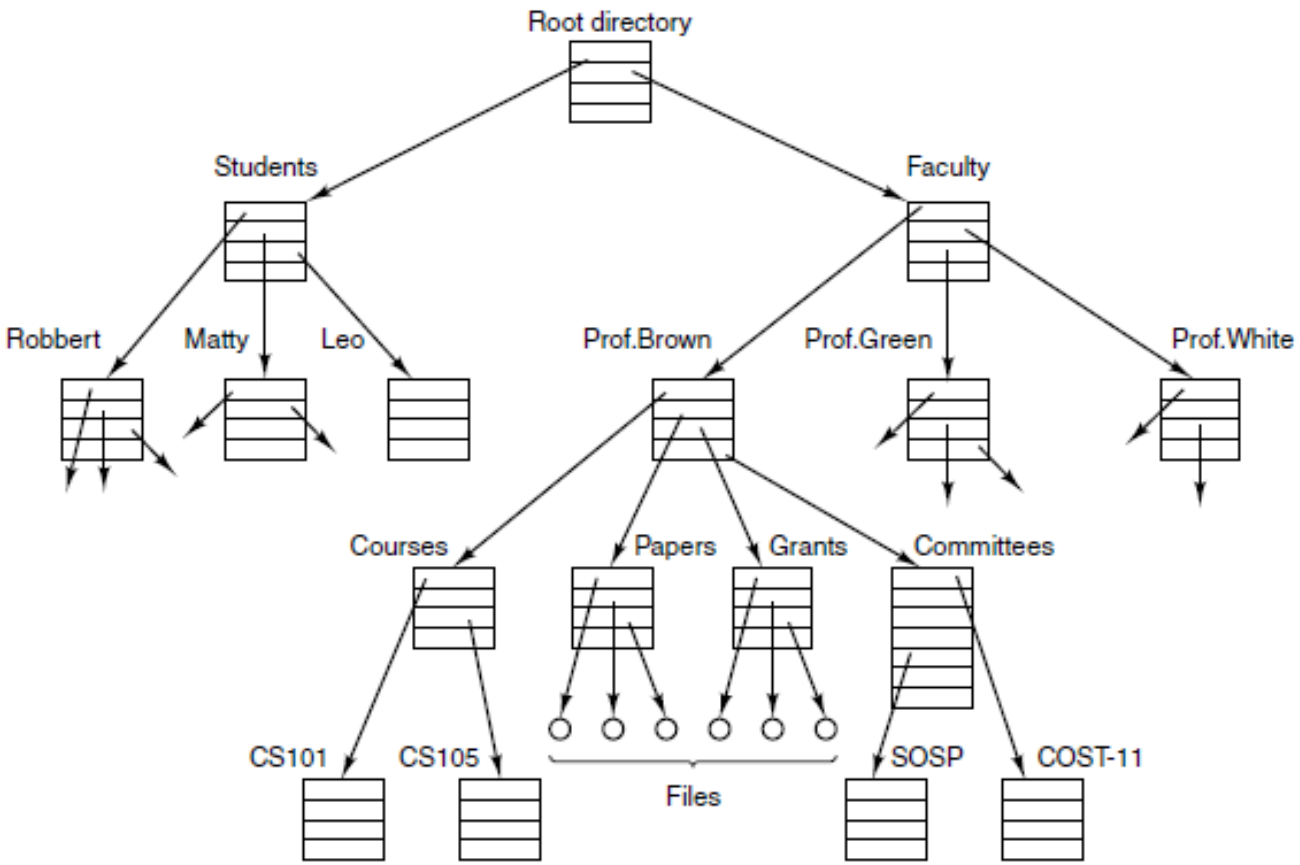
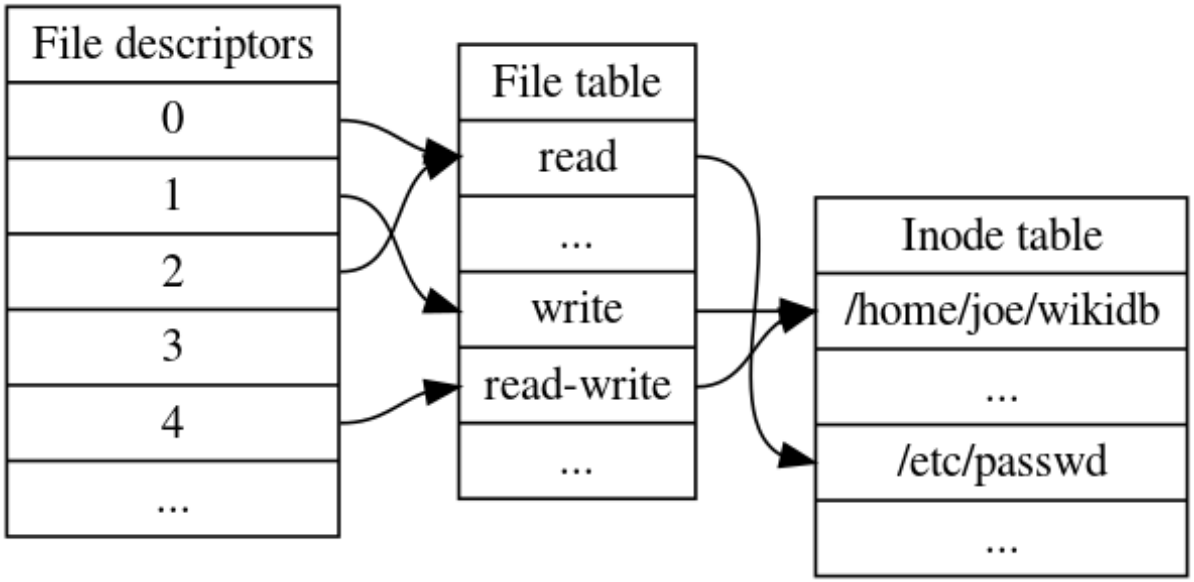


Figure 1-14. A file system for a university department.

File Terms

- Current Working Directory
- Special Files
  - Block special file
  - Character special file
- Pipe
  - pseudo file used to connect two processes

File Terms



File descriptor: an index into a per-process file descriptor table maintained by the kernel, that in turn indexes into a system-wide table of files opened by all processes, called the file table.

Files (2)

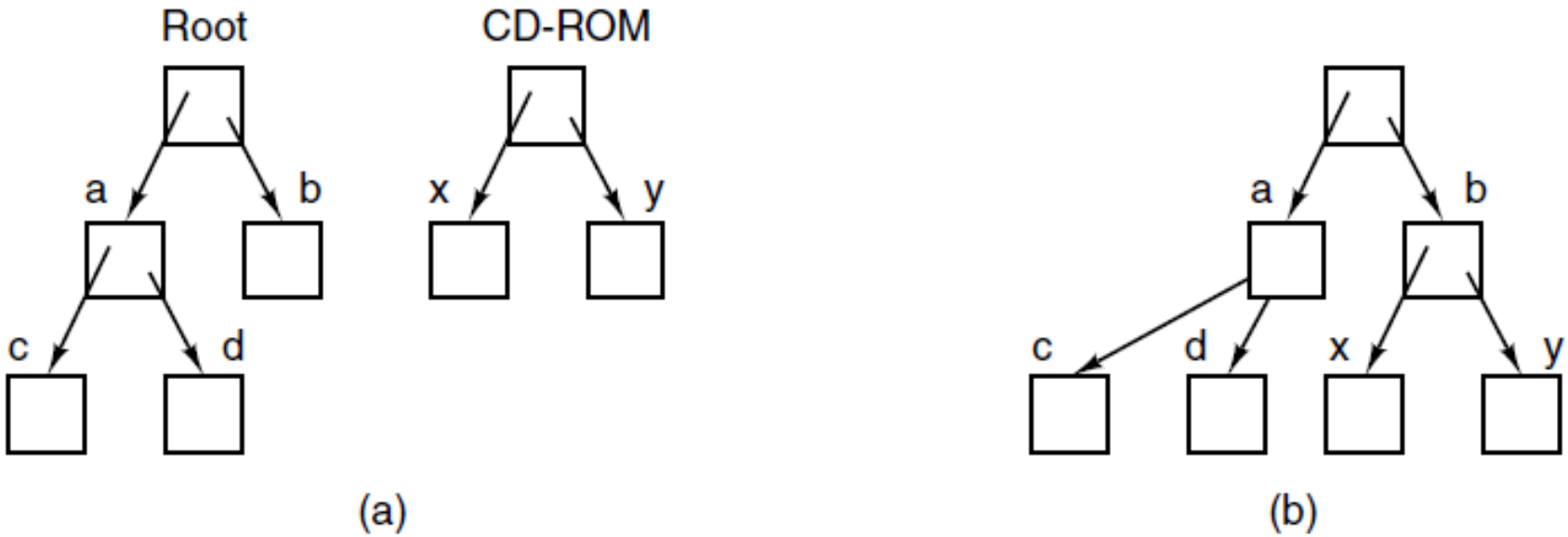


Figure 1-15. (a) Before mounting, the files on the CD-ROM are not accessible. (b) After mounting, they are part of the file hierarchy.

Files (3)

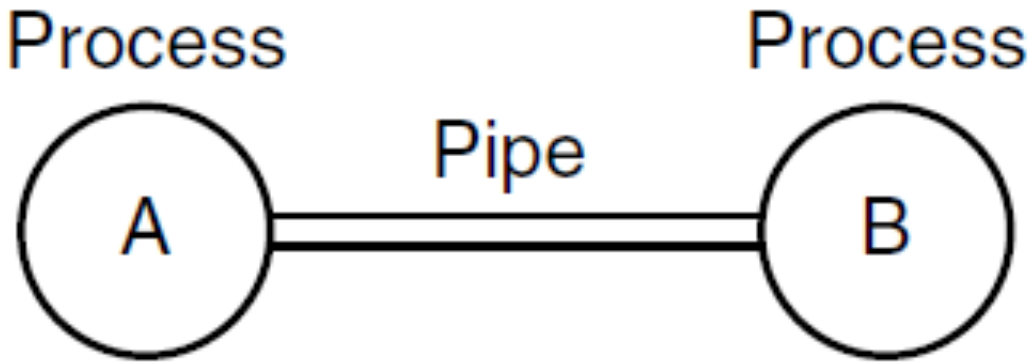


Figure 1-16. Two processes connected by a pipe.

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

- 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.
  - Only system calls enter the kernel. Procedure calls do not.

System Calls (1)

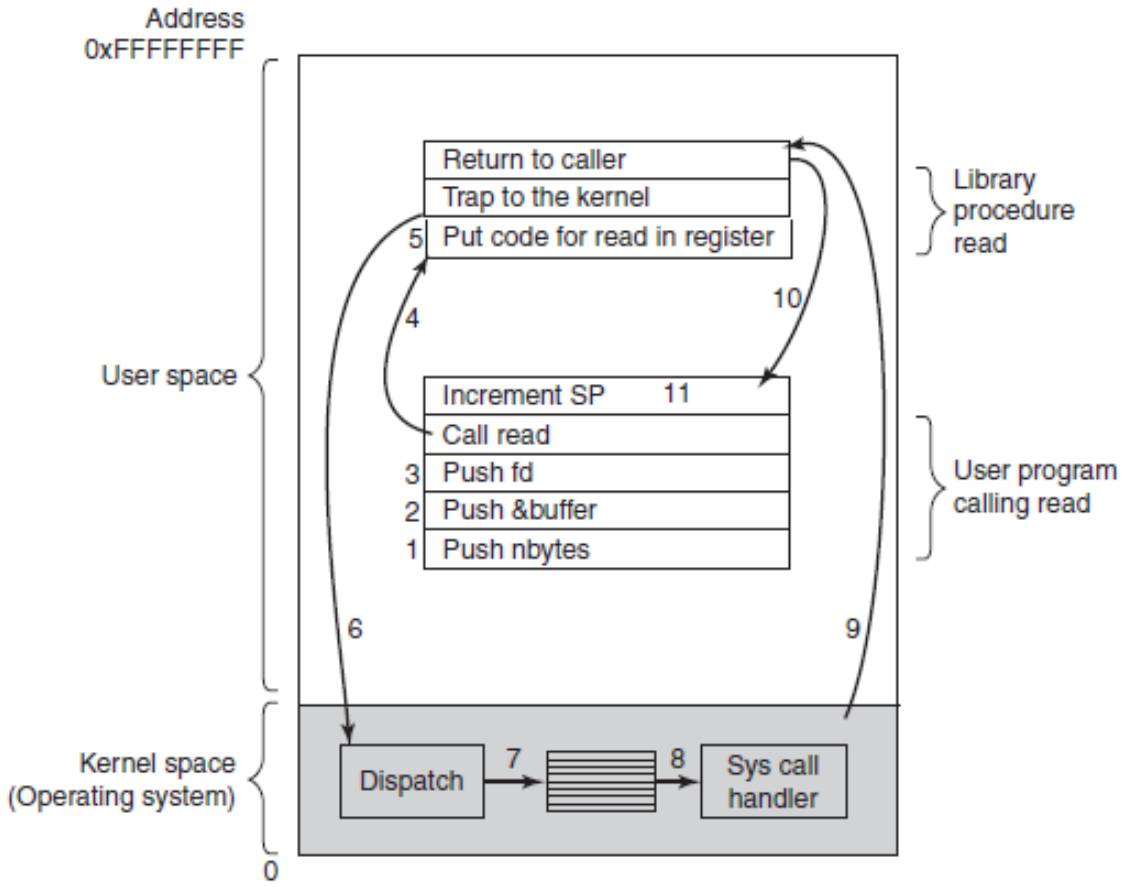


Figure 1-17. The 11 steps in making the system call `read(fd, buffer, nbytes)`.

System Calls (1)

1-3. Push parameters on the stack

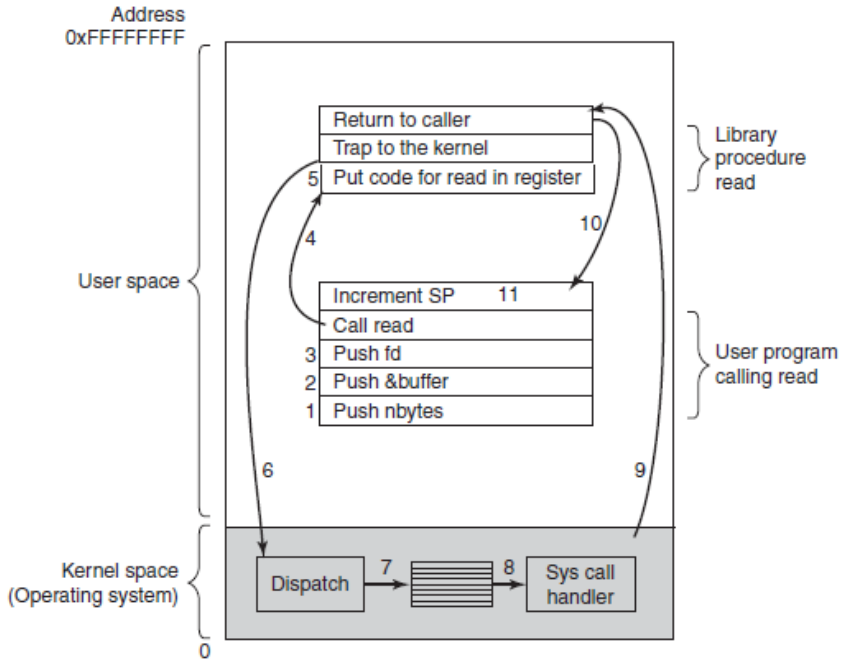
4. Call system function.

    -Same as a procedure call

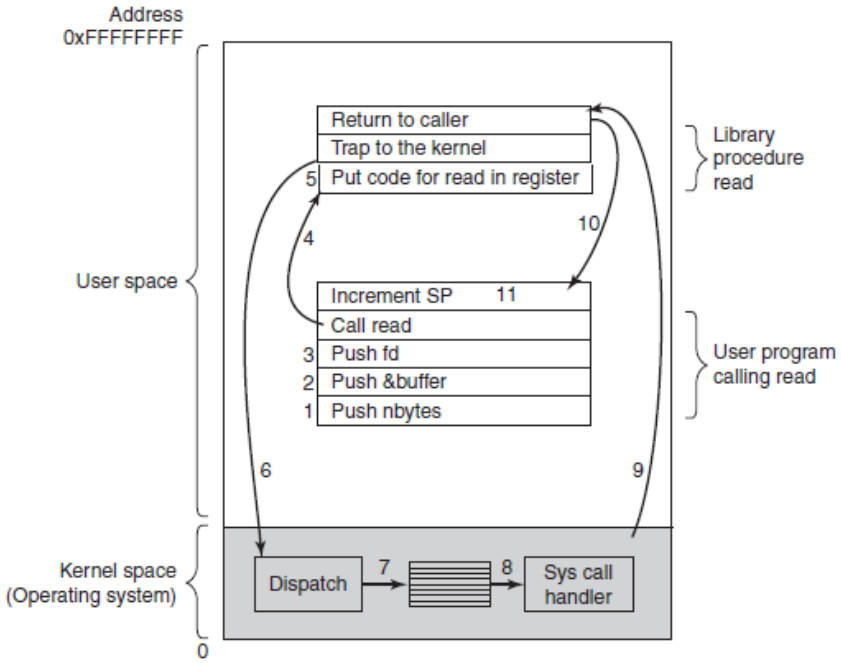
5. Place system call code in register.

6. TRAP instruction to switch to kernel mode.

7. Dispatch to signal call handler.



System Calls (1)



8. Signal call handler runs
9. Return from TRAP to user space of system call
10. Return to user program
11. Increment stack pointer to clear stack, as with all procedure calls returned.

Linux/arch/x86/entry/syscall_64.c

```
1 /* System call table for x86-64. */
2
3 #include <linux/linkage.h>
4 #include <linux/sys.h>
5 #include <linux/cache.h>
6 #include <asm/asm-offsets.h>
7 #include <asm/syscall.h>
8
9 #define __SYSCALL_64_QUAL_(sym) sym
10 #define __SYSCALL_64_QUAL_ptregs(sym) ptregs_##sym
11
12 #define __SYSCALL_64(nr, sym, qual) extern asmlinkage long __SYSCALL_64_QUAL_##qual(sym)
(unsigned long, unsigned long, unsigned long, unsigned long, unsigned long, unsigned long);
13 #include <asm/syscalls_64.h>
14 #undef __SYSCALL_64
15
16 #define __SYSCALL_64(nr, sym, qual) [nr] = __SYSCALL_64_QUAL_##qual(sym),
17
18 extern long sys_ni_syscall(unsigned long, unsigned long, unsigned long, unsigned long,
unsigned long, unsigned long);
19
20 asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
21         /*
22          * Smells like a compiler bug -- it doesn't work
23          * when the & below is removed.
24          */
25         [0 ... __NR_syscall_max] = &sys_ni_syscall,
26 #include <asm/syscalls_64.h>
27 };
```

arch/x86/syscalls/ syscall_64.tbl

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#

0    common    read            sys_read
1    common    write           sys_write
2    common    open            sys_open
3    common    close           sys_close
4    common    stat            sys_newstat
5    common    fstat           sys_newfstat
6    common    lstat           sys_newlstat
7    common    poll            sys_poll
8    common    lseek           sys_lseek
9    common    mmap            sys_mmap
10   common    mprotect        sys_mprotect
11   common    munmap               sys_munmap
12   common    brk             sys_brk
13   64   rt_sigaction         sys_rt_sigaction
14   common    rt_sigprocmask       sys_rt_sigprocmask
15   64   rt_sigreturn         stub_rt_sigreturn
16   64   ioctl          sys_ioctl
```

System Calls (2)

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Figure 1-18. Some of the major POSIX system calls. The return code ***s*** is –1 if an error has occurred. The return codes are as follows: ***pid*** is a process id, ***fd*** is a file descriptor, ***n*** is a byte count, ***position*** is an offset within the file, and ***seconds*** is the elapsed time.

System Calls (3)

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Figure 1-18. Some of the major POSIX system calls. The return code ***s*** is –1 if an error has occurred. The return codes are as follows: ***pid*** is a process id, ***fd*** is a file descriptor, ***n*** is a byte count, ***position*** is an offset within the file, and ***seconds*** is the elapsed time.

System Calls (4)

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Figure 1-18. Some of the major POSIX system calls. The return code ***s*** is –1 if an error has occurred. The return codes are as follows: ***pid*** is a process id, ***fd*** is a file descriptor, ***n*** is a byte count, ***position*** is an offset within the file, and ***seconds*** is the elapsed time.

System Calls (5)

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figure 1-18. Some of the major POSIX system calls. The return code ***s*** is –1 if an error has occurred. The return codes are as follows: ***pid*** is a process id, ***fd*** is a file descriptor, ***n*** is a byte count, ***position*** is an offset within the file, and ***seconds*** is the elapsed time.

System Calls for Process Management

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
        type_prompt( );                         /* display prompt on the screen */
        read_command(command, parameters);      /* read input from terminal */

        if (fork( ) != 0) {                     /* fork off child process */
            /* Parent code. */
            waitpid(–1, &status, 0);            /* wait for child to exit */
        } else {
            /* Child code. */
            execve(command, parameters, 0);     /* execute command */
        }
}
```

Figure 1-19. A stripped-down shell. Throughout this book, ***TRUE*** is assumed to be defined as 1.

Pseudo-code for Assignment 2

Monolithic Systems (1)

Basic structure of OS

1. A main program that invokes the requested service procedure.

2. A set of service procedures that carry out the system calls.

3. A set of utility procedures that help the service procedures.

Monolithic Systems (2)

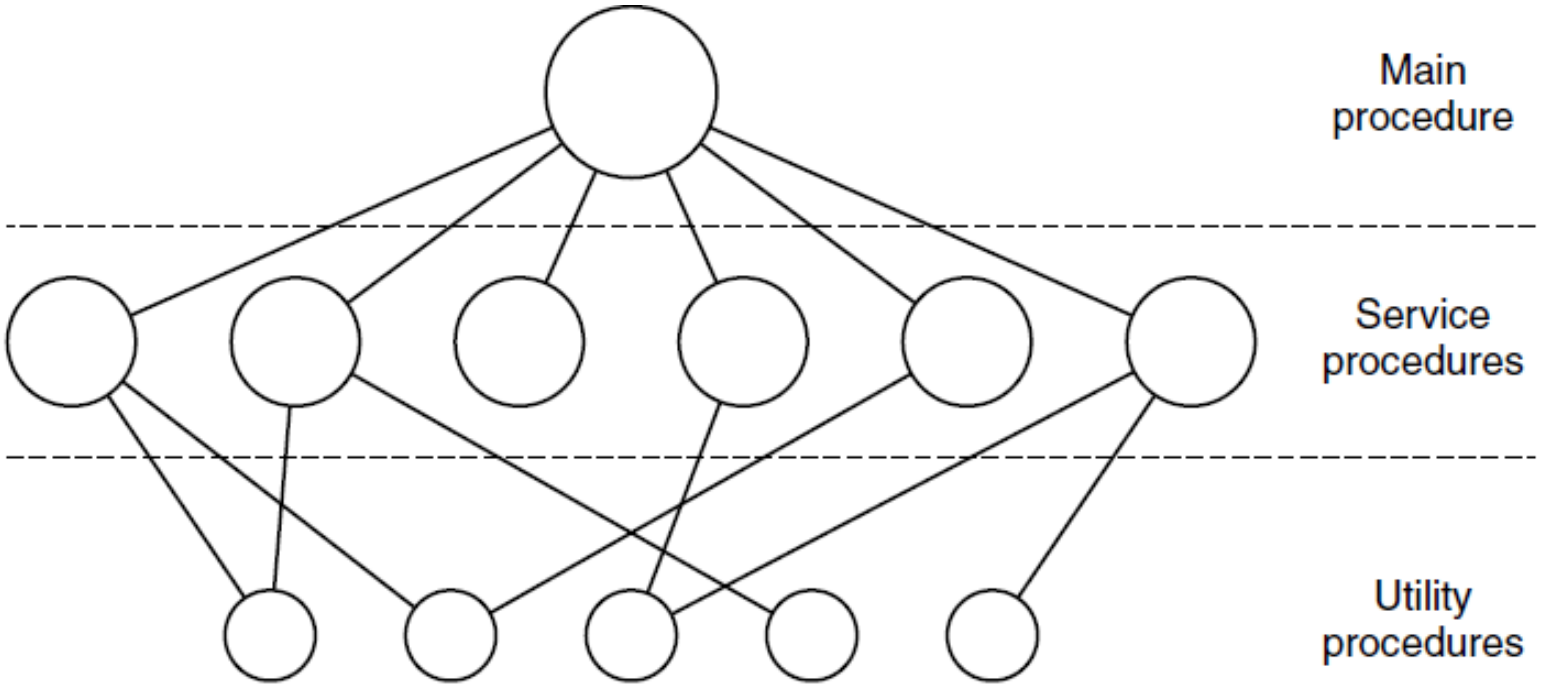
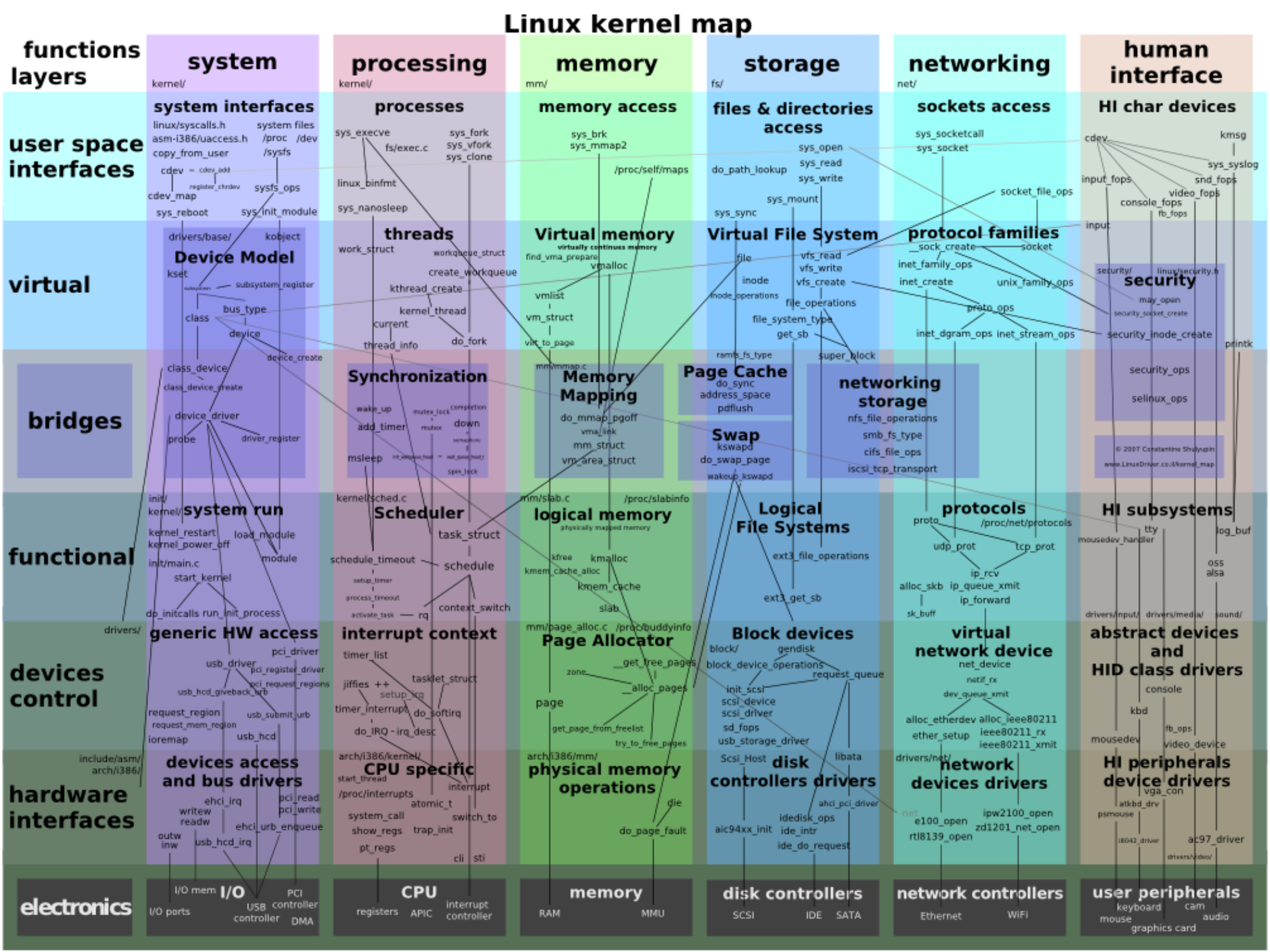


Fig
for

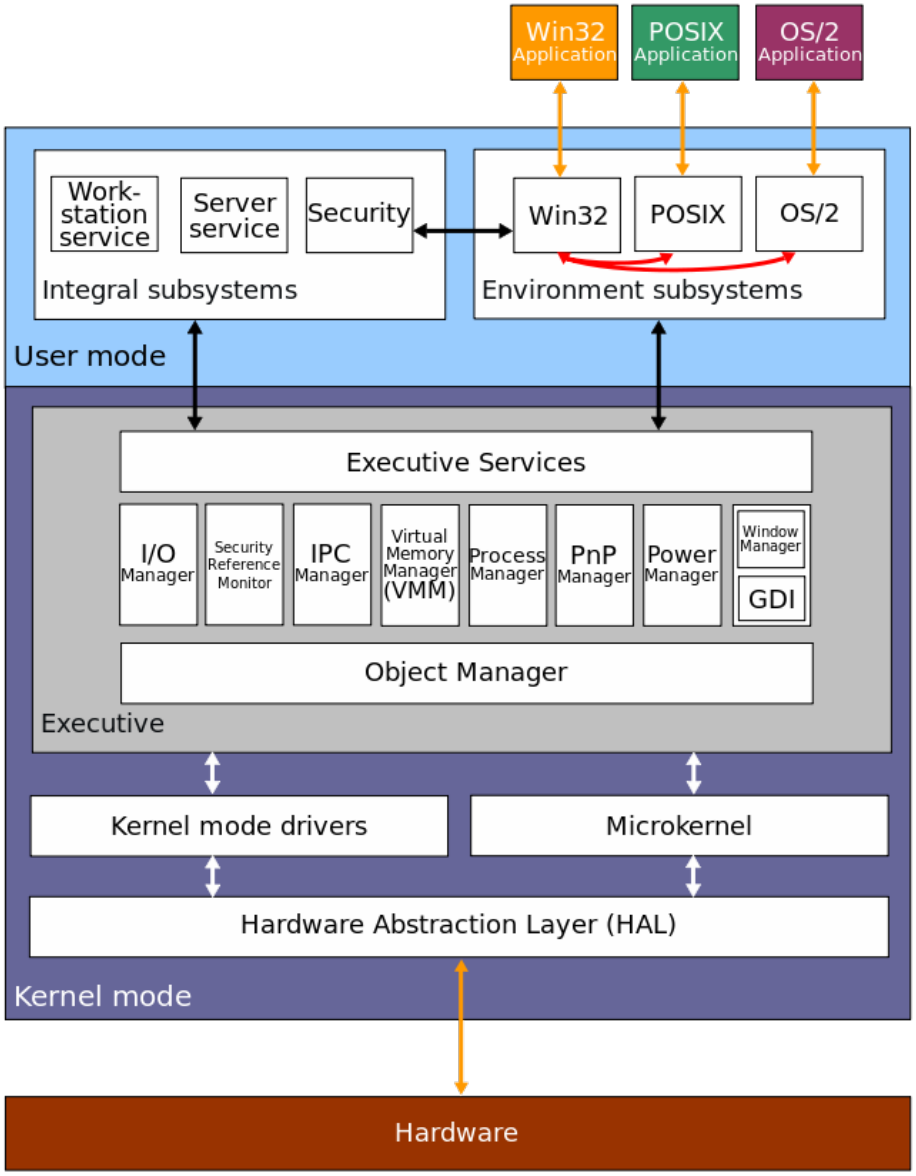

Linux kernel map
functions
layers
system
kernel/
processing
kernel/
memory
mm/
storage
fs/
networking
net/
human interface
user space interfaces
system interfaces
linux/syscalls.h
asm-i386/uaccess.h
copy_from_user
system files
/proc
/dev
/sysfs
cdev
cdev_add
register_chrdev
cdev_map
sysfs_ops
sys_reboot
sys_init_module
processes
sys_execve
fs/exec.c
sys_fork
sys_vfork
sys_clone
linux_binfmt
sys_nanosleep
memory access
sys_brk
sys_mmap2
/proc/self/maps
files & directories access
sys_open
sys_read
sys_write
do_path_lookup
sys_mount
sys_sync
sockets access
sys_socketcall
sys_socket
socket_file_ops
HI char devices
cdev
kmsg
sys_syslog
input_fops
snd_fops
video_fops
console_fops
fb_fops
input
virtual
drivers/base/
kobject
Device Model
kset
subsystem_register
bus_type
class
device
device_create
threads
work_struct
workqueue_struct
create_workqueue
kthread_create
kernel_thread
current
thread_info
do_fork
Virtual memory
virtually continuous memory
find_vma_prepare
vmalloc
vmlist
vm_struct
virt_to_page
mm/mmap.c
Virtual File System
file
inode
inode_operations
vfs_read
vfs_write
vfs_create
file_operations
file_system_type
get_sb
ramfs_fs_type
super_block
protocol families
_sock_create
socket
inet_family_ops
inet_create
unix_family_ops
proto_ops
inet_dgram_ops
inet_stream_ops
security/
linux/security.h
security
may_open
security_socket_create
security_inode_create
security_ops
selinux_ops
printk
bridges
class_device
class_device_create
device_driver
probe
driver_register
Synchronization
wake_up
mutex_lock
completion
add_timer
mutex
down
msleep
spin_lock
Memory Mapping
do_mmap_pgoff
vma_link
mm_struct
vm_area_struct
Page Cache
do_sync
address_space
pdflush
Swap
kswapd
do_swap_page
wakeup_kswapd
networking storage
nfs_file_operations
smb_fs_type
cifs_file_operations
iscsi_tcp_transport
© 2007 Constantine Shulyupin
www.LinuxDriver.co.il/kernel_map
functional
init/
kernel/
system run
kernel_restart
kernel_power_off
load_module
module
init/main.c
start_kernel
do_initcalls
run_init_process
kernel/sched.c
Scheduler
task_struct
schedule_timeout
schedule
setup_timer
process_timeout
activate_task
rq
context_switch
mm/slab.c
/proc/slabinfo
logical memory
physically mapped memory
kfree
kmalloc
kmem_cache_alloc
kmem_cache
slab
Logical File Systems
ext3_file_operations
ext3_get_sb
protocols
proto
/proc/net/protocols
udp_prot
tcp_prot
ip_rcv
alloc_skb
ip_queue_xmit
ip_forward
sk_buff
HI subsystems
tty
mousedev_handler
log_buf
oss
alsa
drivers/input/
drivers/media/
sound/
devices control
drivers/
generic HW access
pci_driver
usb_driver
pci_register_driver
pci_request_regions
usb_hcd_giveback_urb
request_region
request_mem_region
ioremap
usb_submit_urb
usb_hcd
interrupt context
timer_list
tasklet_struct
jiffies ++
setup_irq
timer_interrupt
do_softirq
do_IRQ
irq_desc
mm/page_alloc.c
/proc/buddyinfo
Page Allocator
get_free_pages
zone
alloc_pages
page
get_page_from_freelist
try_to_free_pages
Block devices
block/
gendisk
block_device_operations
request_queue
init_scsi
scsi_device
scsi_driver
sd_fops
usb_storage_driver
virtual network device
net_device
netif_rx
dev_queue_xmit
alloc_etherdev
alloc_ieee80211
ether_setup
ieee80211_rx
ieee80211_xmit
abstract devices and HID class drivers
console
kbd
fb_ops
mousedev
video_device
hardware interfaces
include/asm/
arch/i386/
devices access and bus drivers
ehci_irq
writew
readw
ehci_urb_enqueue
pci_read
pci_write
outw
inw
usb_hcd_irq
arch/i386/kernel/
CPU specific
start_thread
/proc/interrupts
interrupt
atomic_t
system_call
show_regs
trap_init
switch_to
pt_regs
cli
sti
arch/i386/mm/
physical memory operations
die
do_page_fault
Scsi_Host
libata
disk controllers drivers
ahci_pci_driver
idedisk_ops
ide_intr
aic94xx_init
ide_do_request
drivers/net/
network devices drivers
net
e100_open
ipw2100_open
rtl8139_open
zd1201_net_open
HI peripherals device drivers
vga_con
atkbd_drv
psmouse
i8042_driver
ac97_driver
drivers/video/
electronics
I/O
I/O mem
I/O ports
USB controller
PCI controller
DMA
CPU
registers
APIC
interrupt controller
memory
RAM
MMU
disk controllers
SCSI
IDE
SATA
network controllers
Ethernet
WiFi
user peripherals
keyboard
cam
mouse
audio
graphics card

Layered Systems

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure 1-25. Structure of the the operating system.

Windows NT Architecture



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

- Put the mechanism in the kernel and the policy in user-mode
  - For example: job scheduling
    - Kernel runs the highest priority process
    - User-mode job scheduler decides priorities

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

Microkernels

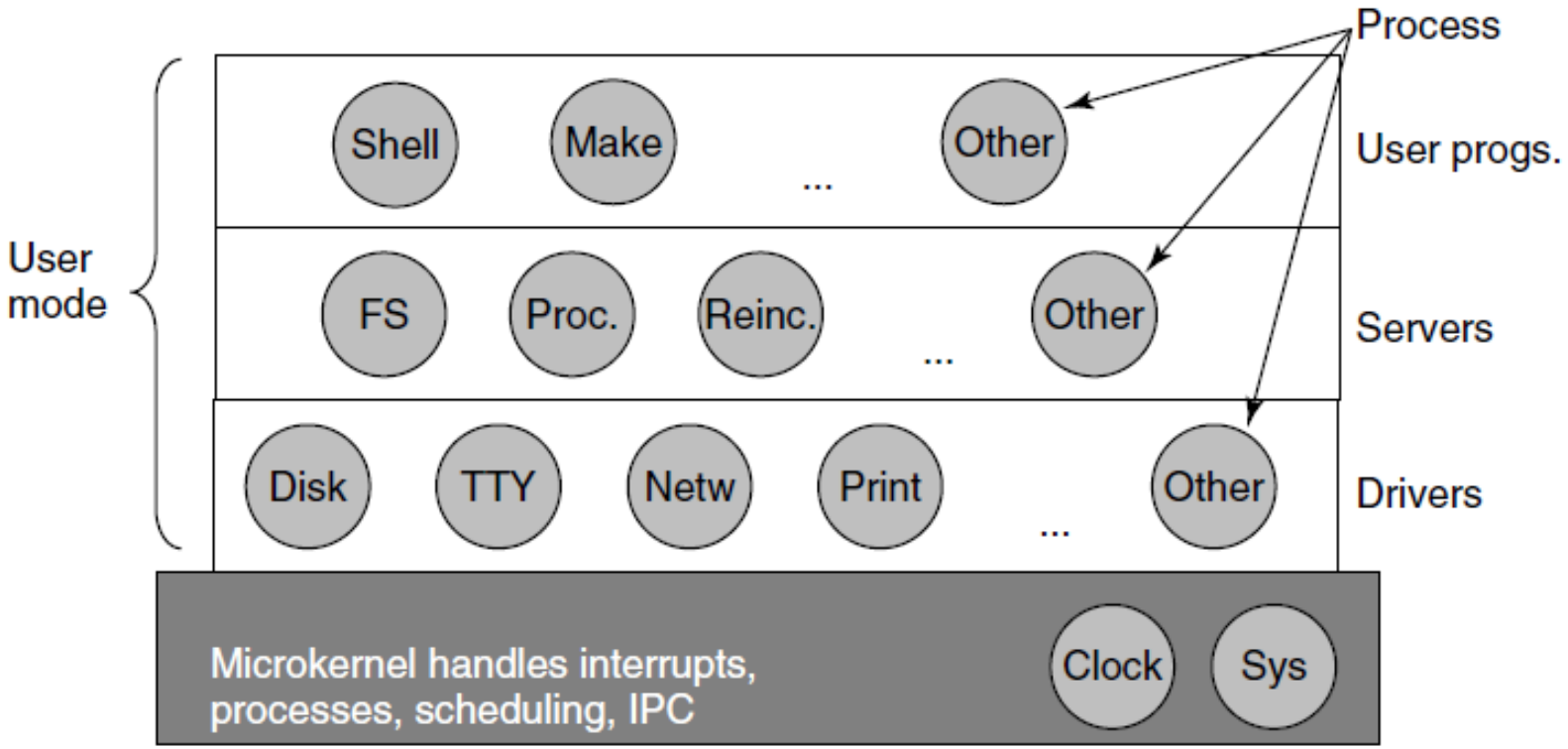
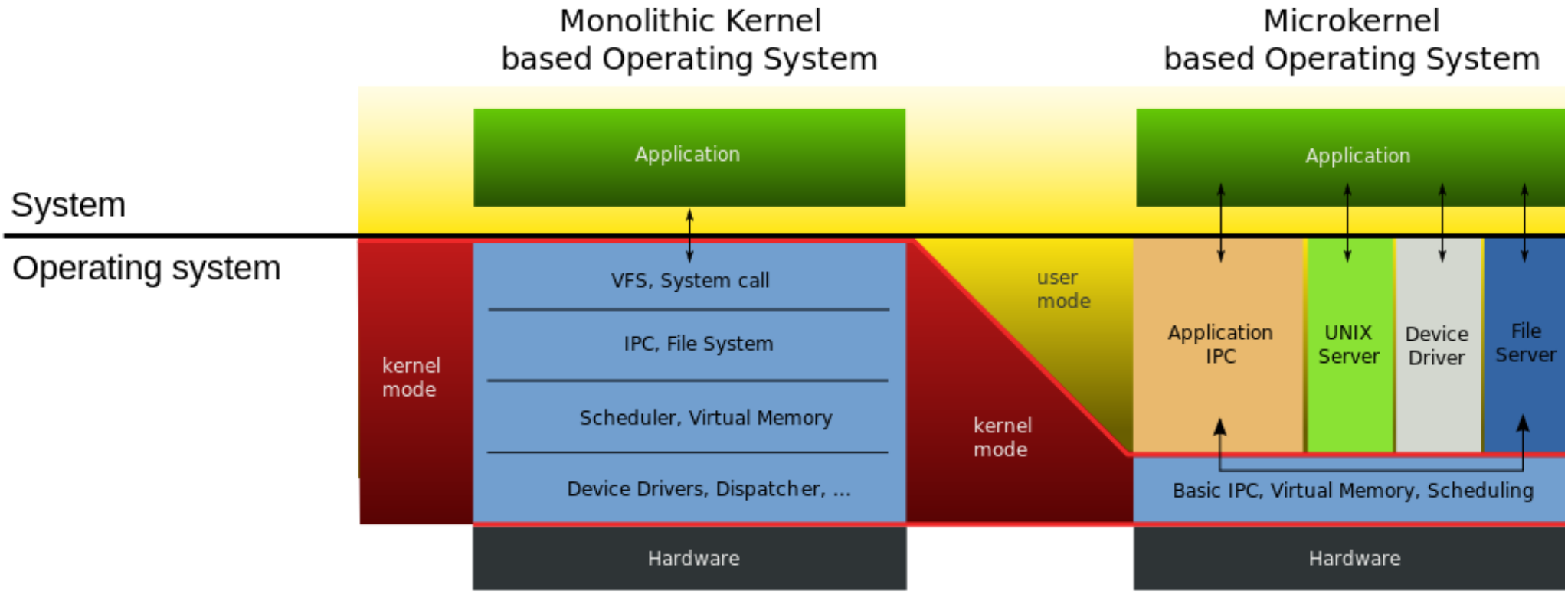


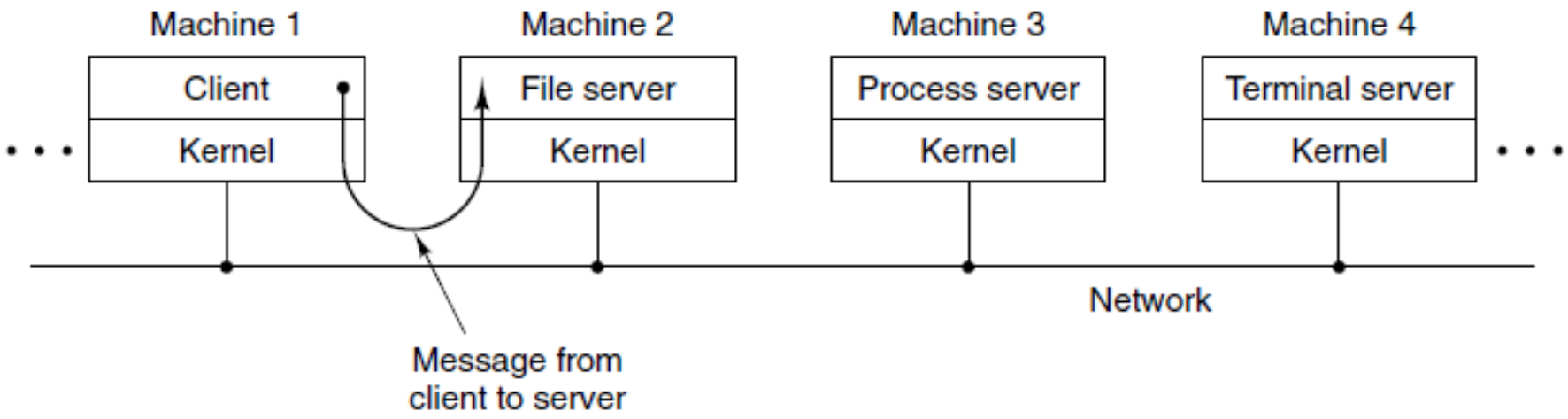
Figure 1-26. Simplified structure of the MINIX 3 system.

Micro v. Monolithic

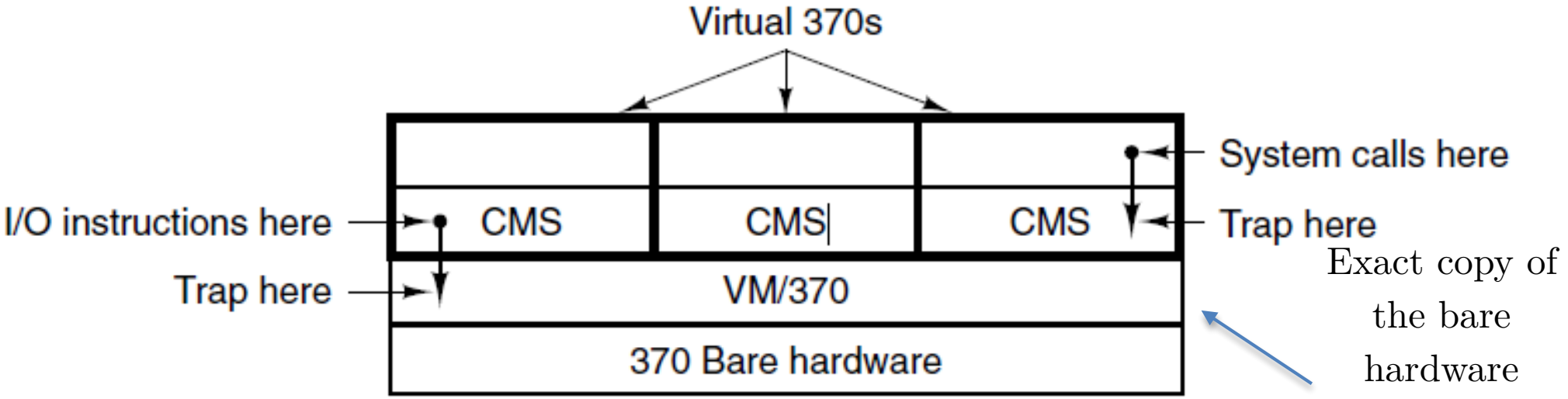


"OS-structure2" by Golftheman - http://en.wikipedia.org/wiki/Image:OS-structure.svg. Licensed under Public domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:OS-structure2.svg#mediaviewer/File:OS-structure2.svg

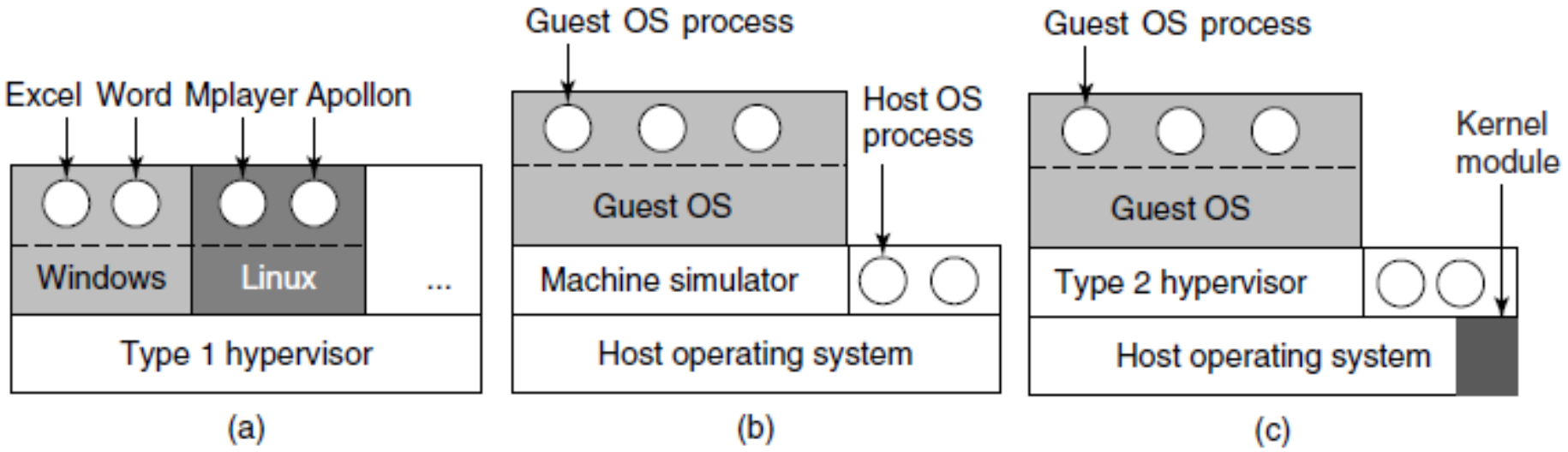
Client-Server Model



Virtual Machines



Virtual Machines Rediscovered



Virtual Machines Rediscovered

- x86 architecture was a problem
  - CPU must be virtualizable
    - When the OS executes a privileged instructions modifying the PSW or executing I/O, the hardware must trap to the VM
    - Pentium did not allow privileged instructions in user mode.

Virtual Machines Rediscovered

- In 2005 and 2006, Intel and AMD (working independently) created new processor extensions to the x86 architecture.
- The first generation of x86 hardware virtualization addressed the issue of privileged instructions.
- The issue of low performance of virtualized system memory was addressed with MMU virtualization that was added to the chipset later.

https://en.wikipedia.org/wiki/X86_virtualization

Powers of 2

n	2^n	n	2^n	n	2^n
0	1	11	2,048	22	4,194,304
1	2	12	4,096	23	8,388,608
2	4	13	8,192	24	16,777,216
3	8	14	16,384	25	33,554,432
4	16	15	32,768	26	67,108,864
5	32	16	65,536	27	134,217,728
6	64	17	131,072	28	268,435,456
7	128	18	262,144	29	536,870,912
8	256	19	524,288	30	1,073,741,824
9	512	20	1,048,576	31	2,147,483,648
10	1,024	21	2,097,152	32	4,254,967,296

Can't emphasize this enough. Learn them. You will need them!

End

Chapter 1