Operating Systems

Lecture Handouts

Wang Xiaolin
wx672ster+os@gmail.com

August 28, 2020

Contents

References

[1] BRYANT R E, O'HALLARON D R. *Computer Systems: A Programmer's Perspective*. 2nd ed. USA: Addison-Wesley, 2010.

[2] TANENBAUM A S. *Modern Operating Systems*. 3rd ed. Prentice Hall Press, 2007.

[3] BOVET D, CESATI M. *Understanding The Linux Kernel*. 3rd ed. OReilly, 2005.

[4] Wikipedia. *Process (computing) — Wikipedia, The Free Encyclopedia*. 2014. http://en.wikipedia.org/w/index.php?title=Process%5C_(computing)&oldid=639847817.

[5] KERRISK M. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.

[6] MATTHEW N, STONES R. *Beginning linux programming*. John Wiley & Sons, 2008.

[7] DOWNEY A B. *The Little Book of Semaphores*. greenteapress.com, 2008.

[8] Silberschatz, Galvin, Gagne. *Operating System Concepts Essentials*. 1st ed. John Wiley & Sons, 2011.

[9] LOVE R. *Linux Kernel Development*. Addison-Wesley, 2010.

[10] Wikipedia. *Compiler — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Compiler&oldid=661266598.

[11] Wikipedia. *Assembly language — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Assembly%5C_language&oldid=661185928.

[12] Wikipedia. *Linker (computing) — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Linker%5C_(computing)&oldid=652892136.

[13] Wikipedia. *Loader (computing) — Wikipedia, The Free Encyclopedia*. 2012. http://en.wikipedia.org/w/index.php?title=Loader%5C_(computing)&oldid=520743198.

[14] Wikipedia. *Dynamic linker — Wikipedia, The Free Encyclopedia*. 2012. http://en.wikipedia.org/w/index.php?title=Dynamic%5C_linker&oldid=517400345.

[15] LEVINE J. *Linkers and Loaders*. Morgan Kaufmann, 2000.

[16] Intel. *INTEL 80386 Programmer's Reference Manual*. http://pdos.csail.mit.edu/6.828/2004/readings/i386/toc.htm. 1986.

[17] Wikipedia. *Executable and Linkable Format — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Executable%5C_and%5C_Linkable%5C_Format&oldid=659380509.

[18] Wikipedia. *Open (system call) — Wikipedia, The Free Encyclopedia*. 2014. http://en.wikipedia.org/w/index.php?title=Open%5C_(system%5C_call)&oldid=611838618.

[19] Wikipedia. *File descriptor — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=File%5C_descriptor&oldid=661810398.

[20] STALLINGS W. *Operating Systems: Internals and Design Principles*. 7th ed. Prentice Hall, 2011.

[21] Wikipedia. *File Allocation Table — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=File%5C_Allocation%5C_Table&oldid=661104239.

[22] Wikipedia. *Inode — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Inode&oldid=647736522.

[23] BACH M. *The design of the UNIX operating system*. Prentice-Hall, 1986.

[24] THOMPSON K. "Unix Implementation". Bell System Technical Journal, 1978, 57: 1931-1946.

[25] Wikipedia. *Directed acyclic graph — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Directed%5C_acyclic%5C_graph&oldid=654052259.

[26] POIRIER D. *The Second Extended File System Internal Layout*. Web, 2011.

[27] RUSLING D A. *The Linux Kernel*. Linux Documentation Project, 1999.

[28] CARD R, TS'O T, TWEEDIE S. "Design and Implementation of the Second Extended Filesystem". Dutch International Symposium on Linux, 1996.

[29] MORGAN D. *Analyzing a filesystem*. http://homepage.smc.edu/morgan_david/cs40/analyze-ext2.htm. 2012.

Course Web Links
Course web site https://cs6.swfu.edu.cn/moodle
Lecture slides https://cs6.swfu.edu.cn/~wx672/lecture_notes/os/slides/
Source code https://cs6.swfu.edu.cn/~wx672/lecture_notes/os/src/
Lab instructions https://cs6.swfu.edu.cn/~wx672/lecture_notes/os/lab.html
Sample lab report https://cs6.swfu.edu.cn/~wx672/lecture_notes/os/sample-report/
System Programming https://github.com/angrave/SystemProgramming/wiki
Beej's Guides http://beej.us/guide/

Homework

Weekly tech question

1. What was I trying to do?
2. How did I do it? (steps)
3. The expected output? The real output?
4. How did I try to solve it? (steps, books, web links)
5. How many hours did I struggle on it?

https://cs6.swfu.edu.cn/moodle/mod/forum/view.php?id=53
✉ *wx672ster+os@gmail.com*
𝔼 Preferably in English
in stackoverflow style
OR simply show me the tech questions you asked on any website

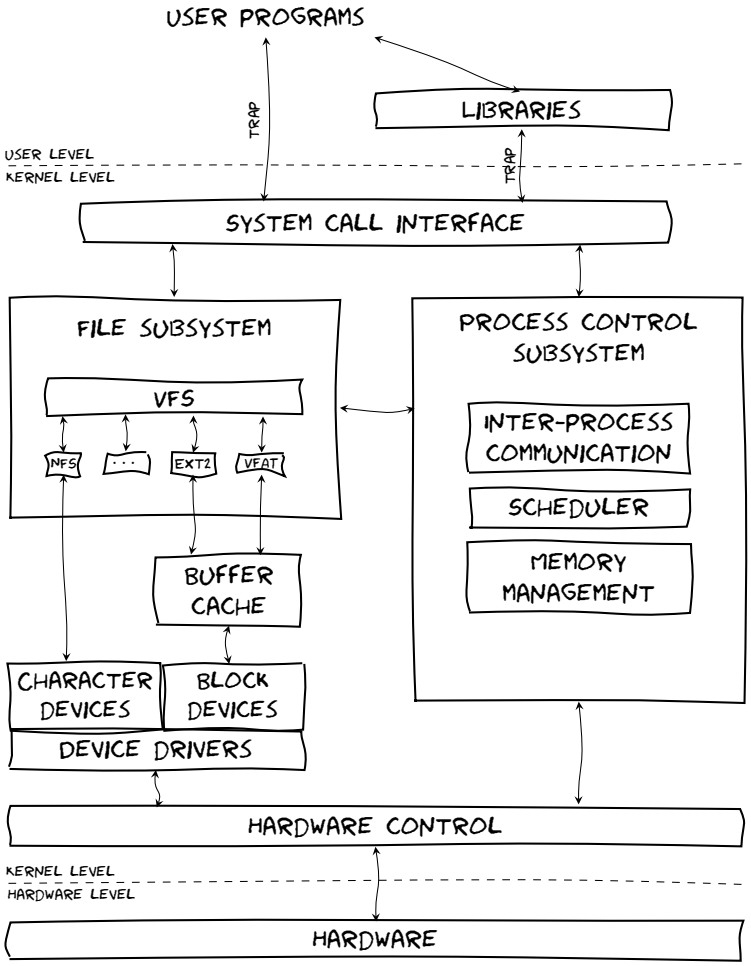


Fig. 1: What's in the OS?

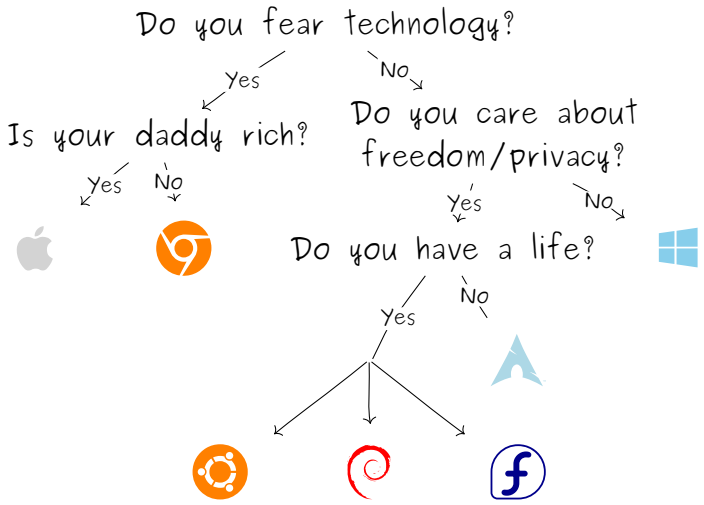
Part I
Introduction

1 What's an Operating System

What's an Operating System?

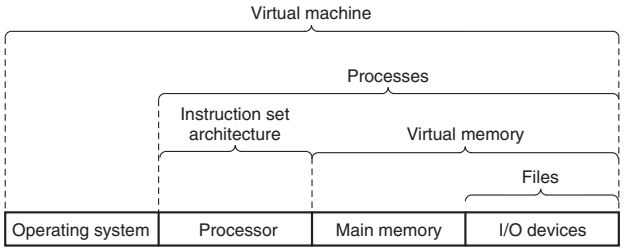
- *"Everything a vendor ships when you order an operating system"*
- It's the program that runs all the time
- It's a *resource manager*
  - Each program gets time with the resource
  - Each program gets space on the resource
- It's a *control program*
  - Controls execution of programs to prevent errors and improper use of the computer
- No universally accepted definition

Choosing an OS



Abstractions

To hide the complexity of the actual implementations



See also: [*Computer Systems: A Programmer's Perspective*, Sec. 1.9.2, *The Importance of Abstractions in Computer Systems*].

System Goals

Convenient vs. Efficient

- Convenient for the user — for PCs
- Efficient — for mainframes, multiusers
- UNIX
  - Started with keyboard + printer, none paid to convenience
  - Now, still concentrating on efficiency, with GUI support

History of Operating Systems

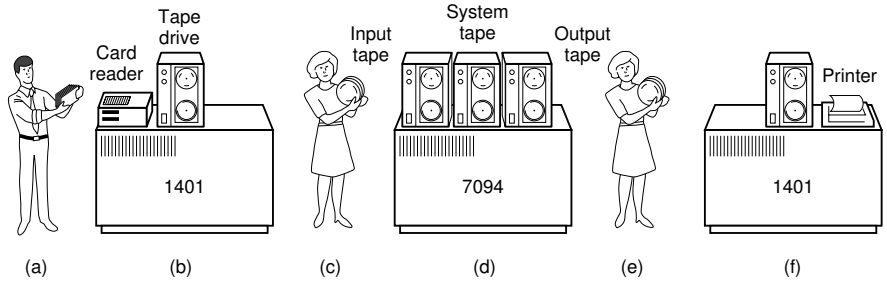


Fig. 1-2. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

1945 - 1955 First generation
- vacuum tubes, plug boards

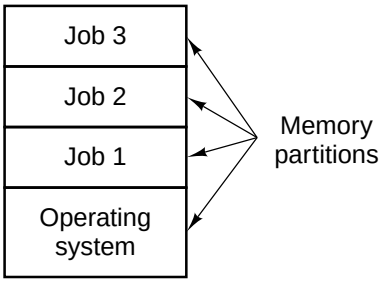
1955 - 1965 Second generation
- transistors, batch systems

1965 - 1980 Third generation
- ICs and multiprogramming

1980 - present Fourth generation
- personal computers

Multi-programming is the first instance where the OS must make decisions for the users

Job scheduling — decides which job should be loaded into the memory.
Memory management — because several programs in memory at the same time
CPU scheduling — choose one job among all the jobs are ready to run
Process management — make sure processes don't offend each other



The Operating System Zoo

- Mainframe OS
- Server OS
- Multiprocessor OS
- Personal computer OS
- Real-time OS
- Embedded OS
- Smart card OS

2 OS Services

OS Services
Like a government

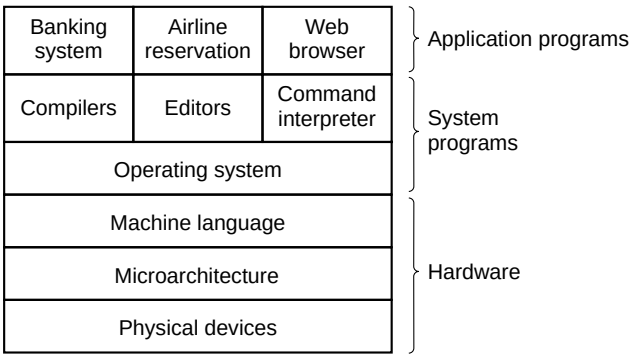
Helping the users:

- User interface
- Program execution
- I/O operation
- File system manipulation
- Communication
- Error detection

Keeping the system efficient:

- Resource allocation
- Accounting
- Protection and security

A Computer System



3 Hardware

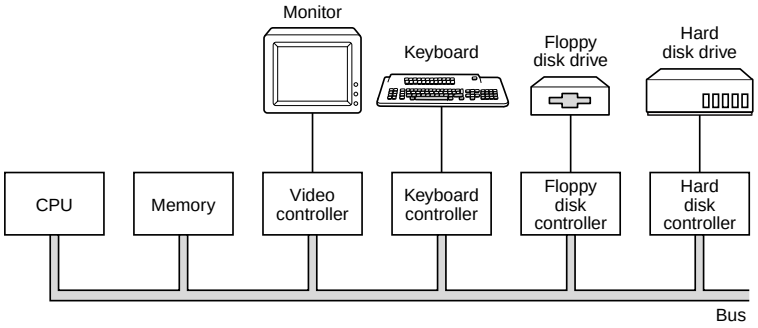
CPU Working Cycle

1. Fetch the first instruction from memory
2. Decode it to determine its type and operands
3. execute it

Special CPU Registers
Program counter (PC): keeps the memory address of the next instruction to be fetched
Stack pointer (SP): ☛ the top of the current stack in memory
Program status (PS): holds

- condition code bits
- processor state

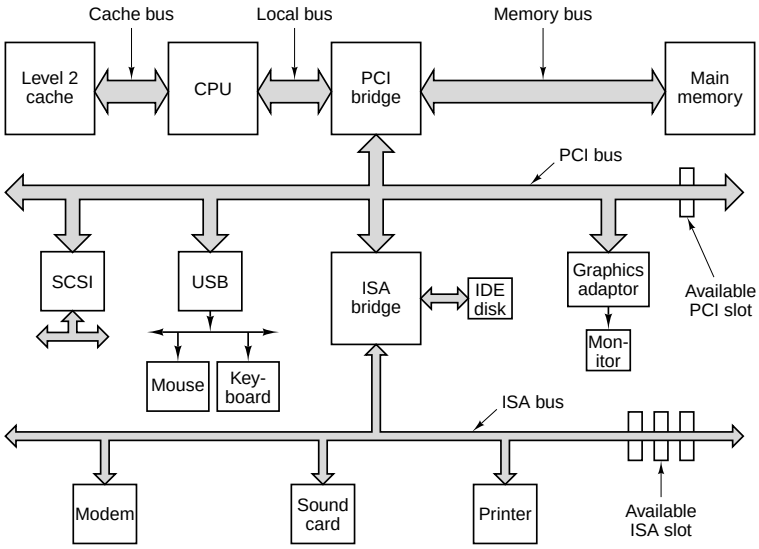
System Bus



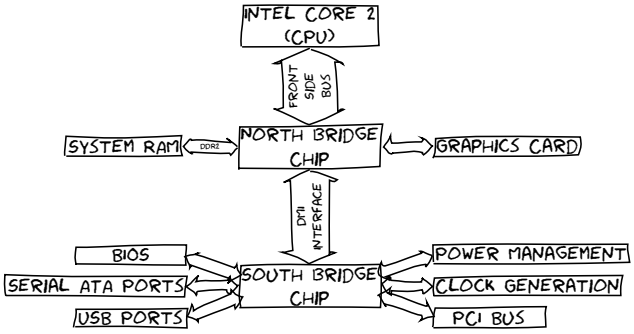
Address Bus: specifies the memory locations (addresses) for the data transfers
Data Bus: holds the data transfered. Bidirectional
Control Bus: contains various lines used to route timing and control signals throughout the system

Controllers and Peripherals

- Peripherals are real devices controlled by controller chips
- Controllers are processors like the CPU itself, have control registers
- Device driver writes to the registers, thus control it
- Controllers are connected to the CPU and to each other by a variety of buses



Motherboard Chipsets



The northbridge

1. receives a physical memory request
2. decides where to route it
   - to RAM? to video card? to …?
   - decision made via the *memory address map*

See also:

- When is the memory address map built? `setup()`.
- *Motherboard Chipsets And The Memory Map* [1].
- The CPU doesn't know what it's connected to
  - CPU test bench? network router? toaster? brain implant?
- The CPU talks to the outside world through its pins
  - some pins to transmit the physical memory address
  - other pins to transmit the values
- The CPU's gateway to the world is the *front-side bus*

Intel Core 2 QX6600

- 33 pins to transmit the physical memory address

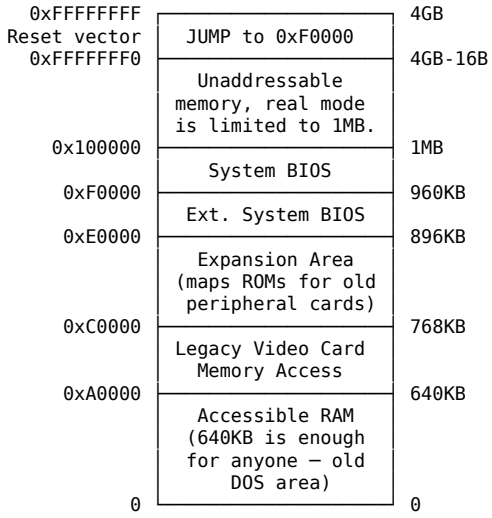
[1] http://duartes.org/gustavo/blog/post/motherboard-chipsets-memory-map

- so there are 2^{33} choices of memory locations

• 64 pins to send or receive data
  - so data path is 64-bit wide, or 8-byte chunks

This allows the CPU to physically address 64GB of memory ($2^{33} \times 8B$)

See also: *Datasheet for Intel Core 2 Quad-Core Q6000 Sequence* [2].



Some physical memory addresses are mapped away!

- only the addresses, not the spaces
- Memory holes
  - $640KB \sim 1MB$
  - `/proc/iomem`

Memory-mapped I/O

- BIOS ROM
- video cards
- PCI cards
- …

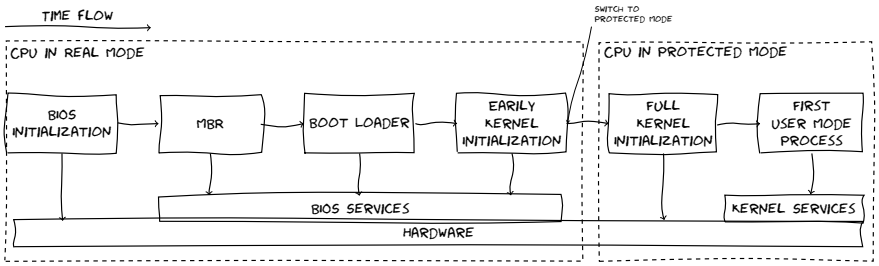
This is why 32-bit OSes have problems using 4 gigs of RAM.

4 Bootstrapping

Bootstrapping

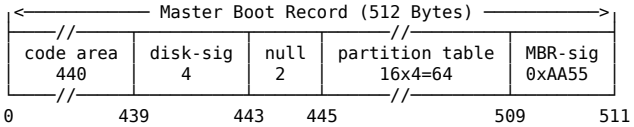
Can you pull yourself up by your own bootstraps?

A computer cannot run without first loading software but must be running before any software can be loaded.



Intel x86 Bootstrapping

1. BIOS (`0xffffffff0`)
   ➠ POST ➠ HW init ➠ Find a boot device (FD,CD,HD…) ➠ Copy *sector zero (MBR)* to RAM (`0x00007c00`)
2. MBR – the first 512 bytes, contains
   - Small code ($< 446\,Bytes$), e.g. GRUB stage 1, for loading GRUB stage 2
   - the primary partition table ($16 \times 4 = 64\,Bytes$)
   - its job is to load the second-stage boot loader.
3. GRUB stage 2 — load the OS kernel into RAM
4. 🐧 startup
5. init — the first user-space program




```
$ sudo hd -n512 /dev/sda
```

[2] http://download.intel.com/design/processor/datashts/31559205.pdf

5 Interrupt

Why Interrupt?

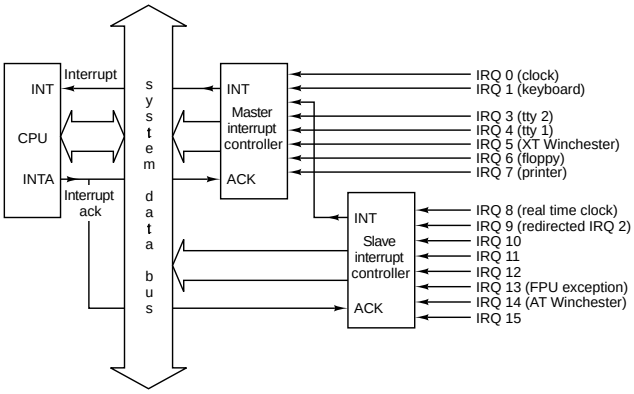
While a process is reading a disk file, can we do...

```
while(!done_reading_a_file())
{
  let_CPU_wait();
  /* or... */
  lend_CPU_to_others();
}
operate_on_the_file();
```

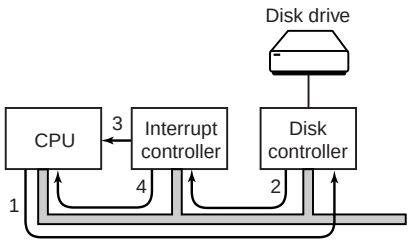
Modern OS are Interrupt Driven
HW INT by sending a signal to CPU
SW INT by executing a *system call*
Trap (exception) is a software-generated INT coursed by an error or by a specific request from an user program
Interrupt vector is an array of pointers ☛ the memory addresses of *interrupt handlers*. This array is indexed by a unique device number

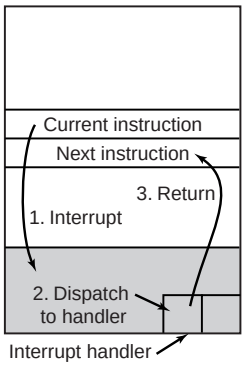
```
$ less /proc/devices
$ less /proc/interrupts
```

Programmable Interrupt Controllers



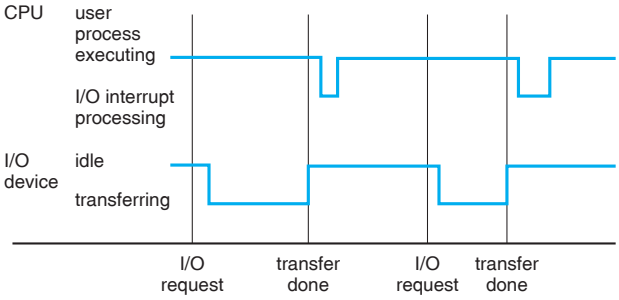
Interrupt Processing





Detailed explanation: in [*Modern Operating Systems*, Sec. 1.3.5, *I/O Devices*].

Interrupt Timeline



6 System Calls

System Calls

A System Call

- is how a program requests a service from an OS kernel
- provides the interface between a process and the OS

```
$ man 2 intro
$ man 2 syscalls
```

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

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

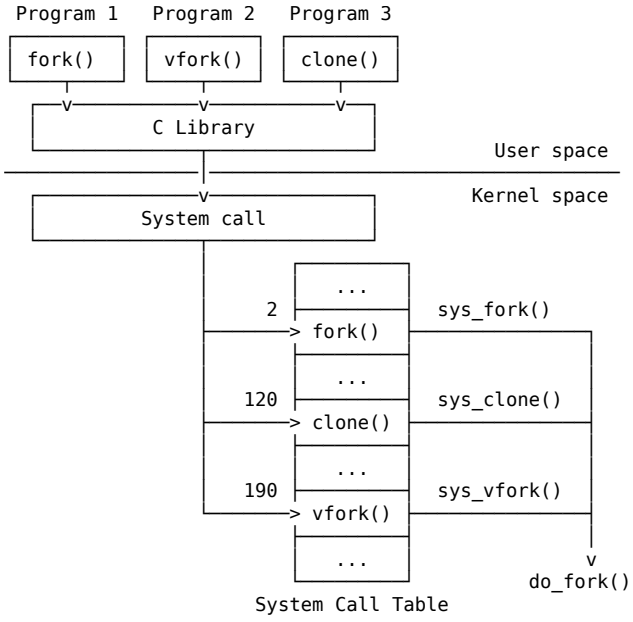
Directory and file system management

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

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

Fig. 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. The parameters are explained in the text.



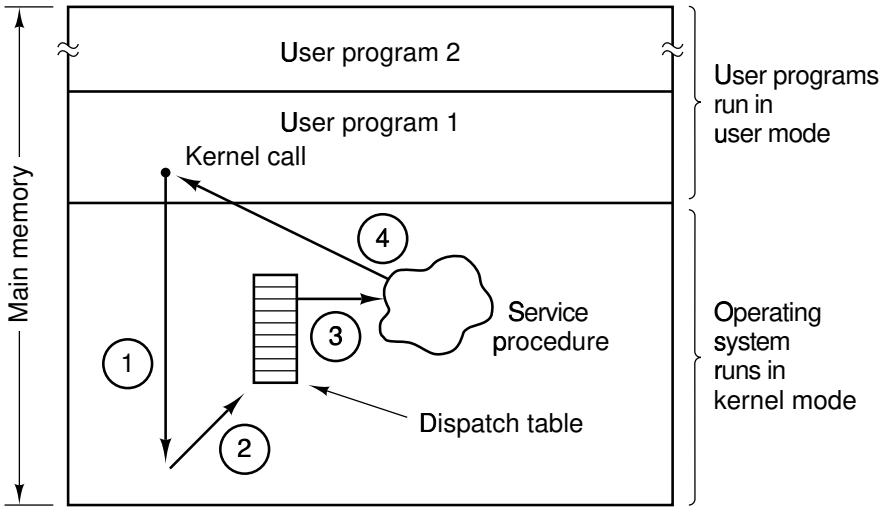
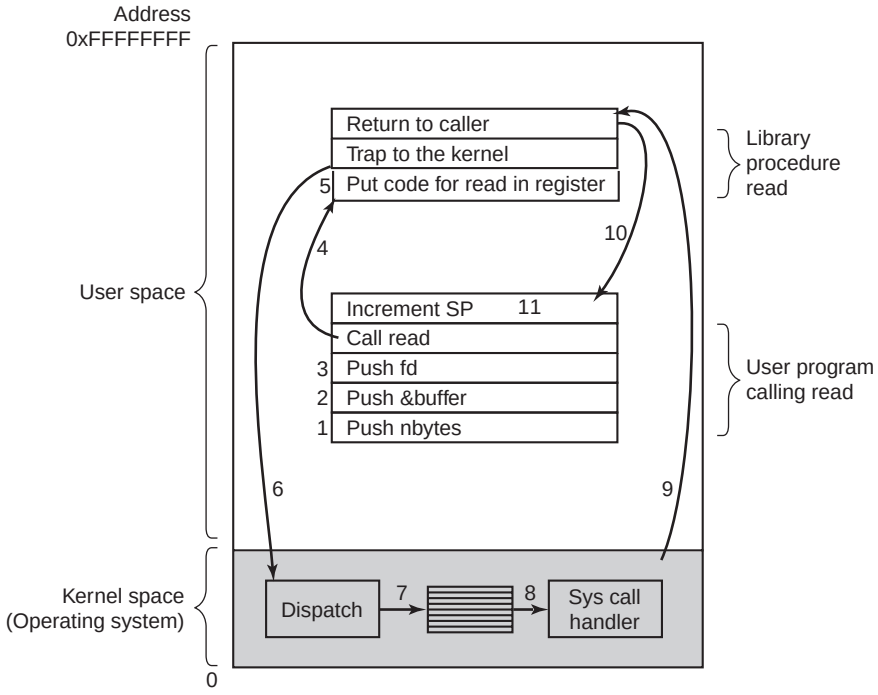


Figure 1-16. How a system call can be made: (1) User program traps to the kernel. (2) Operating system determines service number required. (3) Operating system calls service procedure. (4) Control is returned to user program.

The 11 steps in making the system call `read(fd,buffer,nbytes)`



Example
Linux INT80h
Interrupt Vector Table: The very first 1KiB of x86 memory.

- 256 entries × 4B = 1KiB
- Each entry is a complete memory address (segment:offset)
- It's populated by Linux and BIOS
- Slot 80h: address of the kernel services dispatcher (☛ sys-call table)

- https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl

Example

```
        ; https://www.devdungeon.com/content/hello-world-nasm-assembler
        ; Compile: nasm -f elf hello32.asm -o hello.o
        ;    Link: ld hello.o -o hello -m elf_i386
        ;     Run: ./hello
        ;
        ; strace issue:
                ; https://github.com/strace/strace/issues/103 (need kernel 5.3+)
                ; https://stackoverflow.com/questions/57850588/confused-by-strace-output-of-a-
                ↪  simple-helloworld-nasm-program/57884132#57884132
        ;

SECTION .DATA
Msg:     db 'Hello, world!',10  ; 10 = ascii for LF
MsgLen:  equ $-Msg

SECTION .TEXT
GLOBAL _start

_start:
        mov eax, 4       ; write(
        mov ebx, 1       ;   STDOUT_FILENO,
        mov ecx, Msg     ;   "Hello, world!\n",
        mov edx, MsgLen  ;   sizeof("Hello, world!\n")
        int 80h          ; );

        mov eax, 1       ; exit(
        mov ebx, 0       ;   EXIT_SUCCESS
        int 80h          ; );
```

System call numbers:

```
$ less /usr/include/asm/unistd_32.h
$ less /usr/include/asm/unistd_64.h
```

64-bits version:

```
        ; https://jameshfisher.com/2018/03/10/linux-assembly-hello-world/
        ;
        ; $ nasm -f elf64 -o hello64.o hello64.s
        ; $ ld -o hello64 hello64.o
        ; $ ./hello64

global _start

section .text

_start:
  mov rax, 1        ; write(
  mov rdi, 1        ;   STDOUT_FILENO,
  mov rsi, msg      ;   "Hello, world!\n",
  mov rdx, msglen   ;   sizeof("Hello, world!\n")
  syscall           ; );

  mov rax, 60       ; exit(
  mov rdi, 0        ;   EXIT_SUCCESS
  syscall           ; );

section .rodata
  msg: db "Hello, world!", 10
  msglen: equ $ - msg
```

System Call Examples

```c
#include <unistd.h>

int main(void)
{
  write(1, "Hello, world!\n", 14);

  return 0;
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra write.c -o
↪  write" */
/* End: */
```

- Actually, write() is a wrapper function in glibc.

```
$ man 2 write
$ man 3 write
```

Don't invoke syscall directly whenever possible

```c
/* Calling sys_write using inline assembly code */
/* https://jameshfisher.com/2018/02/20/c-inline-assembly-hello-world/ */
int main(void) {
  register char* arg2 asm("rsi") = "hello, world!\n";

  /* rax: sys_write; rdi: STDOUT; */
  asm("mov $1, %rax; mov $1, %rdi; mov $14, %rdx; syscall;");

  return 0;
}

/* Local Variables: */
/* compile-command: "gcc -Wall write-inlineasm.c -o /tmp/a.out" */
/* End: */
```

- https://jameshfisher.com/2018/02/20/c-inline-assembly-hello-world/
- https://cs.lmu.edu/~ray/notes/gasexamples/
- https://montcs.bloomu.edu/Information/LowLevel/Assembly/assembly-tutorial.html

System Call Examples

```
$ man 2 fork
```

```c
/* Basically, the fork() call, inside a process, creates an exact copy of that process somewhere
   else in the memory (meaning it'll copy variable values, etc...), and runs the copy from the
↪  point
   the call was made (for the assembly kids : it means that the relative value of the next
   instruction pointer is also copied)  */

/* When we launch this program, it first goes through the first puts(). Then, the fork() makes a
   copy of this program. Finally, each one of this program and its copy goes through the second
   puts().  */

#include <stdio.h>
#include <unistd.h>

int main ()
{
  puts("Hello World!");
  fork();
  puts("Goodbye Cruel World!");
  return 0;
}
/* Local Variables: */
/* compile-command: "gcc -Wall fork.c -o /tmp/fork" */
/* End: */
```

```
$ man 3 exec
```

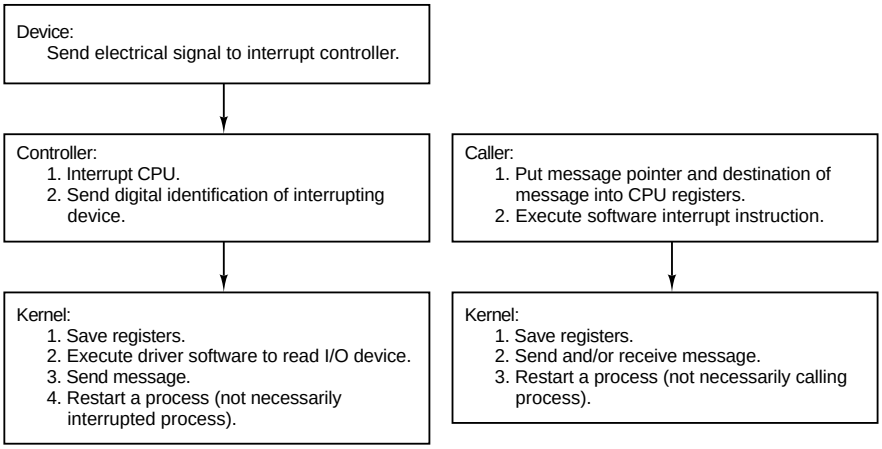
```c
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>

int main()
{
  printf("Main process (%d) saying hello!\n", getpid());

  if(fork() == 0 ){                          /* child */
    printf("Child (%d) is listing the source file...\n", getpid());

    execl("/bin/ls", "", NULL);

    /* useless after execl() */
    puts("You can't see this line unless execl() failed.\n");
  }
  else {                                                 /* parent */
    int i=60;
    printf("parent (%d) is sleeping for %d seconds...\n", getpid(),i);
    sleep(i);                                          /* for zombie */
  }

  /* only the one who doesn't call execl() can do the following. */
  printf("Hello again from process %d\n", getpid());

  return 0;
}

/* https://unix.stackexchange.com/questions/315812/why-does-argv-include-the-program-name */

/* Local Variables: */
/* compile-command: "gcc -Wall fork-exec.c" */
/* End: */
```

Quoted from stackoverflow: What is the difference between the functions of the *exec* family of system calls:

> There is no *exec* system call — this is usually used to refer to all the *execXX* calls as a group. They all do essentially the same thing: loading a new program into the current process, and provide it with arguments and environment variables. The differences are in how the program is found, how the arguments are specified, and where the environment comes from.
>
> - The calls with *v* in the name take an array parameter to specify the `argv[]` array (*vector*) of the new program.
> - The calls with *l* in the name take the arguments of the new program as a variable-length argument *list* to the function itself.
> - The calls with *e* in the name take an extra argument to provide the *environment* of the new program; otherwise, the program inherits the current process's environment.
> - The calls with *p* in the name search the *PATH* environment variable to find the program if it doesn't have a directory in it (i.e. it doesn't contain a / character). Otherwise, the program name is always treated as a path to the executable.

- https://stackoverflow.com/questions/174942/how-should-strace-be-used
- https://unix.stackexchange.com/questions/160578/strace-hello-world-program

Hardware INT vs. Software INT



[1] Wikipedia. *Interrupt — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Interrupt&oldid=646521061.

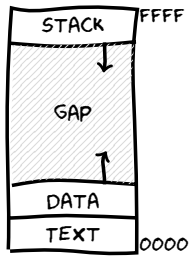
[2] Wikipedia. *System call — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=System%5C_call&oldid=647910319.

Part II
Process And Thread

7 Processes

7.1 What's a Process

Process
A process is an instance of a program in execution



Processes are like human beings:
- they are generated
- they have a life
- they optionally generate one or more child processes, and
- eventually they die

A small difference:
- sex is not really common among processes
- each process has just one parent

The term "process" is often used with several different meanings. In this book, we stick to the usual OS textbook definition: a process is an instance of a program in execution. You might think of it as *the collection of data structures that fully describes how far the execution of the program has progressed*. [*Understanding The Linux Kernel*, Sec. 3.1, *Processes, Lightweight Processes, and Threads*]

Processes are like human beings: they are generated, they have a more or less significant life, they optionally generate one or more child processes, and eventually they die. A small difference is that sex is not really common among processes each process has just one parent.

From the kernel's point of view, the purpose of a process is to act as an entity to which system resources (CPU time, memory, etc.) are allocated.

In general, a computer system process consists of (or is said to 'own') the following resources [*Process (computing) — Wikipedia, The Free Encyclopedia*]:
- An image of the executable machine code associated with a program.
- Memory (typically some region of virtual memory); which includes the executable code, process-specific data (input and output), a call stack (to keep track of active subroutines and/or other events), and a heap to hold intermediate computation data generated during run time.
- Operating system descriptors of resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows), and data sources and sinks.
- Security attributes, such as the process owner and the process' set of permissions (allowable operations).
- Processor state (context), such as the content of registers, physical memory addressing, etc. The state is typically stored in computer registers when the process is executing, and in memory otherwise.

The operating system holds most of this information about active processes in data structures called process control blocks.

Any subset of resource, but typically at least the processor state, may be associated with each of the process' threads in operating systems that support threads or 'daughter' processes.

The operating system keeps its processes separated and allocates the resources they need, so that they are less likely to interfere with each other and cause system failures (e.g., deadlock or thrashing). The operating system may also provide mechanisms for inter-process communication to enable processes to interact in safe and predictable ways.

7.2 PCB

Process Control Block (PCB)

Implementation

A process is *the collection of data structures* that fully describes how far the execution of the program has progressed.

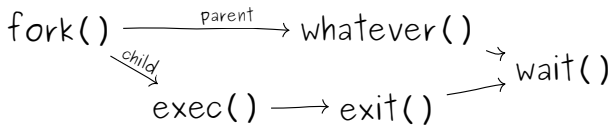
- Each process is represented by a *PCB*
- `task_struct` in 🐧

process state
PID
program counter
registers
memory limits
list of open files
...

To manage processes, the kernel must have a clear picture of what each process is doing. It must know, for instance, the process's priority, whether it is running on a CPU or blocked on an event, what address space has been assigned to it, which files it is allowed to address, and so on. This is the role of the process descriptor a `task_struct` type structure whose fields contain all the information related to a single process. As the repository of so much information, the process descriptor is rather complex. In addition to a large number of fields containing process attributes, the process descriptor contains several pointers to other data structures that, in turn, contain pointers to other structures[*Understanding The Linux Kernel*, Sec. 3.2, *Process Descriptor*].

7.3 Process Creation

Process Creation



- When a process is created, it is almost identical to its parent
  - It receives a (logical) copy of the parent's address space, and
  - executes the same code as the parent
- The parent and child have separate copies of the data (stack and heap)

When a process is created, it is almost identical to its parent. It receives a (logical) copy of the parent's address space and executes the same code as the parent, beginning at the next instruction following the process creation system call. Although the parent and child may share the pages containing the program code (text), they have separate copies of the data (stack and heap), so that changes by the child to a memory location are invisible to the parent (and vice versa)[*Understanding The Linux Kernel*, Sec. 3.1, *Processes, Lightweight Processes, and Threads*].

While earlier Unix kernels employed this simple model, modern Unix systems do not. They support *multi-threaded applications* user programs having many relatively independent execution flows sharing a large portion of the application data structures. In such systems, a process is composed of several *user threads* (or simply *threads*), each of which represents an execution flow of the process. Nowadays, most multi-threaded applications are written using standard sets of library functions called *pthread (POSIX thread) libraries*.

Traditional Unix systems treat all processes in the same way: resources owned by the parent process are duplicated in the child process. This approach makes process creation very slow and inefficient, because it requires copying the entire address space of the parent process. The child process rarely needs to read or modify all the resources inherited from the parent; in many cases, it issues an immediate `execve()` and wipes out the address space that was so carefully copied[*Understanding The Linux Kernel*, Sec. 3.4, *Creating Processes*].

Modern Unix kernels solve this problem by introducing three different mechanisms:

- Copy On Write
- Lightweight processes
- The `vfork()` system call

Forking in C

```
1 /* Basically, the fork() call, inside a process, creates an exact copy of that process somewhere
2    else in the memory (meaning it'll copy variable values, etc...), and runs the copy from the
  ↪  point
3    the call was made (for the assembly kids : it means that the relative value of the next
```

```
   instruction pointer is also copied)  */

/* When we launch this program, it first goes through the first puts(). Then, the fork() makes a
   copy of this program. Finally, each one of this program and its copy goes through the second
   puts().  */

#include <stdio.h>
#include <unistd.h>

int main ()
{
  puts("Hello World!");
  fork();
  puts("Goodbye Cruel World!");
  return 0;
}
/* Local Variables: */
/* compile-command: "gcc -Wall fork.c -o /tmp/fork" */
/* End: */
```

```
$ man fork
```

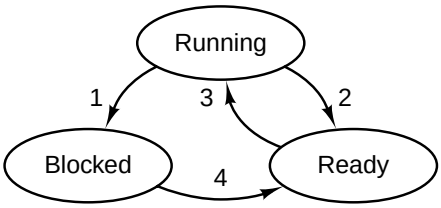
exec()

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

int main()
{
  pid_t pid;

  pid = fork();
  if(pid < 0) {        /* error */
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }
  else if(pid == 0){   /* child */
    execlp("/bin/ls", "ls", NULL);
  }
  else {               /* parent */
    wait(NULL);
    puts("Child Complete");
    exit(EXIT_SUCCESS);
  }
  return 0;
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra fork-exec-osc.c" */
/* End: */
```

```
$ man 3 exec
```

7.4 Process State

Process State Transition

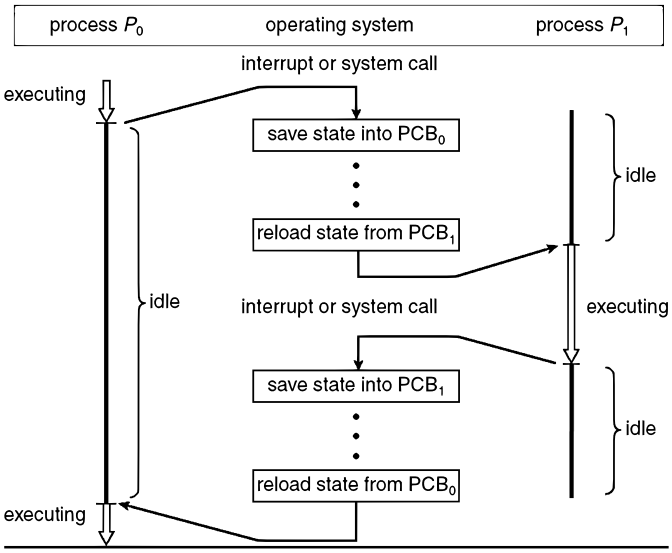


1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

See also: [*Understanding The Linux Kernel*, Sec. 3.2.1, *Process State*].

7.5 CPU Switch From Process To Process

CPU Switch From Process To Process



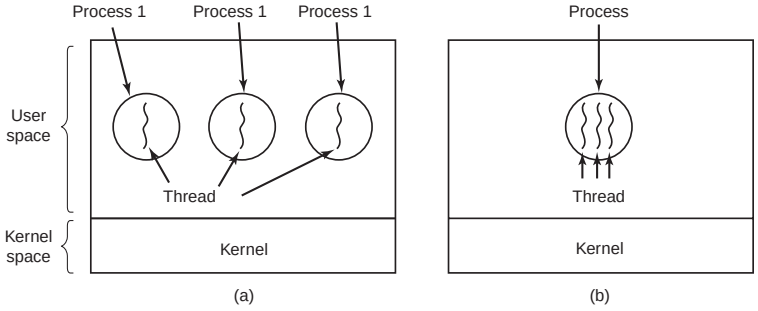
See also: [*Understanding The Linux Kernel*, Sec. 3.3, *Process Switch*].

8 Threads

8.1 Processes vs. Threads

Process vs. Thread

a single-threaded process = resource + execution
a multi-threaded process = resource + executions



A process = a unit of resource ownership, used to group resources together;
A thread = a unit of scheduling, scheduled for execution on the CPU.

Process vs. Thread

multiple threads running in one process:	**multiple processes running in one computer:**
share an address space and other resources	share physical memory, disk, printers ...

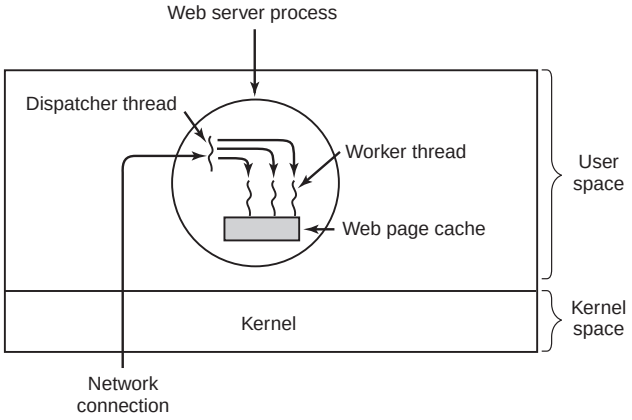
No protection between threads
impossible — because process is the minimum unit of resource management
unnecessary — a process is owned by a single user

Threads

code, data, open files, signals...		
thread ID	thread ID	thread ID
program counter	program counter	program counter
register set	register set	register set
stack	stack	stack

8.2 Why Thread?

A Multi-threaded Web Server



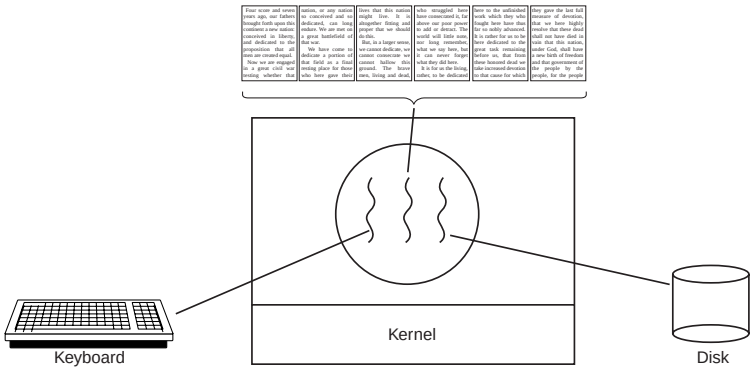

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

A Word Processor With 3 Threads



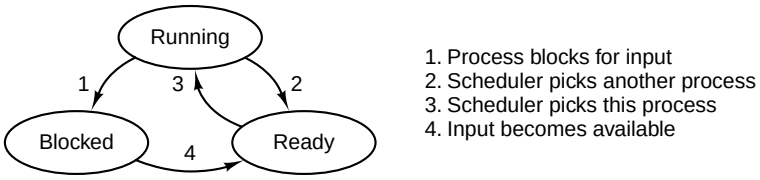
Why Having a Kind of Process Within a Process?

- Responsiveness
  - Good for interactive applications.
  - A process with multiple threads makes a great server (e.g. a web server):
    Have one server process, many ”worker” threads – if one thread blocks (e.g. on a read), others can still continue executing
- Economy – Threads are cheap!
  - Cheap to create – only need a stack and storage for registers
  - Use very little resources – don’t need new address space, global data, program code, or OS resources
  - switches are fast – only have to save/restore PC, SP, and registers
- Resource sharing – Threads can pass data via shared memory; no need for IPC
- Can take advantage of multiprocessors

8.3 Thread Characteristics

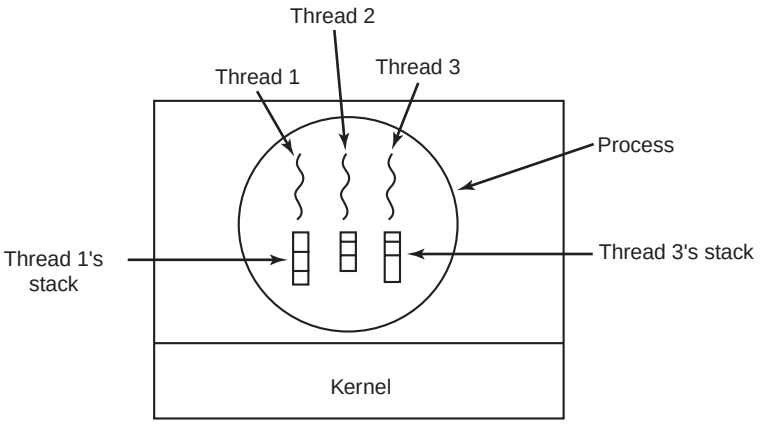
Thread States Transition

Same as process states transition

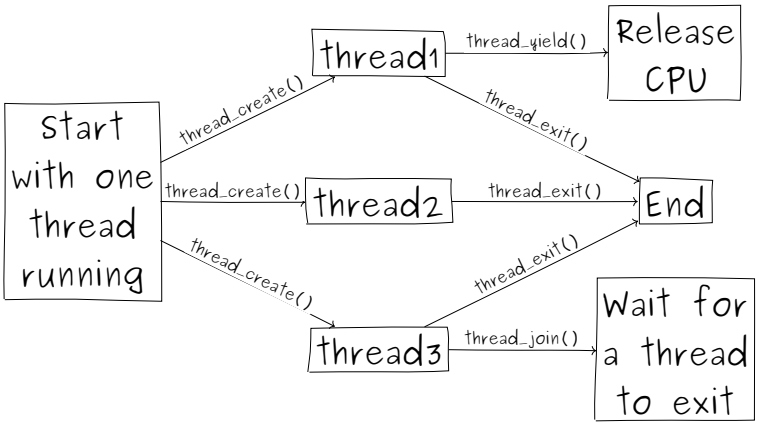


Each Thread Has Its Own Stack

- A typical stack stores local data and call information for (usually nested) procedure calls.
- Each thread generally has a different execution history.



Thread Operations



8.4 POSIX Threads

POSIX Threads

IEEE 1003.1c The standard for writing portable threaded programs. The threads package it defines is called *Pthreads*, including over 60 function calls, supported by most UNIX systems.

Some of the Pthreads function calls

Thread call	Description
`pthread_create`	Create a new thread
`pthread_exit`	Terminate the calling thread
`pthread_join`	Wait for a specific thread to exit
`pthread_yield`	Release the CPU to let another thread run
`pthread_attr_init`	Create and initialize a thread's attribute structure
`pthread_attr_destroy`	Remove a thread's attribute structure

Pthreads

Example 1

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

void *hello(void *arg)
{
  int i;
  for( i=0; i<20; i++ ){
        puts("Thread says hi!");
    sleep(1);
  }
  return NULL;
}

int main(void)
{
  pthread_t t;
  if( pthread_create(&t, NULL, hello, NULL) ){
    perror("error creating thread.");
    abort();
  }

  if( pthread_join(t, NULL) ){
```

```
    perror("error joining thread.");
    abort();
  }
  exit(0);
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra thread1.c -pthread" */
/* End: */
```

See also:

- IBM Developworks: POSIX threads explained[3].
- stackoverflow.com: What is the difference between `exit()` and `abort()`?[4].

Pthreads

`pthread_t` defined in `pthread.h`, is often called a ”thread id” (`tid`);
`pthread_create()` returns zero on success and a non-zero value on failure;
`pthread_join()` returns zero on success and a non-zero value on failure;

How to use pthread?

```
  #include<pthread.h>
$ gcc thread1.c -o thread1 -pthread
$ ./thread1
```

Pthreads

Example 2

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NUMBER_OF_THREADS 5

void *hello(void *tid)
{
  printf ("Hello from thread %d\n", *(int*)tid);
  pthread_exit(NULL);
}

int main(void)
{
  pthread_t t[NUMBER_OF_THREADS];
  int status, i;


  for (i=0; i<NUMBER_OF_THREADS; i++){
        printf("Main: creating thread %d ...", i);

    if( (status = pthread_create(&t[i], NULL, hello, (void *)&i)) ){
          perror("pthread_create");
      exit(-1);
    }
    puts("done.");
  }

  for (i=0; i<NUMBER_OF_THREADS; i++){
    printf("Joining thread %d ...",i);
```

[3] http://www.ibm.com/developerworks/linux/library/l-posix1/index.html
[4] http://stackoverflow.com/questions/397075/what-is-the-difference-between-exit-and-abort

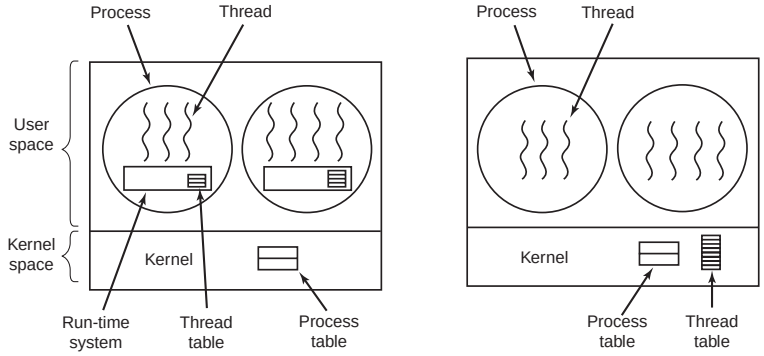
```
32
33      if( pthread_join(t[i], NULL) ){
34        perror("pthread_join");
35        abort() ;
36      }
37      puts("done.");
38    }
39    exit(0);
40  }
41
42  /* Local Variables: */
43  /* compile-command: "gcc -Wall -Wextra thread3.c -o /tmp/a.out -pthread" */
44  /* End: */
```

Pthreads

With or without `pthread_join()`? Check it by yourself.

8.5 User-Level Threads vs. Kernel-level Threads

User-Level Threads vs. Kernel-Level Threads



User-Level Threads

User-level threads provide a library of functions to allow user processes to create and manage their own threads.

- ☺ No need to modify the OS
- ☺ Simple representation
  - each thread is represented simply by a PC, regs, stack, and a small TCB, all stored in the user process' address space
- ☺ Simple Management
  - creating a new thread, switching between threads, and synchronization between threads can all be done without intervention of the kernel
- ☺ Fast
  - thread switching is not much more expensive than a procedure call
- ☺ Flexible
  - CPU scheduling (among threads) can be customized to suit the needs of the algorithm – each process can use a different thread scheduling algorithm

User-Level Threads

- ☹ Lack of coordination between threads and OS kernel
  - Process as a whole gets one time slice
  - Same time slice, whether process has 1 thread or 1000 threads
  - Also – up to each thread to relinquish control to other threads in that process

☹ Requires non-blocking system calls (i.e. a multithreaded kernel)
  – Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the process
  – part of motivation for user-level threads was not to have to modify the OS
☹ If one thread causes a page fault(interrupt!), the entire process blocks

See also: *More about blocking and non-blocking calls*[5].

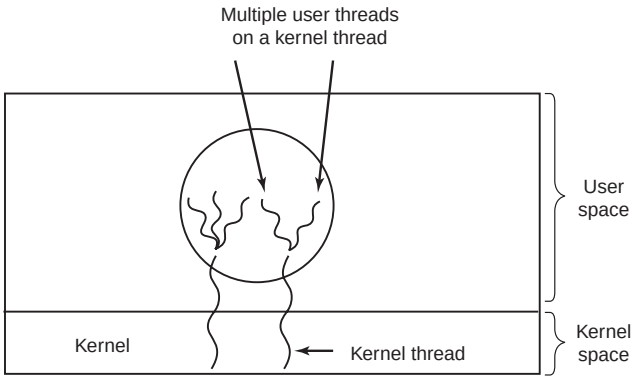
Kernel-Level Threads

Kernel-level threads kernel provides system calls to create and manage threads

☺ Kernel has full knowledge of all threads
  – Scheduler may choose to give a process with 10 threads more time than process with only 1 thread
☺ Good for applications that frequently block (e.g. server processes with frequent interprocess communication)
☹ Slow – thread operations are 100s of times slower than for user-level threads
☹ Significant overhead and increased kernel complexity – kernel must manage and schedule threads as well as processes
  – Requires a full thread control block (TCB) for each thread

Hybrid Implementations

Combine the advantages of two



Programming Complications

- `fork()`: shall the child has the threads that its parent has?
- What happens if one thread closes a file while another is still reading from it?
- What happens if several threads notice that there is too little memory?

And sometimes, threads fix the symptom, but not the problem.

8.6 Linux Threads

Linux Threads

To the Linux kernel, there is no concept of a thread

- Linux implements all threads as standard processes
- To Linux, a thread is merely a process that shares certain resources with other processes
- Some OS (MS Windows, Sun Solaris) have cheap threads and expensive processes.
- Linux processes are already quite lightweight

On a 75MHz Pentium: thread: $1.7\mu s$, fork: $1.8\mu s$

[*Understanding The Linux Kernel*, Sec. 3.1, *Processes, Lightweight Processes, and Threads*] Older versions of the Linux kernel offered no support for multithreaded applications. From the kernel point of view, a multithreaded application was just a normal process. The multiple execution flows of a multithreaded application were created, handled, and scheduled entirely in User Mode, usually by means of a POSIX-compliant *pthread* library.

However, such an implementation of multithreaded applications is not very satisfactory. For instance, suppose a chess program uses two threads: one of them controls the graphical chessboard, waiting for the moves of the human

[5] http://www.daniweb.com/software-development/computer-science/threads/384575/synchronous-vs-asynchronous-blocking-vs-non-blocking

player and showing the moves of the computer, while the other thread ponders the next move of the game. While the first thread waits for the human move, the second thread should run continuously, thus exploiting the thinking time of the human player. However, if the chess program is just a single process, the first thread cannot simply issue a blocking system call waiting for a user action; otherwise, the second thread is blocked as well. Instead, the first thread must employ sophisticated nonblocking techniques to ensure that the process remains runnable.

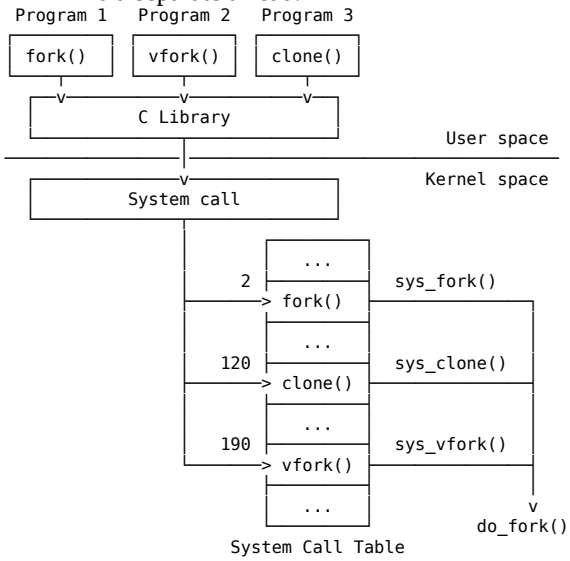
Linux uses *lightweight processes* to offer better support for multithreaded applications. Basically, two lightweight processes may share some resources, like the address space, the open files, and so on. Whenever one of them modifies a shared resource, the other immediately sees the change. Of course, the two processes must synchronize themselves when accessing the shared resource.

A straightforward way to implement multithreaded applications is to associate a light-weight process with each thread. In this way, the threads can access the same set of application data structures by simply sharing the same memory address space, the same set of open files, and so on; at the same time, each thread can be scheduled independently by the kernel so that one may sleep while another remains runnable. Examples of POSIX-compliant pthread libraries that use Linux's lightweight processes are *LinuxThreads*, *Native POSIX Thread Library (NPTL)*, and IBM's *Next Generation Posix Threading Package (NGPT)*.

POSIX-compliant multithreaded applications are best handled by kernels that support "thread groups". In Linux a *thread group* is basically a set of lightweight processes that implement a multithreaded application and act as a whole with regards to some system calls such as `getpid()`, `kill()`, and `_exit()`.

Linux Threads

`clone()` creates a separate process that shares the address space of the calling process. The cloned task behaves *much like* a separate thread.



The `clone()` System Call

```
#include <sched.h>
int clone(int (*fn) (void *), void *child_stack,
          int flags, void *arg, ...);
```

arg 1 the function to be executed, i.e. `fn(arg)`, which returns an `int`;
arg 2 a pointer ☛ a (usually malloced) memory space to be used as the stack for the new thread;
arg 3 a set of flags used to indicate how much the calling process is to be shared. In fact,
`clone(0) == fork()`
arg 4 the arguments passed to the function.
It returns the PID of the child process or -1 on failure.

```
$ man clone
```

Some flags:

flag	Shared
CLONE_FS	File-system info
CLONE_VM	Same memory space
CLONE_SIGHAND	Signal handlers
CLONE_FILES	The set of open files

In practice, one should try to avoid calling `clone()` directly

Instead, use a threading library (such as pthreads) which use `clone()` when starting a thread (such as during a call to `pthread_create()`)

`clone()` Example

```c
/* http://stackoverflow.com/questions/5255320/reason-for-segmentation-fault */
#define _GNU_SOURCE
#include <unistd.h>
#include <sched.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>

int variable;

int do_something() {
   variable = 42;
   _exit(0);
}

int main(void) {
   void *child_stack;

   variable = 9;
   child_stack = (void *) malloc(8192); // WRONG!
   // child_stack = (void **) malloc(8192) + 8192 / sizeof(*child_stack); // Right!
   printf("The variable was %d\n", variable);

   clone(do_something, child_stack, CLONE_FS | CLONE_VM | CLONE_FILES, NULL);
   sleep(1);

   printf("The variable is now %d\n", variable);
   return 0;
}

/* Local Variables: */
/* compile-command: "gcc -Wall clone.c -o /tmp/a.out" */
/* End: */
```


[1] Wikipedia. *Process (computing) — Wikipedia, The Free Encyclopedia*. 2014. http://en.wikipedia.org/w/index.php?title=Process%5C_(computing)&oldid=639847817.

[2] Wikipedia. *Thread (computing) — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Thread%5C_(computing)&oldid=648172980.

[3] Wikipedia. *Process control block — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Process%5C_control%5C_block&oldid=646933587.

9 Process Synchronization

9.1 IPC

Interprocess Communication

Example:

```
$ unicode skull | head -1 | cut -f1 -d' ' | sm -
```

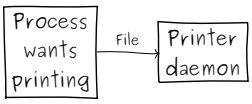
IPC issues:

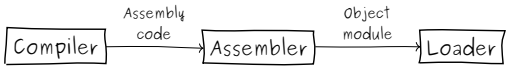
1. How one process can pass information to another
2. Be sure processes do not get into each other's way
   e.g. in an airline reservation system, two processes compete for the last seat
3. Proper sequencing when dependencies are present
   e.g. if A produces data and B prints them, B has to wait until A has produced some data

Two models of IPC:

- Shared memory
- Message passing (e.g. sockets)

Producer-Consumer Problem





9.2 Shared Memory

Process Synchronization

Producer-Consumer Problem

- Consumers don't try to remove objects from Buffer when it is empty.
- Producers don't try to add objects to the Buffer when it is full.

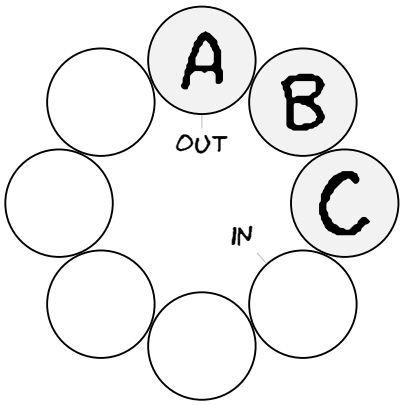
```
while(TRUE){
    while(FULL);
    item = produceItem();
    insertItem(item);
}
```

```
while(TRUE){
    while(EMPTY);
    item = removeItem();
    consumeItem(item);
}
```

How to define *full/empty*?

Producer-Consumer Problem

— Bounded-Buffer Problem (Circular Array)



Front(out): the first full position
Rear(in): the next free position

Full or empty when "$front == rear$"?

Producer-Consumer Problem

Common solution:
Full: when "$(\texttt{in}+1)\%\texttt{BUFFER\_SIZE} == \texttt{out}$"
Actually, this is "$\texttt{full}-1$"
Empty: when "$\texttt{in} == \texttt{out}$"
Can only use "$\texttt{BUFFER\_SIZE}-1$" elements

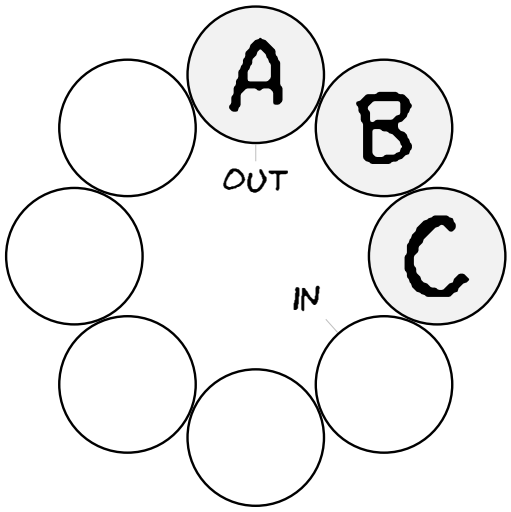
Shared data:

```
#define BUFFER_SIZE 6
typedef struct {
        /* ... */
} item;
item buffer[BUFFER_SIZE];
int in = 0; //the next free position
int out = 0;//the first full position
```

Bounded-Buffer Problem

Producer:

```
while (true) {
  /* do nothing -- no free buffers */
  while (((in + 1) % BUFFER_SIZE) == out);

  produce(buffer[in]);

  in = (in + 1) % BUFFER_SIZE;
}
```



Consumer:

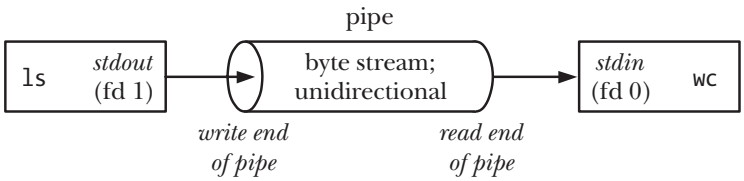
```
while (true) {
  while (in == out); // do nothing

  consume(buffer[out]);

  out = (out + 1) % BUFFER_SIZE;
}
```

9.2.1 Pipes and FIFOs

- [*The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, chap. 44]

Pipe

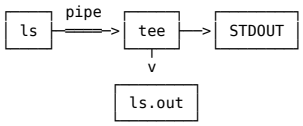
```
$ ls | wc -l
```



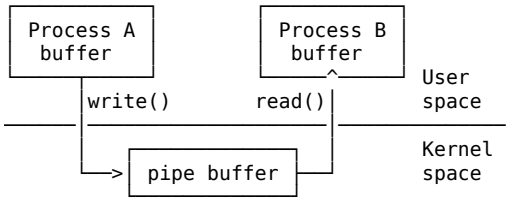
- A pipe is a byte stream
- Unidirectional
- `read()` would be blocked if nothing written at the other end

tee

```
$ ls | tee ls.out
```



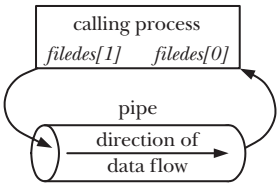
- When we say that a pipe is a byte stream, we mean that there is no concept of messages or message boundaries when using a pipe. The process reading from a pipe can read blocks of data of any size, regardless of the size of blocks written by the writing process. Furthermore, the data passes through the pipe sequentially — bytes are read from a pipe in exactly the order they were written. It is not possible to randomly access the data in a pipe using `lseek()`. [*The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, chap. 44]



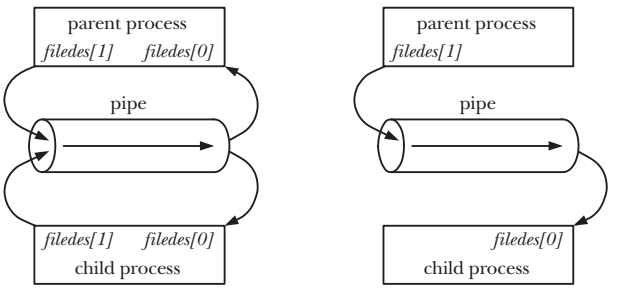
- No direct link between A and B (need system calls)
- A pipe is simply a buffer maintained in kernel memory
  `$ cat /proc/sys/fs/pipe-max-size`

pipe()

```
1 #include <unistd.h>
2
3 int pipe(int fd[2]);
```



pipe() + fork()



a) After *fork()* **b) After closing unused descriptors**

- Pipes must have a reader and a writer. If a process tries to write to a pipe that has no reader, it will be sent the SIGPIPE signal from the kernel. This is imperative when more than two processes are involved in a pipeline. (http://www.tldp.org/LDP/lpg/node20.html)
- While it is possible for the parent and child to both read from and write to the pipe, this is not usual. Therefore, immediately after the `fork()`, one process closes its descriptor for the write end of the pipe, and the other closes its descriptor for the read end. For example, if the parent is to send data to the child, then it would close its read descriptor for the pipe, `filedes[0]`, while the child would close its write descriptor for the pipe, `filedes[1]`. One reason that it is not usual to have both the parent and child reading from a single pipe is that if two processes try to simultaneously read from a pipe, we can't be sure which process will be the first to succeed — the two processes race for data. Preventing such races would require the use of some synchronization mechanism. However,

if we require bidirectional communication, there is a simpler way: just create two pipes, one for sending data in each direction between the two processes. (If employing this technique, then we need to be wary of deadlocks that may occur if both processes block while trying to read from empty pipes or while trying to write to pipes that are already full.) [*The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, chap. 44, p. 893]

- Pipes can be used for communication between any two (or more) related processes, as long as the pipe was created by a common ancestor before the series of `fork()` calls that led to the existence of the processes. For example, a pipe could be used for communication between a process and its grandchild. The first process creates the pipe, and then forks a child that in turn forks to yield the grandchild. A common scenario is that a pipe is used for communication between two siblings — their parent creates the pipe, and then creates the two children. This is what the shell does when building a pipeline.
- **Closing unused pipe file descriptors**. The process reading from the pipe closes its write descriptor for the pipe, so that, when the other process completes its output and closes its write descriptor, the reader sees end-of-file (once it has read any outstanding data in the pipe). If the reading process doesn't close the write end of the pipe, then, after the other process closes its write descriptor, the reader won't see end-of-file, even after it has read all data from the pipe. Instead, a `read()` would block waiting for data, because the kernel knows that there is still at least one write descriptor open for the pipe. That this descriptor is held open by the reading process itself is irrelevant; in theory, that process could still write to the pipe, even if it is blocked trying to read. For example, the `read()` might be interrupted by a signal handler that writes data to the pipe.
  The writing process closes its read descriptor for the pipe for a different reason. When a process tries to write to a pipe for which no process has an open read descriptor, the kernel sends the SIGPIPE signal to the writing process. By default, this signal kills a process. A process can instead arrange to catch or ignore this signal, in which case the `write()` on the pipe fails with the error EPIPE (broken pipe). Receiving the SIGPIPE signal or getting the EPIPE error is a useful indication about the status of the pipe, and this is why unused read descriptors for the pipe should be closed.
  If the writing process doesn't close the read end of the pipe, then, even after the other process closes the read end of the pipe, the writing process will still be able to write to the pipe. Eventually, the writing process will fill the pipe, and a further attempt to write will block indefinitely.
  One final reason for closing unused file descriptors is that it is only after all file descriptors in all processes that refer to a pipe are closed that the pipe is destroyed and its resources released for reuse by other processes. At this point, any unread data in the pipe is lost.

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

#define BUF_SIZE 10

int main(int argc, char *argv[]) /* Over-simplified! */
{
  int pfd[2]; /* Pipe file descriptors */
  char buf[BUF_SIZE];
  ssize_t numRead;

  pipe(pfd); /* Create the pipe */

  switch (fork()) {
  case 0: /* Child - reads from pipe */
    close(pfd[1]); /* Write end is unused */

    for(;;) { /* Read data from pipe, echo on stdout */
      if( (numRead = read(pfd[0], buf, BUF_SIZE)) == 0 )
        break;  /* End-of-file */
      if( write(1, buf, numRead) != numRead ){
        perror("child - partial/failed write");
                exit(EXIT_FAILURE);
          }
```

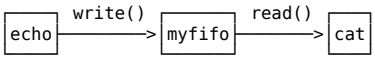
```
    }
    puts(""); /* newline */

    close(pfd[0]); _exit(EXIT_SUCCESS);

  default: /* Parent - writes to pipe */
    close(pfd[0]); /* Read end is unused */

    if( (size_t)write(pfd[1], argv[1], strlen(argv[1])) != strlen(argv[1]) ){
      perror("parent - partial/failed write");
          exit(EXIT_FAILURE);
        }

    close(pfd[1]); /* Child will see EOF */

    wait(NULL); /* Wait for child to finish */
    exit(EXIT_SUCCESS);
  }
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra simple_pipe.c -o simple-pipe" */
/* End: */
```

- https://stackoverflow.com/questions/5422831/what-is-the-difference-between-using-exit-exit-in-a-conventional-linux-fo
- _exit(2)

Named Pipe (FIFO)
PIPEs pass data between related processes.
FIFOs pass data between any processes.

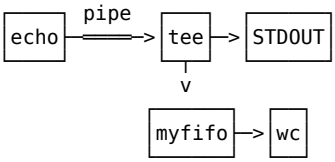
```
$ mkfifo myfifo
```

```
$ echo hello > myfifo
$ cat myfifo
```



tee

```
$ echo hello | tee myfifo
$ wc myfifo
```



- https://en.wikipedia.org/wiki/Named_pipe

IPC With FIFO

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/myfifo"

int main(int argc, char *argv[]) /* Oversimplified */
{
  int fd, i, mode = 0;
```

```
  char c;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <O_RDONLY | O_WRONLY | O_NONBLOCK>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  for(i = 1; i < argc; i++) {
    if (strncmp(*++argv, "O_RDONLY", 8) == 0)  mode |= O_RDONLY;
    if (strncmp(*argv, "O_WRONLY", 8) == 0)    mode |= O_WRONLY;
    if (strncmp(*argv, "O_NONBLOCK", 10) == 0) mode |= O_NONBLOCK;
  }

  if (access(FIFO_NAME, F_OK) == -1) mkfifo(FIFO_NAME, 0777);

  printf("Process %d: FIFO(fd %d, mode %d) opened.\n",
         getpid(), fd = open(FIFO_NAME, mode), mode);

  if( (mode == 0) | (mode == 2048) )
      while( read(fd,&c,1) == 1 ) putchar(c);

  if( (mode == 1) | (mode == 2049) )
      while( (c = getchar()) != EOF ) write(fd,&c,1);

  exit(EXIT_SUCCESS);
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra fifo2.c -o fifo2" */
/* End: */
```

```
$ watch 'lsof -n.1 /tmp/myfifo'
$ ./a.out O_RDONLY
$ ./a.out O_WRONLY
$ ./a.out O_RDONLY O_NONBLOCK
$ ./a.out O_WRONLY O_NONBLOCK
```

`O_NONBLOCK`

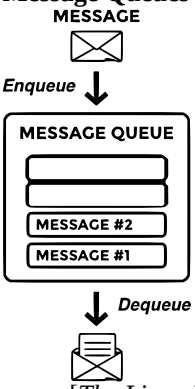
- A `read()`/`write()` will wait on an empty blocking FIFO
- A `read()` on an empty nonblocking FIFO will return 0 bytes
- `open(const char *path, O_WRONLY | O_NONBLOCK);`
  - Returns an error (-1) if FIFO not open
  - Okay if someone's reading the FIFO
- If opened with `O_RDWR`, the result is undefined
- If opened with `O_RDWR`, the result is undefined. If you do want to pass data in both directions, it's much better to use a pair of FIFOs or pipes, one for each direction.
- There are four legal combinations of `O_RDONLY`, `O_WRONLY`, and the `O_NONBLOCK` flag.

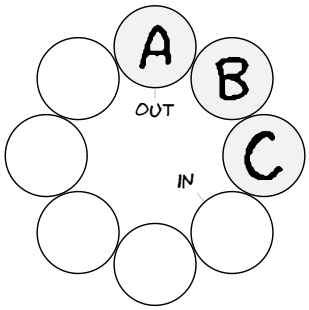
```
open(const char *path, O_RDONLY);
/* In this case, the open call will block; it will not return until a process opens the
   same FIFO for writing. */

open(const char *path, O_RDONLY | O_NONBLOCK);
/* The open call will now succeed and return immediately, even if the FIFO has not been
   opened for writing by any process. */

open(const char *path, O_WRONLY);
/* In this case, the open call will block until a process opens the same FIFO for
   reading. */
```

```
12
13 open(const char *path, O_WRONLY | O_NONBLOCK);
14 /* This will always return immediately, but if no process has the FIFO open for reading,
15    open will return an error, - 1, and the FIFO won't be opened. If a process does have the
16    FIFO open for reading, the file descriptor returned can be used for writing to the
17    FIFO. */
```

9.2.2 Message Queues

Message Queues





- [*The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, Sec. 52.3]
- `mq_overview(7)`
- `sem_overview(7)`
- `shm_overview(7)`
- https://www.uninformativ.de/blog/postings/2016-05-16/0/POSTING-en.html

Message Queues

Send

```c
#include <fcntl.h>
#include <limits.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

int main(int argc, char **argv)
{
  mqd_t queue;
  struct mq_attr attrs;
  size_t msg_len;

  if (argc < 3){
    fprintf(stderr, "Usage: %s <queuename> <message>\n", argv[0]);
    return 1;
  }

  queue = mq_open(argv[1], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR, NULL);
  if (queue == (mqd_t)-1){
        perror("mq_open");
        return 1;
  }

  if (mq_getattr(queue, &attrs) == -1){
    perror("mq_getattr");
    mq_close(queue);
    return 1;
```

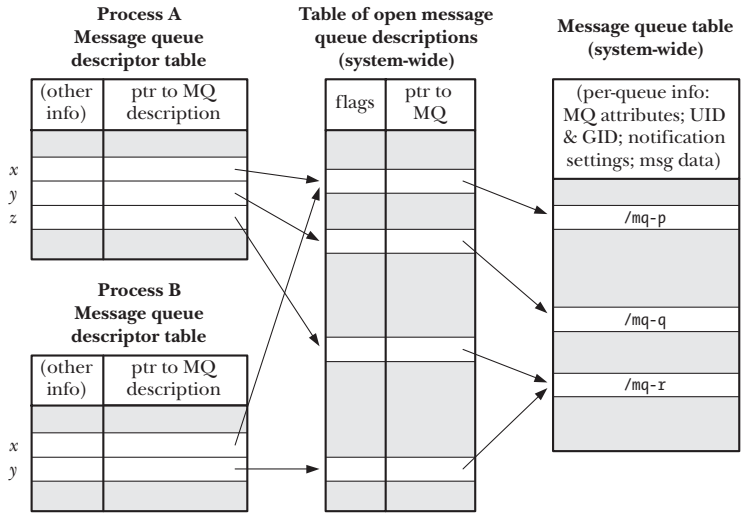
```
  }

  msg_len = strlen(argv[2]);
  if (msg_len > LONG_MAX || (long)msg_len > attrs.mq_msgsize){
    fprintf(stderr, "Your message is too long for the queue.\n");
    mq_close(queue);
    return 1;
  }

  if (mq_send(queue, argv[2], strlen(argv[2]), 0) == -1){
    perror("mq_send");
    mq_close(queue);
    return 1;
  }

  return 0;
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra mq-send.c -o mq-send -lrt" */
/* End: */
```

Message Queues
Receive

```
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main(int argc, char **argv)
{
  mqd_t queue;
  struct mq_attr attrs;
  char *msg_ptr;
  ssize_t recvd;
  size_t i;

  if (argc < 2){
    fprintf(stderr, "Usage: %s <queuename>\n", argv[0]);
    return 1;
  }

  queue = mq_open(argv[1], O_RDONLY | O_CREAT, S_IRUSR | S_IWUSR, NULL);
  if (queue == (mqd_t)-1){
    perror("mq_open");
    return 1;
  }

  if (mq_getattr(queue, &attrs) == -1){
    perror("mq_getattr");
    mq_close(queue);
    return 1;
  }

  msg_ptr = calloc(1, attrs.mq_msgsize);
  if (msg_ptr == NULL){
    perror("calloc for msg_ptr");
    mq_close(queue);
    return 1;
```

```
  }

  recvd = mq_receive(queue, msg_ptr, attrs.mq_msgsize, NULL);
  if (recvd == -1){
    perror("mq_receive");
    return 1;
  }

  printf("Message: ");
  for (i = 0; i < (size_t)recvd; i++)
    putchar(msg_ptr[i]);
  puts("");
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra mq-recv.c -o mq-recv -lrt" */
/* End: */
```

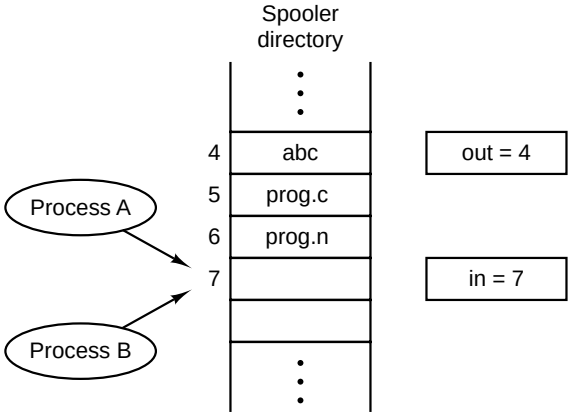
Relationship Between Kernel Data Structures



9.3 Race Condition and Mutual Exclusion

Race Conditions

Now, let's have two producers



Race Conditions

Two producers

```
#define BUFFER_SIZE 100
typedef struct {
        /* ... */
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Process A and B do the same thing:

```
while (true) {
  while (((in + 1) % BUFFER_SIZE) == out);
  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
}
```

Race Conditions

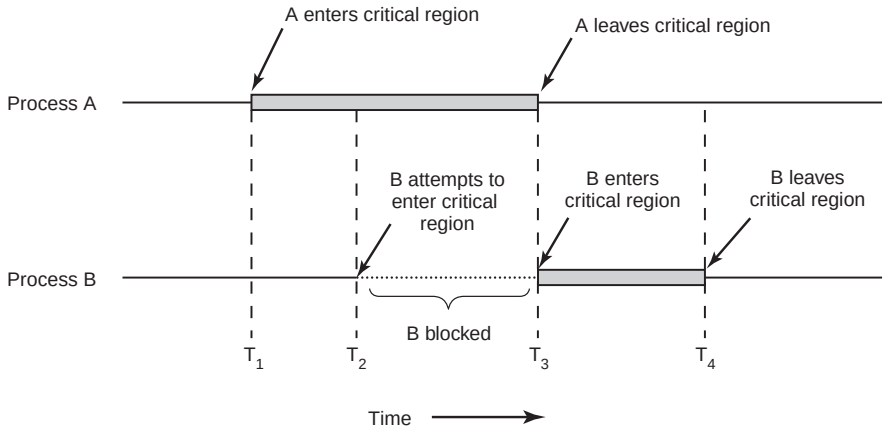
Problem: Process `B` started using one of the *shared variables* before Process `A` was finished with it.

Solution: *Mutual exclusion.* If one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

Critical Regions

Mutual Exclusion

Critical Region: is a piece of code accessing a common resource.



Critical Region

A solution to the critical region problem must satisfy three conditions:

Mutual Exclusion: No two process may be simultaneously inside their critical regions.

Progress: No process running outside its critical region may block other processes.

Bounded Waiting: No process should have to wait forever to enter its critical region.

Mutual Exclusion With Busy Waiting

Disabling Interrupts

```
{
  ...
  disableINT();
  critical_code();
  enableINT();
  ...
}
```

Problems:

- It's not wise to give user process the power of turning off INTs.
  - Suppose one did it, and never turned them on again
- useless for multiprocessor system

Disabling INTs is often a useful technique within the kernel itself but is not a general mutual exclusion mechanism for user processes.

Mutual Exclusion With Busy Waiting

Lock Variables

```
1 int lock=0; //shared variable
2 {
3   ...
4   while(lock); //busy waiting
5   lock=1;
6   critical_code();
7   lock=0;
8   ...
9 }
```

Problem:

- What if an interrupt occurs right at line 5?
- Checking the lock again while backing from an interrupt?

Mutual Exclusion With Busy Waiting

Strict Alternation

```
1 while(TRUE){
2   while(turn != 0);
3   critical_region();
4   turn = 1;
5   noncritical_region();
6 }
```

```
1 while(TRUE){
2   while(turn != 1);
3   critical_region();
4   turn = 0;
5   noncritical_region();
6 }
```

Problem: violates condition-2

- One process can be blocked by another not in its critical region.
- Requires the two processes strictly alternate in entering their critical region.

Mutual Exclusion With Busy Waiting

Peterson's Solution

```
1 int interest[0] = 0;
2 int interest[1] = 0;
3 int turn;
```

P0

```
1 interest[0] = 1;
2 turn = 1;
3 while(interest[1] == 1
4           && turn == 1);
5 critical_section();
6 interest[0] = 0;
```

P1

```
1 interest[1] = 1;
2 turn = 0;
3 while(interest[0] == 1
4           && turn == 0);
5 critical_section();
6 interest[1] = 0;
```

[1] Wikipedia. *Peterson's algorithm — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Peterson%27s%5C_algorithm&oldid=646078826.

Mutual Exclusion With Busy Waiting

Lock file

```
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <fcntl.h>
5  #include <errno.h>
6
7  const char *mylock = "/tmp/LCK.test2";
8
9  int main() {
10   int fd;
```

```
11
12    for(;;){
13      while( (fd = open(mylock, O_RDWR | O_CREAT | O_EXCL, 0444)) != -1 ){
14        printf("Process(%d) - Working in critical region...\n", getpid());
15        sleep(2);           /* working */
16  //      close(fd);
17        if ( unlink(mylock) == 0 ) puts("Done.\nResource unlocked.");
18        sleep(3);           /* non-critical region */
19      }
20      printf("Process(%d) - Waiting for lock...\n", getpid());
21    }
22    exit(EXIT_SUCCESS);
23  }
24
25  /* Local Variables: */
26  /* compile-command: "gcc -Wall -Wextra lock2.c -o /tmp/a.out" */
27  /* End: */
```

☹ Lock file could be left in system after Ctrl+c

```
1   #include <unistd.h>
2   #include <stdlib.h>
3   #include <stdio.h>
4   #include <fcntl.h>
5   #include <errno.h>
6   #include <signal.h>
7
8   const char *mylock = "/tmp/LCK.test2";
9
10  void sigint(int signo){
11    if ( unlink(mylock) == 0 ) puts("Quit. Lock released.");
12    exit(EXIT_SUCCESS);
13  }
14
15  int main() {
16    int fd;
17
18    signal(SIGINT,sigint);
19
20    for(;;){
21      while( (fd = open(mylock, O_RDWR | O_CREAT | O_EXCL, 0444)) != -1 ) {
22        printf("Process(%d) - Working in critical region...\n", getpid());
23        sleep(2);           /* working */
24        close(fd);
25        if ( unlink(mylock) == 0 ) puts("Done.\nResource unlocked.");
26        sleep(3);               /* non-critical region */
27      }
28      printf("Process(%d) - Waiting for lock...\n", getpid());
29    }
30
31    exit(EXIT_SUCCESS);
32  }
33
34  /* Local Variables: */
35  /* compile-command: "gcc -Wall -Wextra lock2-sigint.c -o /tmp/a.out" */
36  /* End: */
```

☹ `sigint()` is too crude to be reliable. Need a more sophisticated design.

Mutual Exclusion With Busy Waiting
Hardware Solution: The TSL Instruction

Lock the memory bus

```
enter_region:
    TSL REGISTER,LOCK          | copy lock to register and set lock to 1
    CMP REGISTER,#0            | was lock zero?
    JNE enter_region           | if it was non zero, lock was set, so loop
    RET                        | return to caller; critical region entered


leave_region:
    MOVE LOCK,#0               | store a 0 in lock
    RET                        | return to caller
```

See also: [*Modern Operating Systems*, Sec. 2.3.3, *Mutual Exclusion With Busy Waiting*, p. 124].

Mutual Exclusion Without Busy Waiting
Sleep & Wakeup

```c
#define N 100  /* number of slots in the buffer */
int count = 0; /* number of items in the buffer */
```

```c
void producer(){
  int item;
  while(TRUE){
    item = produce_item();
    if(count == N)
      sleep();
    insert_item(item);
    count++;
    if(count == 1)
      wakeup(consumer);
  }
}
```

```c
void consumer(){
  int item;
  while(TRUE){
    if(count == 0)
      sleep();
    item = rm_item();
    count--;
    if(count == N - 1)
      wakeup(producer);
    consume_item(item);
  }
}
```


Producer-Consumer Problem
Race Condition

Problem

1. Consumer is going to sleep upon seeing an empty buffer, but INT occurs;
2. Producer inserts an item, increasing `count` to `1`, then call `wakeup(consumer)`;
3. But the consumer is not asleep, though `count` was `0`. So *the `wakeup()` signal is lost*;
4. Consumer is back from INT remembering `count` is `0`, and goes to sleep;
5. Producer sooner or later will fill up the buffer and also goes to sleep;
6. Both will sleep forever, and waiting to be waken up by the other process. Deadlock!

Producer-Consumer Problem
Race Condition

Solution: Add a *wakeup waiting bit*

1. The bit is set, when a wakeup is sent to an awaken process;
2. Later, when the process wants to sleep, it checks the bit first. Turns it off if it's set, and stays awake.

What if many processes try going to sleep?

9.4 Semaphores

What is a Semaphore?

- A locking mechanism
- An integer or ADT

Atomic Operations	
P()	V()
Wait()	Signal()
Down()	Up()
Decrement()	Increment()
...	...

```
down(S){
  while(S<=0);
  S--;
}
```

```
up(S){
  S++;
}
```

More meaningful names:

- `increment_and_wake_a_waiting_process_if_any()`
- `decrement_and_block_if_the_result_is_negative()`

Semaphore

How to ensure atomic?

1. For single CPU, implement `up()` and `down()` as system calls, with the OS disabling all interrupts while accessing the semaphore;
2. For multiple CPUs, to make sure only one CPU at a time examines the semaphore, a lock variable should be used with the TSL instructions.

Semaphore is a Special Integer

A semaphore is like an integer, with three differences:

1. You can initialize its value to any integer, but after that the only operations you are allowed to perform are *increment* (`S++`) and *decrement* (`S--`).
2. When a thread decrements the semaphore, if the result is negative ($S \leq 0$), the thread blocks itself and cannot continue until another thread increments the semaphore.
3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

Why Semaphores?

We don't need semaphores to solve synchronization problems, but there are some advantages to using them:

- Semaphores impose deliberate constraints that help programmers avoid errors.
- Solutions using semaphores are often clean and organized, making it easy to demonstrate their correctness.
- Semaphores can be implemented efficiently on many systems, so solutions that use semaphores are portable and usually efficient.

The Simplest Use of Semaphore

Signaling

- One thread sends a signal to another thread to indicate that something has happened
- it solves the serialization problem

  Signaling makes it possible to guarantee that a section of code in one thread will run before a section of code in another thread

```
statement a1
sem.signal()
```

```
sem.wait()
statement b1
```

What's the initial value of `sem`?

Semaphore

Rendezvous Puzzle

```
statement a1
statement a2
```

```
statement b1
statement b2
```

Q: How to guarantee that

1. a1 happens before b2, and
2. b1 happens before a2

a1 → b2; b1 → a2

Hint: Use two semaphores initialized to 0.

	Thread A:	Thread B:

Solution 1:	`statement a1`
	`sem1.wait()`
	`sem2.signal()`
	`statement a2`
		`statement b1`
		`sem1.signal()`
		`sem2.wait()`
		`statement b2`

Solution 2:	`statement a1`
	`sem2.signal()`
	`sem1.wait()`
	`statement a2`
		`statement b1`
		`sem1.signal()`
		`sem2.wait()`
		`statement b2`

Solution 3:	`statement a1`
	`sem2.wait()`
	`sem1.signal()`
	`statement a2`
		`statement b1`
		`sem1.wait()`
		`sem2.signal()`
		`statement b2`

Solution 3 has deadlock!

Example: Signaling With Semaphore

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *func(void *arg);
sem_t sem;
#define BUFSIZE 1024
char buf[BUFSIZE];                                  /* critical region */

int main() {
  pthread_t t;

  if( sem_init(&sem, 0, 0) != 0 ) {
    perror("Semaphore initialization failed");
    exit(EXIT_FAILURE);
  }

  if( pthread_create(&t, NULL, func, NULL) != 0 ) {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);
  }

  puts("Please input some text. Ctrl-d to quit.");

  while( fgets(buf, BUFSIZE, stdin) )
        sem_post(&sem);

  sem_post(&sem);       /* in case of Ctrl-d */

  if( pthread_join(t, NULL) != 0) {
```

```
    perror("Thread join failed");
    exit(EXIT_FAILURE);
  }

  sem_destroy(&sem);  exit(EXIT_SUCCESS);
}

void *func(void *arg) {
  sem_wait(&sem);
  while( buf[0] != '\0' ) {
    printf("You input %ld characters\n", strlen(buf)-1);
        buf[0] = '\0';           /* in case of Ctrl-d */
    sem_wait(&sem);
  }
  pthread_exit(NULL);
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra thread-semaphore.c -o /tmp/a.out -pthread" */
/* End: */
```

- [*Beginning linux programming*, Sec. 12.5]
- https://stackoverflow.com/questions/368322/differences-between-system-v-and-posix-semaphores

Mutex

- A second common use for semaphores is to enforce mutual exclusion
- It guarantees that only one thread accesses the shared variable at a time
- A `mutex` is like a token that passes from one thread to another, allowing one thread at a time to proceed

Q: Add semaphores to the following example to enforce mutual exclusion to the shared variable `i`.

Thread A: `i++` **Thread B:** `i++`

Why? Because `i++` is not atomic.

`i++` can go wrong!

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

static int glob = 0;

static void *threadFunc(void *arg) /* loop 'arg' times */
{
  int j;
  for (j = 0; j < *((int *) arg); j++) glob++; /* not atomic! */
  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t t1, t2;
  int loops;

  loops = (argc > 1) ? atoi(argv[1]) : 10000000;

  if( pthread_create(&t1, NULL, threadFunc, &loops) != 0 ){
          perror("pthread_create 1");
          exit(EXIT_FAILURE);
  }

```

```
  if( pthread_create(&t2, NULL, threadFunc, &loops) != 0 ){
          perror("pthread_create 2");
          exit(EXIT_FAILURE);
  }

  if( pthread_join(t1, NULL) != 0 ){
          perror("pthread_join 1");
          exit(EXIT_FAILURE);
  }

  if( pthread_join(t2, NULL) != 0 ){
          perror("pthread_join 2");
          exit(EXIT_FAILURE);
  }

  printf("glob = %d\n", glob);
  exit(EXIT_SUCCESS);
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra atomic-non.c -o /tmp/a.out -pthread" */
/* End: */
```

`i++` is not atomic in assembly language

```
LOAD    [i], r0   ;load the value of 'i' into
                  ;a register from memory
ADD     r0, 1     ;increment the value
                  ;in the register
STORE   r0, [i]   ;write the updated
                  ;value back to memory
```

Interrupts might occur in between. So, `i++` needs to be protected with a `mutex`.

Mutex Solution

Create a semaphore named `mutex` that is initialized to `1`

1: a thread may proceed and access the shared variable
0: it has to wait for another thread to release the mutex

```
mutex.wait()
    i++
mutex.signal()
```

```
mutex.wait()
    i++
mutex.signal()
```

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void *threadFunc(void *arg)
{
  int j;
  for (j = 0; j < *((int *) arg); j++) {
    if ( pthread_mutex_lock(&mtx) != 0 ){
        perror("pthread_mutex_lock");
                exit(EXIT_FAILURE);
    }
```

```
    glob++;
    if ( pthread_mutex_unlock(&mtx) != 0){
        perror("pthread_mutex_unlock");
                exit(EXIT_FAILURE);
    }
  }
  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t t1, t2;
  int loops;

  loops = (argc > 1) ? atoi(argv[1]) : 10000000;

  if( pthread_create(&t1, NULL, threadFunc, &loops) != 0 ){
      perror("pthread_create");
          exit(EXIT_FAILURE);
  }

  if( pthread_create(&t2, NULL, threadFunc, &loops) != 0 ){
      perror("pthread_create");
          exit(EXIT_FAILURE);
  }

  if( pthread_join(t1, NULL) != 0 ){
      perror("pthread_join");
          exit(EXIT_FAILURE);
  }

  if( pthread_join(t2, NULL) != 0 ){
      perror("pthread_join");
          exit(EXIT_FAILURE);
  }

  printf("glob = %d\n", glob);
  exit(EXIT_SUCCESS);
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra atomic-mutex.c -o /tmp/a.out -pthread" */
/* End: */
```

```
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

static int glob = 0;
static sem_t sem;

static void *threadFunc(void *arg)
{
  int j;
  for (j = 0; j < *((int *) arg); j++) {
    if (sem_wait(&sem) == -1){
                perror("sem_wait");
                exit(EXIT_FAILURE);
    }
    glob++;
    if (sem_post(&sem) == -1){
```

```
      perror("sem_post");
          exit(EXIT_FAILURE);
    }
  }
  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t t1, t2;
  int loops;

  loops = (argc > 1) ? atoi(argv[1]) : 10000000;

  if( sem_init(&sem, 0, 1) == -1 ){
      perror("sem_init");
          exit(EXIT_FAILURE);
  }

  if( pthread_create(&t1, NULL, threadFunc, &loops) != 0 ){
          perror("pthread_create");
          exit(EXIT_FAILURE);
  }

  if( pthread_create(&t2, NULL, threadFunc, &loops) != 0 ){
          perror("pthread_create");
          exit(EXIT_FAILURE);
  }

  if( pthread_join(t1, NULL) != 0 ){
     perror("pthread_join");
         exit(EXIT_FAILURE);
  }

  if( pthread_join(t2, NULL) != 0 ){
      perror("pthread_join");
          exit(EXIT_FAILURE);
  }

  printf("glob = %d\n", glob);
  exit(EXIT_SUCCESS);
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra mutex.c -o mutex -pthread" */
/* End: */
```

Multiplex — Without Busy Waiting

```c
typedef struct{
  int space;         //number of free resources
  struct process *P; //a list of queueing producers
  struct process *C; //a list of queueing consumers
} semaphore;
semaphore S;
S.space = 5;
```

```c
void down(S){
  S.space--;
  if(S.space == 4){
    rmFromQueue(S.C);
    wakeup(S.C);
  }
  if(S.space < 0){
    addToQueue(S.P);
    sleep();
  }
}
```

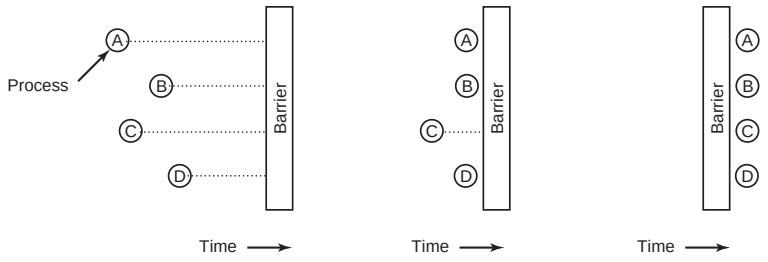
```c
void up(S){
  S.space++;
  if(S.space > 5){
    addToQueue(S.C);
    sleep();
  }
  if(S.space >= 0){
    rmFromQueue(S.P);
    wakeup(S.P);
  }
}
```

```
if S.space < 0,
   S.space == Number of queuing producers
if S.space > 5,
   S.space == Number of queuing consumers + 5
```

The work flow There are several processes running simultaneously. They all need to access some common resources.

1. Assuming `S.space == 3` in the beginning
2. Process P1 comes and take one resource away. `S.space == 2` now.
3. Process P2 comes and take the 2nd resource away. `S.space == 1` now.
4. Process P3 comes and take the last resource away. `S.space == 0` now.
5. Process P4 comes and sees nothing left. It has to sleep. `S.space == -1` now.
6. Process P5 comes and sees nothing left. It has to sleep. `S.space == -2` now.
7. At this moment, there are 2 processes (P4 and P5) sleeping. In another word, they are queuing for resources.
8. Now, P1 finishes using the resource, and released it. After it does a `S.space++`, it finds out that `S.space <= 0`. So it wakes up a Process (say P4) in the queue.
9. P4 wakes up, and back to execute the instruction right after `sleep()`.
10. P4 (or P2|P3) finishes using the resource, and releases it. After it does a `S.space++`, it finds out that `S.space <= 0`. So it wakes up P5 in the queue.
11. the queue is empty now.

Barrier



1. Processes approaching a barrier
2. All processes but one blocked at the barrier
3. When the last process arrives at the barrier, all of them are let through

Synchronization requirement:

```
specific_task()
critical_point()
```

No thread executes `critical_point()` until after all threads have executed `specific_task()`.

Barrier Solution

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

count: keeps track of how many threads have arrived
mutex: provides exclusive access to `count`
barrier: is locked (≤ 0) until all threads arrive
When `barrier.value<0`,
`barrier.value == Number of queueing processes`

```
#define CHAIRS 5
semaphore customers = 0;
semaphore bber = ?;
semaphore mutex = 1;
int waiting = 0;

void barber(void)
{
  while (TRUE) {
    wait(&customers);
    cutHair();
  }
}
```

```
void customer(void)
{
  if (waiting == CHAIRS)
    goHome();
  else {
    wait(&mutex);
    waiting++;
    signal(&mutex);
    signal(&customers);
    wait(&bber);
    getHairCut();
    signal(&bber);
    wait(&mutex);
      waiting--;
    signal(&mutex);
  }
}
```

Only one thread can pass the barrier!

Barrier Solution

```
specific_task();

mutex.wait();
   count++;
mutex.signal();

if (count == n)
   barrier.signal();

barrier.wait();
barrier.signal();

critical_point();
```

```
specific_task();

mutex.wait();
   count++;

   if (count == n)
      barrier.signal();

   barrier.wait();
   barrier.signal();
mutex.signal();

critical_point();
```

☠ Blocking on a semaphore while holding a mutex! ☠

```
barrier.wait();
barrier.signal();
```

Turnstile

This pattern, a `wait` and a `signal` in rapid succession, occurs often enough that it has a name called a *turnstile*, because

- it allows one thread to pass at a time, and
- it can be locked to bar all threads

See also: [*The Little Book of Semaphores*, Sec. 4.1, *Producer-Consumer Problem*].

Producer-Consumer Problem With Bounded-Buffer

Given:

```
semaphore items = 0;
semaphore spaces = BUFFER_SIZE;
```

Can we?

```
if (items >= BUFFER_SIZE)
        producer.block();
```

if: the buffer is full
then: the producer blocks until a consumer removes an item

No! We can't check the current value of a semaphore, because
! *the only operations are* `wait` *and* `signal`.
? But...

Why can't check the current value of a semaphore? We DO have seen:

```
void S.down(){
  S.value--;
  if(S.value < 0){
    addToQueue(S.L);
    sleep();
  }
}
```

```
void S.up(){
  S.value++;
  if(S.value <= 0){
    rmFromQueue(S.L);
    wakeup(S.L);
  }
}
```

Notice that the checking is within `down()` and `up()`, and is not available to user process to use it directly.

9.5 Monitors

Monitors
Monitor a high-level synchronization object for achieving mutual exclusion.

- It's a language concept, and C does not have it.
- Only one process can be active in a monitor at any instant.
- It is up to the compiler to implement mutual exclusion on monitor entries.
  - The programmer just needs to know that by turning all the critical regions into monitor procedures, no two processes will ever execute their critical regions at the same time.

```
monitor example
  integer i;
  condition c;

  procedure producer();
  ...
  end;

  procedure consumer();
  ...
  end;
end monitor;
```

Monitor
The producer-consumer problem

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;
```

```
procedure producer;
begin
  while true do
  begin
    item = produce_item;
    ProducerConsumer.insert(item)
  end
end;

procedure consumer;
begin
  while true do
  begin
    item = ProducerConsumer.remove;
    consume_item(item)
  end
end;
```

9.6 Message Passing

Message Passing

- Semaphores are too low level
- Monitors are not usable except in a few programming languages
- Neither monitor nor semaphore is suitable for distributed systems
- No conflicts, easier to implement

Message passing uses two primitives, `send` and `receive` system calls:

- `send(destination, &message);`
- `receive(source,  &message);`

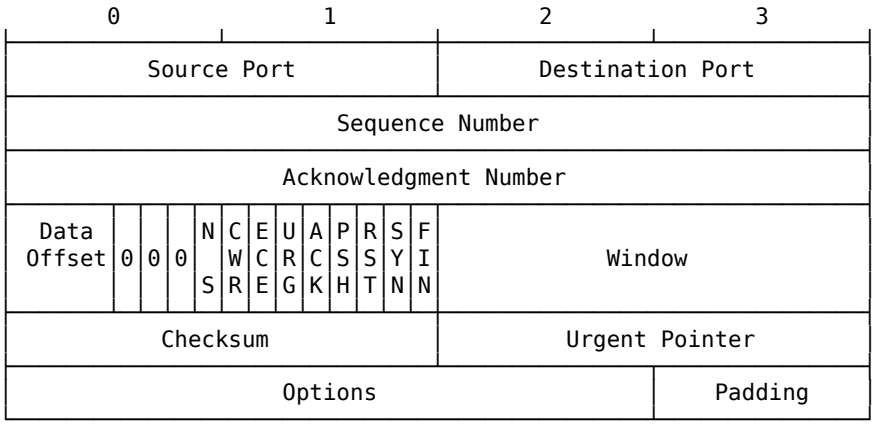
Message Passing

Design issues

- Message can be lost by network; — `ACK`
- What if the ACK is lost? — `SEQ`
- What if two processes have the same name? — `socket`
- Am I talking with the right guy? Or maybe a MIM? — `authentication`
- What if the sender and the receiver on the same machine? — Copying messages is always slower than doing a semaphore operation or entering a monitor.

Message Passing

TCP Header Format



Message Passing

The producer-consumer problem

```
#define N 100  /* number of slots in the buffer */
void producer(void)
{
  int item;
  message m;                  /* message buffer */
  while (TRUE) {
    item = produce_item();    /* generate something to put in buffer */
    receive(consumer, &m);    /* wait for an empty to arrive */
    build_message(&m, item);  /* construct a message to send */
    send(consumer, &m);       /* send item to consumer */
  }
}

void consumer(void)
{
  int item, i;
  message m;
  for (i=0; i<N; i++) send(producer, &m);  /* send N empties */
  while (TRUE) {
    receive(producer, &m);    /* get message containing item */
    item = extract_item(&m);  /* extract item from message */
    send(producer, &m);       /* send back empty reply */
    consume_item(item);       /* do something with the item */
  }
}
```

Sockets

To create a socket:

```
fd = socket(domain, type, protocol)
```

Domain Determines address format and the range of communication (local or remote). The most commonly used domains are:

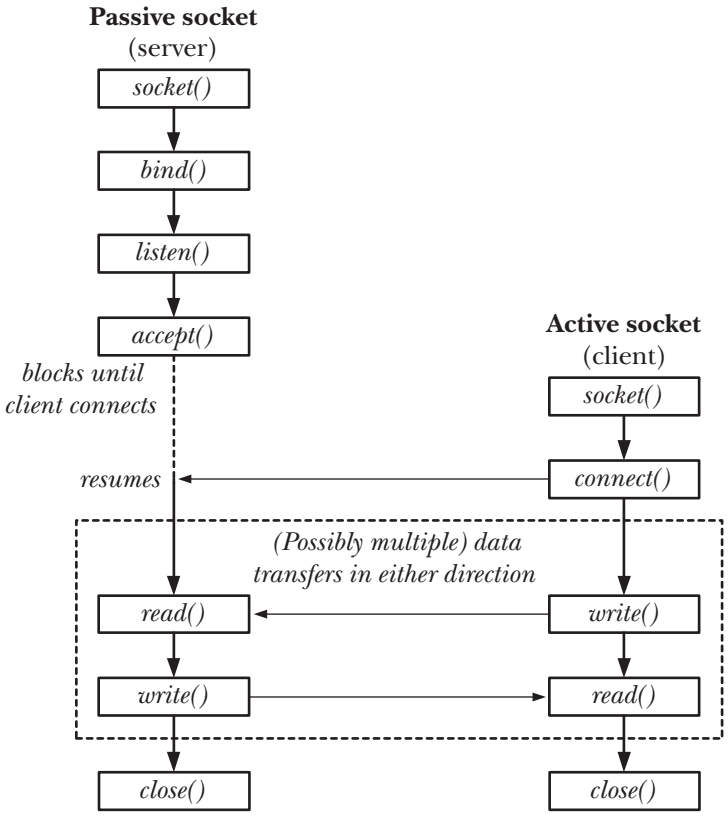
Domain	Addr structure	Addr format
AF_UNIX	sockaddr_un	/path/name
AF_INET	sockaddr_in	ip:port
AF_INET6	sockaddr_in6	ip6:port

Type SOCK_STREAM(☎), SOCK_DGRAM (✉)
Protocol always 0

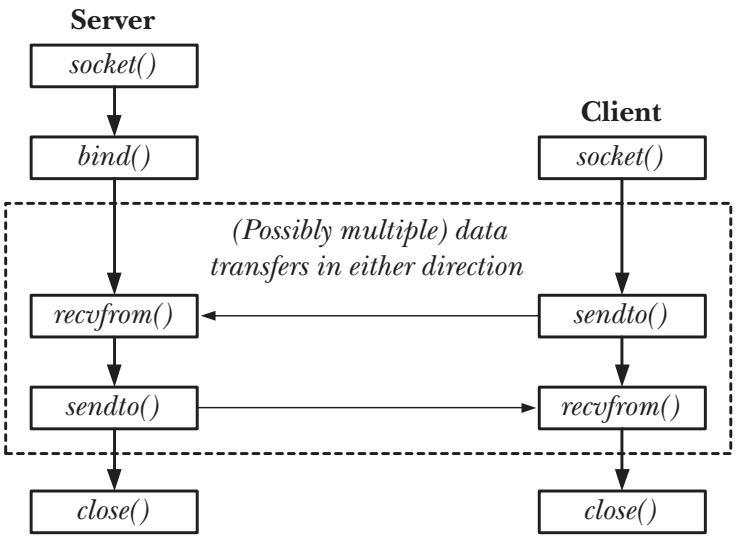
Socket System Calls

`socket()` creates a new socket
`bind()` binds a socket to an address (usually a well-known address on server side)
`listen()` waits for incoming connection requests
`connect()` sends a connection request to peer
`accept()` accepts a connection request
`send()`/`recv()` data transfer

Stream Sockets



Datagram Sockets

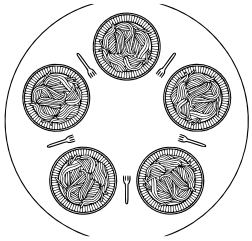


9.7 Classical IPC Problems

9.7.1 The Dining Philosophers Problem

The Dining Philosophers Problem

```
while True:
    think()
    get_forks()
    eat()
    put_forks()
```



How to implement `get_forks()` and `put_forks()` to ensure
1. No deadlock
2. No starvation
3. Allow more than one philosopher to eat at the same time

The Dining Philosophers Problem

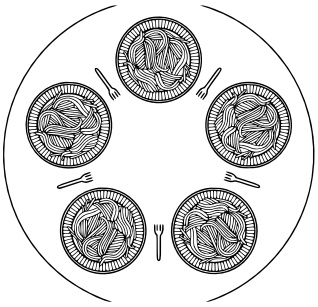
```
#define N 5                 /* number of philosophers */

void philosopher(int i)     /* i: philosopher number, from 0 to 4 */
{
  while (TRUE) {
    think();                /* philosopher is thinking */
    take_fork(i);           /* take left fork */
    take_fork((i+1) % N);   /* take right fork; % is modulo operator */
    eat();                  /* yum-yum, spaghetti */
    put_fork(i);            /* put left fork back on the table */
    put_fork((i+1) % N);    /* put right fork back on the table */
  }
}
```

? Put down the left fork and wait for a while if the right one is not available? Similar to CSMA/CD — *Starvation*

The Dining Philosophers Problem

With One Mutex

```
#define N 5
semaphore mutex=1;

void philosopher(int i)
{
  while (TRUE) {
    think();
    wait(&mutex);
      take_fork(i);
      take_fork((i+1) % N);
      eat();
      put_fork(i);
      put_fork((i+1) % N);
    signal(&mutex);
  }
}
```



☹ Only one philosopher can eat at a time.

? How about 2 mutexes? 5 mutexes?

The Dining Philosophers Problem

AST Solution (Part 1)

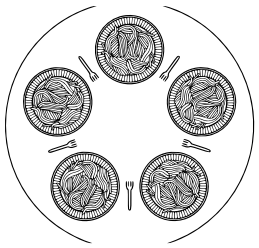
A philosopher may only move into eating state if neither neighbor is eating

```
#define N 5                 /* number of philosophers */
#define LEFT (i+N-1)%N      /* number of i's left neighbor */
#define RIGHT (i+1)%N       /* number of i's right neighbor */
#define THINKING 0          /* philosopher is thinking */
#define HUNGRY 1            /* philosopher is trying to get forks */
#define EATING 2            /* philosopher is eating */
typedef int semaphore;
int state[N];               /* state of everyone */
semaphore mutex = 1;        /* for critical regions */
semaphore s[N];             /* one semaphore per philosopher */

void philosopher(int i)  /* i: philosopher number, from 0 to N-1 */
{
  while(TRUE) {
    think();
    take_forks(i); /* acquire two forks or block */
    eat();
    put_forks(i);  /* put both forks back on table */
  }
}
```

The Dining Philosophers Problem

AST Solution (Part 2)

```
1  void take_forks(int i)          /* i: philosopher number */
2  {
3    down(&mutex);                 /* enter critical region */
4    state[i] = HUNGRY;
5    test(i);                      /* try to acquire 2 forks */
6    up(&mutex);                   /* exit critical region */
7    down(&s[i]);                  /* block if forks were not acquired */
8  }
9  void put_forks(i)               /* i: philosopher number */
10 {
11   down(&mutex);                 /* enter critical region */
12   state[i] = THINKING;
13   test(LEFT);                   /* see if left neighbor can now eat */
14   test(RIGHT);                  /* see if right neighbor can now eat */
15   up(&mutex);                   /* exit critical region */
16 }
17 void test(i)                    /* i: philosopher number */
18 {
19   if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
     ↪  EATING)
20         {
21       state[i] = EATING;
22       up(&s[i]);
23     }
24 }
```



Starvation can happen!

Step by step

1. If 5 philosophers `take_forks(i)` at the same time, only one can get `mutex`.
2. The one who gets `mutex` sets his `state` to `HUNGRY`. And then,
3. `test(i);` try to get 2 forks.
   If his `LEFT` and `RIGHT` are not `EATING`, success to get 2 forks.
   i. sets his `state` to `EATING`
   ii. `up(&s[i]);` The initial value of `s(i)` is 0.
   Now, his `LEFT` and `RIGHT` will fail to get 2 forks, even if they could grab `mutex`.
   If either `LEFT` or `RIGHT` are `EATING`, fail to get 2 forks.
4. release `mutex`
5. `down(&s[i]);`
   (a) block if forks are not acquired
   (b) `eat()` if 2 forks are acquired
6. After `eat()`ing, the philosopher doing `put_forks(i)` has to get `mutex` first.
   - because `state[i]` can be changed by more than one philosopher.
7. After getting `mutex`, set his `state` to `THINKING`
8. `test(LEFT);` see if `LEFT` can now eat?
   If `LEFT` is `HUNGRY`, and `LEFT`'s `LEFT` is not `EATING`, and `LEFT`'s `RIGHT` (me) is not `EATING`
   i. set `LEFT`'s `state` to `EATING`
   ii. `up(&s[LEFT]);`
   If `LEFT` is not `HUNGRY`, or `LEFT`'s `LEFT` is `EATING`, or `LEFT`'s `RIGHT` (me) is `EATING`, `LEFT` fails to get 2 forks.
9. `test(RIGHT);` see if `RIGHT` can now eat?
10. release `mutex`

The Dining Philosophers Problem

More Solutions

- If there is at least one leftie and at least one rightie, then deadlock is not possible
- Wikipedia: Dining philosophers problem

9.7.2 The Readers-Writers Problem

The Readers-Writers Problem
Constraint: no process may access the shared data for reading or writing while another process is writing to it.

```
semaphore mutex = 1;
semaphore noOther = 1;
int readers = 0;

void writer(void)
{
  while (TRUE) {
    wait(&noOther);
      writing();
    signal(&noOther);
  }
}
```

```
void reader(void)
{
  while (TRUE) {
    wait(&mutex);
      readers++;
      if (readers == 1)
         wait(&noOther);
    signal(&mutex);
    reading();
    wait(&mutex);
      readers--;
      if (readers == 0)
        signal(&noOther);
    signal(&mutex);
    anything();
  }
}
```

Starvation The writer could be blocked forever if there are always someone reading.

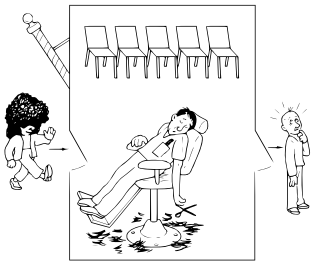
The Readers-Writers Problem
No starvation

```
semaphore mutex = 1;
semaphore noOther = 1;
semaphore turnstile = 1;
int readers = 0;

void writer(void)
{
  while (TRUE) {
    turnstile.wait();
      wait(&noOther);
        writing();
      signal(&noOther);
    turnstile.signal();
  }
}
```

```
void reader(void)
{
  while (TRUE) {
    turnstile.wait();
    turnstile.signal();

    wait(&mutex);
      readers++;
      if (readers == 1)
         wait(&noOther);
    signal(&mutex);
    reading();
    wait(&mutex);
      readers--;
      if (readers == 0)
        signal(&noOther);
    signal(&mutex);
    anything();
  }
}
```

9.7.3 The Sleeping Barber Problem

The Sleeping Barber Problem



Where's the problem?

- the barber saw an empty room right before a customer arrives the waiting room;
- Several customer could race for a single chair;

Solution

```
#define CHAIRS 5
semaphore customers = 0; // any customers or not?
semaphore bber = 0;      // barber is busy
semaphore mutex = 1;
int waiting = 0;         // queueing customers
```

```
void barber(void)
{
  while (TRUE) {
    wait(&customers);
    wait(&mutex);
       waiting--;
    signal(&mutex);
    cutHair();
    signal(&bber);
  }
}
```

```
void customer(void)
{
  if(waiting == CHAIRS)
    goHome();
  else {
    wait(&mutex);
    waiting++;
    signal(&mutex);
    signal(&customers);
    wait(&bber);
    getHairCut();
  }
}
```

Solution2

```
#define CHAIRS 5
semaphore customers = 0;
semaphore bber = ???;
semaphore mutex = 1;
int waiting = 0;

void barber(void)
{
  while (TRUE) {
    wait(&customers);
    cutHair();
  }
}
```

```
void customer(void)
{
  wait(&mutex);
  if (waiting == CHAIRS){
      signal(&mutex);
      goHome();
  } else {
      waiting++;
      signal(&customers);
      signal(&mutex);
      wait(&bber);
      getHairCut();
      wait(&mutex);
         waiting--;
      signal(&mutex);
      signal(&bber);
  }
}
```

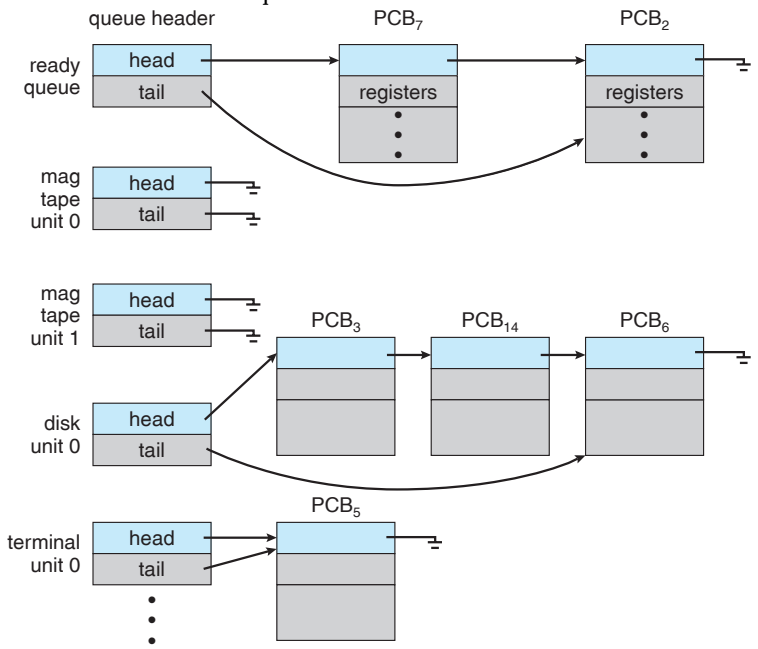

[1] Wikipedia. *Inter-process communication — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Inter-process%5C_communication&oldid=645037874.

[2] Wikipedia. *Semaphore (programming) — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Semaphore%5C_(programming)&oldid=647556304.

10 CPU Scheduling

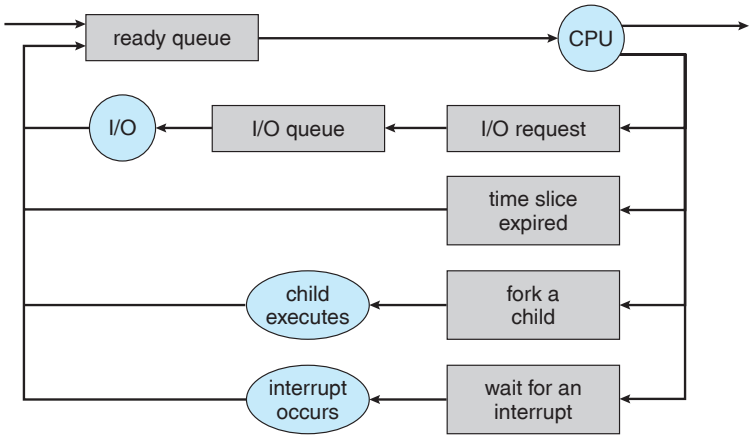
10.1 Process Scheduling Queues

Scheduling Queues
Job queue consists all the processes in the system
Ready queue A linked list consists processes in the main memory ready for execute
Device queue Each device has its own device queue



- The *tail* pointer — When adding a new process to the queue, don't have to find the tail by traversing the list

Queueing Diagram



10.2 Scheduling

Scheduling
- *Scheduler* uses *scheduling algorithm* to choose a process from the ready queue
- Scheduling doesn't matter much on simple PCs, because
  1. Most of the time there is only one active process
  2. The CPU is too fast to be a scarce resource any more
- Scheduler has to make efficient use of the CPU because process switching is expensive
  1. User mode $\rightarrow$ kernel mode
  2. Save process state, registers, memory map...

3. Selecting a new process to run by running the scheduling algorithm
4. Load the memory map of the new process
5. The process switch usually invalidates the entire memory cache

Scheduling Algorithm Goals

All systems
Fairness giving each process a fair share of the CPU
Policy enforcement seeing that stated policy is carried out
Balance keeping all parts of the system busy

Batch systems
Throughput maximize jobs per hour
Turnaround time minimize time between submission and termination
CPU utilization keep the CPU busy all the time

Interactive systems
Response time respond to requests quickly
Proportionality meet users' expectations

Real-time systems
Meeting deadlines avoid losing data
Predictability avoid quality degradation in multimedia systems

See also: [*Modern Operating Systems*, Sec. 2.4.1.5, *Scheduling Algorithm Goals*, p. 150].

Process Classification

Traditionally
CPU-bound processes vs. I/O-bound processes

Alternatively
Interactive processes responsiveness
- command shells, editors, graphical apps

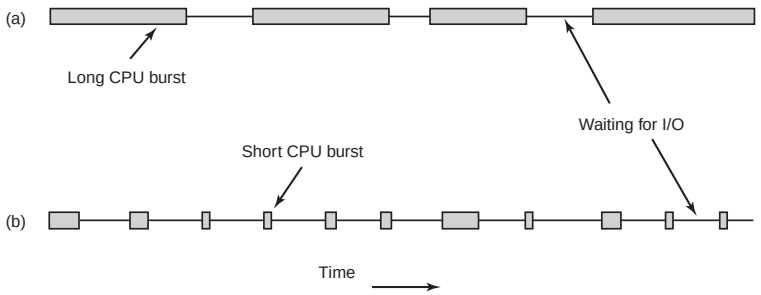
Batch processes no user interaction, run in background, often penalized by the scheduler
- programming language compilers, database search engines, scientific computations

Real-time processes video and sound apps, robot controllers, programs that collect data from physical sensors
- should never be blocked by lower-priority processes
- should have a short guaranteed response time with a minimum variance

The two classifications we just offered are somewhat independent. For instance, a batch process can be either I/O-bound (e.g., a database server) or CPU-bound (e.g., an image-rendering program).

While real-time programs are explicitly recognized as such by the scheduling algorithm in Linux, there is no easy way to distinguish between interactive and batch programs. The Linux 2.6 scheduler implements a sophisticated heuristic algorithm based on the past behavior of the processes to decide whether a given process should be considered as interactive or batch. Of course, the scheduler tends to favor interactive processes over batch ones.

Process Behavior
CPU-bound vs. I/O-bound
Types of CPU bursts:
- long bursts – CPU bound (i.e. batch work)
- short bursts – I/O bound (i.e. emacs)



As CPUs get faster, processes tend to get more I/O-bound.

Schedulers
Long-term scheduler (or job scheduler) - selects which processes should be brought into the ready queue.
Short-term scheduler (or CPU scheduler) - selects which process should be executed next and allocates CPU.
Midium-term scheduler swapping.

- LTS is responsible for a good *process mix* of I/O-bound and CPU-bound process leading to best performance.
- Time-sharing systems, e.g. UNIX, often have no long-term scheduler.

Nonpreemptive vs. preemptive
A nonpreemptive scheduling algorithm lets a process run as long as it wants until it blocks (I/O or waiting for another process) or until it voluntarily releases the CPU.
A preemptive scheduling algorithm will forcibly suspend a process after it runs for sometime. — clock interruptable

10.3 Scheduling In Batch Systems

Scheduling In Batch Systems
First-Come First-Served

- nonpreemptive
- simple
- also has a disadvantage

  What if a CPU-bound process (e.g. runs 1s at a time) followed by many I/O-bound processes (e.g. 1000 disk reads to complete)?

  * In this case, a preemptive scheduling is preferred.

Scheduling In Batch Systems
Shortest Job First

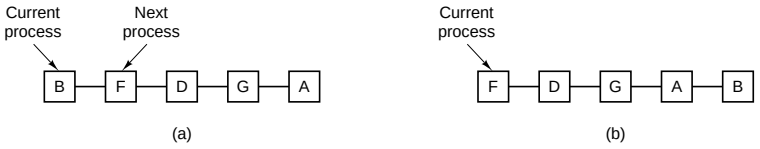
Average turnaround time
(a) $(8 + 12 + 16 + 20) \div 4 = 14$
(b) $(4 + 8 + 12 + 20) \div 4 = 11$

How to know the length of the next CPU burst?

- For long-term (job) scheduling, user provides
- For short-term scheduling, no way

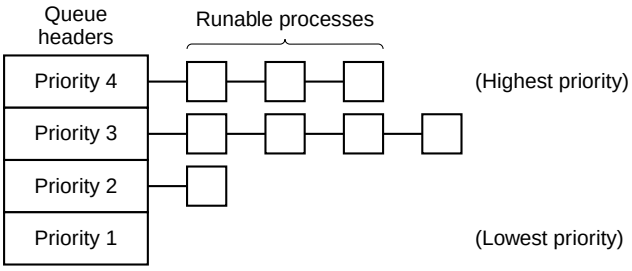
10.4 Scheduling In Interactive Systems

Scheduling In Interactive Systems
Round-Robin Scheduling



- Simple, and most widely used;
- Each process is assigned a time interval, called its *quantum*;
- How long shoud the quantum be?
  - too short — too many process switches, lower CPU efficiency;
  - too long — poor response to short interactive requests;
  - usually around $20 \sim 50$ms.

Scheduling In Interactive Systems
Priority Scheduling

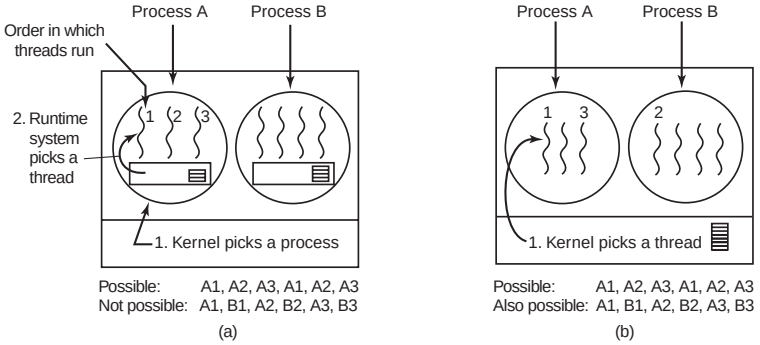


- SJF is a priority scheduling;
- *Starvation* — low priority processes may never execute;
  - *Aging* — as time progresses increase the priority of the process;

`$ man nice`

10.5 Thread Scheduling

Thread Scheduling



- With kernel-level threads, sometimes a full context switch is required
- Each process can have its own application-specific thread scheduler, which usually works better than kernel can

10.6 Linux Scheduling

- [*Operating System Concepts Essentials*, Sec. 5.6.3, *Example: Linux Scheduling*].
- [*Operating System Concepts Essentials*, Sec. 15.5, *Scheduling*].
- [*Understanding The Linux Kernel*, Chap. 7, *Process Scheduling*].
- [*Linux Kernel Development*, Chap. 4, *Process Scheduling*].

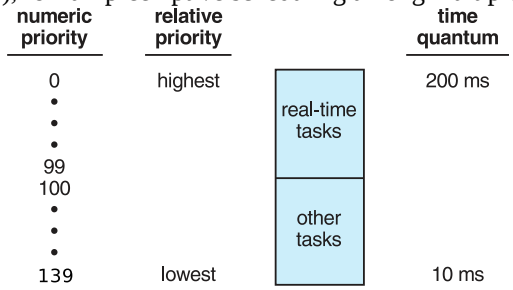
Call graph:

```
└ cpu_idle()
    └ schedule()
        └ context_switch()
            └ switch_to()
```

Process Scheduling In Linux

A preemptive, priority-based algorithm with two separate priority ranges:

1. *real-time* range ($0 \sim 99$), for tasks where absolute priorities are more important than fairness
2. *nice value* range ($100 \sim 139$), for fair preemptive scheduling among multiple processes



In Linux, Process Priority is Dynamic

The scheduler keeps track of what processes are doing and adjusts their priorities periodically

- Processes that have been denied the use of a CPU for a long time interval are boosted by dynamically increasing their priority (usually I/O-bound)
- Processes running for a long time are penalized by decreasing their priority (usually CPU-bound)
- Priority adjustments are performed only on user tasks, not on real-time tasks

Tasks are determined to be I/O-bound or CPU-bound based on an interactivity heuristic

A task's interactiveness metric is calculated based on how much time the task executes compared to how much time it sleeps

Problems With The Pre-2.6 Scheduler

- 😐 an algorithm with $O(n)$ complexity
- a single runqueue for all processors
  - 😊 good for load balancing
  - ☹ bad for CPU caches, when a task is rescheduled from one CPU to another
- 😐 a single runqueue lock — only one CPU working at a time

The scheduling algorithm used in earlier versions of Linux was quite simple and straightforward: at every process switch the kernel scanned the list of runnable processes, computed their priorities, and selected the "best" process to run. The main drawback of that algorithm is that the time spent in choosing the best process depends on the number of runnable processes; therefore, the algorithm is too costly, that is, it spends too much time in high-end systems running thousands of processes[*Understanding The Linux Kernel*, Sec. 7.2, *The Scheduling Algorithm*].

Scheduling In Linux 2.6 Kernel

- 😊 $O(1)$ — Time for finding a task to execute depends not on *the number of active tasks* but instead on *the number of priorities*
- 😊 Each CPU has its own *runqueue*, and schedules itself independently; better cache efficiency
- The job of the scheduler is simple — Choose the task on the highest priority list to execute

How to know there are processes waiting in a priority list?

A priority bitmap (5 32-bit words for 140 priorities) is used to define when tasks are on a given priority list.

- `find-first-bit-set` instruction is used to find the highest priority bit.

Each runqueue has two priority arrays

10.6.1 Completely Fair Scheduling

Completely Fair Scheduling (CFS)

Linux's Process Scheduler
up to 2.4: simple, scaled poorly

- $O(n)$
- non-preemptive in kernel mode
- single run queue (cache? SMP?)

from 2.5 on: $O(1)$ scheduler
- ☺ 140 priority lists — scaled well
- ☺ one run queue per CPU — true SMP support
- ☺ preemptive
- ☺ ideal for large server workloads
- ☹ showed latency on desktop systems

from 2.6.23 on: Completely Fair Scheduler (CFS)
- ☺ improved interactive performance

Completely Fair Scheduler (CFS)

For a perfect (unreal) multitasking CPU
- n runnable processes can run at the same time
- each process should receive $\frac{1}{n}$ of CPU power

For a real world CPU
- can run only a single task at once — unfair
  - ☺ while one task is running
  - ☹ the others have to wait
- `p->wait_runtime` is the amount of time the task should now run on the CPU for it becomes completely fair and balanced.
  - ☺ on ideal CPU, the `p->wait_runtime` value would always be zero
- CFS always tries to run the task with the largest `p->wait_runtime` value

See also: *Discussing the Completely Fair Scheduler*[6].

CFS

In practice it works like this:
- While a task is using the CPU, its `wait_runtime` decreases

```
wait_runtime = wait_runtime - time_running
```

if: its `wait_runtime` $\neq$ $\texttt{MAX}_{\texttt{wait\_runtime}}$ (among all processes)
then: it gets preempted
- Newly woken tasks (`wait_runtime = 0`) are put into the tree more and more to the right
- slowly but surely giving a chance for every task to become the "leftmost task" and thus get on the CPU within a deterministic amount of time

[1] Wikipedia. *Scheduling (computing) — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Scheduling%5C_(computing)&oldid=647892123.

[6] http://kerneltrap.org/node/8208

11 Deadlock

11.1 Resources

A Major Class of Deadlocks Involve Resources

Processes need access to resources in reasonable order
Suppose...
- a process holds resource A and requests resource B. At same time,
- another process holds B and requests A

Both are blocked and remain so

Examples of computer resources
- printers
- memory space
- data (e.g. a locked record in a DB)
- semaphores

Resources

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

(b)

Deadlock!

Resources

Deadlocks occur when ...
processes are granted exclusive access to *resources*
e.g. devices, data records, files, ...

Preemptable resources can be taken away from a process with no ill effects
e.g. memory

Nonpreemptable resources will cause the process to fail if taken away
e.g. CD recorder

In general, deadlocks involve nonpreemptable resources.

Resources

Sequence of events required to use a resource

request	use	release
`open()`		`close()`
`allocate()`		`free()`

What if request is denied?
Requesting process
- may be blocked
- may fail with error code

11.2 Introduction to Deadlocks

The Best Illustration of a Deadlock

A law passed by the Kansas legislature early in the 20th century
"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Deadlock
A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.
- Usually the event is release of a currently held resource
- None of the processes can ...
  - run
  - release resources
  - be awakened

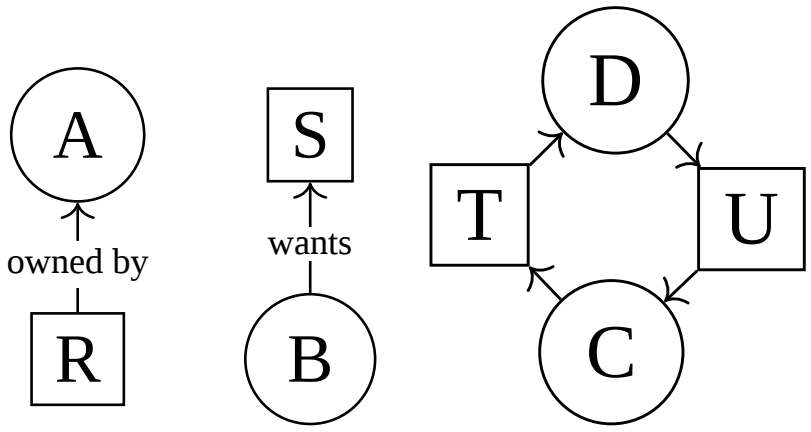
Four Conditions For Deadlocks
Mutual exclusion condition each resource can only be assigned to one process or is available
Hold and wait condition process holding resources can request additional
No preemption condition previously granted resources cannot forcibly taken away
Circular wait condition
- must be a circular chain of 2 or more processes
- each is waiting for resource held by next member of the chain

Four Conditions For Deadlocks

Unlocking a deadlock is to answer 4 questions:
1. Can a resource be assigned to more than one process at once?
2. Can a process hold a resource and ask for another?
3. can resources be preempted?
4. Can circular waits exits?

11.3 Deadlock Modeling

Resource-Allocation Graph



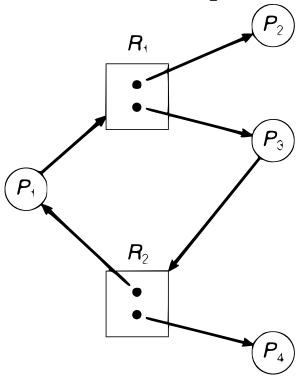
Strategies for dealing with Deadlocks

1. detection and recovery
2. dynamic avoidance — careful resource allocation
3. prevention — negating one of the four necessary conditions
4. just ignore the problem altogether

Resource-Allocation Graph

- The right graph has deadlock

Resource-Allocation Graph



Basic facts:

- No cycles ➠ no deadlock
- If graph contains a cycle ➠
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

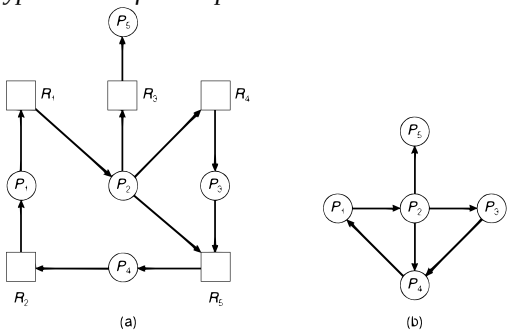
11.4 Deadlock Detection and Recovery

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Deadlock Detection

Single Instance of Each Resource Type — Wait-for Graph



- $\boxed{P_i \rightarrow P_j}$ if P_i is waiting for P_j;
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Deadlock Detection

Several Instances of a Resource Type

Tape drives Plotters Scanners CD Roms

E = (4 2 3 1)

Tape drives Plotters Scanners CD Roms

A = (2 1 0 0)

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Row n:

C: current allocation to process n

R: current requirement of process n

Column m:

C: current allocation of resource class m

R: current requirement of resource class m

Deadlock Detection

Several Instances of a Resource Type

Resources in existence $(E_1, E_2, E_3, ..., E_m)$

Resources available $(A_1, A_2, A_3, ..., A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

e.g.

$$(C_{13} + C_{23} + \ldots + C_{n3}) + A_3 = E_3$$

n: number of processes;

m: number of resource classes;

E**:** a vector of existing resources
- $E = [E_1, E_2, ..., E_m]$
- $E_i = 2$ means system has 2 resources of class $i, (1 \leq i \leq m)$;

A**:** a vector of available resources;
- $A = [A_1, A_2, ...A_m]$
- $A_i = 2$ means system has 2 resources of class i left unassigned;

C_{ij}**:** is the number of instances of resource j that process i holds;
e.g. $C_{31} = 2$ means P_3 has 2 resources of class 1;

R_{ij}**:** is the number of instances of resource j that process i wants;
e.g. $R_{43} = 2$ means P_4 wants 2 resources of class 3;

Maths recall: vectors comparison
For two vectors, X and Y

$$X \leq Y \quad \text{iff} \quad X_i \leq Y_i \quad \text{for} \quad 0 \leq i \leq m$$

e.g.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \leq \begin{bmatrix} 2 & 3 & 4 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & \mathbf{3} & 4 \end{bmatrix} \not\leq \begin{bmatrix} 2 & 3 & \mathbf{2} & 4 \end{bmatrix}$$

Deadlock Detection
Several Instances of a Resource Type

	Tape drives	Plotters	Scanners	CD Roms
E = (	4	2	3	1)

	Tape drives	Plotters	Scanners	CD Roms
A = (	2	1	0	0)

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

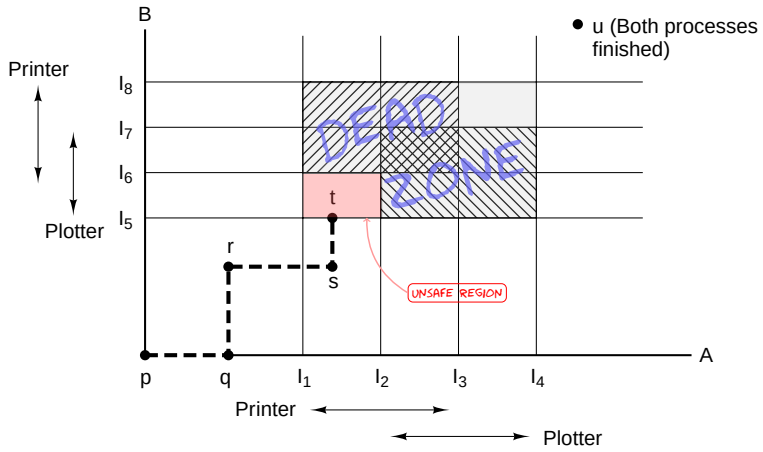
$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

$$\begin{matrix} A & & R \end{matrix}$$
$$(2\ 1\ 0\ 0) \geq R_3, (2\ 1\ 0\ 0)$$
$$(2\ 2\ 2\ 0) \geq R_2, (1\ 0\ 1\ 0)$$
$$(4\ 2\ 2\ 1) \geq R_1, (2\ 0\ 0\ 1)$$

Recovery From Deadlock
- Recovery through preemption
  - take a resource from some other process
  - depends on nature of the resource
- Recovery through rollback
  - checkpoint a process periodically
  - use this saved state
  - restart the process if it is found deadlocked
- Recovery through killing processes

11.5 Deadlock Avoidance

Deadlock Avoidance
Resource Trajectories



- B is requesting a resource at point `t`. The system must decide whether to grant it or not.
- Deadlock is unavoidable if you get into *unsafe region*.

Deadlock Avoidance
Safe and Unsafe States
Assuming $E = 10$

Unsafe

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

UNSAFE

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4
(d)

Safe

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7
(e)

1. Given totally 10 resources, for process A, B, C:
   - A has 3, and need 6 more
   - B has 2, and need 2 more
   - C has 2, and need 5 more
   - 3 left available
2. allocate 1 to A,
   - A has 4, and need 5 more
   - B unchange
   - C unchange
   - 2 left available
3. ...

Deadlock Avoidance
The Banker's Algorithm for a Single Resource
The banker's algorithm considers each request as it occurs, and sees if granting it leads to a safe state.

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

UNSAFE!

Free: 1

(c)

$$unsafe \neq deadlock$$

Deadlock Avoidance
The Banker's Algorithm for Multiple Resources

Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

$D \rightarrow A \rightarrow B, C, E$
$D \rightarrow E \rightarrow A \rightarrow B, C$

Deadlock Avoidance
Mission Impossible

In practice
- processes rarely know in advance their max future resource needs;
- the number of processes is not fixed;
- the number of available resources is not fixed.

Conclusion: Deadlock avoidance is essentially a mission impossible.

11.6 Deadlock Prevention

Deadlock Prevention
Break The Four Conditions

Attacking the Mutual Exclusion Condition
- For example, using a *printing daemon* to avoid exclusive access to a printer.
- Not always possible
  - Not required for sharable resources;
  - must hold for nonsharable resources.
- The best we can do is to avoid mutual exclusion as much as possible
  - Avoid assigning a resource when not really necessary
  - Try to make sure as few processes as possible may actually claim the resource

See also: [*Modern Operating Systems*, Sec. 6.6.1, *Attacking the Mutual Exclusion Condition*, p. 452] for the *printer daemon* example.

Attacking the Hold and Wait Condition
Must guarantee that whenever a process requests a resource, it does not hold any other resources.
Try: the processes must request all their resources before starting execution

if everything is available
then can run
if one or more resources are busy
then nothing will be allocated (just wait)

Problem:

- many processes don't know what they will need before running
- Low resource utilization; starvation possible

Attacking the No Preemption Condition

if a process that is holding some resources requests another resource that cannot be immediately allocated to it
then
1. All resources currently being held are released
2. Preempted resources are added to the list of resources for which the process is waiting
3. Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

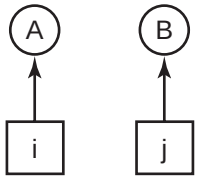
Low resource utilization; starvation possible

Attacking Circular Wait Condition

Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

(a)



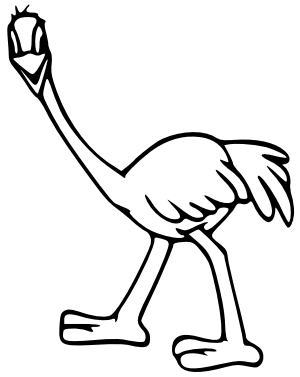
(b)

It's hard to find an ordering that satisfies everyone.

11.7 The Ostrich Algorithm

The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
- UNIX and Windows takes this approach
- It is a trade off between
  - convenience
  - correctness



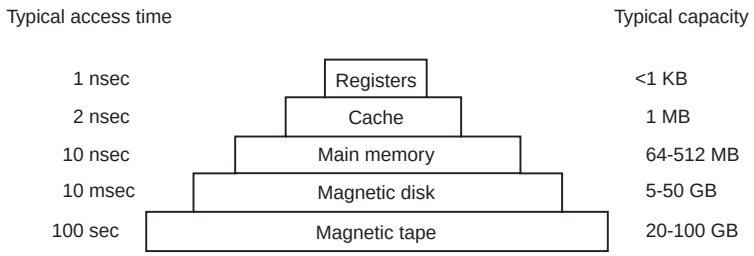
[1] Wikipedia. *Deadlock — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Deadlock&oldid=645602687.

Part III
Memory Management

12 Background

Memory Management
In a perfect world Memory is *large, fast, non-volatile*

In real world ...

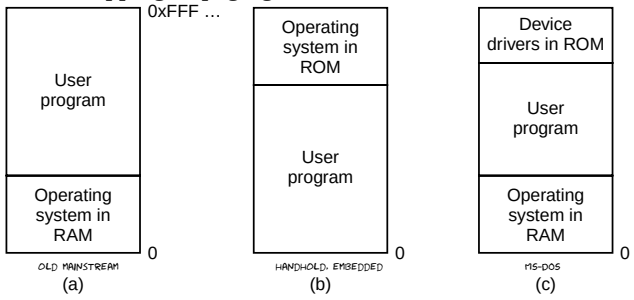


Memory manager handles the memory hierarchy.

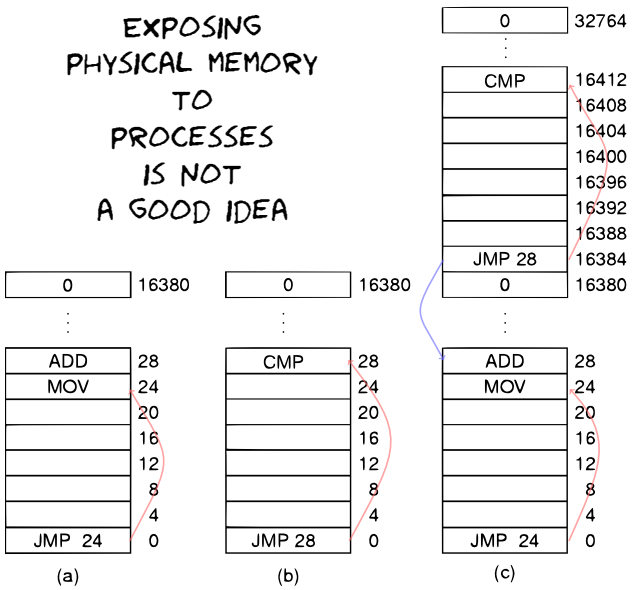
Basic Memory Management
Real Mode

In the old days ...

- Every program simply saw the physical memory
- mono-programming without swapping or paging



Basic Memory Management
Relocation Problem



(a) only one program in memory
(b) only another program in memory
(c) both in memory

Memory Protection
Protected mode

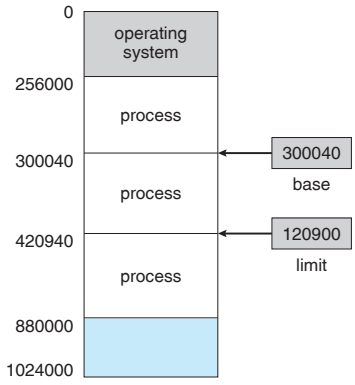
We need

- Protect the OS from access by user programs
- Protect user programs from one another

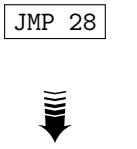
Protected mode is an operational mode of x86-compatible CPU.

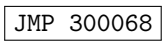
- The purpose is to protect everyone else (including the OS) from your program.

Memory Protection
Logical Address Space
Base register holds the smallest legal physical memory address
Limit register contains the size of the range

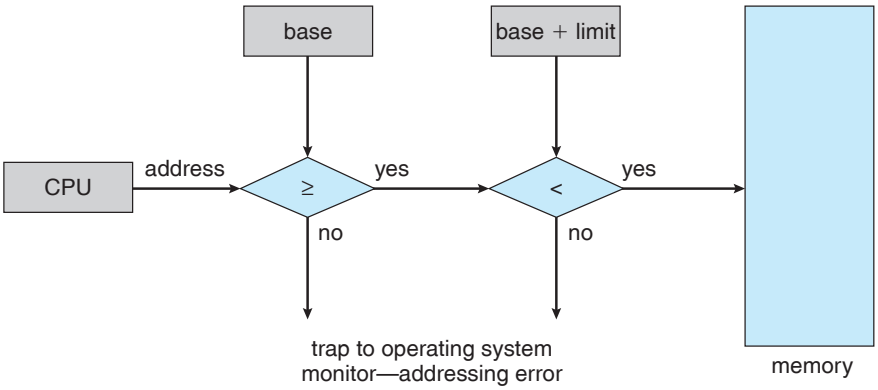


A pair of `base` and `limit` registers define the logical address space

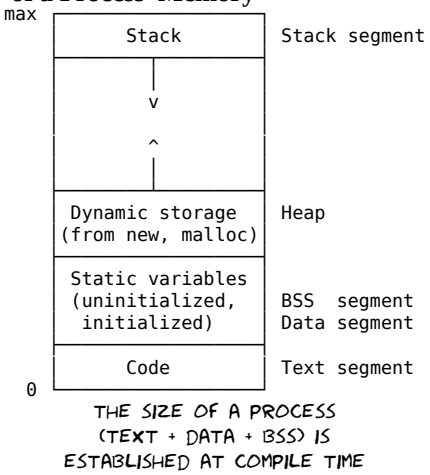




Memory Protection
Base and limit registers



UNIX View of a Process' Memory



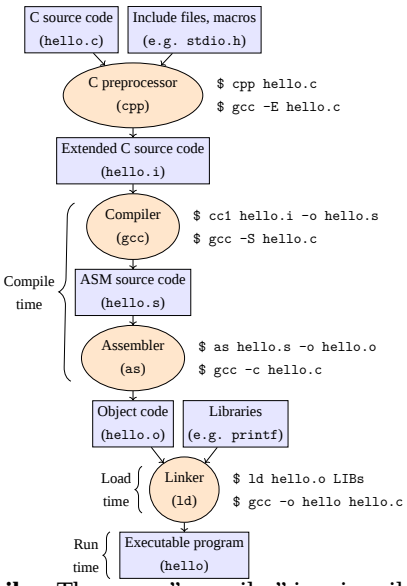
text: program code
data: initialized global and static data
bss: uninitialized global and static data
heap: dynamically allocated with `malloc, new`
stack: local variables

Stack vs. Heap

Stack	Heap
compile-time allocation	run-time allocation
auto clean-up	you clean-up
inflexible	flexible
smaller	bigger
quicker	slower

How large is the ...
stack: `ulimit -s`
heap: could be as large as your virtual memory
text|data|bss: `size a.out`

Multi-step Processing of a User Program
When is space allocated?



Static: before program start running
- Compile time
- Load time

Dynamic: as program runs
- Execution time

Compiler The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code)[*Compiler — Wikipedia, The Free Encyclopedia*].

Assembler An assembler creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities[*Assembly language — Wikipedia, The Free Encyclopedia*].

Linker Computer programs typically comprise several parts or modules; all these parts/modules need not be contained within a single object file, and in such case refer to each other by means of symbols[*Linker (computing) — Wikipedia, The Free Encyclopedia*].

When a program comprises multiple object files, the linker combines these files into a unified executable program, resolving the symbols as it goes along.

Linkers can take objects from a collection called a library. Some linkers do not include the whole library in the output; they only include its symbols that are referenced from other object files or libraries. Libraries exist for diverse purposes, and one or more system libraries are usually linked in by default.

The linker also takes care of arranging the objects in a program's address space. This may involve relocating code that assumes a specific base address to another base. Since a compiler seldom knows where an object will reside, it often assumes a fixed base location (for example,zero).

Loader An assembler creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities. ... Loading a program involves reading the contents of executable file, the file containing the program text, into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code[*Loader (computing) — Wikipedia, The Free Encyclopedia*]

Dynamic linker A dynamic linker is the part of an operating system (OS) that loads (copies from persistent storage to RAM) and links (fills jump tables and relocates pointers) the shared libraries needed by an executable at run time, that is, when it is executed. The specific operating system and executable format determine how the dynamic linker functions and how it is implemented. Linking is often referred to as a process that is performed at compile time of the executable while a dynamic linker is in actuality a special loader that loads external shared libraries into a running process and then binds those shared libraries dynamically to the running process. The specifics of how a dynamic linker functions is operating-system dependent[*Dynamic linker — Wikipedia, The Free Encyclopedia*]

Linkers and Loaders allow programs to be built from modules rather than as one big monolith.

See also:
- [*Computer Systems: A Programmer's Perspective*, Chap. 7, *Linking*].
- COMPILER, ASSEMBLER, LINKER AND LOADER: A BRIEF STORY[7].
- Linkers and Loaders[8].

[7] http://www.tenouk.com/ModuleW.html
[8] http://www.iecc.com/linker/

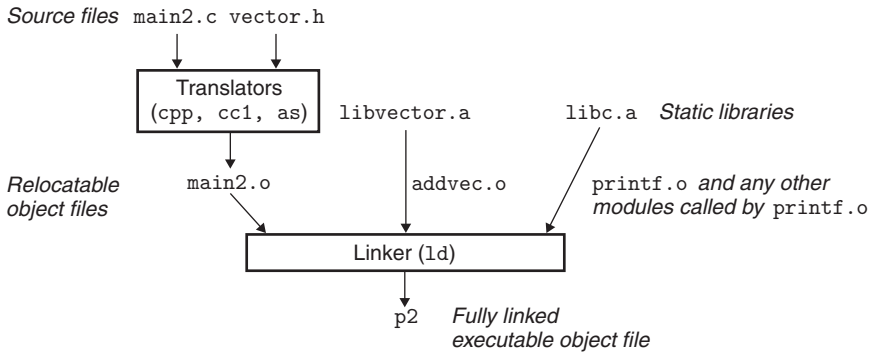
- [*Linkers and Loaders*, *Links and loaders*].
- Linux Journal: Linkers and Loaders[9]. Discussing how compilers, links and loaders work and the benefits of shared libraries.

Address Binding
Who assigns memory to segments?

Static-binding: before a program starts running
Compile time: *Compiler* and *assembler* generate an object file for each source file
Load time:
- *Linker* combines all the object files into a single executable object file
- *Loader* (part of OS) loads an executable object file into memory at location(s) determined by the OS
  - invoked via the `execve` system call

Dynamic-binding: as program runs
- Execution time:
  - uses `new` and `malloc` to dynamically allocate memory
  - gets space on stack during function calls
- Address binding has nothing to do with physical memory (RAM). It determines the addresses of objects in the address space (virtual memory) of a process.

Static loading
- The entire program and all data of a process must be in physical memory for the process to execute
- The size of a process is thus limited to the size of physical memory

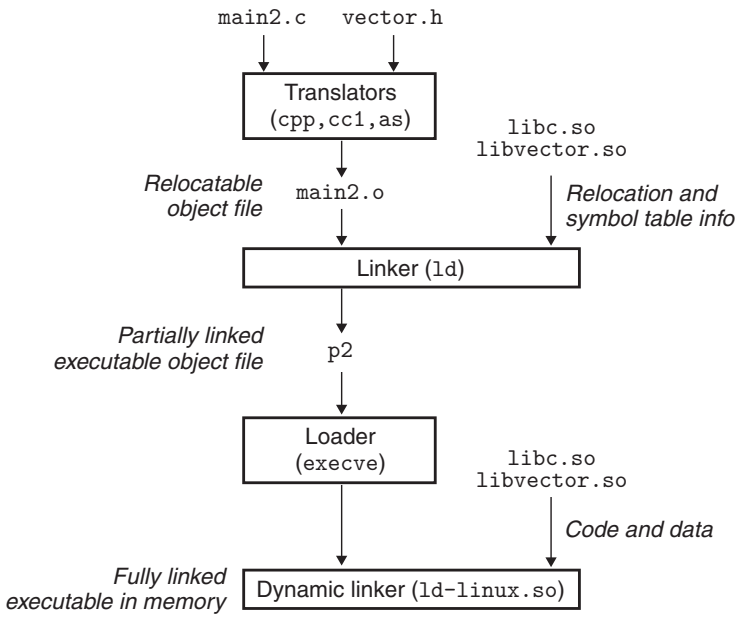


Dynamic Linking
A *dynamic linker* is actually a special loader that loads external shared libraries into a running process
- Small piece of code, `stub`, used to locate the appropriate memory-resident library routine
- Only one copy in memory
- Don't have to re-link after a library update

```
if ( stub_is_executed ){
  if ( !routine_in_memory )
     load_routine_into_memory();
  stub_replaces_itself_with_routine();
  execute_routine();
}
```

[9] http://www.linuxjournal.com/article/6463

Dynamic Linking



More about dynamic linking Many operating system environments allow dynamic linking, that is the postponing of the resolving of some undefined symbols until a program is run. That means that the executable code still contains undefined symbols, plus a list of objects or libraries that will provide definitions for these. Loading the program will load these objects/libraries as well, and perform a final linking. Dynamic linking needs no linker[*Linker (computing) — Wikipedia, The Free Encyclopedia*].

This approach offers two advantages:

- Often-used libraries (for example the standard system libraries) need to be stored in only one location, not duplicated in every single binary.
- If an error in a library function is corrected by replacing the library, all programs using it dynamically will benefit from the correction after restarting them. Programs that included this function by static linking would have to be re-linked first.

There are also disadvantages:

- Known on the Windows platform as ”DLL Hell”, an incompatible updated DLL will break executables that depended on the behavior of the previous DLL.
- A program, together with the libraries it uses, might be certified (e.g. as to correctness, documentation requirements, or performance) as a package, but not if components can be replaced. (This also argues against automatic OS updates in critical systems; in both cases, the OS and libraries form part of a qualified environment.)

Library Files

Static libraries *.a* files. Very old ones, but still alive.

```
$ find /usr/lib -name "*.a"
```

Shared libraries *.so* files. The preferred ones.

```
$ find /usr/lib -name "*.so.*"
```

Examples:

```
$ gcc -o hello hello.c /usr/lib/libm.a
$ gcc -o hello hello.c -lm
```

Build A Static Library

Source codes

main.c

```
#include "lib.h"

int main(int argc, char*
↪  argv[])
{
  int i=1;

  for (; i<argc; i++)
    {
      hello(argv[i]);
      hi(argv[i]);
    }
  return 0;
}
```

lib.h

```
#include <stdio.h>

void hello(char *);
void hi(char *);
```

hello.c

```
#include <stdio.h>

void hello(char *arg)
{
  printf("Hello, %s!\n", arg);
}
```

hi.c

```
#include <stdio.h>

void hi(char *arg)
{
  printf("Hi, %s!\n", arg);
}
```

Build A Static Library

Step by step

1. Get *hello.o* and *hi.o*
   $ gcc -c hello.c hi.c
2. Put **.o* into *libhi.a*
   $ ar crv libhi.a hello.o hi.o
3. Use *libhi.a*
   $ gcc main.c libhi.a

Build A Static Library

Makefile

```
main: main.c lib.h libhi.a
        gcc -Wall -o main main.c libhi.a

libhi.a: hello.o hi.o
        ar crv libhi.a hello.o hi.o

hello.o: hello.c
        gcc -Wall -c hello.c

hi.o: hi.c
        gcc -Wall -c hi.c

clean:
        rm -f *.o *.a main
```

Build A Shared Library

Source codes

hello.c

```
#include "hello.h"

```

```
int main(int argc, char *argv[])
{
  if (argc != 2)
    printf ("Usage: %s needs an argument.\n", argv[0]);
  else
    hi(argv[1]);
  return 0;
}
```

hello.h

```
#include <stdio.h>
#include <stdlib.h>

int hi(char*);
```

hi.c

```
#include "hello.h"

int hi(char* s)
{
  printf ("Hi, %s\n",s);
  return 0;
}
```

Build A Shared Library

Step by step

1. Get *hi.o*
   ```
   $ gcc -fPIC -c hi.c
   ```
2. Get *libhi.so*
   ```
   $ gcc -shared -o libhi.so hi.o
   ```
3. Use *libhi.so*
   ```
   $ gcc -L. -Wl,-rpath=. hello.c -lhi
   ```
4. Check it
   ```
   $ ldd a.out
   ```

Build A Shared Library

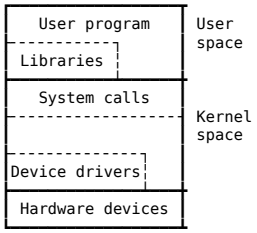
Makefile

```
# http://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html
# http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html
#
# gcc -fPIC -c hi.c
# gcc -shared -o libhi.so hi.o
# gcc -L/current/dir -Wl,option -Wall -o hello hello.c -lhi
#
# -L         - tells ld where to search libraries
# -Wl,option - pass option as an option to the linker (ld)
# -rpath=dir - Add a directory to the runtime library search path

hello: hello.c hello.h libhi.so
        gcc -L. -Wl,-rpath=. -Wall -o hello hello.c -lhi
libhi.so: hi.o hello.h
        gcc -shared -o libhi.so hi.o
hi.o: hi.c hello.h
        gcc -fPIC -c hi.c
clean:
        rm *.o *.so hello
```

GNU C Library

Linux API > POSIX API

```
$ man 7 libc
$ man 3 intro
$ man gcc
$ info gcc
sudo apt install gcc-doc
```



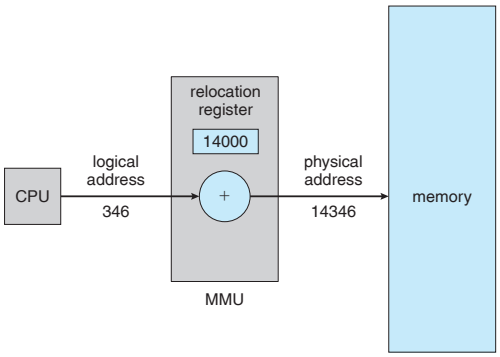
Logical vs. Physical Address Space

- Mapping logical address space to physical address space is central to MM
  Logical address generated by the CPU; also referred to as *virtual address*
  Physical address address seen by the memory unit
- In compile-time and load-time address binding schemes, LAS and PAS are identical in size
- In execution-time address binding scheme, they are differ.

Logical vs. Physical Address Space

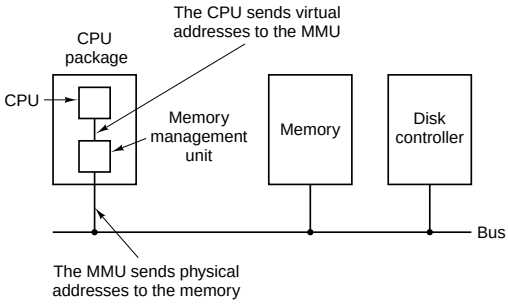
The user program

- deals with logical addresses
- never sees the real physical addresses

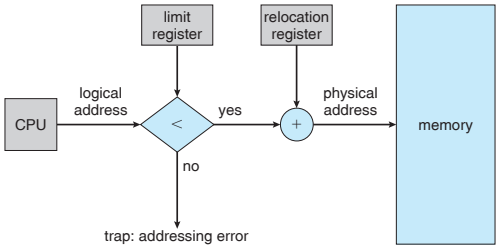


MMU

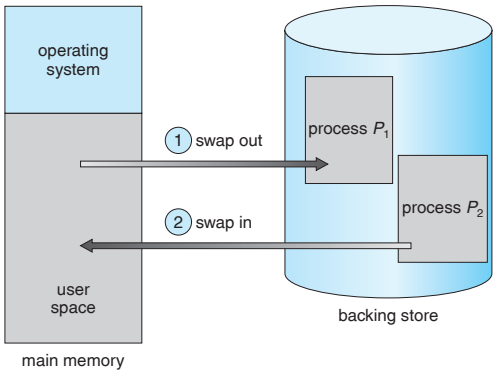
Memory Management Unit



Memory Protection



Swapping



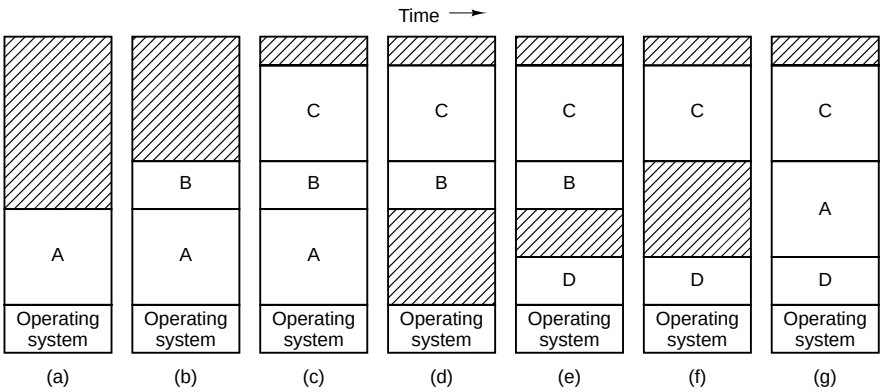
Major part of swap time is transfer time

Total transfer time is directly proportional to the amount of memory swapped

13 Contiguous Memory Allocation

Contiguous Memory Allocation

Multiple-partition allocation



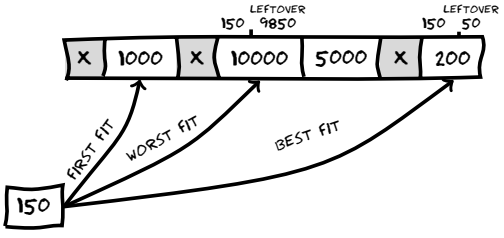
Operating system maintains information about:

a allocated partitions

b free partitions (hole)

Dynamic Storage-Allocation Problem

First Fit, Best Fit, Worst Fit



First-fit: The first hole that is big enough

Best-fit: The smallest hole that is big enough

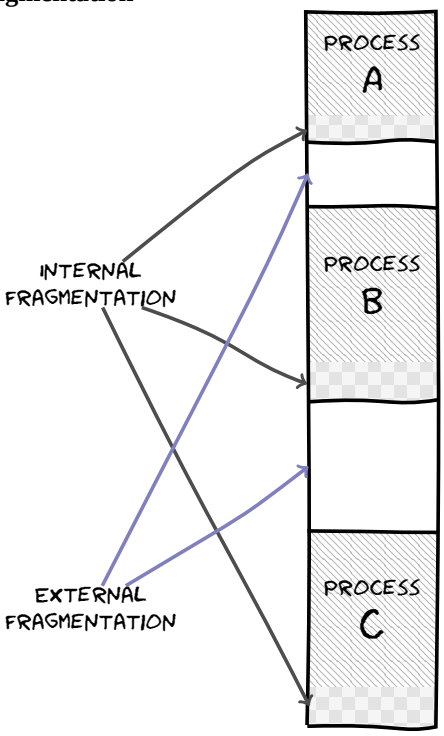
- Must search entire list, unless ordered by size
- Produces the smallest leftover hole

Worst-fit: The largest hole

- Must also search entire list

- Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
- First-fit is generally faster

Fragmentation



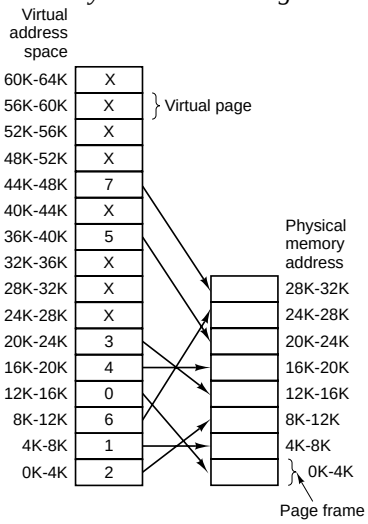
Reduce external fragmentation by

- *Compaction* is possible only if relocation is dynamic, and is done at execution time
- *Noncontiguous memory allocation*
  - Paging
  - Segmentation

14 Virtual Memory

Virtual Memory

Logical memory can be much larger than physical memory



Address translation

$$\underset{address}{virtual} \xrightarrow{page\ table} \underset{address}{physical}$$

$$Page\ 0 \xrightarrow{map\ to} Frame\ 2$$

$$0_{virtual} \xrightarrow{map\ to} 8192_{physical}$$

$$\underset{(20k+20)_{vir}}{20500_{vir}} \xrightarrow{map\ to} \underset{(12k+20)_{phy}}{12308_{phy}}$$

14.1 Paging

Paging

Address Translation Scheme

Address generated by CPU is divided into:

Page number(p): an index into a *page table*

Page offset(d): to be copied into memory

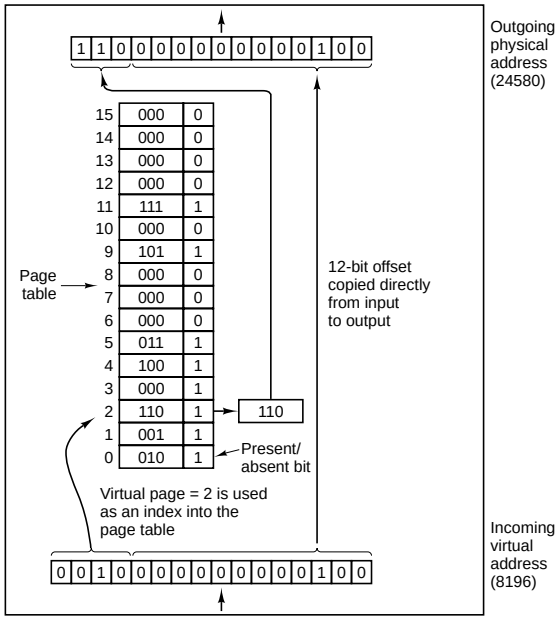
Given *logical address space* (2^m) and *page size* (2^n),

$$\text{number of pages} = \frac{2^m}{2^n} = 2^{m-n}$$

Example: addressing to 0010000000000100

$$\overbrace{\underbrace{0\,0\,1\,0}_{m-n=4}\ \underbrace{0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0}_{n=12}}^{m=16}$$

$$\text{page number} = 0010 = 2, \quad \text{page offset} = 000000000100$$



Virtual pages:	16
Page size:	4k
Virtual memory:	64K
Physical frames:	8
Physical memory:	32K

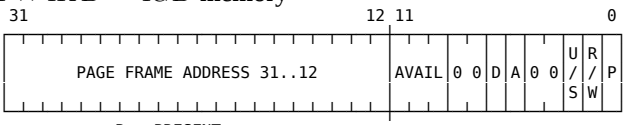
Page Table Entry

Intel i386 Page Table Entry

- Commonly 4 bytes (32 bits) long
- Page size is usually 4k (2^{12} bytes). OS dependent

  `$ getconf PAGESIZE`
- Could have $2^{32-12} = 2^{20} = 1M$ pages

  Could address $1M \times 4KB = 4GB$ memory




```
   P - PRESENT
 R/W - READ/WRITE
 U/S - USER/SUPERVISOR
   A - ACCESSED
   D - DIRTY
AVAIL - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

Page Table

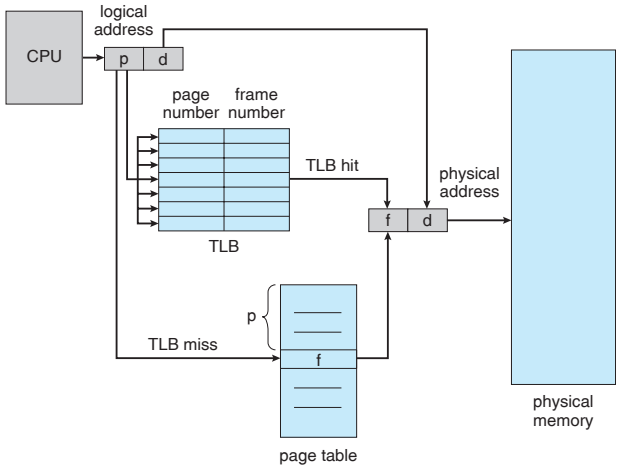
- Page table is kept in main memory
- Usually one page table for each process
- *Page-table base register (PTBR):* A pointer to the page table is stored in PCB
- *Page-table length register (PRLR):* indicates size of the page table
- Slow

– Requires two memory accesses. One for the page table and one for the data/instruction.

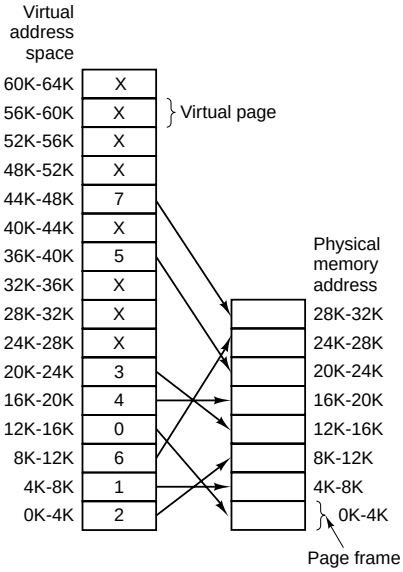
- TLB

Translation Lookaside Buffer (TLB)

80-20 rule Only a small fraction of the PTEs are heavily read; the rest are barely used at all



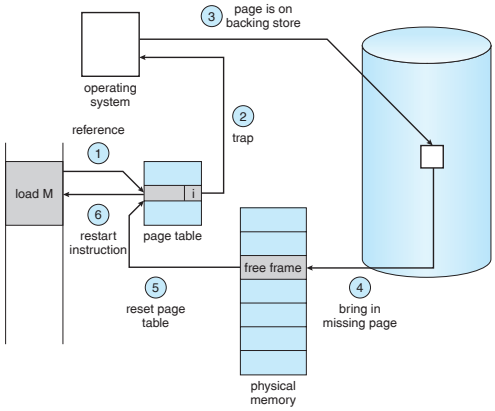
Page Fault



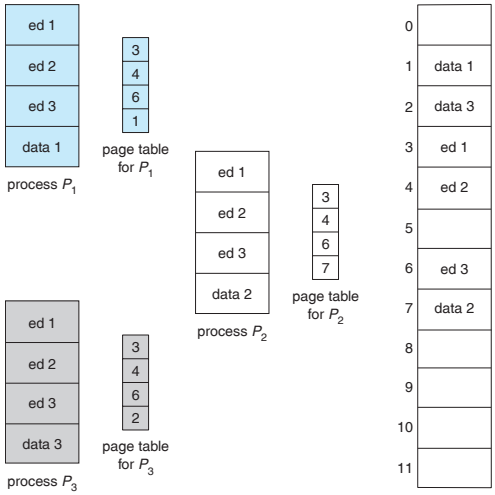
`MOV REG, 32780`?

➠ Page fault & swapping

Page Fault Handling



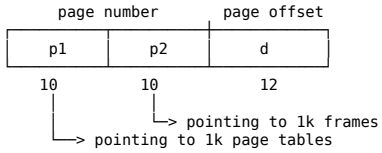
Shared Pages



Multilevel Page Tables

- a $1M$-entry page table eats $4M$ memory
- while 100 processes running, $400M$ memory is gone for page tables
- avoid keeping all the page tables in memory all the time

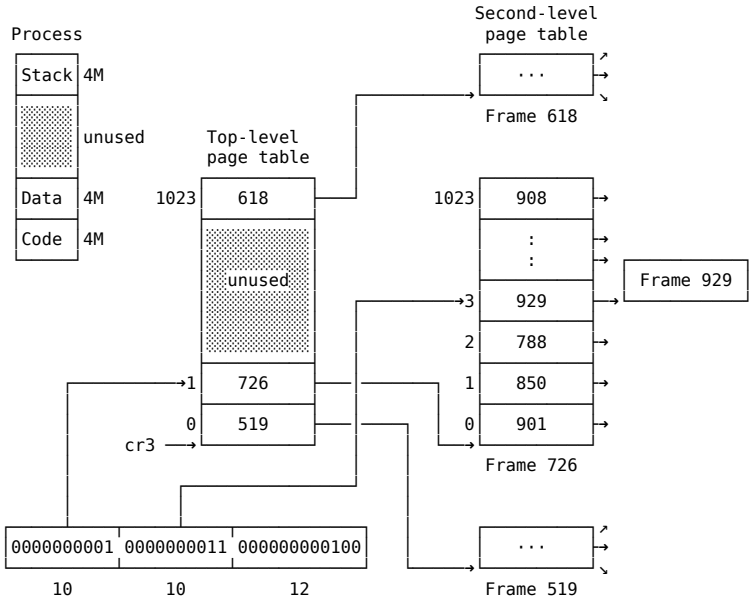
A two-level scheme



p1: is an index into the outer page table
p2: is the displacement within the page of the outer page table

- Split one huge page table into 1k small page tables
  - i.e. the huge page table has 1k entries.
  - Each entry keeps a page frame number of a small page table.
- Each small page table has 1k entries
  - Each entry keeps a page frame number of a physical frame.

Two-Level Page Tables
Example



Problem With 64-bit Systems
Given:

- virtual address space $= 64\,bits$
- page size $= 4\,KB = 2^{12}\,B$

? How much space would a simple single-level page table take?

if Each page table entry takes $4\,Bytes$
then The whole page table (2^{64-12} entries) will take

$$2^{64-12} \times 4\,B = 2^{54}\,B = 16\,PB \quad (peta \Rightarrow tera \Rightarrow giga)!$$

And this is for ONE process!

Multi-level?

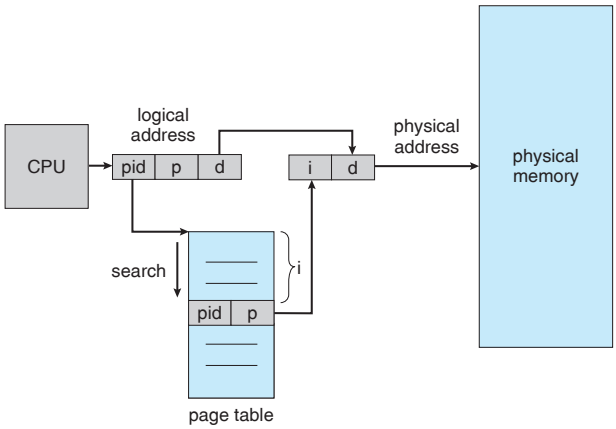
if $10\,bits$ for each level
then $\frac{64-12}{10} = 5$ levels are required
5 memory accress for each address translation!

Inverted Page Tables
Index with frame number

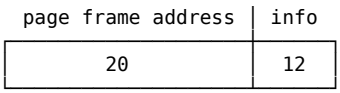
Inverted Page Table:

- One entry for each *physical* frame
  - The physical frame number is the table index
- A single global page table for all processes
  - The table is shared — PID is required
- Physical pages are now mapped to virtual — each entry contains a virtual page number instead of a physical one
- Information bits, e.g. protection bit, are as usual

Inverted Page Tables
Index with frame number
$(pid,\, p) \Rightarrow i$ Find index according to entry contents

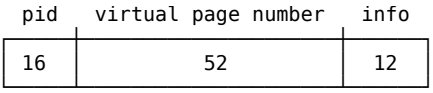


Std. PTE (32-bit sys.):



if $2^{20}\,entries, 4\,B$ each
then $SIZE_{page\,table} = 2^{20} \times 4 = 4\,MB$
(for *each* process)

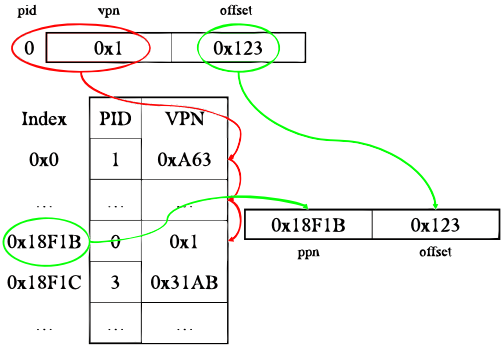
Inverted PTE (64-bit sys.):



if assuming
- 16 bits for PID
- 52 bits for virtual page number
- 12 bits of information

then each entry takes $16 + 52 + 12 = 80\,bits = 10\,bytes$
if physical mem $= 1G\,(2^{30}\,B)$, and page size $= 4K\,(2^{12}\,B)$,
we'll have $2^{30-12} = 2^{18}\,pages$
then $SIZE_{page\,table} = 2^{18} \times 10\,B = 2.5\,MB$
(for *all* processes)

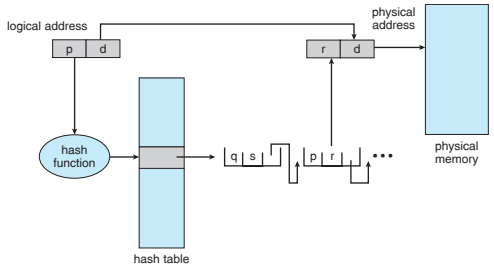
Inefficient: Require searching the entire table



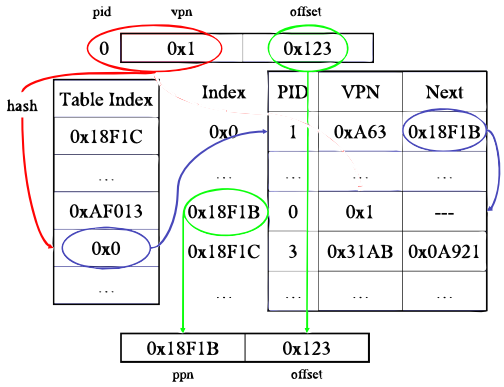
Hashed Inverted Page Tables

A hash anchor table — an extra level before the actual page table

- maps $\overset{\text{process IDs}}{\underset{\text{virtual page numbers}}{}}$ $\Rightarrow$ page table entries
- Since collisions may occur, the page table must do chaining



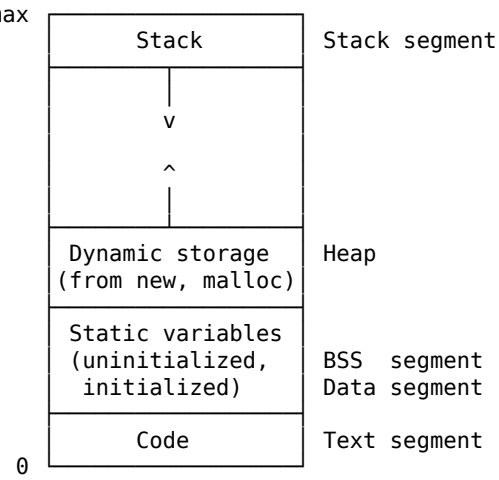
Hashed Inverted Page Table



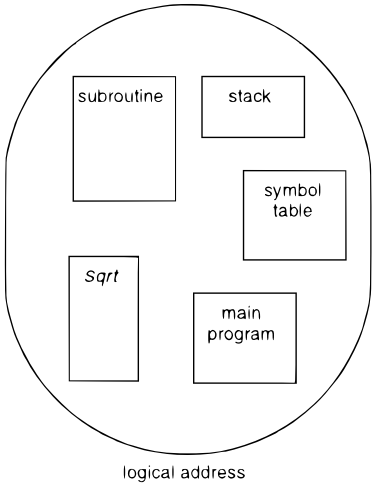
14.2 Segmentation

Two Views of A Virtual Address Space

One-dimensional
a linear array of bytes



Two-dimensional
a collection of variable-sized segments

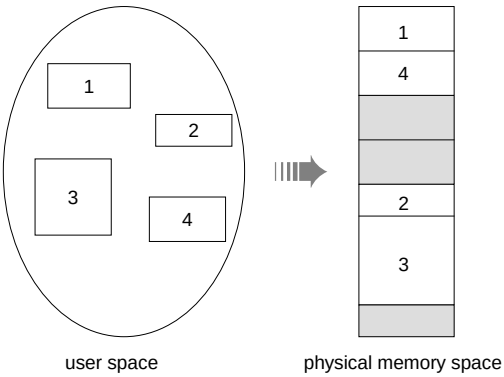


User's View

- A program is a collection of segments
- A segment is a logical unit such as:

main program	procedure	function
method	object	local variables
global variables	common block	stack
symbol table	arrays	

Logical And Physical View of Segmentation



Segmentation Architecture

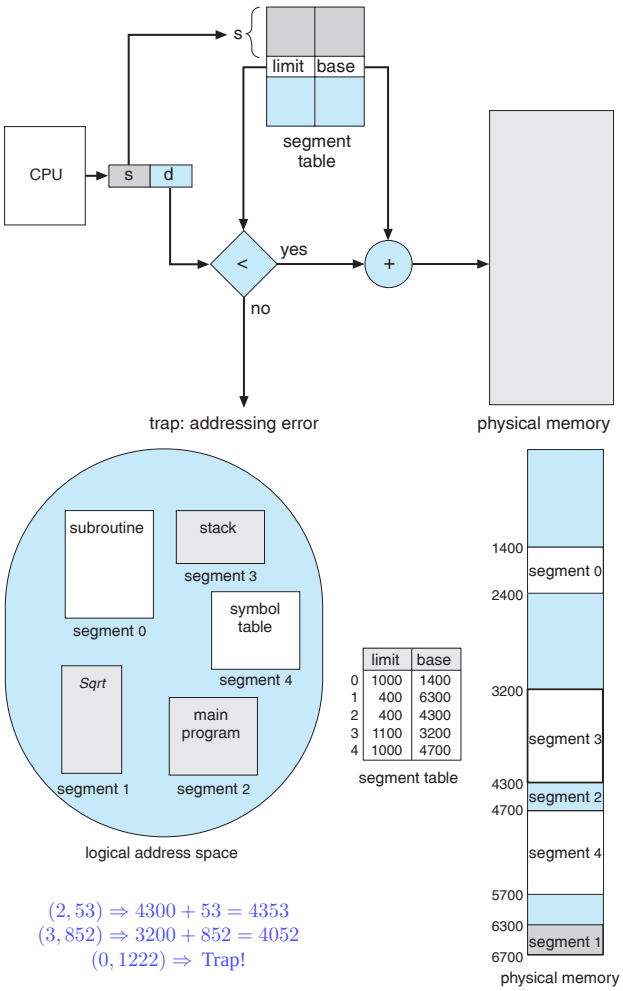
- Logical address consists of a *two tuple*:

  `<segment-number, offset>`

- *Segment table* maps 2D virtual addresses into 1D physical addresses; each table entry has:
  - *base* contains the starting physical address where the segments reside in memory
  - *limit* specifies the length of the segment
- *Segment-table base register (STBR)* ☛ the segment table's location in memory
- *Segment-table length register (STLR)* indicates number of segments used by a program;

  segment number `s` is legal if `s < STLR`

Segmentation hardware



$(2, 53) \Rightarrow 4300 + 53 = 4353$
$(3, 852) \Rightarrow 3200 + 852 = 4052$
$(0, 1222) \Rightarrow$ Trap!

Advantages of Segmentation

- Each segment can be
  - located independently
  - separately protected
  - grow independently
- Segments can be shared between processes

Problems with Segmentation

- Variable allocation
- Difficult to find holes in physical memory
- Must use one of non-trivial placement algorithm
  - first fit, best fit, worst fit
- External fragmentation

See also: http://cseweb.ucsd.edu/classes/fa03/cse120/Lec08.pdf.

Linux prefers paging to segmentation

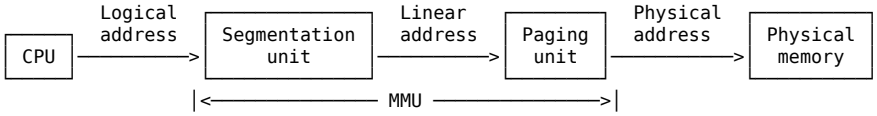
Because

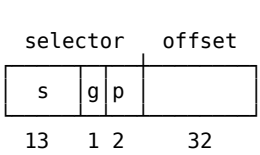
- Segmentation and paging are somewhat redundant
- Memory management is simpler when all processes share the same set of linear addresses
- Maximum portability. RISC architectures in particular have limited support for segmentation

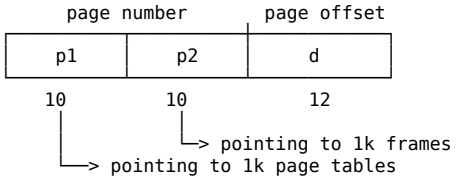
The Linux 2.6 uses segmentation only when required by the 80x86 architecture.

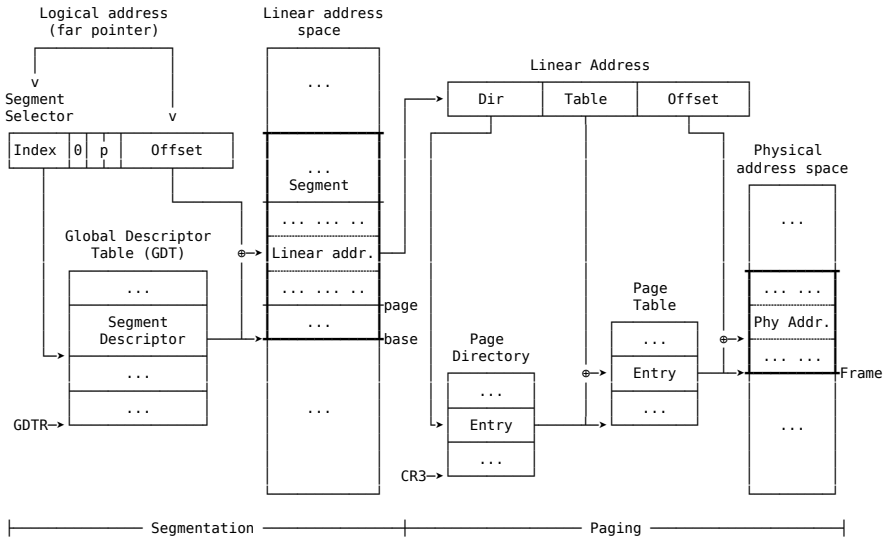
Case Study: The Intel Pentium

Segmentation With Paging



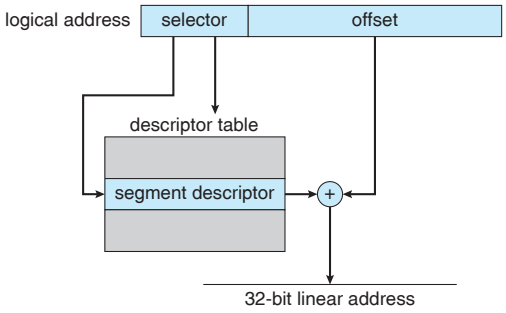






Segmentation

Logical Address ⇒ Linear Address



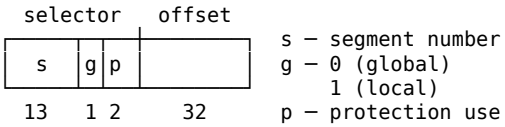
Segment Selectors

A logical address consists of two parts:

segment selector : offset

16 bits 32 bits

Segment selector is an index into GDT/LDT



Segment Descriptor Tables

All the segments are organized in 2 tables:

GDT *Global Descriptor Table*

- shared by all processes
- GDTR stores address and size of the GDT

LDT *Local Descriptor Table*

- one process each
- LDTR stores address and size of the LDT

Segment descriptors are entries in either GDT or LDT, 8-byte long

Analogy

Process	$\Longleftrightarrow$	Process Descriptor(PCB)
File	$\Longleftrightarrow$	Inode
Segment	$\Longleftrightarrow$	Segment Descriptor

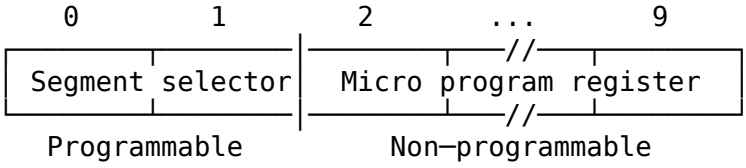
See also:

- Memory Tanslation And Segmentation[10].
- The GDT[11].
- The GDT and IDT[12].

Segment Registers

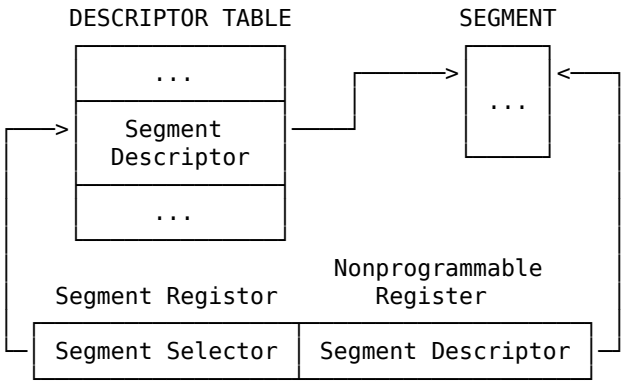
The Intel Pentium has

- *6 segment registers*, allowing 6 segments to be addressed at any one time by a process
  - Each segment register ☛ an entry in LDT/GDT
- *6 8-byte micro program registers* to hold descriptors from either LDT or GDT
  - avoid having to read the descriptor from memory for every memory reference



Fast access to segment descriptors

An additional nonprogrammable register for each segment register



[10] http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation
[11] http://www.osdever.net/bkerndev/Docs/gdt.htm
[12] http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html

Segment registers hold segment selectors

cs code segment register

CPL 2-bit, specifies the Current Privilege Level of the CPU

00 - Kernel mode

11 - User mode

ss stack segment register

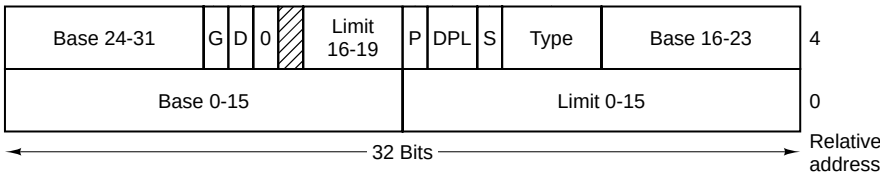
ds data segment register

es/fs/gs general purpose registers, may refer to arbitrary data segments

See also:

- [*INTEL 80386 Programmer's Reference Manual*, Sec. 6.3.2, *Restricting Access to Data*].
- CPU Rings, Privilege, and Proctection[13].

Example: A LDT entry for code segment



Base: Where the seg starts

G: Granularity flag
0 - seg size in bytes
1 - in 4096 bytes

D/B: 0 - 16-bit seg
1 - 32-bit seg

DPL: Descriptor Privilege Level. 0 or 3

AVL: ignored by Linux

Limit: 20 bit, $\Rightarrow 2^{20}$ in size

S: System flag
0 - system seg, e.g. LDT
1 - normal code/data seg

Type: seg type (cs/ds/tss)

P: Seg-Present flag
0 - not in memory
1 - in memory

The Four Main Linux Segments

Every process in Linux has these 4 segments

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xfffff	1	10	3	1	1
user data	0x00000000	1	0xfffff	1	2	3	1	1
kernel code	0x00000000	1	0xfffff	1	10	0	1	1
kernel data	0x00000000	1	0xfffff	1	2	0	1	1

All linear addresses start at 0, end at 4G-1

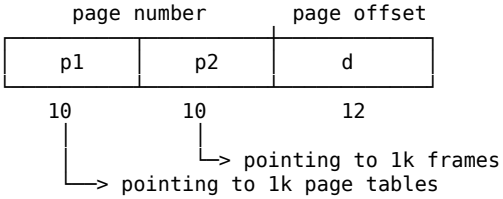
- All processes share the same set of linear addresses
- Logical addresses coincide with linear addresses

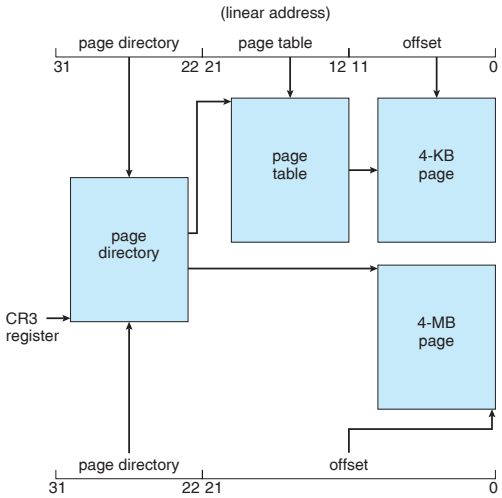
Pentium Paging

Linear Address $\Rightarrow$ *Physical Address*

[13] http://duartes.org/gustavo/blog/post/cpu-rings-privilege-and-protection

Two page size in Pentium:
4K: 2-level paging
4M: 1-level paging

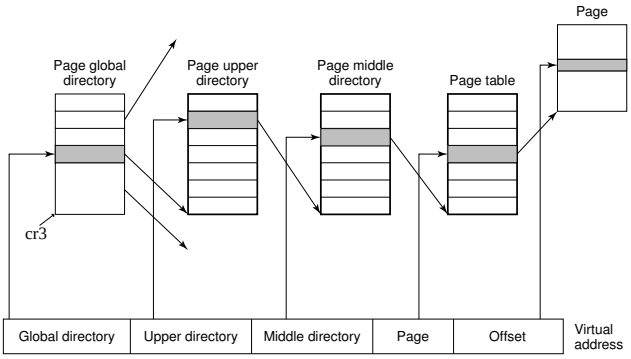




- The CR3 register ☛ the top level page table for the current process.

Paging In Linux

4-level paging for both 32-bit and 64-bit



4-level paging for both 32-bit and 64-bit

- *64-bit: four-level paging*
  1. Page Global Directory
  2. Page Upper Directory
  3. Page Middle Directory
  4. Page Table
- *32-bit: two-level paging*
  1. Page Global Directory
  2. Page Upper Directory — 0 bits; 1 entry
  3. Page Middle Directory — 0 bits; 1 entry
  4. Page Table

The same code can work on 32-bit and 64-bit architectures

Arch	Page size	Address bits	Paging levels	Address splitting
x86	4KB(12bits)	32	2	$10+0+0+10+12$
x86-PAE	4KB(12bits)	32	3	$2+0+9+\ \ 9+12$
x86-64	4KB(12bits)	48	4	$9+9+9+\ \ 9+12$

[1] Wikipedia. *Memory management — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Memory%5C_management&oldid=647917932.

[2] Wikipedia. *Virtual memory — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Virtual%5C_memory&oldid=644430956.

Part IV
File Systems

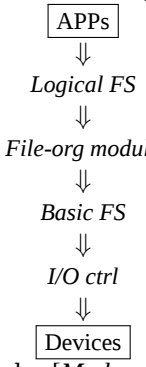
15 File System Structure

Long-term Information Storage Requirements
- Must store large amounts of data
- Information stored must survive the termination of the process using it
- Multiple processes must be able to access the information concurrently

File-System Structure

File-system design addressing two problems:
1. defining how the FS should look to the user
   - defining a file and its attributes
   - the operations allowed on a file
   - directory structure
2. creating algorithms and data structures to map the logical FS onto the physical disk

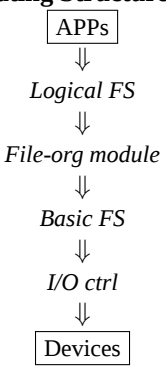
File-System — A Layered Design



- *logical file system* — manages metadata information
  - maintains all of the file-system structure (directory structure, FCB)
  - responsible for protection and security
- *file-organization module*
  - $\text{logical block address} \xrightarrow{translate} \text{physical block address}$
  - keeps track of free blocks
- *basic file system* issues generic commands to device driver, e.g
  - "`read drive 1, cylinder 72, track 2, sector 10`"
- *I/O Control* — device drivers, and INT handlers
  - *device driver:* $\text{high-level commands} \xrightarrow{translate} \text{hardware-specific instructions}$

See also [*Modern Operating Systems*, Sec. 1.3.5, *I/O Devices*].

The Operating Structure



Example — To create a file
1. *APP* calls `creat()`
2. *Logical FS*
   (a) allocates a new FCB
   (b) updates the in-mem dir structure
   (c) writes it back to disk
   (d) calls the file-org module
3. *file-organization module*
   (a) allocates blocks for storing the file's data
   (b) maps the directory I/O into disk-block numbers

Benefit of layered design
The I/O control and sometimes the basic file system code can be used by multiple file systems.

16 Files

File
A Logical View Of Information Storage

User's view

A file is the smallest storage unit on disk.
- Data cannot be written to disk unless they are within a file

UNIX view

Each file is a sequence of 8-bit bytes
- It's up to the application program to interpret this byte stream.

File

What Is Stored In A File?

Source code, object files, executable files, shell scripts, PostScript...

Different type of files have different structure

- UNIX looks at contents to determine type
  Shell scripts start with "`#!`"
  PDF start with "`%PDF...`"
  Executables start with *magic number*
- Windows uses file naming conventions
  executables end with "`.exe`" and "`.com`"
  MS-Word end with "`.doc`"
  MS-Excel end with "`.xls`"

File Naming

Vary from system to system

- Name length?
- Characters? Digits? Special characters?
- Extension?
- Case sensitive?

File Types

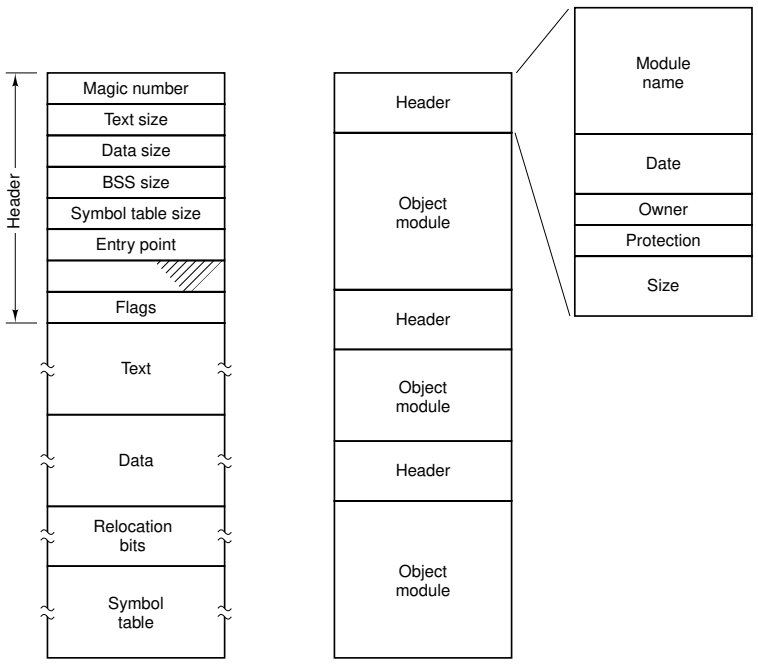
Regular files: ASCII, binary
Directories: Maintaining the structure of the FS

In UNIX, everything is a file

Character special files: I/O related, such as terminals, printers ...
Block special files: Devices that can contain file systems, i.e. disks

disks — logically, linear collections of blocks; disk driver translates them into physical block addresses

Binary files



(a) An Unix executable file (b) An Unix archive file

See also:

- [*Executable and Linkable Format — Wikipedia, The Free Encyclopedia*, *Wikipedia:ELF*].
- OSDev: ELF[14].

File Attributes — Metadata

- *Name* - only information kept in human-readable form
- *Identifier* - unique tag (number) identifies file within file system
- *Type* - needed for systems that support different types
- *Location* - pointer to file location on device
- *Size* - current file size
- *Protection* - controls who can do reading, writing, executing
- *Time, date, and user identification* - data for protection, security, and usage monitoring

File Operations

POSIX file system calls

[14] http://wiki.osdev.org/ELF

```
creat(name, mode)          | read(fd, buffer, byte_count)
open(name, flags)          | write(fd, buffer, byte_count)
close(fd)                  | lseek(fd, offset, whence)
link(oldname, newname)     | chown(name, owner, group
unlink(name)               | fchown(fd, owner, group)
truncate(name, size)       | chmod(name, mode
ftruncate(fd, size)        | fchmod(fd, mode)
stat(name, buffer)         | utimes(name, times)
fstat(fd, buffer)          |
```

write()

```c
#include <unistd.h>

int main(void)
{
  write(1, "Hello, world!\n", 14);

  return 0;
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra write.c -o write" */
/* End: */
```

```
$ man 2 write
$ man 3 write
```

read()

```c
#include <unistd.h>

int main(void)
{
  char buffer[10];

  read(0, buffer, 10);

  write(1, buffer, 10);

  return 0;
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra read.c -o read" */
/* End: */
```

```
$ man 2 read
$ man 3 read
```

- No need to open() STDIN, STDOUT, and STDERR

cp

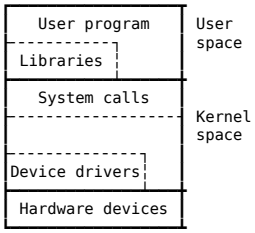
```c
#include <sys/types.h>   /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

#define BUF_SIZE 4096
#define OUTPUT_MODE 0700

int main(int argc, char *argv[])
{
  int in, out, rbytes, wbytes;
  char buf[BUF_SIZE];

  if (argc != 3) exit(1);

  if ( (in = open(argv[1], O_RDONLY)) < 0 ) exit(2);

  if ( (out = creat(argv[2], OUTPUT_MODE)) < 0 ) exit(3);

  while (1) { /* Copy loop */
    if ( (rbytes = read(in, buf, BUF_SIZE)) <= 0 ) break;
    if ( (wbytes = write(out, buf, rbytes)) <= 0 ) exit(4);
  }

```

```
  close(in); close(out);
  if (rbytes == 0) exit(0); /* no error on last read */
  else exit(5);             /* error on last read */
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra cp-syscall.c -o cp-syscall" */
/* End: */
```

`stdio` — The Standard I/O Library
System calls: `open()`, `read()`, `write()`, `close()`...
Library functions: `fopen()`, `fread()`, `fwrite`, `fclose()`...

Avoid calling syscalls directly as much as you can

- Portability
- Buffered I/O



`open()` vs. `fopen()`

`open()`

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
  char c;
  int in;
  in = open("/tmp/1m.test", O_RDONLY);

  while (read(in, &c, 1) == 1);

  return 0;
}


/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra
 ↪  open.c -o /tmp/open" */
/* End: */
```

```
$ strace -c ./open
```

```
$ dd if=/dev/zero of=/tmp/1m.test bs=1k count=1024
```

- https://stackoverflow.com/questions/1658476/c-fopen-vs-open

`fopen()` — Buffered I/O

```
#include <stdio.h>

int main(void)
{
  FILE *stream;

  stream = fopen("/tmp/1m.test", "r");

  while ( fgetc(stream) != EOF );

  fclose(stream);

  return 0;
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra
 ↪  fopen.c -o /tmp/fopen" */
/* End: */
```

```
$ strace -c ./fopen
```

`cp` — With `stdio`

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
```

```
{
  FILE *in, *out;
  int c=0;

  if (argc != 3) exit(1);

  in = fopen(argv[1], "r");
  out = fopen(argv[2], "w");

  while( (c = fgetc(in)) != EOF )
    fputc(c, out);

  return 0;
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra cp-libc.c -o cp-libc" */
/* End: */
```

Try `fread()/fwrite()` instead.

- https://stackoverflow.com/questions/32742430/is-getc-a-macro-or-a-function
- https://stackoverflow.com/questions/9104568/macro-vs-function-in-c

`open()`

`fd open(pathname, flags)`

A per-process *open-file table* is kept in the OS

- upon a successful `open()` syscall, a new entry is added into this table
- indexed by *file descriptor (fd)*

To see files opened by a process, e.g. `init`

```
$ lsof -p 1
```

Why `open()` is needed?

To avoid constant searching

- Without `open()`, every file operation involves searching the directory for the file.

The purpose of the `open()` call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls[*Modern Operating Systems*, Sec. 4.1.6, *File Operations*].

See also:

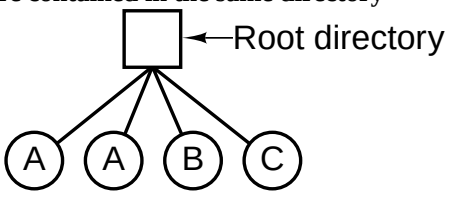
- [*Open (system call) — Wikipedia, The Free Encyclopedia*, *Wikipedia:open() syscall*].
- [*File descriptor — Wikipedia, The Free Encyclopedia*, *Wikipedia:File descriptor*].

17 Directories

Directories

Single-Level Directory Systems

All files are contained in the same directory



- contains 4 files
- owned by 3 different people, A, B, and C

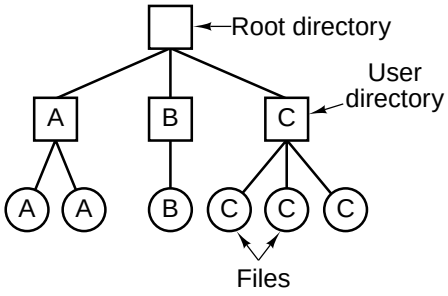
Limitations

- name collision
- file searching

Often used on simple embedded devices, such as telephone, digital cameras...

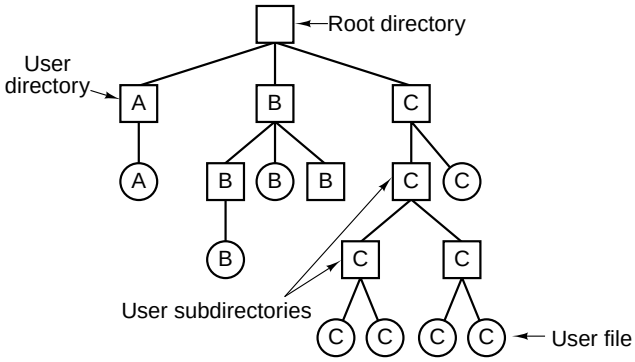
Directories
Two-level Directory Systems

A separate directory for each user

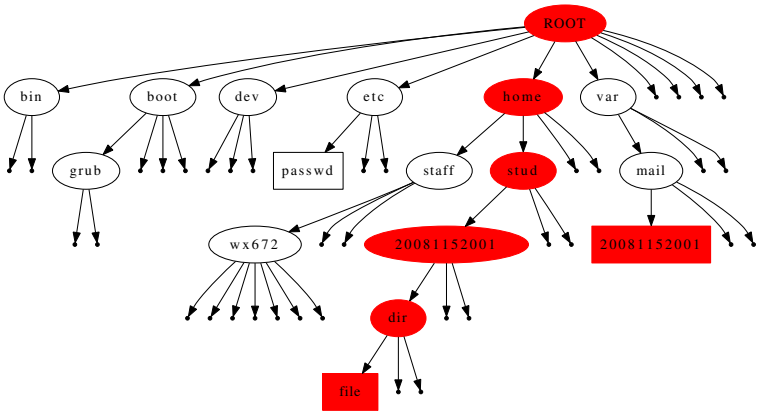


Limitation: hard to access others files

Directories
Hierarchical Directory Systems



Directories
Path Names



Directories
Directory Operations

```
opendir()   mkdir()   rename()   link()
closedir()  rmdir()   readdir()  unlink()
…
```

ls

```c
#include <sys/types.h>
#include <dirent.h>
#include <stddef.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  DIR *dp;
  struct dirent *entry;

  dp = opendir(argv[1]);

  while ( (entry = readdir(dp)) != NULL ){
    printf("%s\n", entry->d_name);
  }

  closedir(dp);

  return 0;
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra ls.c -o
↪  ls" */
/* End: */
```

The real `ls.c`?

- 116 A4 pages
- 5308 lines

Do one thing, and do it really well.

```
$ apt source coreutils
```

mkdir(), chdir(), rmdir(), getcwd()

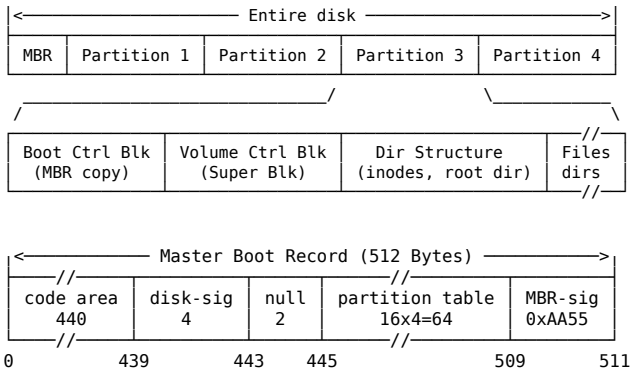
```c
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  char s[100];
  if ( mkdir(argv[1], S_IRUSR|S_IXUSR) == 0 )
    chdir(argv[1]);
  printf("PWD = %s\n", getcwd(s,100));
  rmdir(argv[1]);
  return 0;
}

/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra mkdir.c -o mkdir" */
/* End: */
```

18 File System Implementation

18.1 Basic Structures

File System Implementation

A typical file system layout



MBR, partition table, and booting File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the MBR (Master Boot Record) and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, called the boot block , and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every partition starts with a boot block, even if it does not contain a bootable operating system. Besides, it might contain one in the future, so reserving a boot block is a good idea anyway[*Modern Operating Systems*, Sec 4.3.1, *File System Layout*].

The *superblock* is read into memory when the computer is booted or the file system is first touched.

On-Disk Information Structure

Boot control block a MBR copy

UFS: Boot block

NTFS: Partition boot sector

Volume control block Contains volume details

number of blocks	size of blocks
free-block count	free-block pointers
free FCB count	free FCB pointers

UFS: Superblock

NTFS: Master File Table

Directory structure Organizes the files *FCB, File control block*, contains file details (metadata).

UFS: I-node

NTFS: Stored in MFT using a relational database structure, with one row per file

Each File-System Has a Superblock

Superblock

Keeps information about the file system

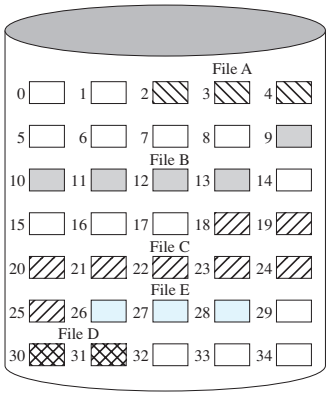
- Type — ext2, ext3, ext4...
- Size
- Status — how it's mounted, free blocks, free inodes, ...
- Information about other metadata structures

```
$ sudo dumpe2fs /dev/sda1 | less
```

18.2 Implementing Files

Implementing Files

Contiguous Allocation

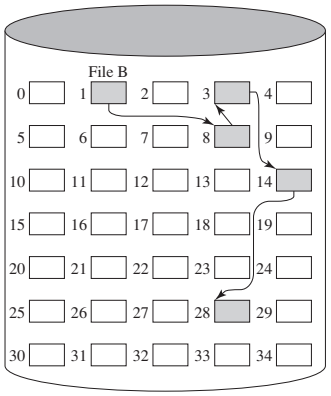


File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

- simple;
- good for read only;
- fragmentation

Linked List (Chained) Allocation A pointer in each disk block

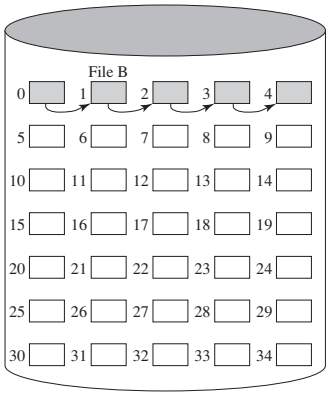


File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •

- no waste block;
- slow random access;
- not 2^n

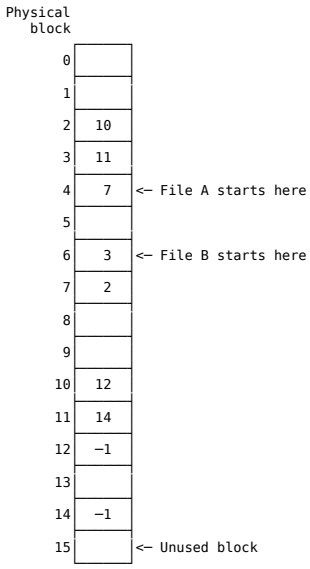
Consolidation One consequence of chaining, as described so far, is that there is *no accommodation of the principle of locality*. Thus, if it is necessary to bring in several blocks of a file at a time, as in sequential processing, then a series of accesses to different parts of the disk are required. This is perhaps a more significant effect on a single-user system but may also be of concern on a shared system. To overcome this problem, some systems periodically consolidate files (fig. 300)[*Operating Systems: Internals and Design Principles*, Sec. 12.7, *Secondary Storage Management*, P. 547]
Linked List (Chained) Allocation Though there is no external fragmentation, consolidation is still preferred.



File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	0	5
• • •	• • •	• • •

FAT: Linked list allocation with a table in RAM

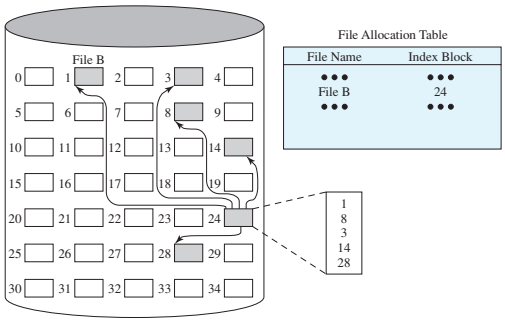


- Taking the pointer out of each disk block, and putting it into a table in memory
- fast random access (chain is in RAM)
- is 2^n
- the entire table must be in RAM

$$disk \nearrow \Rightarrow FAT \nearrow \Rightarrow RAM_{used} \nearrow$$

See also: [*File Allocation Table — Wikipedia, The Free Encyclopedia*, *Wikipedia:FAT*].

Indexed Allocation

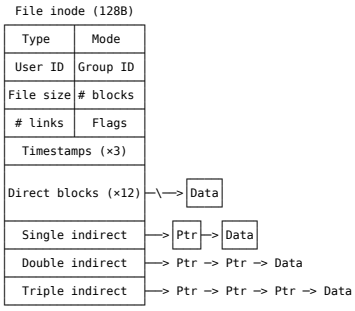


I-node A data structure for each file. An i-node is in memory *only if* the file is open

$$files_{opened} \nearrow \Rightarrow RAM_{used} \nearrow$$

See also: [*Inode — Wikipedia, The Free Encyclopedia*, *Wikipedia:inode*].

I-node — FCB in UNIX



File type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

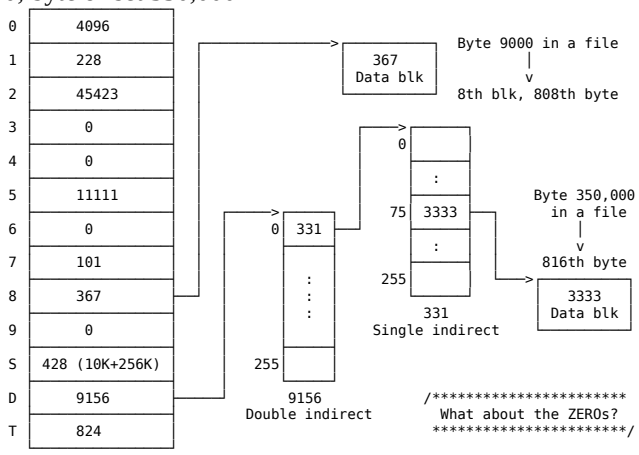
Mode: 9-bit pattern

A little quiz

- in one terminal, to create a file “a”, do:
  `$ echo hello > /tmp/a`
- to track its contents and keep it open, do:
  `$ tail -f /tmp/a`
- in another terminal, delete this file “a”, do:
  `$ rm -f /tmp/a`

- make sure it's gone, do:
  ```
  $ ls -l /tmp/a
  $ ls -li /proc/`pidof tail`/fd
  $ lsof -p `pidof tail` | grep deleted
  ```
  as you can see, `/tmp/a` is marked as "deleted". Now, do:
  ```
  $ echo "another a" >> /tmp/a
  $ ls -li /tmp/a
  ```
  Is this `/tmp/a` same as the deleted one? (check the inodes)

Inode Quiz
Given: block size is 1KB; pointer size is 4B
Addressing: byte offset 9000; byte offset 350,000



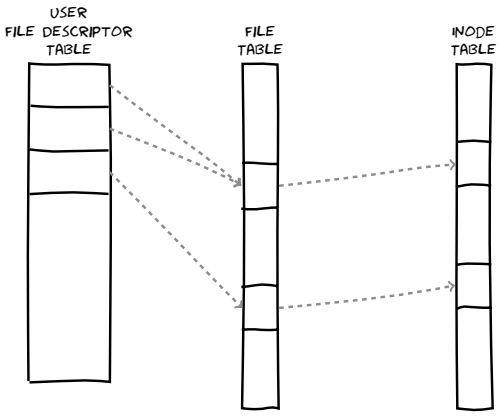
Several block entries in the inode are 0, meaning that the logical block entries contain no data. This happens if no process ever wrote data into the file at any byte offsets corresponding to those blocks and hence the block numbers remain at their initial value, 0. No disk space is wasted for such blocks. Process can cause such a block layout in a file by using the `lseek()` and `write()` system calls[*The design of the UNIX operating system*, Sec. 4.2, *Structure of a Regular File*].

UNIX In-Core Data Structure
mount table — Info about each mounted FS
directory-structure cache — Dir-info of recently accessed dirs
inode table — An in-core version of the on-disk inode table
file table
- global
- keeps inode of each open file
- keeps track of
  - how many processes are associated with each open file
  - where the next read and write will start
  - access rights

user file descriptor table
- per process
- identifies all open files for a process

Find them in the kernel source
user file descriptor table — `struct fdtable` in `include/linux/fdtable.h`
open file table — `struct files_struct` in `include/linux/fdtable.h`
inode — `struct inode` in `include/linux/fs.h`
inode table — ?
superblock — `struct super_block` in `include/linux/fs.h`
dentry — `struct dentry` in `include/linux/dcache.h`
file — `struct file` in `include/linux/fs.h`

UNIX In-Core Data Structure



`open()/creat()`

1. add entry in each table
2. returns a *file descriptor* — an index into the user file descriptor table

Open file descriptor table a global table, whose address is contained in the `files` field of the process descriptor, specifies which files are currently opened by the process. It is a `files_struct` structure whose fields are illustrated in Table 1[*Understanding The Linux Kernel*, Sec. 12.2.6, *Files Associated with a Process*].

Tbl. 1: The fields of the `files_struct` structure

Type	Field	Description
`atomic_t`	`count`	Number of processes sharing this table
`rwlock_t`	`file_lock`	Read/write spin lock for the table fields
`int`	`max_fds`	Current maximum number of file objects
`int`	`max_fdset`	Current maximum number of file descriptors
`int`	`next_fd`	Maximum file descriptors ever allocated plus 1
`struct file **`	`fd`	Pointer to array of file object pointers
`fd_set *`	`close_on_exec`	Pointer to file descriptors to be closed on `exec()`
`fd_set *`	`open_fds`	Pointer to open file descriptors
`fd_set`	`close_on_exec_init`	Initial set of file descriptors to be closed on `exec()`
`fd_set`	`open_fds_init`	Initial set of file descriptors
`struct file *[]`	`fd_array`	Initial array of file object pointers

The `fd` field points to an array of pointers to file objects. The size of the array is stored in the `max_fds` field. Usually, `fd` points to the `fd_array` field of the `files_struct` structure, which includes 32 file object pointers. If the process opens more than 32 files, the kernel allocates a new, larger array of file pointers and stores its address in the `fd` fields; it also updates the `max_fds` field.

For every file with an entry in the `fd` array, the array index is the file descriptor. Usually, the first element (index 0) of the array is associated with the standard input of the process, the second with the standard output, and the third with the standard error (See fig. 12-3[15]). Unix processes use the file descriptor as the main file identifier. Notice that, thanks to the `dup()` , `dup2()` , and `fcntl()` system calls, two file descriptors may refer to the same opened file, that is, two elements of the array could point to the same file object. Users see this all the time when they use shell constructs such as `2>&1` to redirect the standard error to the standard output.

`open()` A call to `open()` creates a new *open file description*, an entry in the system-wide table of open files. This entry records the file offset and the file status flags (modifiable via the `fcntl(2)` `F_SETFL` operation). *A file descriptor is a reference to one of these entries; this reference is unaffected if `pathname` is subsequently removed or modified to refer to a different file. The new* open file description *is initially not shared with any other process, but sharing may arise via `fork(2)` [`man 2 open`].*

[15] In book Understanding The Linux Kernel

The internal representation of a file is given by an *inode*, which contains a description of the disk layout of the file data and other information such as the file owner, access permissions, and access times. The term inode is a contraction of the term index node and is commonly used in literature on the UNIX system. Every file has one inode, but it may have several names, all of which map into the inode. Each name is called a *link*. When a process refers to a file by name, the kernel parses the file name one component at a time, checks that the process has permission to search the directories in the path, and eventually retrieves the inode for the file. For example, if a process calls

```
1 open("/fs2/mjb/rje/sourcefile",1);
```

the kernel retrieves the inode for "`/fs2/mjb/rje/sourcefile`". When a process creates a new file, the kernel assigns it an unused inode. Inodes are stored in the file system, as will be seen shortly, but the kernel reads them into an in-core inode table when manipulating files.

The kernel contains two other data structures, the *file table* and the *user file descriptor table*. The file table is a global kernel structure, but the user file descriptor table is allocated per process. When a process `opens` or `creats` a file, the kernel allocates an entry from each table, corresponding to the file's inode. *Entries in the three structures — user file descriptor table, file table, and inode table — maintain the state of the file and the user's access to it.* The file table keeps track of the byte offset in the file where the user's next `read` or `write` will start, and the access rights allowed to the `open`ing process. The user file descriptor table identifies all open files for a process. Fig. 306 shows the tables and their relationship to each other. The kernel returns a *file descriptor* for the `open` and `creat` system calls, which is an index into the user file descriptor table. When executing `read` and `write` system calls, the kernel uses the file descriptor to access the user fie descriptor table, follows pointers to the file table and inode table entries, and, from the inode, finds the data in the file. Chapters 4 and 5 describe these data structures in great detail. For now, suffice it to say that use of three tables allows various degrees of sharing access to a file[*The design of the UNIX operating system*, Sec. 2.2.1, *An Overview of the File Subsystem*].

The `open` system call is the first step a process must take to access the data in a file. The syntax for the `open` system call is

```
1 fd = open(pathname, flags, modes);
```

where *pathname* is a file name, *flags* indicate the type of open (such as for reading or writing), and *modes* give the file permissions if the file is being created. The `open` system call returns an integer called the user *file descriptor*. Other file operations, such as reading, writing, seeking, duplicating the file descriptor, setting file I/O parameters, determining file status, and closing the file, use the file descriptor that the `open` system call returns[*The design of the UNIX operating system*, Sec. 5.1, *Open*].

The kernel searches the file system for the file name parameter using algorithm *namei* (see fig. 2). It checks permissions for opening the file after it finds the in-core inode and allocates an entry in the file table for the open file. The file table entry contains a pointer to the inode of the open file and a field that indicates the byte offset in the file where the kernel expects the next `read` or `write` to begin. The kernel initializes the offset to 0 during the `open` call, meaning that the initial `read` or `write` starts at the beginning of a file by default. Alternatively, a process can `open` a file in *write-append* mode, in which case the kernel initializes the offset to the size of the file. The kernel allocates an entry in a private table in the process *u area*, called the user file descriptor table, and notes the index of this entry. The index is the file descriptor that is returned to the user. The entry in the user file table points to the entry in the global file table.

Q1: Can `open()` return an inode number?

Q2: Can we keep the I/O pointers in the inode? Possibly adding a few pointers in the inode data structure in a similar fashion of those block pointers (direct/single-indirect/double-indirect/triple-indirect). Each pointer pointing to an I/O pointer record.

The Tables

Two levels of internal tables in the OS

A per-process table tracks all files that a process has open. Stores

- the current-file-position pointer (not really)
- access rights
- more...

a.k.a file descriptor table

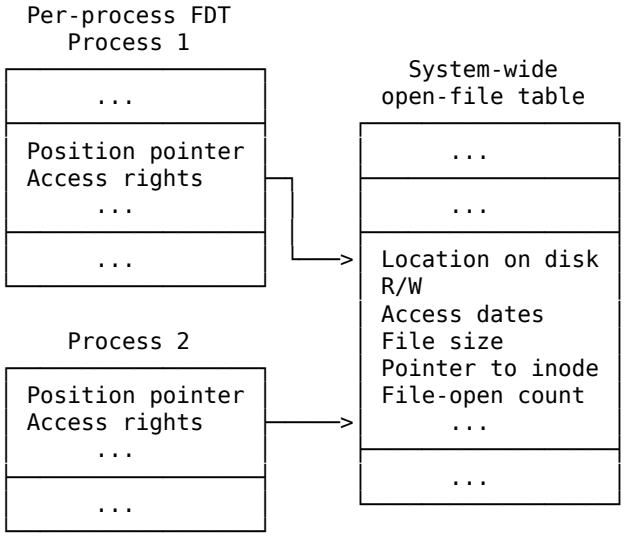
A system-wide table keeps process-independent information, such as

- the location of the file on disk
- access dates

```
algorithm open
inputs: file name
        type of open
        file permissions (for creation type of open)
output: file descriptor
{
   convert file name to inode (algorithm namei);
   if (file does not exist or not permitted access)
          return (error);
   allocate file table entry for inode, initialize count, offset;
   allocate user file descriptor entry, set pointer to file table entry;
   if (type of open specifies truncate file)
         free all file blocks (algorithm free);
   unlock (inode); /* locked above in namei */
   return (user file descriptor);
}
```

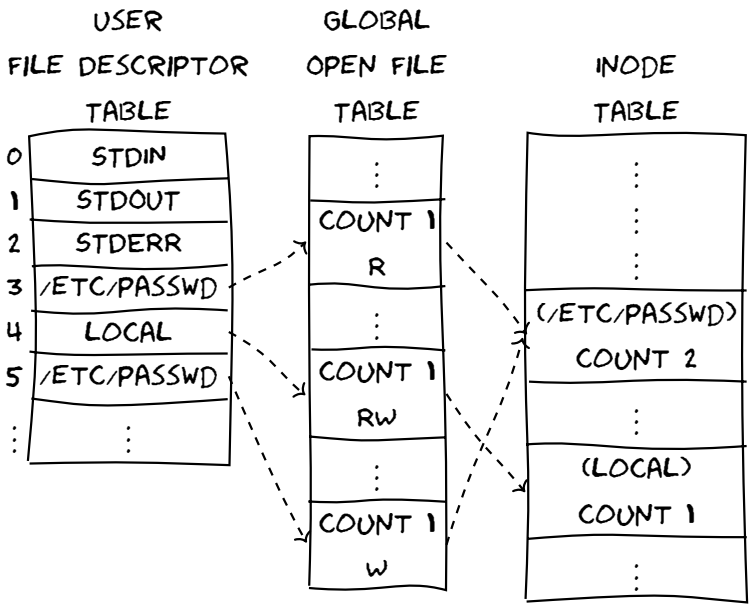
Fig. 2: Algorithm for opening a file

- file size
- file open count — the number of processes opening this file



A process executes the following code:

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("local", O_RDWR);
fd3 = open("/etc/passwd", O_WRONLY);
```

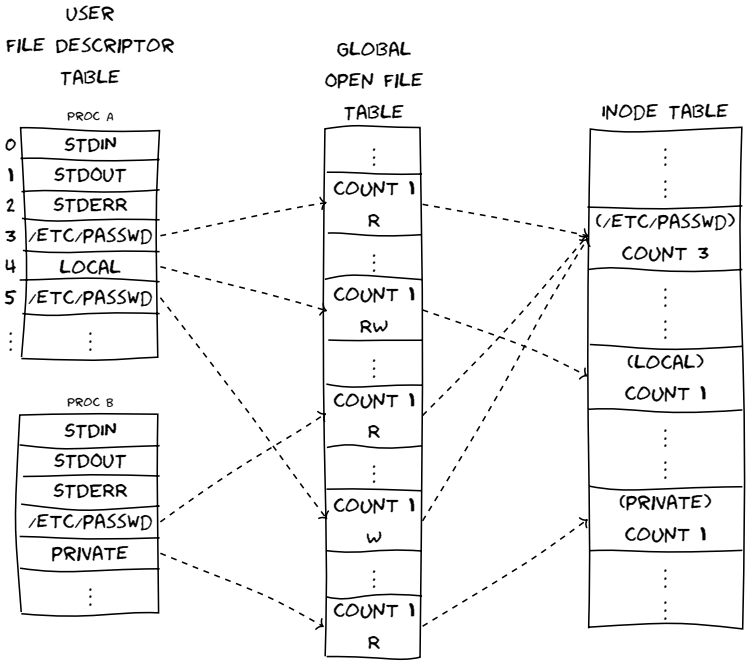


See also:

- [*The design of the UNIX operating system*, Sec. 5.1, *Open*].
- File descriptor manipulation[16].

One more process B:

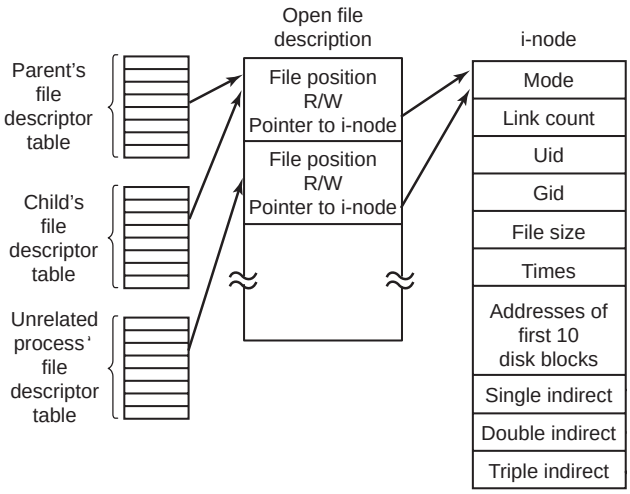
```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("private", O_RDONLY);
```



Why File Table?

To allow a parent and child to share a file position, but to provide unrelated processes with their own values.

[16] http://www.cim.mcgill.ca/~franco/OpSys-304-427/lecture-notes/node27.html#SECTION00063000000000000000



Why File Table?

Where To Put File Position Info?
Inode table? No. Multiple processes can open the same file. Each one has its own file position.
User file descriptor table? No. Trouble in file sharing.

Example

```
1 #!/bin/bash
2
3 echo hello
4 echo world
```

```
$ ./hello.sh > A
```

? Where should the "`world`" be?

Why file table?

File system implementation With file sharing, it is necessary to allow related processes to share a common I/O pointer and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that share the same open file (the result of `forks`) share a common open file table entry. A separate open of the same file will only share the i-node table entry, but will have distinct open file table entries[“Unix Implementation”, Sec. 4.1].

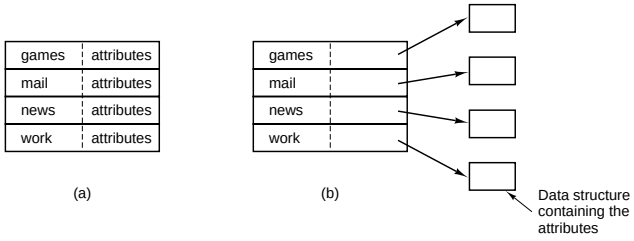
Open The user file descriptor table entry could conceivably contain the file offset for the position of the next I/O operation and point directly to the in-core inode entry for the file, eliminating the need for a separate kernel file table. The examples above show a one-to-one relationship between user file descriptor entries and kernel file table entries. Thompson notes, however, that he implemented the file table as a separate structure to allow sharing of the offset pointer between several user file descriptors[“Unix Implementation”, *Thompson 78, p. 1943*]. The `dup` and `fork` system calls, explained in [*The design of the UNIX operating system*, Sec. 5.13] and [*The design of the UNIX operating system*, Sec. 7.1], manipulate the data structures to allow such sharing [*The design of the UNIX operating system*, Sec. 5.1].

The Linux File System ... The idea is to start with this file descriptor and end up with the corresponding i-node. Let us consider one possible design: just put a pointer to the i-node in the file descriptor table. Although simple, unfortunately this method does not work. The problem is as follows. Associated with every file descriptor is a file position that tells at which byte the next read (or write) will start. Where should it go? One possibility is to put it in the i-node table. However, this approach fails if two or more unrelated processes happen to open the same file at the same time because each one has its own file position[*Modern Operating Systems*, Sec. 10.6].
A second possibility is to put the file position in the file descriptor table. In that way, every process that opens a file gets its own private file position. Unfortunately this scheme fails too, but the reasoning is more subtle and has to do with the nature of file sharing in Linux. Consider a shell script, `s`, consisting of two commands, `p1` and `p2`, to be run in order. If the shell script is called by the command line
`S >X`
it is expected that `p1` will write its output to `x`, and then `p2` will write its output to `x` also, starting at the place where `p1` stopped.

When the shell forks off `p1`, `x` is initially empty, so `p1` just starts writing at file position 0. However, when `p1` finishes, some mechanism is needed to make sure that the initial file position that `p2` sees is not 0 (which it would be if the file position were kept in the file descriptor table), but the value `p1` ended with.
The way this is achieved is shown in Fig 311. The trick is to introduce a new table, the *open file description table*, between the file descriptor table and the i-node table, and put the file position (and read/write bit) there. In this figure, the parent is the shell and the child is first `p1` and later `p2`. When the shell forks off `p1`, its user structure (including the file descriptor table) is an exact copy of the shell's, so both of them point to the same open file description table entry. When `p1` finishes, the shell's file descriptor is still pointing to the open file description containing p1's file position. When the shell now forks off `p2`, the new child automatically inherits the file position, without either it or the shell even having to know what that position is.

18.3 Implementing Directories

Implementing Directories



(a) A simple directory (Windows)
  – fixed size entries
  – disk addresses and attributes in directory entry
(b) Directory in which each entry just refers to an i-node (UNIX)

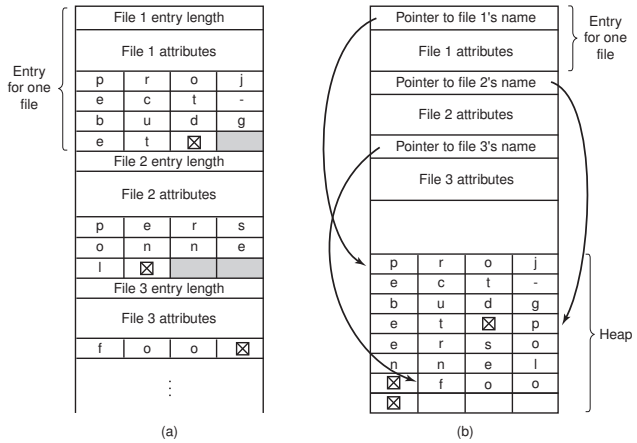
The maximum possible size for a file on a FAT32 volume is 4 GiB minus 1 byte or 4,294,967,295 ($2^{32}-1$) bytes. This limit is a consequence of the file length entry in the directory table and would also affect huge FAT16 partitions with a sufficient sector size[*File Allocation Table — Wikipedia, The Free Encyclopedia*].

Directory entry in `glibc`

```
struct dirent {
  ino_t          d_ino;       /* Inode number */
  off_t          d_off;       /* Not an offset; see below */
  unsigned short d_reclen;    /* Length of this record */
  unsigned char  d_type;      /* Type of file; not supported
                                 by all filesystem types */
  char           d_name[256]; /* Null-terminated filename */
};
```

```
$ man readdir
$ view /usr/include/x86_64-linux-gnu/bits/dirent.h
```

How Long A File Name Can Be?

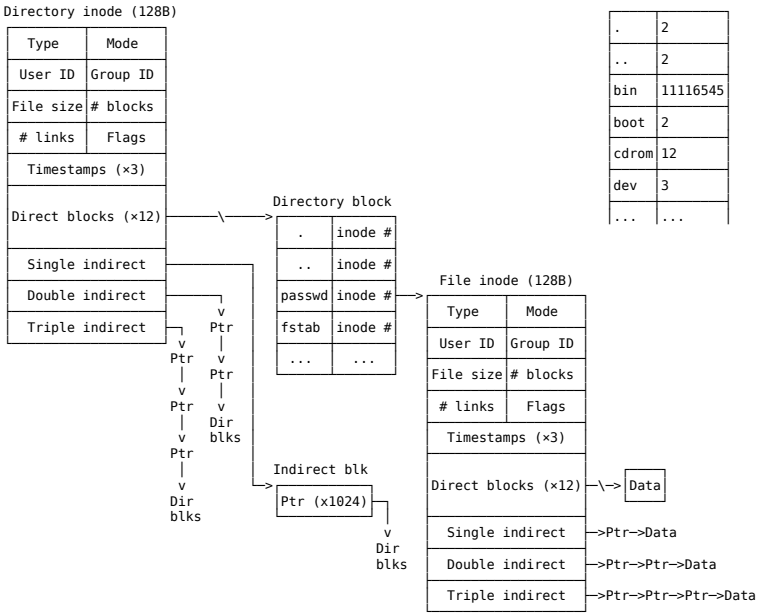


How long file name is implemented?

1. The simplest approach is to set a limit on file name length, typically 255 characters, and then use one of the designs of fig. 313 with 255 characters reserved for each file name. This approach is simple, but wastes a great deal of directory space, since few files have such long names. For efficiency reasons, a different structure is desirable.
2. One alternative is to give up the idea that all directory entries are the same size. With this method, each directory entry contains a fixed portion, typically starting with the length of the entry, and then followed by data with a fixed format, usually including the owner, creation time, protection information, and other attributes. This fixed-length header is followed by the actual file name, however long it may be, as shown in fig. 315(a) in big-endian format (e.g., SPARC). In this example we have three files, project-budget, personnel, and foo. Each file name is terminated by a special character (usually 0), which is represented in the figure by a box with a cross in it. To allow each directory entry to begin on a word boundary, each file name is filled out to an integral number of words, shown by shaded boxes in the figure.
   A disadvantage of this method is that when a file is removed, a variable-sized gap is introduced into the directory into which the next file to be entered may not fit. This problem is the same one we saw with contiguous disk files, only now compacting the directory is feasible because it is entirely in memory. Another problem is that a single directory entry may span multiple pages, so a page fault may occur while reading a file name.
3. Another way to handle variable-length names is to make the directory entries themselves all fixed length and keep the file names together in a heap at the end of the directory, as shown in fig. 315(b). This method has the advantage that when an entry is removed, the next file entered will always fit there. Of course, the heap must be managed and page faults can still occur while processing file names. One minor win here is that there is no longer any real need for file names to begin at word boundaries, so no filler characters are needed after file names in fig. 315(b) as they are in fig. 315(a).
4. Ext2's approach is a bit different. See Sec. 19.5.

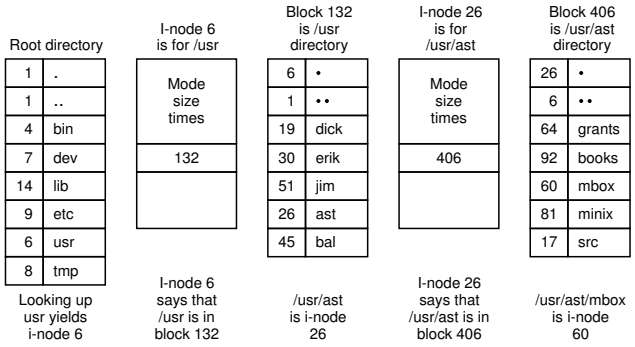
See also: [*Modern Operating Systems*, Sec. 4.3.3, *Implementating Directories*].

UNIX Treats a Directory as a File



- A directory is a file whose data is a sequence of entries, each consisting of an inode number and the name of a file contained in the directory.
- Each (disk) block in a directory file consists of a linked list of entries; each entry contains the length of the entry, the name of a file, and the inode number of the inode to which that entry refers[*Operating System Concepts Essentials*, Sec. 15.7.2, *The Linux ext2fs File System*].

The steps in looking up `/usr/ast/mbox`



- First the file system locates the root directory. In UNIX its i-node is located at a fixed place on the disk. From this i-node, it locates the root directory, which can be anywhere on the disk, but say block 1[*Modern Operating Systems*, Sec. 4.5, *Example File Systems*].
  Then it reads the root directory and looks up the first component of the path, `usr`, in the root directory to find the i-node number of the file `/usr`. Locating an i-node from its number is straightforward, since each one has a fixed location on the disk. From this i-node, the system locates the directory for `/usr` and looks up the next component, `ast`, in it. When it has found the entry for `ast`, it has the i-node for the directory `/usr/ast`. From this i-node it can find the directory itself and look up `mbox`. The i-node for this file is then read into memory and kept there until the file is closed. The lookup process is illustrated in fig. 317.
  Relative path names are looked up the same way as absolute ones, only starting from the working directory instead of starting from the root directory. Every directory has entries for `.` and `..` which are put there when the directory is created. The entry `.` has the i-node number for the current directory, and the entry for `..` has the i-node number for the parent directory. Thus, a procedure looking up `../dick/prog.c` simply looks up `..` in the working directory, finds the i-node number for the parent directory, and searches that directory for `dick`. No special mechanism is needed to handle these names. As far as the directory system is concerned, they are just ordinary ASCII strings, just the same as any other names. The only bit of trickery here is that `..` in the root directory points to itself.

18.4 Shared Files

File Sharing
Multiple Users
User IDs identify users, allowing permissions and protections to be per-user
Group IDs allow users to be in groups, permitting group access rights

Example: 9-bit pattern
`rwxr-x---` means:

user	group	other
rwx	r-x	---
111	1-1	000
7	5	0

File Sharing
Remote File Systems

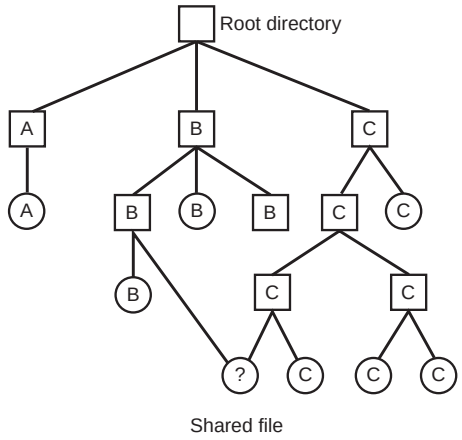
Networking — allows file system access between systems
- Manually via programs like FTP
- Automatically, seamlessly using distributed file systems
- Semi automatically, via the world wide web

C/S model — allows clients to *mount* remote file systems from servers
- NFS — standard UNIX client-server file sharing protocol
- CIFS — standard Windows protocol
- Standard system calls are translated into remote calls

Distributed Information Systems (distributed naming services)
- such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing

File Sharing
Protection
- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

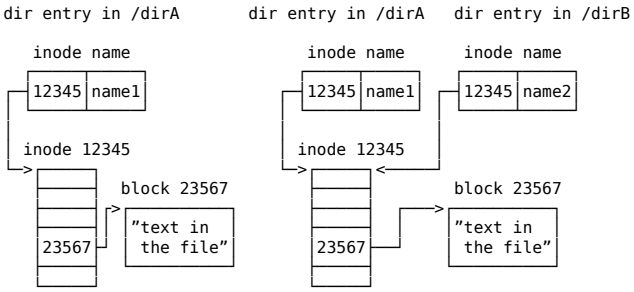
Shared Files
Hard Links vs. Soft Links



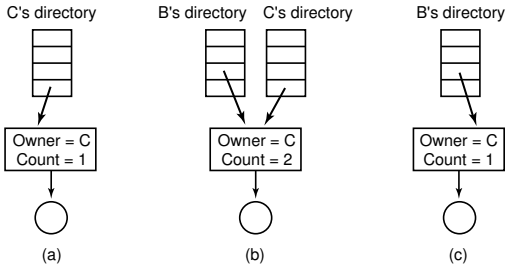
See also: [*Directed acyclic graph — Wikipedia, The Free Encyclopedia*, *Wikipedia:dag*].

Hard Links

Hard links ☛ the same inode



Drawback



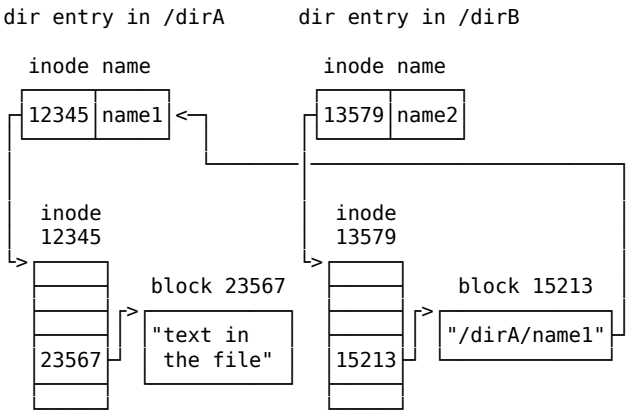
- Both hard and soft links have drawbacks[*Modern Operating Systems*, Sec. 4.3.4, *Shared Files*].

```
# echo 0 > /proc/sys/fs/protected_hardlinks
```

- Why hard links not allowed to directories in UNIX/Linux[17]?
  ... if you were allowed to do this for directories, two different directories in different points in the filesystem could point to the same thing. In fact, a subdir could point back to its grandparent, creating a loop.
  Why is this loop a concern? Because when you are traversing, there is no way to detect you are looping (without keeping track of inode numbers as you traverse). Imagine you are writing the `du` command, which needs to recurse through subdirs to find out about disk usage. How would `du` know when it hit a loop? It is error prone and a lot of bookkeeping that `du` would have to do, just to pull off this simple task.
- The Ultimate Linux Soft and Hard Link Guide (10 `Ln` Command Examples)[18]

Symbolic Links

A symbolic link has its own inode ☛ a directory entry



[17] http://unix.stackexchange.com/questions/22394/why-hard-links-not-allowed-to-directories-in-unix-linux
[18] http://www.thegeekstuff.com/2010/10/linux-ln-command-examples/

link(), unlink(), symlink()

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* link(argv[1], argv[2]); */
  /* symlink(argv[1], argv[2]); */
  unlink(argv[1]);
  perror(argv[0]);
  return 0;
}


/* Local Variables: */
/* compile-command: "gcc -Wall -Wextra link.c -o ln" */
/* End: */
```

18.5 Disk Space Management

Disk Space Management

Statistics

Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69

Length	VU 1984	VU 2005	Web
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

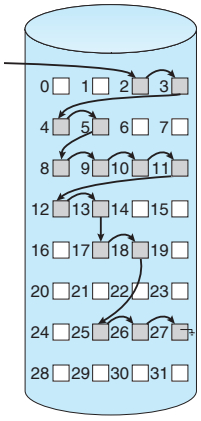
See also: [*Modern Operating Systems*, Sec. 4.4.1, *Disk Space Management*].

- Block size is chosen while creating the FS
- Disk I/O performance is conflict with space utilization
  - smaller block size $\Rightarrow$ better space utilization
  - larger block size $\Rightarrow$ better disk I/O performance

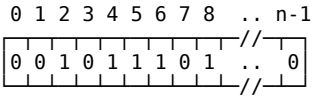
  `$ dumpe2fs /dev/sda1 | grep "Block size"`

Keeping Track of Free Blocks

1. Linked List



2. Bit map (n blocks)



$$bit[i] = \begin{cases} 0 \Rightarrow block[i]\ is\ free \\ 1 \Rightarrow block[i]\ is\ occupied \end{cases}$$

Journaling File Systems

Operations required to remove a file in UNIX:

1. Remove the file from its directory
   - set inode number to 0
2. Release the i-node to the pool of free i-nodes
   - clear the bit in inode bitmap
3. Return all the disk blocks to the pool of free disk blocks
   - clear the bits in block bitmap

What if crash occurs between 1 and 2, or between 2 and 3?

Suppose that the first step is completed and then the system crashes. The i-node and file blocks will not be accessible from any file, but will also not be available for reassignment; they are just off in limbo somewhere, decreasing the available resources. If the crash occurs after the second step, only the blocks are lost.

If the order of operations is changed and the i-node is released first, then after rebooting, the i-node may be reassigned, but the old directory entry will continue to point to it, hence to the wrong file. If the blocks are released first, then a crash before the i-node is cleared will mean that a valid directory entry points to an i-node listing blocks now in the free storage pool and which are likely to be reused shortly, leading to two or more files randomly sharing the same blocks. None of these outcomes are good[*Modern Operating Systems*, Sec. 4.3.6, *Journaling File Systems*].

See also: [*Operating System Concepts Essentials*, Sec. 15.7.3, *Journaling*].

Journaling File Systems

Keep a log of what the file system is going to do before it does it

- so that if the system crashes before it can do its planned work, upon rebooting the system can look in the log to see what was going on at the time of the crash and finish the job.
- NTFS, EXT3/4, and ReiserFS use journaling among others

19 Ext2 File System

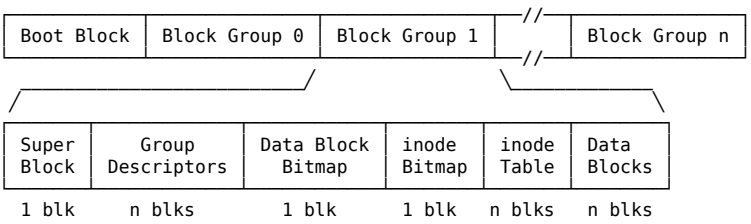
References:

- [*The Second Extended File System Internal Layout*, *The Second Extented File System*].
- [*Operating System Concepts Essentials*, Sec. 15.7, *File Systems*].
- [*The Linux Kernel*, Chap. 9, *The File System*].
- ["Design and Implementation of the Second Extended Filesystem", *Design and Implementation of the Second Extended Filesystem*].
- [*Analyzing a filesystem*, *Analyzing a filesystem*].

19.1 Ext2 File System Layout

Ext2 File System

Physical Layout



See also: [*Operating System Concepts Essentials*, Sec. 15.7.2, *The Linux ext2fs File System*].

19.2 Ext2 Block groups

Ext2 Block groups

The partition is divided into *Block Groups*

- Block groups are same size — easy locating
- Kernel tries to keep a file's data blocks in the same block group — reduce fragmentation
- Backup critical info in each block group
- The Ext2 inodes for each block group are kept in the *inode table*
- The *inode-bitmap* keeps track of allocated and unallocated inodes

  When allocating a file, ext2fs must first select the block group for that file[*Operating System Concepts Essentials*, Sec. 15.7.2, *The Linux ext2fs File System*, P. 625].

  - For data blocks, it attempts to allocate the file to the block group to which the file's inode has been allocated.
  - For inode allocations, it selects the block group in which the file's parent directory resides, for nondirectory files.
  - Directory files are not kept together but rather are dispersed throughout the available block groups.

  These policies are designed not only to keep related information within the same block group but also to spread out the disk load among the disk's block groups to reduce the fragmentation of any one area of the disk.

Group descriptor

- Each block group has a group descriptor
- All the group descriptors together make the *group descriptor table*
- The table is stored along with the superblock
- *Block Bitmap:* tracks free blocks
- *Inode Bitmap:* tracks free inodes
- *Inode Table:* all inodes in this block group
- *Free blocks count*, *Free Inodes count*, *Used dir count* } counters

```
# dumpe2fs /dev/sda1
```

Maths

Given $\begin{matrix} block\,size = 4K \\ block\,bitmap = 1\,blk \end{matrix}$ **, then**

$$blocks\,per\,group = 8\,bits \times 4K = 32K$$

How large is a group?

$$group\,size = 32K \times 4K = 128\,M$$

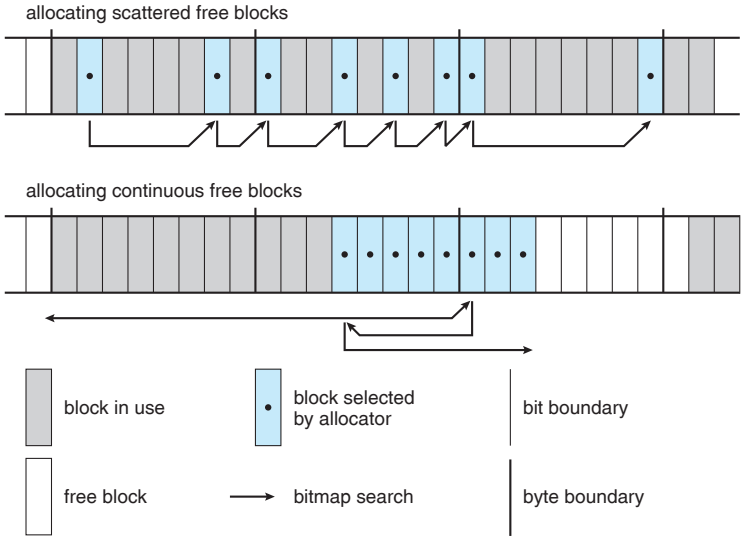
How many block groups are there?

$$\approx \frac{partition\,size}{group\,size} = \frac{partition\,size}{128M}$$

How many files can I have in max?

$$\approx \frac{partition\,size}{block\,size} = \frac{partition\,size}{4K}$$

Ext2 Block Allocation Policies



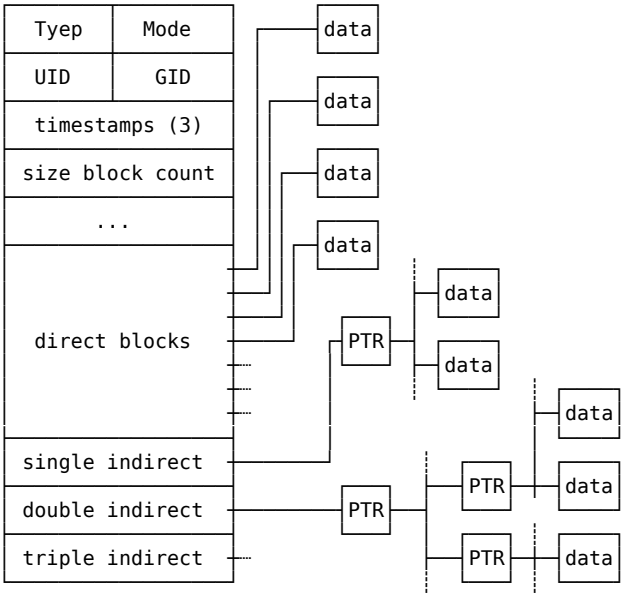
More about block bitmap It maintains a bitmap of all free blocks in a block group. When allocating the first blocks for a new file, it starts searching for a free block from the beginning of the block group; when extending a file, it continues the search from the block most recently allocated to the file. The search is performed in two stages. First, ext2fs searches for an entire free byte in the bitmap; if it fails to find one, it looks for any free bit. The search for free bytes aims to allocate disk space in chunks of at least eight blocks where possible[*Operating System Concepts Essentials*, Sec. 15.7.2, *The Linux ext2fs File System*].

Once a free block has been identified, the search is extended backward until an allocated block is encountered. When a free byte is found in the bitmap, this backward extension prevents ext2fs from leaving a hole between the most recently allocated block in the previous nonzero byte and the zero byte found. Once the next block to be allocated has been found by either bit or byte search, ext2fs extends the allocation forward for up to eight blocks and *preallocates* these extra blocks to the file. This preallocation helps to reduce fragmentation during interleaved writes to separate files and also reduces the CPU cost of disk allocation by allocating multiple blocks simultaneously. *The preallocated blocks are returned to the free-space bitmap when the file is closed*.

Fig. 335 illustrates the allocation policies. Each row represents a sequence of set and unset bits in an allocation bitmap, indicating used and free blocks on disk. In the first case, if we can find any free blocks sufficiently near the start of the search, then we allocate them no matter how fragmented they may be. The fragmentation is partially compensated for by the fact that the blocks are close together and can probably all be read without any disk seeks, and allocating them all to one file is better in the long run than allocating isolated blocks to separate files once large free areas become scarce on disk. In the second case, we have not immediately found a free block close by, so we search forward for an entire free byte in the bitmap. If we allocated that byte as a whole, we would end up creating a fragmented area of free space between it and the allocation preceding it, so before allocating we back up to make this allocation flush with the allocation preceding it, and then we allocate forward to satisfy the default allocation of eight blocks.

19.3 Ext2 Inode

Ext2 inode



Ext2 inode
Mode: holds two pieces of information

1. Is it a {*file*|*dir*|*sym-link*|*blk-dev*|*char-dev*|*FIFO*}?
2. Permissions

Owner info: Owners' ID of this file or directory
Size: The size of the file in bytes
Timestamps: Accessed, created, last modified time
Datablocks: 15 pointers to data blocks ($12 + S + D + T$)

Max File Size
Given:

$$\begin{cases} block\,size = 4K \\ pointer\,size = 4B \end{cases},$$

We get:

$$Max\,File\,Size = number\,of\,pointers \times block\,size$$

$$= (\overbrace{\underbrace{12}_{direct} + \underbrace{1K}_{1-indirect} + \underbrace{1K \times 1K}_{2-indirect} + \underbrace{1K \times 1K \times 1K}_{3-indirect}}^{number\,of\,pointers}) \times 4K$$

$$= 48K + 4M + 4G + 4T$$

19.4 Ext2 Superblock

Ext2 Superblock
Magic Number: `0xEF53`
Revision Level: determines what new features are available
Mount Count and Maximum Mount Count: determines if the system should be fully checked
Block Group Number: indicates the block group holding this superblock
Block Size: usually $4K$
Blocks per Group: $8bits \times block\,size$
Free Blocks: System-wide free blocks
Free Inodes: System-wide free inodes
First Inode: First inode number in the file system

See more:

```
# dumpe2fs /dev/sda1
```

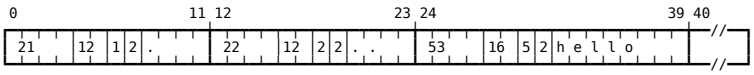
Ext2 File Types

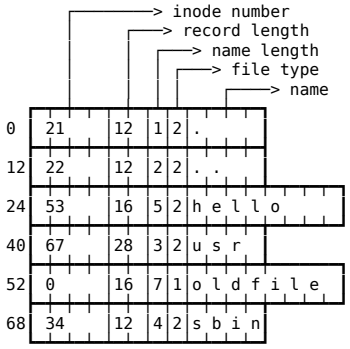
File type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

Device file, pipe, and socket: No data blocks are required. All info is stored in the inode
Fast symbolic link: Short path name ($< 60\,chars$) needs no data block. Can be stored in the 15 pointer fields

19.5 Ext2 Directory

Ext2 Directories





- Directories are special files
- "." and ".." first
- Padding to $4\times$
- inode number is 0 — deleted file

Directory files are stored on disk just like normal files, although their contents are interpreted differently. Each block in a directory file consists of a linked list of entries; each entry contains the length of the entry, the name of a file, and the inode number of the inode to which that entry refers[*Operating System Concepts Essentials*, Sec. 15.7.2, *The Linux ext2fs File System*].

20 Vitural File Systems

Many different FS are in use

Windows

uses drive letter (`C:`, `D:`, ...) to identify each FS

UNIX

integrates multiple FS into a single structure

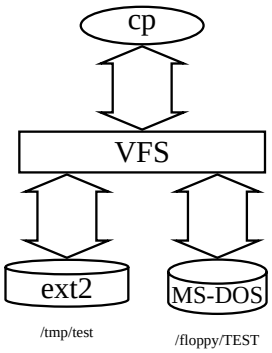
- From user's view, there is only one FS hierarchy

```
$ man fs
```

Windows handles these disparate file systems by identifying each one with a different drive letter, as in `C:`, `D:`, etc. When a process opens a file, the drive letter is explicitly or implicitly present so Windows knows which file system to pass the request to. There is no attempt to integrate heterogeneous file systems into a unified whole[*Modern Operating Systems*, Sec. 4.3.7, *Virtual File Systems*].

A filesystem is a hierarchical storage of data adhering to a specific structure. Filesystems contain files, directories, and associated control information.Typical operations performed on filesystems are creation, deletion, and mounting. In Unix, filesystems are mounted at a specific mount point in a global hierarchy known as a *namespace*. This enables all mounted filesystems to appear as entries in a single tree. Contrast this single, unified tree with the behavior of DOS and Windows, which break the file namespace up into drive letters, such as `C:`. This breaks the namespace up among device and partition boundaries, "leaking" hardware details into the filesystem abstraction. As this delineation may be arbitrary and even confusing to the user, it is inferior to Linux's unified namespace[*Linux Kernel Development*, Sec. 13.3, *Unix Filesystems*].

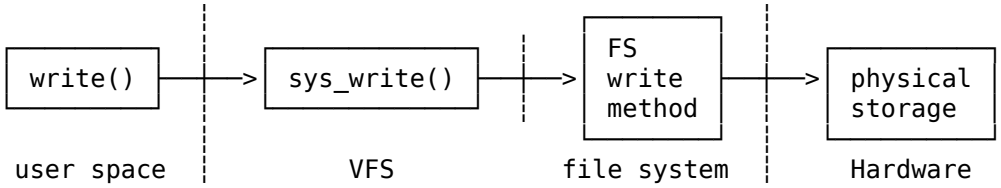
```
$ cp /floppy/TEST /tmp/test
```




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
            O_WRONLY|O_CREAT|O_TRUNC, 0600);
do{
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

Where `/floppy` is the mount point of an MS-DOS diskette and `/tmp` is a normal Second Extended Filesystem (Ext2) directory. The VFS is an abstraction layer between the application program and the filesystem implementations. Therefore, the `cp` program is not required to know the filesystem types of `/floppy/TEST` and `/tmp/test`. Instead, `cp` interacts with the VFS by means of generic system calls known to anyone who has done Unix programming[*Understanding The Linux Kernel*, Sec. 12.1].

```
ret = write(fd, buf, len);
```

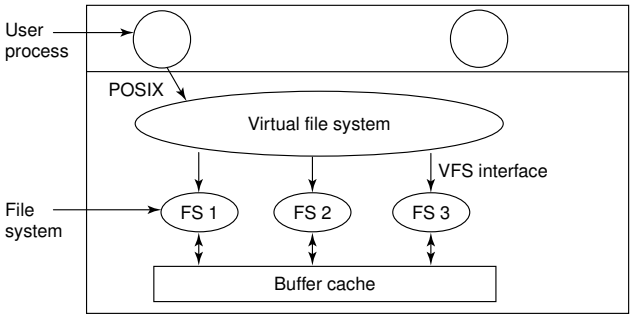


This system call writes the `len` bytes pointed to by `buf` into the current position in the file represented by the file descriptor `fd`. This system call is first handled by a generic `sys_write()` system call that determines the actual file writing method for the filesystem on which `fd` resides. The generic `write` system call then invokes this method, which is part of the filesystem implementation, to write the data to the media (or whatever this filesystem does on write). Fig. 344 shows the flow from user-space's `write()` call through the data arriving on the physical media. On one side of the system call is the generic VFS interface, providing the frontend to user-space; on the other side of the system call is the filesystem-specific backend, dealing with the implementation details[*Linux Kernel Development*, Sec. 13.2, P. 262].

Virtural File Systems

Put common parts of all FS in a separate layer

- It's a layer in the kernel
- It's a common interface to several kinds of file systems
- It calls the underlying concrete FS to actual manage the data



To the VFS layer and the rest of the kernel, however, each filesystem looks the same. They all support notions such as files and directories, and they all support operations such as creating and deleting files. ... In fact, nothing in the kernel needs to understand the underlying details of the filesystems, except the filesystems themselves[*Linux Kernel Development*, Sec. 13.2, *Filesystem Abstraction Layer*].

The key idea is to abstract out that part of the file system that is common to all file systems and put that code in a separate layer that calls the underlying concrete file systems to actual manage the data[*Modern Operating Systems*, Sec. 4.3.7, *Virtual File Systems*]. The overall structure is illustrated in Fig 345.

All system calls relating to files are directed to the virtual file system for initial processing. These calls, coming from user processes, are the standard POSIX calls, such as `open`, `read`, `write`, `lseek`, and so on. Thus the VFS has an "upper" interface to user processes and it is the well-known POSIX interface.

The VFS also has a "lower" interface to the concrete file systems, which is labeled VFS interface in fig. 345 . This interface consists of several dozen function calls that the VFS can make to each file system to get work done. *Thus to create a new file system that works with the VFS, the designers of the new file system must make sure that it supplies the function calls the VFS requires*.

Virtual File System

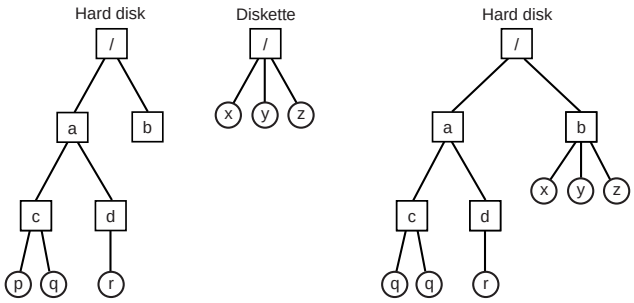
- Manages kernel level file abstractions in one format for all file systems
- Receives system call requests from user level (e.g. `write`, `open`, `stat`, `link`)
- Interacts with a specific file system based on mount point traversal
- Receives requests from other parts of the kernel, mostly from memory management

Real File Systems

- managing file & directory data
- managing meta-data: timestamps, owners, protection, etc.
- disk data, NFS data... $\xleftrightarrow{translate}$ VFS data

Historically, Unix has provided four basic filesystem-related abstractions: files, directory entries, inodes, and mount points. ... Traditionally, Unix filesystems implement these notions as part of their physical on-disk layout. For example, file information is stored as an inode in a separate block on the disk; directories are files; control information is stored centrally in a superblock, and so on. The Unix file concepts are physically mapped on to the storage medium. The Linux VFS is designed to work with filesystems that understand and implement such concepts. Non-Unix filesystems, such as FAT or NTFS, still work in Linux, but their filesystem code must provide the appearance of these concepts. For example, *even if a filesystem does not support distinct inodes, it must assemble the inode data structure in memory as if it did*. Or if a filesystem treats directories as a special object, to the VFS they must represent directories as mere files. *Often, this involves some special processing done on-the-fly by the non-Unix filesystems* to cope with the Unix paradigm and the requirements of the VFS. Such filesystems still work, however, and the overhead is not unreasonable[*Linux Kernel Development*, Sec. 13.3, *Unix Filesystems*].

File System Mounting



A FS must be mounted before it can be used

Mount — The file system is registered with the VFS

- The superblock is read into the VFS superblock
- The table of addresses of functions the VFS requires is read into the VFS superblock
- The FS' topology info is mapped onto the VFS superblock data structure

The VFS keeps a list of the mounted file systems together with their superblocks

The VFS superblock contains:

- Device, blocksize
- Pointer to the *root inode*
- Pointer to a set of superblock routines
- Pointer to `file_system_type` data structure
- more...

See also: [*Linux Kernel Development*, Sec. 13.13, *Data Structures Associated with Filesystems*].

- `struct file_system_type`: There is only one `file_system_type` per filesystem, regardless of how many instances of the filesystem are mounted on the system, or whether the filesystem is even mounted at all.
- `struct vfsmount`: represents a specific instance of a filesystem — in other words, a mount point.

V-node

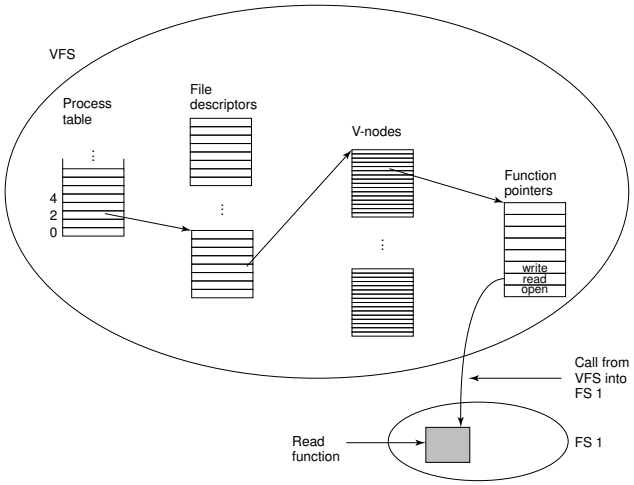
- Every file/directory in the VFS has a VFS inode, kept in the VFS *inode cache*
- The real FS builds the VFS inode from its own info

Like the EXT2 inodes, the VFS inodes describe

- files and directories within the system
- the contents and topology of the Virtual File System

VFS Operation

`read()`



To understand how the VFS works, let us run through an example chronologically. When the system is booted, the root file system is registered with the VFS. In addition, when other file systems are mounted, either at boot time or

during operation, they, too must register with the VFS. *When a file system registers, what it basically does is provide a list of the addresses of the functions the VFS requires, either as one long call vector (table) or as several of them, one per VFS object, as the VFS demands*. Thus once a file system has registered with the VFS, the VFS knows how to, say, read a block from it — it simply calls the fourth (or whatever) function in the vector supplied by the file system. Similarly, the VFS then also knows how to carry out every other function the concrete file system must supply: it just calls the function whose address was supplied when the file system registered[*Modern Operating Systems*, Sec. 4.3.7, *Virtual File Systems*, P. 288].

After a file system has been mounted, it can be used. For example, if a file system has been mounted on `/usr` and a process makes the call

```
1 open("/usr/include/unistd.h", O_RDONLY);
```

while parsing the path, the VFS sees that a new file system has been mounted on `/usr` and locates its superblock by searching the list of superblocks of mounted file systems. Having done this, it can find the root directory of the mounted file system and look up the path `include/unistd.h` there. The VFS then creates a v-node and makes a call to the concrete file system to return all the information in the file's i-node. This information is copied into the v-node (in RAM), along with other information, most importantly the pointer to the table of functions to call for operations on v-nodes, such as `read`, `write`, `close`, and so on.

After the v-node has been created, the VFS makes an entry in the file descriptor table for the calling process and sets it to point to the new v-node. (For the purists, the file descriptor actually points to another data structure that contains the current file position and a pointer to the v-node, but this detail is not important for our purposes here.) Finally, the VFS returns the file descriptor to the caller so it can use it to `read`, `write`, and `close` the file.

Later when the process does a `read` using the file descriptor, the VFS locates the v-node from the process and file descriptor tables and follows the pointer to the table of functions, all of which are addresses within the concrete file system on which the requested file resides. The function that handles `read` is now called and code within the concrete file system goes and gets the requested block. *The VFS has no idea whether the data are coming from the local disk, a remote file system over the network, a CD-ROM, a USB stick, or something different*. The data structures involved are shown in Fig 350. Starting with the caller's process number and the file descriptor, successively the v-node, read function pointer, and access function within the concrete file system are located.

In this manner, it becomes relatively straightforward to add new file systems. To make one, the designers first get a list of function calls the VFS expects and then write their file system to provide all of them. Alternatively, if the file system already exists, then they have to provide wrapper functions that do what the VFS needs, usually by making one or more native calls to the concrete file system.

See also: [*Linux Kernel Development*, Sec. 13.14, *Data Structures Associated with a Process*].

Linux VFS

The Common File Model

All other filesystems must map their own concepts into the common file model

For example, FAT filesystems do not have inodes.

- The main components of the common file model are
  superblock – information about mounted filesystem
  inode – information about a specific file
  file – information about an open file
  dentry – information about directory entry
- Geared toward Unix FS

Dentry Note that because the VFS treats directories as normal files, there is not a specific directory object. Recall from earlier in this chapter that a dentry represents a component in a path, which might include a regular file. In other words, a dentry is not the same as a directory, but a directory is just another kind of file. Got it?[*Linux Kernel Development*, Sec. 13.4, *VFS Objects and Their Data Structures*]

The operations objects An operations object is contained within each of these primary objects.These objects describe the methods that the kernel invokes against the primary objects[*Linux Kernel Development*, Sec. 13.4, *VFS Objects and Their Data Structures*]:

- The `super_operations` object, which contains the methods that the kernel can invoke on a specific filesystem, such as `write_inode()` and `sync_fs()`

- The `inode_operations` object, which contains the methods that the kernel can invoke on a specific file, such as `create()` and `link()`
- The `dentry_operations` object, which contains the methods that the kernel can invoke on a specific directory entry, such as `d_compare()` and `d_delete()`
- The `file_operations` object, which contains the methods that a process can invoke on an open file, such as `read()` and `write()`

The operations objects are implemented as a structure of pointers to functions that operate on the parent object. For many methods, the objects can inherit a generic function if basic functionality is sufficient. Otherwise, the specific instance of the particular filesystem fills in the pointers with its own filesystem-specific methods.

The Superblock Object

- is implemented by each FS and is used to store information describing that specific FS
- usually corresponds to the *filesystem superblock* or the *filesystem control block*
- Filesystems that are not disk-based (such as `sysfs`, `proc`) generate the superblock on-the-fly and store it in memory
- `struct super_block` in `<linux/fs.h>`
- `s_op` in `struct super_block`☛`struct super_operations` — the superblock operations table
  - Each item in this table is a pointer to a function that operates on a superblock object

The code for creating, managing, and destroying superblock objects lives in `fs/super.c`. A superblock object is created and initialized via the `alloc_super()` function. When mounted, a filesystem invokes this function, reads its superblock off of the disk, and fills in its superblock object[*Linux Kernel Development*, Sec. 13.5, *The Superblock Object*].

When a filesystem needs to perform an operation on its superblock, it follows the pointers from its superblock object to the desired method. For example, if a filesystem wanted to write to its superblock, it would invoke

```
sb->s_op->write_super(sb);
```

In this call, `sb` is a pointer to the filesystem's superblock. Following that pointer into `s_op` yields the superblock operations table and ultimately the desired `write_super()` function, which is then invoked. Note how the `write_super()` call must be passed a superblock, despite the method being associated with one. This is because of the lack of object-oriented support in C. In C++, a call such as the following would suffice:

```
sb.write_super();
```

In C, there is no way for the method to easily obtain its parent, so you have to pass it[*Linux Kernel Development*, Sec. 13.6, *Superblock Operations*].

The Inode Object

- For Unix-style filesystems, this information is simply read from the on-disk inode
- For others, the inode object is constructed in memory in whatever manner is applicable to the filesystem
- `struct inode` in `<linux/fs.h>`
- An `inode` represents each file on a FS, but the `inode object` is constructed in memory only as files are accessed
  - includes special files, such as device files or pipes
- `i_op` ☛ `struct inode_operations`

The Dentry Object

- components in a path
- makes path name lookup easier
- `struct dentry` in `<linux/dcache.h>`
- created on-the-fly from a string representation of a path name

Dentry State

- used
- unused
- negative

Dentry Cache

consists of three parts:

1. Lists of “ used ” dentries
2. A doubly linked “ least recently used ” list of unused and negative dentry objects
3. A hash table and hashing function used to quickly resolve a given path into the associated dentry object

A **used** dentry corresponds to a valid inode (`d_inode` points to an associated inode) and indicates that there are one or more users of the object (`d_count` is positive). A used dentry is in use by the VFS and points to valid data and, thus, cannot be discarded[*Linux Kernel Development*, Sec. 13.9.1, *Dentry State*].

An *unused* dentry corresponds to a valid inode (`d_inode` points to an inode), but the VFS is not currently using the dentry object (`d_count` is zero). Because the dentry object still points to a valid object, the dentry is kept around — cached — in case it is needed again. Because the dentry has not been destroyed prematurely, the dentry need not be re-created if it is needed in the future, and path name lookups can complete quicker than if the dentry was not cached. If it is necessary to reclaim memory, however, the dentry can be discarded because it is not in active use.

A *negative* dentry is not associated with a valid inode (`d_inode` is NULL) because either the inode was deleted or the path name was never correct to begin with. The dentry is kept around, however, so that future lookups are resolved quickly. For example, consider a daemon that continually tries to open and read a config file that is not present. The `open()` system calls continually returns `ENOENT`, but not until after the kernel constructs the path, walks the on-disk directory structure, and verifies the file’s inexistence. Because even this failed lookup is expensive, caching the “ negative ” results are worthwhile. Although a negative dentry is useful, it can be destroyed if memory is at a premium because nothing is actually using it.

A dentry object can also be freed, sitting in the slab object cache, as discussed in the previous chapter. In that case, there is no valid reference to the dentry object in any VFS or any filesystem code.

The File Object

- is the in-memory representation of an open file
- `open()` ⇒ create; `close()` ⇒ destroy
- there can be multiple file objects in existence for the same file
  - Because multiple processes can open and manipulate a file at the same time
- `struct file` in `<linux/fs.h>`

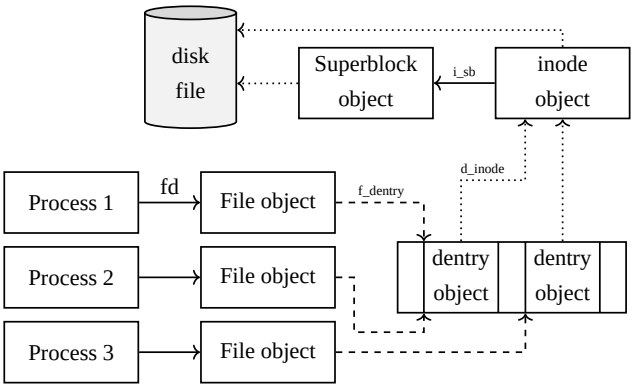


Fig. 357 illustrates with a simple example how processes interact with files. Three different processes have opened the same file, two of them using the same hard link. In this case, each of the three processes uses its own file object, while only two dentry objects are required one for each hard link. Both dentry objects refer to the same inode object, which identifies the superblock object and, together with the latter, the common disk file[*Understanding The Linux Kernel*, Sec. 12.1.1].

[1] Wikipedia. *File system — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=File%5C_system&oldid=646603121.

[2] Wikipedia. *Computer file — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Computer%5C_file&oldid=647724614.

[3] Wikipedia. *Inode — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Inode&oldid=647736522.

[4] Wikipedia. *Ext2 — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Ext2&oldid=642312602.

[5] Wikipedia. *Virtual file system — Wikipedia, The Free Encyclopedia*. 2014. http://en.wikipedia.org/w/index.php?title=Virtual%5C_file%5C_system&oldid=640354992.