Special Edition for CSEDU12 Students

TOUCH-N-PASS EXAM CRAM GUIDE SERIES

SYSTEM PROGRAMMING

Prepared By

Sharafat Ibn Mollah Mosharraf

CSE, DU

12th Batch (2005-2006)

TABLE OF CONTENTS

APIs at a Glance

1 FILE APIs

```
int open(const char *path_name, int access_mode, mode_t permission);
```

Opens a file.

```
size_t read(int fd, void *buf, size_t size);
```

Fetches a fixed size block of data from a file.

```
size_t write(int fd, void *buf, size_t size);
```

Puts a fixed size block of data to a file.

```
int close(int fd);
```

Disconnects a file from a process.

```
off_t lseek(int fd, off_t pos, int where);
```

Used to perform random access of data by changing the file offset to a different value.

2 Process APIs

```
pid_t fork(void);
pid_t vfork(void);
```

Creates a child process.

```
int execl(const char *path, const char *arg, ..., NULL)
int execv(const char *path, char *const argv[])
int execlp(const char *file, const char *arg, ..., NULL)
int execvp(const char *file, char *const argv[])
int execle(const char *path, const char *arg, ..., NULL, char *const env[])
int execve(const char *path, char *const argv[], char *const env[])
```

Causes a calling process to change its context and execute a different program.

```
int pipe(int fifo[2]);
```

Creates a communication channel between two *related* processes.

3 Signal APIs

```
void (*signal(int signal_num, void(*handler)(int)))(int);
void (*sigset(int signal_num, void(*handler)(int)))(int);
int sigaction(int signal_num, struct sigaction *action, struct sigaction *old_action);
```

Used to define / install per-signal handling method.

```
int sigprocmask(int how, const sigset_t *new_mask, sigset_t *old_mask);
```

Used by a process to query or set its signal mask.

```
int sigemptyset(sigset_t *sigmask);
```

Clears all signal flags in the *sigmask* argument.

```
int sigfillset(sigset_t *sigmask);
```

Sets all signal flags in the *sigmask* argument.

```
int sigaddset(sigset_t *sigmask, const int signal_mask);
```

Sets the flag corresponding to the *signal_num* signal in the *sigmask* argument.

```
int sigdelset(sigset_t *sigmask, const int signal_mask);
```

Clears the flag corresponding to the *signal_num* signal in the *sigmask* argument.

```
int sigismember(const sigset_t *sigmask, const int signal_num);
```

Returns 1 if the flag corresponding to the *signal_num* signal in the *sigmask* argument is set, 0 if it is not set, and -1 if the call fails.

	`int sigpending(sigset_t *sigmask);`
	Used to query which signals are pending for a process.
	`int kill(pid_t pid, int signal_num);`
	Used to send a signal to a related *process* by another *process.*
	`int raise(int signal_num);`
	Used to send a signal to the calling process.

4	**UNIX Message APIs**
	`int msgget(key_t key, int flag);`
	Opens a message queue whose key ID is given in the *key* argument.
	`int msgsnd(int msgd, const void* msgPtr, int len, int flag);`
	Sends a message (pointed to by *msgPtr*) to a message queue designated by the *msgd* descriptor.
	`int msgrcv(int msgd, const void* msgPtr, int len, int msgType, int flag);`
	Receives a message of type *msgType* from a message queue designated by *msgd*.
	`int msgctl(int msgd, int cmd, struct msqid_ds* msgbufptr);`
	This API can be used to query the control data of a message queue designated by the *msgd* argument, to change the information within the control data of the queue, or to delete the queue from the system.

5	**Shared Memory APIs**
	`int shmget(key_t key, int size, int flag);`
	Opens a shared memory whose key ID is given in the *key* argument.
	`void *shmat(int shmid, void* addr, int flag);`
	Attaches a shared memory referenced by *shmid* to the calling process virtual address space.
	`int shmdt(void* addr);`
	Detaches (or unmaps) a shared memory from the specified *addr* virtual address of the calling process.
	`int shmctl(int shmid, int cmd, struct shmid_ds* buf);`
	This API can either query or change the control data of a shared memory designated by *shmid*, or delete the memory altogether.

6	**UNIX Semaphore APIs**
	`int semget(key_t key, int num, int flag);`
	Opens a semaphore set whose key ID is given in the *key* argument.
	`int semop(int semd, struct sembuf* opPtr, int len);`
	This API may be used to change the value of one or more semaphores in a set (as designated by *semd*) and/or o test whether their values are 0.
	`int semctl(int semd, int num, int cmd, union semun arg);`
	This API can be used to query or change the control data of a semaphore set designated by the *semd* argument or to delete the set altogether.

7	**Socket APIs**
	`int socket(int domain, int type, int protocol);`
	Creates a socket of the given domain, type and protocol.
	`int bind(int sd, struct sockaddr *addr_p, int lenght);`
	Binds a name to a socket.

```
int listen(int sd, int size);
```

This is called in a server process to establish a connection-based socket (of type SOCK_STREAM or SOCK_SEQPACKET) for communication.

```
int connect(int sd, struct sockaddr *addr_p, int lenght);
```

This is called in a client process in requesting a connection to a server socket.

```
int accept(int sd, struct sockaddr *addr_p, int* lenght);
```

This is called in a server process to establish a connection-based socket connection with a client socket.

```
int send(int sd, const char *buf, int len, int flag);
int sendto(int sd, const char *buf, int len, int flag, struct sockaddr *addr_p, int addr_p_len);
```

The *send* function sends a message, contained in *buf*, of size *len* bytes, to a socket that is connected to this socket, as designated by *sd*.

```
int recv(int sd, const char *buf, int len, int flag);
int recvfrom(int sd, const char *buf, int len, int flag, struct sockaddr *addr_p, int* addr_p_len);
```

The *recv* function receives a message via a socket designated by *sid*. The message received is copied to *buf*, and the maximum size of *buf* is specified in the *len* argument.

```
int htons(short var);
int htonl(long var);
int ntohs(short var);
int ntohl(long var);
```

The *htons* (*host to network short*) function converts *short* values from host byte order (little-endian) to network byte order (big-endian).

The *htonl* (*host to network long*) function converts *long* values from host byte order (little-endian) to network byte order (big-endian).

The *ntohs* and *ntohl* functions do the opposites of *htons* and *htonl* functions.

```
int inet_aton(const char *cp, struct in_addr *addr);
```

Converts the specified string, in the Internet standard dot notation, to an integer value suitable for use as an Internet address. The converted address is in network byte order.

```
char* inet_ntoa(struct in_addr in);
```

Converts the specified Internet host address to a string in the Internet standard dot notation.

CHAPTER 1
ASSEMBLER, LINKER & LOADER

Theories

1.1 Assembler, Compiler, Linker and Loader

An *assembler* is software whose task is to convert *processor-specific* human readable instructions to processor native machine language. Examples: TASM, MASM etc.

A *compiler* is a similar utility but it generates native machine language from generally *processor-independent* source code. Examples: C/C++ compiler, Fortran compiler etc.

A *linker* or *link editor* is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

A *loader* is a system program which brings the object program (i.e., machine language code) into memory for execution.

1.2 Compile, link and execute stages for a running program (process) written in C

Normally, the C's program building process involves four stages and utilizes different *tools* such as a preprocessor, compiler, assembler, and linker.

At the end, there should be a single executable file. Below are the stages that happen in order regardless of the operating system/compiler. The stages are graphically illustrated in *figure 1.1*.

1. **Preprocessing** is the first pass of any C compilation. It processes *include-files* (*#include*), *conditional compilation instructions* (*#ifdef*, *#endif*) and *macros* (*#define*, *#typedef*).
2. **Compilation** is the second pass. It takes the output of the preprocessor and the source code, and generates *assembler source code*.
3. **Assembly** is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an *object file*.
4. **Linking** is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called *relocation*).

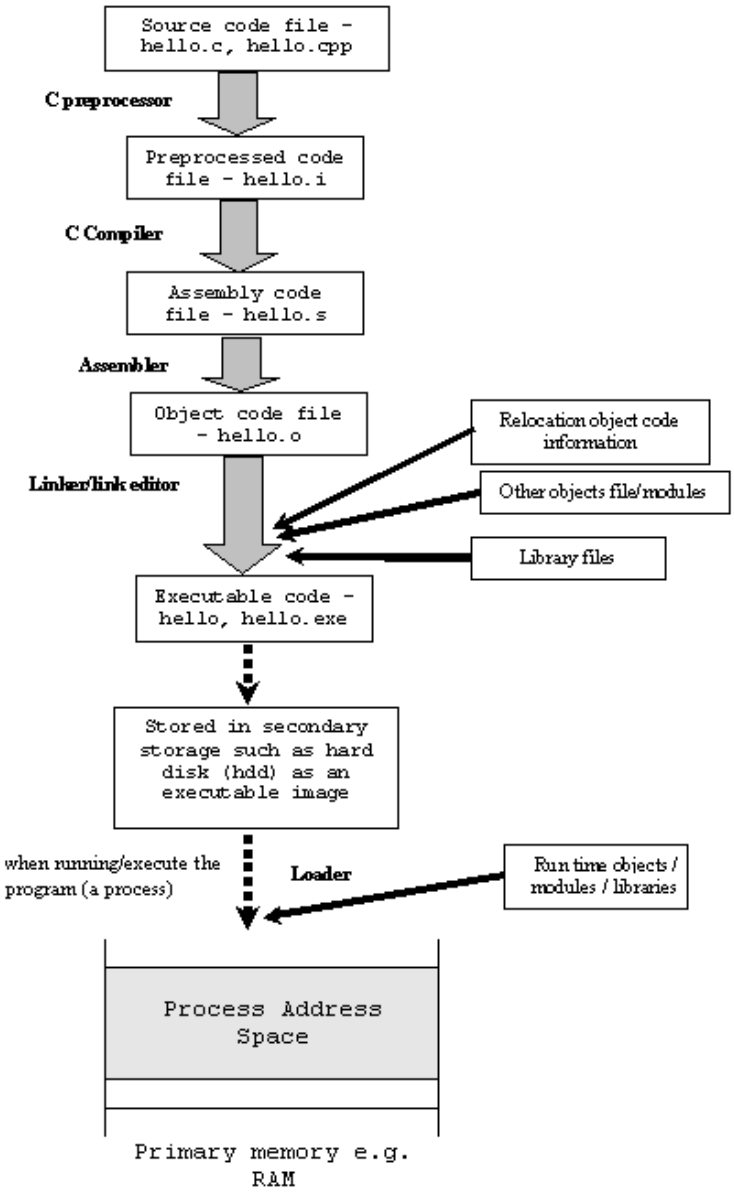


Figure 1.2: Compile, link and execute stages for a running Program (process) written in C.

In UNIX/Linux, the executable or binary file doesn't have any extension, whereas in Windows the executables may have .exe, .com, .dll etc.

File extension	Description
file_name.c	C source code which must be preprocessed.
file_name.i	C source code which should not be preprocessed.
file_name.ii	C++ source code which should not be preprocessed.
file_name.h	C header file (not to be compiled or linked).
file_name.cc file_name.cp file_name.cxx file_name.cpp file_name.c++ file_name.C	C++ source code which must be preprocessed. For file_name.cxx, the xx must both be literally character x and file_name.C, is capital c.
file_name.s	Assembler code.
file_name.S	Assembler code which must be preprocessed.
file_name.o	Object file by default, the object file name for a source file is made by replacing the extension .c, .i, .s etc with .o

1.3 Linker

A *linker* or *link editor* is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

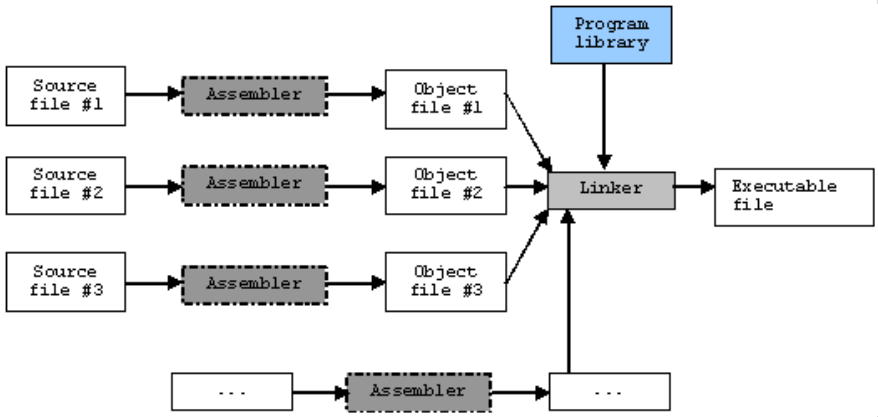


Figure 1.3: The object file linking process.

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. Typically, an object file can contain three kinds of symbols:

- **Defined symbols**, which allow it to be called by other modules.
- **Undefined symbols**, which call the other modules where these symbols are defined.
- **Local symbols**, used internally within the object file to facilitate relocation.

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.

1.4 Relocator

A *Relocator* is a program which modifies the object program so that it can be loaded at an address different from the location originally specified.

Compilers or assemblers typically generate the executable with zero as the lower-most, starting address. Before the execution of object code, these addresses should be adjusted so that they denote the correct runtime addresses. A Relocator inserts some modification records in the object file so that a loader can load the program adjusting the addresses.

1.5 Loader

A *loader* is a system program which brings the object program (i.e., machine language code) into memory for execution.

Functions of Loader

1. Allocate space in memory for the program (allocation).
2. Resolve symbolic references between object programs (linking).
3. Adjust all address dependent locations, such as address constraints, to correspond to the allocated space (relocation).
4. Physically place the machine instruction and data into memory (loading) for execution.

Types of Loader

There are different types of loaders:

1. Absolute loader
2. Linking loader
3. Relocating loader
4. Dynamic loader
5. Bootstrap loader

Absolute loader

An absolute loader simply loads an object program directly into memory for execution without bringing any modification in addresses.

Linking Loader

A linking loader performs all linking and relocation operations – including automatic library search if specified – in the object program and loads the linked program directly into memory for execution.

Relocating Loader

A relocating loader loads a *linked* object program into memory by relocating the addresses.

Dynamic loader

A dynamic loader loads an object program (usually a *library*) during run-time and links it with the calling program.

Bootstrap loader

When a computer is first turned on or restarted, a special type of absolute loader, called a bootstrap loader, is executed. This bootstrap loads the first program to be run by the computer – usually an operating system.

1.6 Differences between Linking Loader and Linkage Editor (or Link editor)

Definition: A linking loader performs all linking and relocation operations – including automatic library search if specified – in the object program and loads the linked program directly into memory for execution.

A linkage editor, on the other hand, produces a linked version of the program (often called a *load module* or an *executable image*), which is written to a file or library for later execution.

Performance: A linking loader searches libraries and resolves external references every time the program is executed. In contrast, a linkage editor performs these tasks only the first time. Hence, the loading can be accomplished in one pass using a relocating loader. This involves much less overhead than using a linking loader.

Application: If a program can be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required. However, if a program is assembled for nearly every execution (for example, during program development and testing), it is more efficient to use a linking loader which avoids the steps of writing and reading the linked programs.

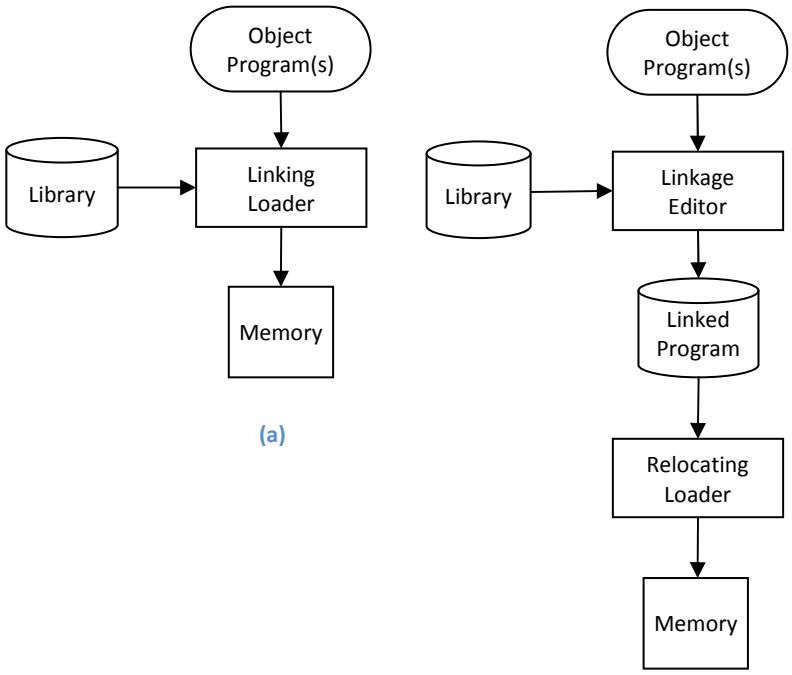


Figure 1.6: Processing of an object program using (a) linking loader, and (b) linkage editor.

1.7 Dynamic Linking / Dynamic Binding / Load on Call

Linkage editor performs linking operation before the program is loaded for execution. Linking loader perform linking operation at load time. *Dynamic linking* (or *dynamic binding*, *load on call*) perform linking operation while the program is executing.

When a subroutine call is encountered and the subroutine is not resident in memory, the subroutine is loaded into memory, linking is performed, and finally program execution jumps to the subroutine.

Advantages

Loading the routines when they are needed, the memory space will be saved.

Implementation of Dynamic Linking

Implementation of dynamic linking needs the help of the operating system. The OS should provide load-and-call system call. The OS has an internal table to keep the names the entry points and the use condition of the routines in memory.

Processing procedures of Dynamic Linking

1. The program makes a load-and-call service request to the operating system. The parameter of this request is the symbolic name of the routine to be called. [*See figure 1.7(a)*]
2. The operating system examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded from the specified user or system libraries as shown in *figure 1.7(b)*. Control is then passed from the OS to the routine being called. [*figure 1.7(c)*]
3. When the called subroutine completes its processing, it returns to its caller (that is, to the operating system routine that handles the load-and-call service request). The operating system then returns control to the program that issued the request. This process is illustrated in *figure 1.7(d)*.
4. After the subroutine is completed, the memory that was allocated to load it may be released and used for other purposes. However, this is not always done immediately. Sometimes it is desirable to retain the routine in memory for later use as long as the storage space is not needed for other processing. If a subroutine is still in memory, a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to the called routine, as shown in *figure 1.7(e)*.

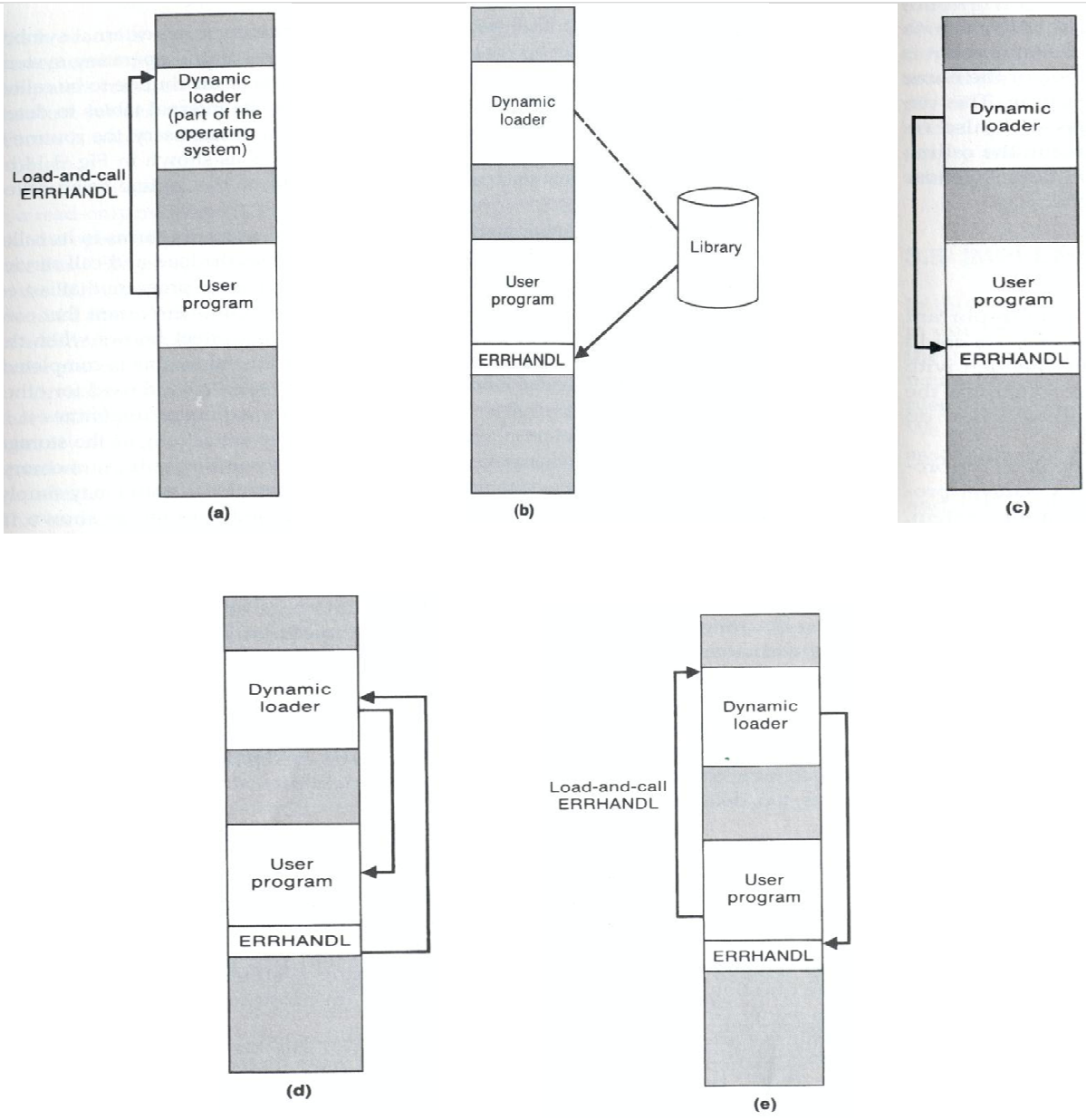


Figure 1.7: Loading and calling of a subroutine using dynamic linking.

CHAPTER 2
KERNEL

Theories

2.1 Kernel

The kernel is the part of the Operating System that runs in privileged or protected mode and interacts directly with the hardware of computer.

The kernel is the central component of most computer operating systems. Its responsibilities include managing the system's resources (the communication between hardware and software components). As a basic component of an operating system, a kernel provides the lowest-level abstraction layer for the resources (especially memory, processors and I/O devices) that application software must control to perform its function. It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls.

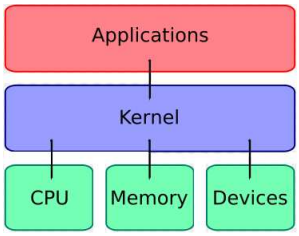


Figure 2.1.1: Kernel Layout.

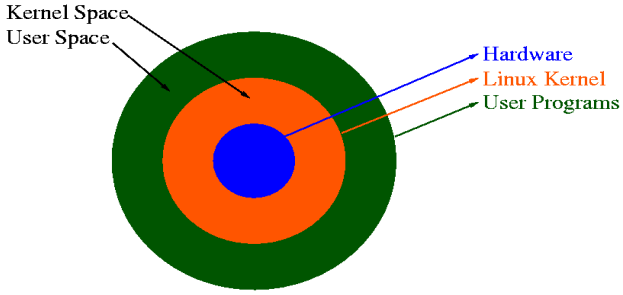


Figure 2.1.2: A typical kernel.

2.2 Kernel Basic Facilities / Purpose of Kernel

The kernel's primary purpose is to manage the computer's resources and allow other programs to run and use these resources. Typically, the resources consist of:

1. **The Central Processing Unit (CPU, the processor).** This is the most central part of a computer system, responsible for running or executing programs on it. The kernel takes responsibility for deciding at any time which of the many running programs should be allocated to the processor or processors (each of which can usually run only one program at a time)
2. **The computer's memory.** Memory is used to store both program instructions and data. Typically, both need to be present in memory in order for a program to execute. Often multiple programs will want access to memory, frequently demanding more memory than the computer has available. The kernel is responsible for deciding which memory each process can use, and determining what to do when not enough is available.
3. **Any Input/Output (I/O) devices present in the computer, such as keyboard, mouse, disk drives, printers, displays, etc.** The kernel allocates requests from applications to perform I/O to an appropriate device (or subsection of a device, in the case of files on a disk or windows on a display) and provides convenient methods for using the device (typically abstracted to the point where the application does not need to know implementation details of the device)

Kernels also usually provide methods for synchronization and communication between processes (called inter-process communication or IPC).

Finally, a kernel must provide running programs with a method to make requests to access these facilities.

2.3 Process Management

The main task of a kernel is to allow the execution of applications and support them with features such as hardware abstractions. A process defines which memory portions the application can access. Kernel process management must take into account the hardware built-in equipment for memory

protection.

To run an application, a kernel typically sets up an address space for the application, loads the file containing the application's code into memory (perhaps via demand paging), sets up a stack for the program and branches to a given location inside the program, thus starting its execution.

Multi-tasking kernels are able to give the user the illusion that the number of processes being run simultaneously on the computer is higher than the maximum number of processes the computer is physically able to run simultaneously.

The operating system might also support multiprocessing; in that case, different programs and threads may run on different processors. A kernel for such a system must be designed to be re-entrant, meaning that it may safely run two different parts of its code simultaneously. This typically means providing synchronization mechanisms (such as spinlocks) to ensure that no two processors attempt to modify the same data at the same time.

2.4 System Calls

To actually perform useful work, a process must be able to access the services provided by the kernel. This is implemented differently by each kernel, but most provide a C library or an API, which in turn invokes the related kernel functions.

The method of invoking the kernel function varies from kernel to kernel. If memory isolation is in use, it is impossible for a user process to call the kernel directly, because that would be a violation of the processor's access control rules. A few possibilities are:

1. **Using a software-simulated interrupt.** This method is available on most hardware, and is therefore very common.
2. **Using a call gate.** A call gate is a special address which the kernel has added to a list stored in kernel memory and which the processor knows the location of. When the processor detects a call to that location, it instead redirects to the target location without causing an access violation. Requires hardware support, but the hardware for it is quite common.
3. **Using a special system call instruction.** This technique requires special hardware support, which common architectures (notably, x86) may lack. System call instructions have been added to recent models of x86 processors, however, and some (but not all) operating systems for PCs make use of them when available.
4. **Using a memory-based queue.** An application that makes large numbers of requests but does not need to wait for the result of each may add details of requests to an area of memory that the kernel periodically scans to find requests.

CHAPTER 3
UNIX ELF FILE FORMAT

Concepts

3.1 ELF File Format

The a.out format served the Unix community well for over 10 years. However, to better support cross-compilation, dynamic linking, initializer/finalizer (e.g., the constructor and destructor in C++) and other advanced system features, a.out has been replaced by the ELF file format. ELF stands for "Executable and Linking Format." ELF has been adopted by FreeBSD and Linux as the current standard.

3.2 ELF File Types

Elf defines the format of executable binary files. There are four different types:

1. **Relocatable:** Created by compilers or assemblers. Need to be processed by the linker before running.
2. **Executable:** Have all relocation done and all symbol resolved except perhaps shared library symbols that must be resolved at run time.
3. **Shared object:** Shared library containing both symbol information for the linker and directly runnable code for run time.
4. **Core file:** A core dump file.

3.3 ELF Structure

ELF files have a dual nature:

1. Compilers, assemblers, and linkers treat the file as a set of logical sections described by a section header table.
2. The system loader treats the file as a set of segments described by a program header table.

A single segment usually consists of several sections. For example, a loadable read-only segment could contain sections for executable code, read-only data, and symbols for the dynamic linker.

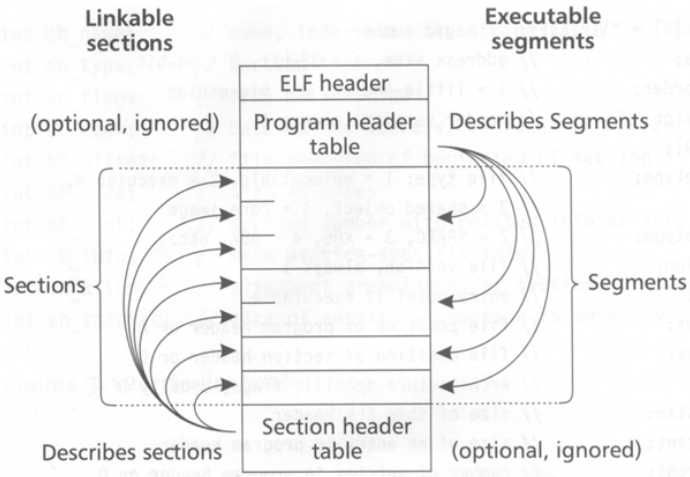


FIGURE • Two views of an ELF file.

Relocatable files have section header tables. Executable files have program header tables. Shared object files have both.

Sections are intended for further processing by a linker, while the segments are intended to be mapped into memory.

3.4 ELF Header

The ELF header is always at offset zero of the file.

The program header table and the section header table's offset in the file are defined in the ELF header.

The header is decodable even on machines with a different byte order from the file's target architecture. After reading class and byte order fields, the rest of the fields in the ELF header can be decoded. The ELF format can support two different address sizes: 32 bits and 64 bits.

```
char magic[4] = "\177ELF"; // magic number
char class;              // address size, 1 = 32-bit, 2 = 64-bit
char byteorder;          // 1 = little-endian, 2 = big-endian
char hversion;           // header version, always 1
char pad[9];
short filetype;          // file type: 1 = relocatable, 2 = executable,
                         // 3 = shared object, 4 = core image
short archtype;          // 2 = SPARC, 3 = x86, 4 = 68K, etc.
int fversion;            // file version, always 1
int entry;               // entry point if executable
int phdrpos;             // file position of program header or 0
int shdrpos;             // file position of section header or 0
int flags;               // architecture-specific flags, usually 0
short hdrsize;           // size of this ELF header
short phdrent;           // size of an entry in program header
short phdrcnt;           // number of entries in program header or 0
short shdrent;           // size of an entry in section header
short shdrcnt;           // number of entries in section header or 0
short strsec;            // section number that contains section name strings
```

FIGURE 3.11 • ELF header.

3.5 Relocatable Files

A relocatable or shared object file is a collection of sections. Each section contains a single type of information, such as program code, read-only data or read/write data, relocation entries, or symbols. Every symbol's address is defined relative to a section. Therefore, a procedure's entry point is relative to the program code section that contains that procedure's code.

Section Header

```
int sh_name;     // name, index into the string table
int sh_type;     // section type
int sh_flags;    // flag bits, below
int sh_addr;     // base memory address, if loadable, or zero
int sh_offset;   // file position of beginning of section
int sh_size;     // size in bytes
int sh_link;     // section number with related info or zero
int sh_info;     // more section-specific info
int sh_align;    // alignment granularity if section is moved
int sh_entsize;  // size of entries if section is an array
```

FIGURE 3.12 • Section header.

Types in Section Header

1. **PROGBITS:** This holds program contents including code, data, and debugger information.
2. **NOBITS:** Like PROGBITS. However, it occupies no space.
3. **SYMTAB** and **DYNSYM:** These hold symbol table.
4. **STRTAB:** This is a string table, like the one used in a.out.
5. **REL** and **RELA:** These hold relocation information.
6. **DYNAMIC** and **HASH:** This holds information related to dynamic linking.

Flags in Section Header

1. **WRITE:** This section contains data that is writable during process execution.
2. **ALLOC:** This section occupies memory during process execution.
3. **EXECINSTR:** This section contains executable machine instructions.

Various Sections

- **.rel.text**, **.rel.data**, and **.rel.rodata**:
  - These contain the relocation information for the corresponding text or data sections.
  - **Type:** REL
  - **Flags:** ALLOC is turned on if the file has a loadable segment that includes relocation.
- **.symtab:**
  - This section hold a symbol table.
- **.strtab:**
  - This section holds strings.
- **.init:** (Used only in C++)
  - This section holds executable instructions that contribute to the process initialization code.
  - **Type:** PROGBITS
  - **Flags:** ALLOC + EXECINSTR
- **.fini:** (Used only in C++)
  - This section hold executable instructions that contribute to the process termination code.
  - **Type:** PROGBITS
  - **Flags:** ALLOC + EXECINSTR
- **.interp:**
  - This section holds the pathname of a program interpreter.
  - **Type:** ALLOC
  - **Flags:** PROGBITS
  - If this section is present, rather than running the program directly, the system runs the interpreter and passes it the elf file as an argument.
  - For many years (used in a.out), UNIX has had self-running interpreted text files, using #! /bin/csh as the first line of the file.
  - ELF extends this facility to interpreters that run nontext programs.
  - In practice, this is used to run the run-time dynamic linker to load the program and to link in any required shared libraries.
- **.debug:**
  - This section holds symbolic debugging information.
  - **Type:** PROGBIT
- **.line:**
  - This section holds line number information for symbolic debugging, which describes the correspondence between the program source and the machine code.
  - **Type:** PROGBIT
- **.comment**
  - This section may store extra information.
- **.got:**
  - This section holds the global offset table.
  - **Type:** PROGBIT
- **.plt:**
  - This section holds the procedure linkage table.

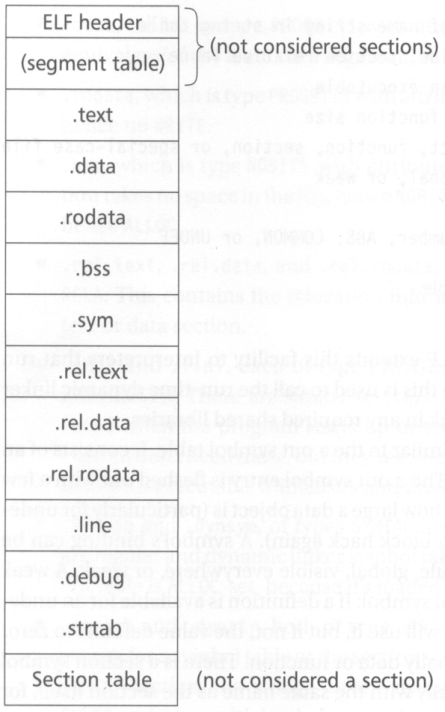


FIGURE 3.14 • Sample relocatable ELF file.

- **Type:** PROGBIT

- **.note:**
  - This section contains some extra information.

String Table

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. We use an index into the string table section to reference a string. The reason why we separate symbol names from symbol tables is that in C or C++, there is no limitation on the length of a symbol.

Symbol Table

An object file's symbol table holds information needed to locate and relocate a program's symbolic definition and references. A symbol table index is a subscript into this array.

3.6 Executable Files

An executable file usually has only a few segments. For example,

- A read-only one for the code.
- A read-only one for read-only data.
- A read/write one for read/write data.

All of the loadable sections are packed into the appropriate segments so that the system can map the file with just one or two operations. For example, if there is a *.init* and *.fini* sections, those sections will be put into the read-only text segment.

Program Header

```
int type;      // loadable code or data, dynamic linking info, etc.
int offset;    // file offset of segment
int virtaddr;  // virtual address to map segment
int physaddr;  // physical address, not used
int filesize;  // size of segment in file
int memsize;   // size of segment in memory (bigger if contains bss)
int flags;     // Read, Write, Execute bits
int align;     // required alignment, invariably hardware page size
```

FIGURE 3.15 • ELF program header.

Types in Program Header

This field tells what kind of segment this array element describes:

- **PT_LOAD:** This segment is a loadable segment.
- **PT_DYNAMIC:** This array element specifies dynamic linking information.
- **PT_INTERP:** This element specified the location and size of a null-terminated path name to invoke as an interpreter.

3.7 ELF Linking

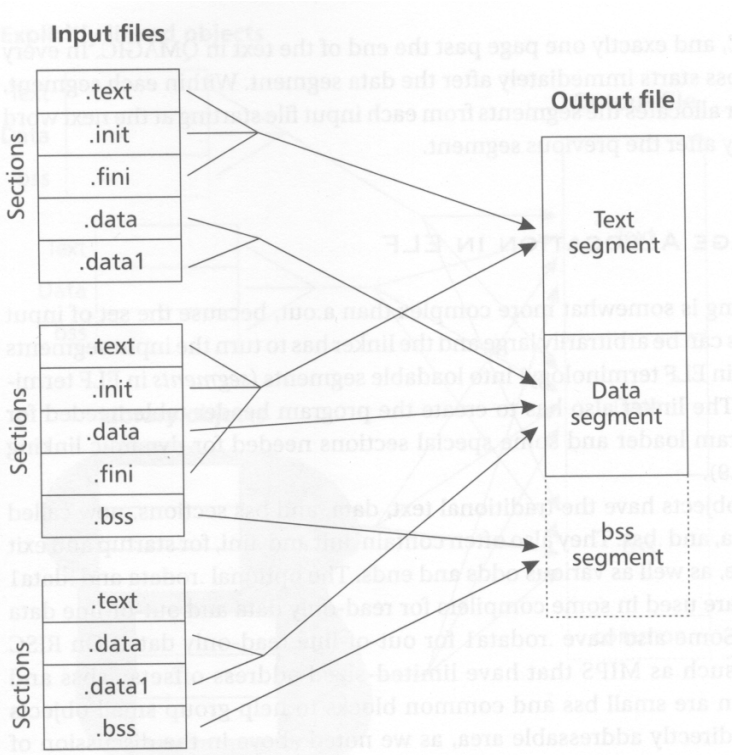


FIGURE 4.9 • ELF linking.

CHAPTER 4
DEVICE DRIVERS

Concepts

4.1 Device Driver

Device driver takes a special role in Linux Kernel. It enables a particular hardware respond to well-defined internal programming interface. User activities are performed by means of standard system call independent of specific device driver. The kernel maps those calls to device-specific operations that act on real hardware. That is the function of a device driver. The programming interface is such that drivers can be built separately from the rest of the kernel. The drivers are plugged in at running kernel when needed.

4.2 Why We are Going for Writing Device Driver

There are various reasons behind this:

- The rate at which new hardware become available and obsolete!
- Individual person may need to know about device driver on which he is interested.
- Hardware vendors, by making a Linux driver available for their products, can add the large and growing Linux user base to their potential markets.
- And the open source nature of the Linux system means that if the driver writer wishes, the source to a driver can be quickly disseminated to millions of users.

4.3 Issues with Writing Device Drivers

- Each driver is different. As a driver writer, you need to understand your specific device well. But most of the principles and basic techniques are the same for all drivers. We will not deal with any specific devices rather give you a handle on the background you need to make your device work.
- A programmer should pay attention to fundamental concepts of kernel code to access the hardware, as well as the software layer between application program and hardware devices.
- Particular policy on user programs should not be enforced.
- A single device may be used by many processes at the same time – synchronization is required in this case. For this purpose, every process must have different data structures to access the device driver.
- There are three classes of devices – character, block and network devices. Other devices such as USB have separate module in kernel and can be accessed through the file system node /dev.
- **Security Issues:** Security is an important issue today. The system call *init_module* checks if the invoking process has the user authorization to access or load the kernel module.

4.4 Hello World Module Example

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENCE("Dual BSD/GPL");
static int hello_init(void) {
       printk(KERN_ALERT "Hello, world\n");
       return 0;
}
static void hello_exit(void) {
       printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

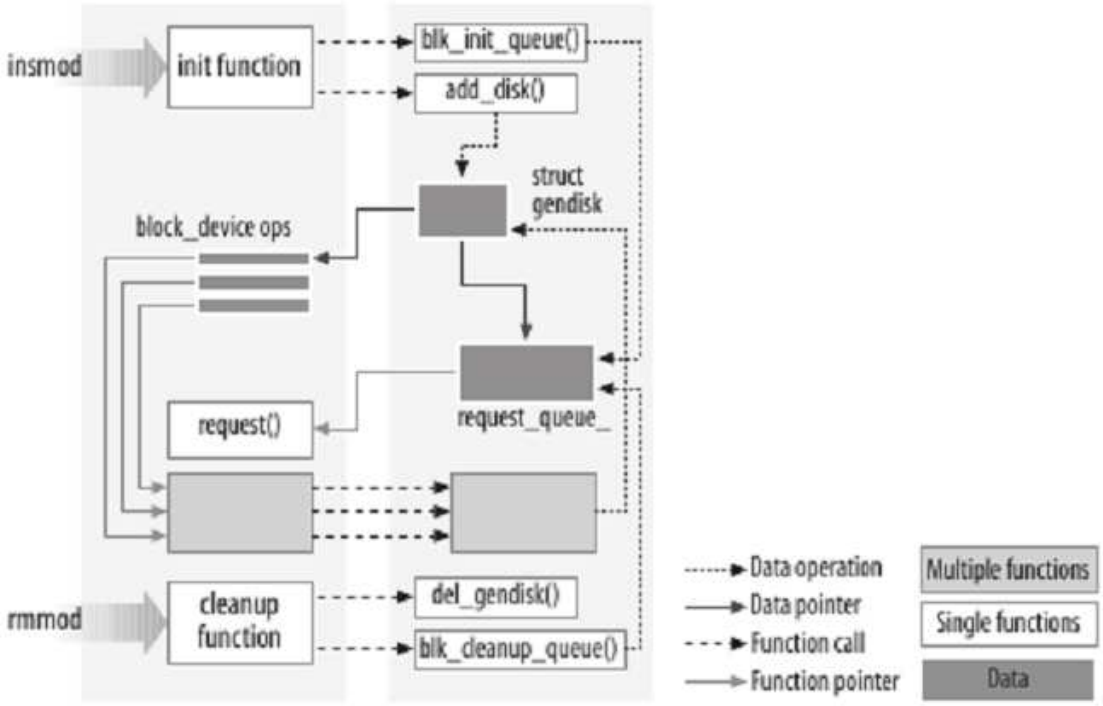
Make

```
% make
make[1]: Entering directory `/usr/src/linux-2.6.10'
  CC [M]  /home/ldd3/src/misc-modules/hello.o
  Building modules, stage 2.
  MODPOST
  CC      /home/ldd3/src/misc-modules/hello.mod.o
  LD [M]  /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory `/usr/src/linux-2.6.10'
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

Makefile

```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
    obj-m := hello.o

# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else

    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD  := $(shell pwd)

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif
```

4.5 Linking Module to the Kernel



When a module is installed, the device is registered to the kernel using a structure – e.g., *gendisk*. The object of the structure collects data from device and delivers them to the requested process. The inverse task is also performed through this object. Requests for accessing the device are queued up into

the request queue.

When the module is removed, some cleanup functions are executed. Any memory allocations allocated by the module is cleaned, and the *gendisk* structure object is destroyed.

4.6 Differences Between Module and Application

1. Applications perform a single task from beginning to end; whereas every kernel module just registers itself in order to serve future requests, and its initialization function terminates immediately.
2. Application exit or termination is not aware of cleanup. Exit in module cleans up everything initialized or allocated during *init* call.
3. Application is usually linked with library (such as libc), a module on the other hand is linked only to the kernel. [*Think about printf being used in Application and printk being used in Modules.*]
4. Modules run in kernel space but applications run in user space.

4.7 Kernel Symbol Table

insmod resolves undefined symbols against the table of public kernel symbols. The kernel symbol table contains the addresses of global kernel items — functions and variables. It also contains the symbols which are needed to implement modularized drivers.

When a module is loaded, any symbol exported by the module becomes part of the kernel symbol table. In the usual case, a module implements its own functionality without the need to export any symbols at all. Other modules may benefit from using the symbols.

Module Stacking and Example of Symbols Used by Other Modules

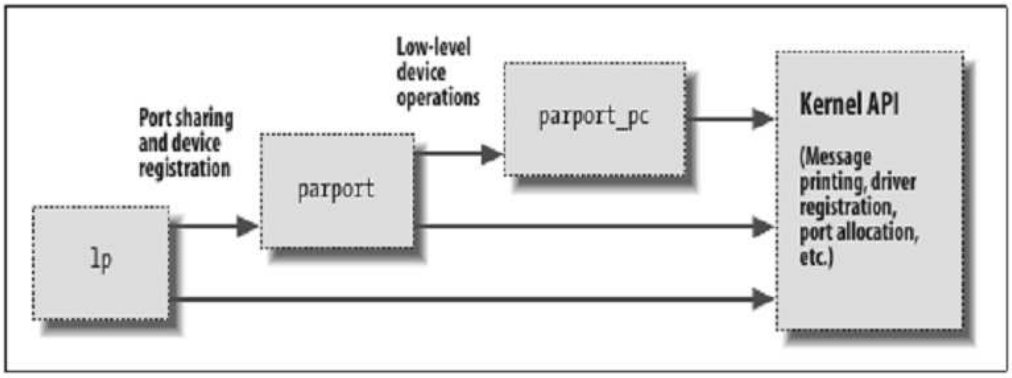


Figure 2-2. Stacking of parallel port driver modules

Exporting Symbols

```
EXPORT_SYMBOL(symbol);
EXPORT_SYMBOL_GPL(symbol);
```

The above are the macros used to export a symbol to the kernel. The first form exports without using versioning information, and the second limits the export to GPL-licensed modules.

4.8 How Device Drivers Work

1. User process or application calls a system call.
2. Control moves to kernel mode from user mode.
3. Kernel gets the module information from registration.
4. Module function for that system call starts running.
5. Data is copied to/from devices or resources are copied to/from application space. This copying should be done by module code / device driver. The module has to know the requesting process ID to access its memory space (user space).

Questions

4.1	**Classify the devices in respect of Linux OS. What do you understand by major and minor number of a device? *[2007. Marks: 2]*** There are three classes of devices – character, block and network devices. Major and minor numbers are associated with the device special files in the /dev directory and are used by the operating system to determine the actual driver and device to be accessed by the user-level request for the special device file. A given software driver can be working with one or more hardware controllers, each of which has its own major number. Each device connected to a given controller then would have its own minor number. Thus, any single device can be identified through the major/minor number combination.
4.2	**What are the essential interface functions at least to install and remove a module from the Linux Kernel? *[2007. Marks: 2]*** To install module – MODULE_INIT(). To remove module – MODULE_EXIT().
4.3	**With an example, briefly explain the steps and resources involved to link/add kernel module to GNU-Linux. *[2007. Marks: 4]*** *See Concept 4.5.*

CHAPTER 5
INTERRUPT

Concepts

5.1 Interrupt

An *interrupt* is an asynchronous signal indicating the need for attention or a synchronous event in software indicating the need for a change in execution.

A *hardware interrupt* causes the processor to save its state of execution via a context switch, and begin execution of an interrupt handler.

Software interrupts are usually implemented as instructions in the instruction set, which cause a context switch to an interrupt handler similar to a hardware interrupt.

Interrupts are a commonly used technique for computer multitasking, especially in real-time computing. Such a system is said to be interrupt-driven.

An act of interrupting is referred to as an *interrupt request* (IRQ).

5.2 Why Hardware Interrupt

Hardware interrupts were introduced as a way to avoid wasting the processor's valuable time in polling loops, waiting for external events.

Implementing Hardware Interrupts

They may be implemented in hardware as a distinct system with control lines, or they may be integrated into the memory subsystem.

If implemented in hardware, an interrupt controller circuit such as the IBM PC's Programmable Interrupt Controller (PIC) may be connected between the interrupting device and the processor's interrupt pin to multiplex several sources of interrupt onto the one or two CPU lines typically available. If implemented as part of the memory controller, interrupts are mapped into the system's memory address space.

5.3 Categories of Interrupts

Interrupts can be categorized into: *maskable interrupt* (IRQ), *non-maskable interrupt* (NMI), *interprocessor interrupt* (IPI), *software interrupt*, and *spurious interrupt*.

A **maskable interrupt** (IRQ) is a hardware interrupt that may be ignored by setting a bit in an interrupt mask register's (IMR) bit-mask.

Likewise, a **non-maskable interrupt** (NMI) is a hardware interrupt that does not have a bit-mask associated with it – meaning that it can never be ignored. NMIs are often used for timers, especially watchdog timers.

An **interprocessor interrupt** is a special case of interrupt that is generated by one processor to interrupt another processor in a multiprocessor system.

A **software interrupt** is an interrupt generated within a processor by executing an instruction. Software interrupts are often used to implement system calls because they implement a subroutine call with a CPU ring level change.

A **spurious interrupt** is a hardware interrupt that is unwanted. They are typically generated by system conditions such as electrical interference on an interrupt line or through incorrectly designed hardware.

Processors typically have an internal interrupt mask which allows software to ignore all external hardware interrupts while it is set. This mask may offer faster access than accessing an interrupt mask register (IMR) in a PIC, or disabling interrupts in the device itself. In some cases, such as the x86 architecture, disabling and enabling interrupts on the processor itself acts as a memory barrier, in which case it may actually be slower.

5.4 Precise Interrupt

An interrupt that leaves the machine in a well-defined state is called a *precise interrupt*. Such an interrupt has four properties:

1. The Program Counter (PC) is saved in a known place.
2. All instructions before the one pointed to by the PC have fully executed.
3. No instruction beyond the one pointed to by the PC has been executed (that is no prohibition on instruction beyond that in PC, it is just that any changes they make to registers or memory must be undone before the interrupt happens).
4. The execution state of the instruction pointed to by the PC is known.

Imprecise Interrupt

An interrupt that does not meet these requirements is called an *imprecise interrupt*.

Interrupt Storm

The phenomenon where the overall system performance is severely hindered by excessive amounts of processing time spent handling interrupts is called an *interrupt storm*.

5.5 Types of Interrupts

Level-Triggered Interrupt

A *level-triggered interrupt* is a class of interrupts where the presence of an unserviced interrupt is indicated by a high level (1), or low level (0), of the interrupt request line. A device wishing to signal an interrupt drives the line to its active level, and then holds it at that level until serviced. It ceases asserting the line when the CPU commands it to or otherwise handles the condition that caused it to signal the interrupt.

The original PCI standard mandated level-triggered interrupts. Newer versions of PCI allow, and PCI Express requires, the use of message-signaled interrupts.

Edge-Triggered Interrupt

An *edge-triggered interrupt* is a class of interrupts that are signaled by a level transition on the interrupt line, either a falling edge (1 to 0) or a rising edge (0 to 1). A device wishing to signal an interrupt drives a pulse onto the line and then releases the line to its quiescent state. If the pulse is too short to be detected by polled I/O, then special hardware may be required to detect the edge.

The elderly Industry Standard Architecture (ISA) bus uses edge-triggered interrupts. The parallel port also uses edge-triggered interrupts.

Hybrid Interrupt

Some systems use a hybrid of level-triggered and edge-triggered signaling. The hardware not only looks for an edge, but it also verifies that the interrupt signal stays active for a certain period of time.

A common use of a hybrid interrupt is for the NMI (non-maskable interrupt) input. Because NMIs generally signal major – or even catastrophic – system events, a good implementation of this signal tries to ensure that the interrupt is valid by verifying that it remains active for a period of time. This 2-step approach helps to eliminate false interrupts from affecting the system.

Message Signaled Interrupt

A message-signalled interrupt does not use a physical interrupt line. Instead, a device signals its request for service by sending a short message over some communications medium, typically a computer bus. The message might be of a type reserved for interrupts, or it might be of some pre-existing type such as a memory write.

Message-signaled interrupts behave very much like edge-triggered interrupts, in that the interrupt is a momentary signal rather than a continuous condition. Interrupt-handling software treats the two in much the same manner. Typically, multiple pending message-signaled interrupts with the same message (the same virtual interrupt line) are allowed to merge, just as closely-spaced edge-triggered

interrupts can merge.

PCI Express, a serial computer bus, uses message-signaled interrupts exclusively.

Doorbell Interrupt

In a push button analogy applied to computer systems, the term *doorbell* or *doorbell interrupt* is often used to describe a mechanism whereby a software system can signal or notify a hardware device that there is some work to be done. Typically, the software system will place data in some well known and mutually agreed upon memory location(s), and "ring the doorbell" by writing to a different memory location. This different memory location is often called the doorbell region, and there may even be multiple doorbells serving different purposes in this region. It's this act of writing to the doorbell region of memory that "rings the bell" and notifies the hardware device that the data is ready and waiting. The hardware device would now know that the data is valid and can be acted upon. It would typically write the data to a hard disk drive, or send it over a network, or encrypt it, etc.

Doorbell interrupts can be compared to Message Signaled Interrupts, as they have some similarities.

5.6 BIOS Interrupt Calls

BIOS Interrupt Calls are a facility that DOS programs, and some other software such as boot loaders, use to invoke the BIOS's facilities. Some operating systems also use the BIOS to probe and initialize hardware resources during their early stages of booting.

5.7 Interrupt Handler / Interrupt Service Routine (ISR)

An *interrupt handler*, also known as an *interrupt service routine* (ISR), is a callback subroutine in an operating system or device driver whose execution is triggered by the reception of an interrupt. Interrupt handlers have a multitude of functions, which vary based on the reason the interrupt was generated and the speed at which the Interrupt Handler completes its task.

An interrupt handler is a low-level counterpart of event handlers. These handlers are initiated by either hardware interrupts or interrupt instructions in software, and are used for servicing hardware devices and transitions between protected modes of operation such as system calls.

Parts of Interrupt Handler

In modern operating systems, interrupt handlers are divided into two parts:

1. First-Level Interrupt Handler (FLIH) / Hard Interrupt Handlers / Fast Interrupt Handlers / Top-Half of Interrupt.
2. Second-Level Interrupt Handlers (SLIH) / Interrupt Threads / Slow Interrupt Handlers / Bottom-Half of Interrupt.

A FLIH implements at minimum platform-specific interrupt handling similarly to interrupt routines. In response to an interrupt, there is a context switch, and the code for the interrupt is loaded and executed. The job of a FLIH is to quickly service the interrupt, or to record platform-specific critical information which is only available at the time of the interrupt, and schedule the execution of a SLIH for further long-lived interrupt handling.

FLIHs which service hardware typically mask their associated interrupt (or keep it masked as the case may be) until they complete their execution. A (unusual) FLIH which unmasks its associated interrupt before it completes is called a reentrant interrupt handler. Reentrant interrupt handlers might cause a stack overflow from multiple preemptions by the same interrupt vector, and so they are usually avoided. In a priority interrupt system, the FLIH also (briefly) masks other interrupts of equal or lesser priority.

A SLIH completes long interrupt processing tasks similarly to a process. SLIHs either have a dedicated kernel thread for each handler, or are executed by a pool of kernel worker threads. These threads sit on a run queue in the operating system until processor time is available for them to perform processing for the interrupt. SLIHs may have a long-lived execution time, and thus are typically scheduled similarly to threads and processes.

It is worth noting that in many systems the FLIH and SLIH are referred to as upper halves and lower halves, hardware and software interrupts, or a derivation of those names.

5.8 Interrupt Vector and Interrupt Vector Table (IVT) / Dispatch Table

An *interrupt vector* is the memory address of an interrupt handler, or an index into an array called an *interrupt vector table* or *dispatch table*. Interrupt vector tables contain the memory addresses of interrupt handlers. When an interrupt is generated, the processor saves its execution state via a context switch, and begins execution of the interrupt handler at the interrupt vector.

Interrupt Descriptor Table (IDT)

The *Interrupt Descriptor Table* (IDT) is a data structure used by the x86 architecture to implement an interrupt vector table. The IDT is used by the processor to determine the correct response to interrupts and exceptions.

5.9 Programmable Interrupt Controller (PIC)

A programmable interrupt controller (PIC) is a device which allows priority levels to be assigned to its interrupt outputs. When the device has multiple interrupt outputs to assert, it will assert them in the order of their relative priority.

Common modes of a PIC include hard priorities, rotating priorities, and cascading priorities. PICs often allow the cascading of their outputs to inputs between each other.

One of the best known PICs, the 8259A, was included in the x86 PC. In modern times, this is not included as a separate chip in an x86 PC. Rather, its function is included as part of the motherboard's southbridge chipset. In other cases, it has been completely replaced by the newer Advanced Programmable Interrupt Controllers which support many more interrupt outputs and more flexible priority schemas.

CHAPTER 6
SYSTEM CALL

Concepts

6.1 System Call

A system call is the mechanism used by an application program to request service from the operating system for performing tasks which the said program does not have required permissions to execute in its own flow of execution.

System calls provide the interface between a process and the operating system. Most operations interacting with the system require permissions not available to a user level process, e.g. I/O performed with a device present on the system or any form of communication with other processes requires the use of system calls.

Why System Call

The fact that improper use of the system call can easily cause a system crash necessitates some level of control. The design of the microprocessor architecture on practically all modern systems (except some embedded systems) offers a series of privilege levels – the (low) privilege level in which normal applications execute limits the address space of the program so that it cannot access or modify other running applications nor the operating system itself. It also prevents the application from directly using devices (e.g. the frame buffer or network devices). But obviously many normal applications need these abilities; thus they can call the operating system. The operating system executes at the highest level of privilege and allows the applications to request services via system calls, which are often implemented through interrupts. If allowed, the system enters a higher privilege level, executes a specific set of instructions which the interrupting program has no direct control over, then returns control to the former flow of execution.

The Library as an Intermediary

Generally, systems provide a library that sits between normal programs and the operating system, usually an implementation of the C library (libc), such as glibc. This library handles the low-level details of passing information to the operating system and switching to supervisor mode, as well as any data processing and preparation which does not need to be done in privileged mode. Ideally, this reduces the coupling between the OS and the application, and increases portability.

6.2 Implementing System Calls

Implementing system calls requires a control transfer which involves some sort of architecture-specific feature. A typical way to implement this is to use a software interrupt or trap. Interrupts transfer control to the OS so software simply needs to set up some register with the system call number they want and execute the software interrupt.

An older x86 mechanism is called a call gate and is a way for a program to literally call a kernel function directly using a safe control transfer mechanism the OS sets up in advance. This approach has been unpopular, presumably due to the requirement of a far call which uses x86 memory segmentation and the resulting lack of portability it causes.

6.3 How System Calls are Carried Out / Steps in Making System Calls

System calls are performed in a series of steps. To make this concept clearer, let us examine the *read* system call.

In preparation for calling the *read* library procedure, which actually makes the *read* system call, the calling program first pushes the parameters onto the stack, as shown in steps 1-3 in *figure* 5.3. C and C++ compilers push the parameters onto the stack in reverse order for historical reasons (having to do with making the first parameter to *printf*, the format string, appear on top of the stack). The first and third parameters are called by value, but the second parameter is passed by reference, meaning that the

address of the buffer (indicated by &) is passed, not the contents of the buffer.

Then comes the actual call to the library procedure (step 4). This instruction is the normal procedure call instruction used to call all procedures.

The library procedure, possibly written in assembly language, typically puts the system call number in a place where the operating system expects it, such as a register (step 5).

Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel (step 6).

The kernel code that starts examines the system call number and then dispatches to the correct system call handler, usually via a table of pointers to system call handlers indexed on system call number (step 7).

At that point the system call handler runs (step 8).

Once the system call handler has completed its work, control may be returned to the user-space library procedure at the instruction following the TRAP instruction (step 9).

This procedure then returns to the user program in the usual way procedure calls return (step 10).

To finish the job, the user program has to clean up the stack, as it does after any procedure call (step 11). Assuming the stack grows downward, as it often does, the compiled code increments the stack pointer exactly enough to remove the parameters pushed before the call to *read*. The program is now free to do whatever it wants to do next.

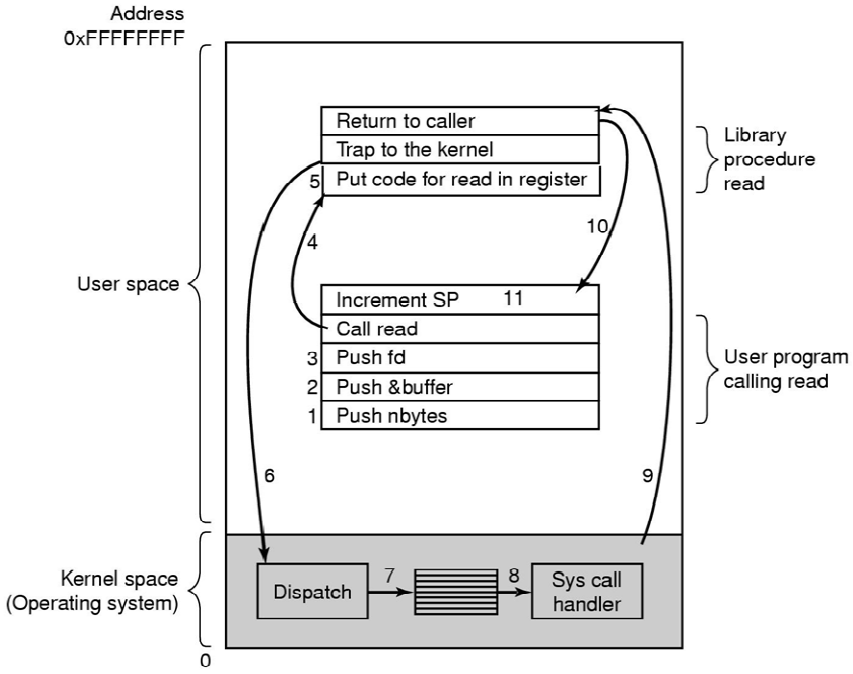


Figure 6.3: The 11 steps in making the system call `read(id, buffer, nbytes)`.

CHAPTER 7
FILE APIS

APIs

7.1 The open API

```
#include <sys/types.h>     //for mode_t
#include <fcntl.h>         //for open()
int open(const char *path_name, int access_mode, mode_t permission);
```

Opens a file.

Parameters:

path_name – The path name of a file.

access_mode – Can be any of the following:

O_RDONLY	–	Opens the file for read-only
O_WRONLY	–	Opens the file for write-only
O_RDWR	–	Opens the file for read and write

Furthermore, one or more of the following modifier flags can be specified by bitwise-ORing them with one of the above access mode flags to alter the access mechanism of the file:

O_APPEND	–	Appends data to the end of the file
O_CREAT	–	Creates the file if it does not exist

permission – This argument is required only if the O_CREAT flag is set in the *access_mode* argument. It specifies the access permission of the file for its owner, group member and all other people. Its value is usually specified as an octal integer literal, such as 0764. Specifically, the left-most, middle and right-most digits of an octal value specify the access permission for owner, group, and others, respectively.

Returns:

File descriptor on success, -1 on failure.

7.2 The read API

```
#include <sys/types.h>     //for size_t
#include <unistd.h>        //for open()
size_t read(int fd, void *buf, size_t size);
```

Fetches a fixed size block of data from a file referenced by a given file descriptor.

Parameters:

fd – The file descriptor that refers to an opened file.

buf – The address of a buffer holding any data read.

size – Specifies how many bytes of data are to be read from the file.

Returns:

The number of bytes of data successfully read and stored in the *buf* argument.

7.3 The `write` API

```
#include <sys/types.h>     //for size_t
#include <unistd.h>        //for write()
size_t write(int fd, void *buf, size_t size);
```

Puts a fixed size block of data to a file referenced by a given file descriptor.

Parameters:

fd – The file descriptor that refers to an opened file.

buf – The address of a buffer which contains data to be written to the file.

size – Specifies how many bytes of data are in the *buf* argument.

Returns:

The number of bytes of data successfully written to a file.

7.4 The `close` API

```
#include <unistd.h>        //for close()
int close(int fd);
```

Disconnects a file from a process.

Parameters:

fd – The file descriptor that refers to an opened file.

Returns:

0 on success, -1 on failure.

7.5 The `lseek` API

```
#include <sys/types.h>     //for off_t
#include <unistd.h>        //for lseek()
off_t lseek(int fd, off_t pos, int where);
```

Used to perform random access of data by changing the file offset to a different value.

Parameters:

fd – The file descriptor that refers to an opened file.

pos – The byte offset to be added to a reference location in deriving the new file offset value.

where – Specifies the reference location. Possible values are:

SEEK_CUR	–	Current file pointer address
SEEK_SET	–	The beginning of a file
SEEK_END	–	The end of a file

Returns:

The new file offset address where the next read or write operation will occur on success, -1 on failure.

CHAPTER 8
PROCESSES

Theories and Concepts

8.1 The `fork` API (Used to create a child process) [POSIX.1]

- Creates a child process whose address space is duplicate to that of its parent.
- Both the child and the parent process will be scheduled by the kernel to run independently, and the order of which process will run first is implementation-dependent.
- Both processes will resume their execution at the return of the `fork` call.
- After the fork call, the return value is used to distinguish whether a process is the parent or the child. In this way, the parent and child processes can do different tasks concurrently.

```
#include <sys/types.h>      //for pid_t
#include <unistd.h>         //for fork()
pid_t[1] fork(void);
```

Returns:

Returns the child PID (Process ID) to parent process and 0 to child process on success, -1 on failure.

`errno` conditions:

ENOMEM	There is insufficient memory to create the new process
EAGAIN	The number of processes currently existing in a system exceeds a system-imposed limit, so try the call again later.

Example:

```
#include <sys/types.h>          //for pid_t
#include <stdio.h>              //for NULL, stderr, fprintf()
#include <unistd.h>             //for fork(), execlp()
#include <sys/wait.h>           //for wait()
#include <stdlib.h>             //for exit()

int main() {
    pid_t pid;

    //fork a child process
    pid = fork();
    if (pid < 0) {              //error occurred
        perror("Fork Failed.");
        exit(-1);
    } else if (pid == 0) {      //child process
        execl("/bin/ls", "ls", NULL);
    } else {                    //parent process
        wait(NULL);             //parent will wait for the child to complete
        printf("Child Completed.");
        exit(0);
    }
}
```

8.2 The `vfork` API (Used to create a child process) [BSD UNIX and System V.4]

```
#include <sys/types.h>      //for pid_t
#include <unistd.h>         //for vfork()
pid_t[2] vfork(void);
```

`vfork` has similar function as `fork`, and it returns the same possible values as does `fork`.

[1] `pid_t` is defined as `unsigned int`.
[2] `pid_t` is defined as `unsigned int`.

Why `vfork`?

The idea of `vfork` is that many programs call exec (in child processes) right after `fork`. Thus, it will improve the system efficiency if the kernel does not create a separate virtual address space for the child process until exec is executed.

This is what happens in `vfork`: After the function is called, the kernel suspends the execution of the parent process while the child process is executing in the parent's virtual address space. When the child process calls `exec` or `_exit`, the parent will resume execution, and the child process will either get its own virtual address space after `exec` or will terminate via the `_exit` call.

Problems with `vfork`

`vfork` is unsafe to use, because:

1. If the child process modifies any data of the parent (e.g., closes files or modifies variables) before it calls `exec` or `_exit`, those changes will remain when the parent process resumes execution. This may cause unexpected behavior in the parent.
2. The child should not call exit or return to any calling function, because this will cause the parent's stream files being closed or modify the parent run-time stack, respectively.

Trade-off between the advantages and disadvantages of `fork` and `vfork`

The latest UNIX systems have improved the efficiency of `fork` by allowing parent and child processes to share a common virtual address space until the child calls either the `exec` or `_exit` function. If either the parent or the child modifies any data in the shared virtual address space, the kernel will create new memory pages that cover the virtual address space modified. Thus, the process that made changes will reference the new memory pages with the modified data, whereas the counterpart process will continue referencing the old memory pages.

8.3 The exec family functions [POSIX.1]

Causes a calling process to change its context and execute a different program.

```
#include <unistd.h>
int execl(const char *path, const char *arg, ..., NULL)
int execv(const char *path, char *const argv[])
int execlp(const char *file, const char *arg, ..., NULL)
int execvp(const char *file, char *const argv[])
int execle(const char *path, const char *arg, ..., NULL, char *const env[])
int execve(const char *path, char *const argv[], char *const env[])
```

Parameters

path	-	The program file to be executed without its path.
file	-	The program file to be executed along with its path.
arg / argv	-	Command line arguments.
env	-	Environment variables to be set for the new process.

Returns

There shall be no return from a successful `exec`, because the calling process image is overlaid by the new process image.

Explanation of mnemonics used in these functions

l	-	Arguments are provided via a null-terminated *list*.
v	-	Arguments are provided via an array (*vector*) of string, where the last element of the array must be NULL.
p	-	The user's full *path* is searched for the given file.
e	-	A new *environment* is also supplied for the new process.

Examples of using exec functions

```
//execl
int ret = execl("/bin/vi", "vi", "/home/docs/class.txt", NULL);
```

```
//execvp
char *args[] = {"vi", "/home/docs/class.txt", NULL};
int ret = execvp("vi", args);

//execve
char *args[] = {"vi", "/home/docs/class.txt", NULL};
char *env[] = {"HOME=/usr/home", "LOGNAME=home", NULL};
int ret = execve("vi", args, env);
```

8.4 The pipe API [POSIX.1]

- The pipe system call creates a communication channel between two *related* processes (for example, between a parent process and a child process, or between two sibling processes with the same parent).
- Specifically the function creates a pipe device file that serves as a temporary buffer for a calling process to read and write data with another process. The pipe device file has no assigned name in any file system; thus, it is called an *unnamed* pipe.

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

Parameters

fifo - An array of two integers that are assigned by the pipe API. On most UNIX systems, a pipe is unidirectional in that *fifo[0]* is a file descriptor that a process can use to *read* data from the pipe. And *fifo [1]* is a different file descriptor that a process can use to *write* data into the pipe.

Returns

0 on success, -1 on failure.

errno conditions:

EFAULT	The *fifo* argument is illegal
ENFILE	The system file table is full

How a pipe is created

- **Between parent and child processes:**

  The parent calls pipe to create a pipe, then forks a child. Since the child has a copy of the parent file descriptors, the parent and child can communicate through the pipe via their respective *fifo[0]* and *fifo [1]* descriptors.

- **Among sibling processes:**

  The parent calls pipe to create a pipe, then forks two or more child processes. The child processes can communicate through the pipe via their respective *fifo[0]* and *fifo[1]* descriptors.

Notable points on pipe / How a pipe works

How data are written into and read from pipes

- Data stored in a pipe is accessed sequentially in a first-in-first-out manner. A process can't use *lseek* to do random data access of a pipe.
- Data is consumed from a pipe when it is read.

Synchronization while reading and writing pipes

- Because the buffer associated with a pipe device file has a finite size (PIPE_BUF), when a process tries to write to a pipe that is already filled with data, it will be blocked by the kernel until another process reads sufficient data from the pipe to make room for the blocked process. Conversely, if a pipe is empty and a process tries to read data from a pipe, it will be blocked until another process writes data into the pipe.

How many processes can concurrently attach to either end of a pipe

- There is no limit on how many processes can concurrently attach to either end of a pipe. But considering its drawbacks, it is conventional to set up a pipe as a *unidirectional* communication channel between *only* two processes such as one process will be designated as the sender of the pipe and the other process will be the receiver of the pipe. For example, if A and B need a bidirectional communication channel, they will create two pipes: one for process A to write data to process B, and vice versa.

How a pipe is closed and what if a process tries to read/write after the pipe has been closed from the other end

- A pipe is deallocated once *all* processes close their file descriptors referencing the pipe.
- If there is no file descriptor in the process to reference the write-end of a pipe, the pipe write-end is considered *close* and any process attempting to read data from the pipe will receive the *remaining* data.
- After the write-end of a pipe is closed, once *all* data in the pipe is consumed, a process that attempts to read more data from the pipe will receive an end-of-file.
- On the other hand, if no file descriptor references the read-end of a pipe, and a process attempts to write data into the pipe, it will receive the SIGPIPE (broken pipe) signal from the kernel. This is because no data written to the pipe can be retrieved by the process; thus the write operation is considered illegal. The process that does the write will be penalized by the signal (the default action of the signal is to abort the process).

Implementation of pipes

- Pipe is used by the UNIX shell to implement the command pipe ('|') for connecting the standard output of one process to the standard input of another process.

CHAPTER 9
SIGNALS

Theories and Concepts

9.1 Signal

Signals are software version of hardware interrupts, which are triggered by events and are posted on a process to notify it that something has happened and requires some action.

9.2 Who generate signals

1. **Kernel** – e.g., when a process performs a divide-by-zero operation, the kernel will send the process a signal to interrupt it.
2. **User** – e.g., when a user hits the `<Ctrl-C>` key at the keyboard, the kernel will send the foreground process a signal to interrupt it.
3. **Process** – e.g., a parent and its child processes can send signals to each other for process synchronization.

9.3 Signal reference list

Signals are defined as integer flags. The `<bits/signum.h>` header depicts the list of signals defined for a UNIX system.

```
/* Signals.  */
#define SIGHUP          1       /* Hangup (POSIX).  */
#define SIGINT          2       /* Interrupt (ANSI).  */
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGILL          4       /* Illegal instruction (ANSI).  */
#define SIGTRAP         5       /* Trace trap (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGIOT          6       /* IOT trap (4.2 BSD).  */
#define SIGBUS          7       /* BUS error (4.2 BSD).  */
#define SIGFPE          8       /* Floating-point exception (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define SIGUSR1         10      /* User-defined signal 1 (POSIX).  */
#define SIGSEGV         11      /* Segmentation violation (ANSI).  */
#define SIGUSR2         12      /* User-defined signal 2 (POSIX).  */
#define SIGPIPE         13      /* Broken pipe (POSIX).  */
#define SIGALRM         14      /* Alarm clock (POSIX).  */
#define SIGTERM         15      /* Termination (ANSI).  */
#define SIGSTKFLT       16      /* Stack fault.  */
#define SIGCLD          SIGCHLD /* Same as SIGCHLD (System V).  */
#define SIGCHLD         17      /* Child status has changed (POSIX).  */
#define SIGCONT         18      /* Continue (POSIX).  */
#define SIGSTOP         19      /* Stop, unblockable (POSIX).  */
#define SIGTSTP         20      /* Keyboard stop (POSIX).  */
#define SIGTTIN         21      /* Background read from tty (POSIX).  */
#define SIGTTOU         22      /* Background write to tty (POSIX).  */
#define SIGURG          23      /* Urgent condition on socket (4.2 BSD).  */
#define SIGXCPU         24      /* CPU limit exceeded (4.2 BSD).  */
#define SIGXFSZ         25      /* File size limit exceeded (4.2 BSD).  */
#define SIGVTALRM       26      /* Virtual alarm clock (4.2 BSD).  */
#define SIGPROF         27      /* Profiling alarm clock (4.2 BSD).  */
#define SIGWINCH        28      /* Window size change (4.3 BSD, Sun).  */
#define SIGPOLL         SIGIO   /* Pollable event occurred (System V).  */
#define SIGIO           29      /* I/O now possible (4.2 BSD).  */
#define SIGPWR          30      /* Power failure restart (System V).  */
#define SIGSYS          31      /* Bad system call.  */
#define SIGUNUSED       31
```

Linux supports the standard signals listed below. Several signal numbers are architecture dependent, as indicated in the *Value* column. (Where three values are given, the first one is usually valid for *alpha* and *sparc*, the middle one for *i386*, *ppc* and *sh*, and the last one for *mips*. A '-' denotes that a signal is absent on the corresponding architecture.)

The entries in the *Action* column of the tables below specify the default disposition for each signal, as follows:

Term - Default action is to terminate the process.
Ign - Default action is to ignore the signal.
Core - Default action is to terminate the process and dump core.
Stop - Default action is to stop the process.
Cont - Default action is to continue the process if it is currently stopped.

First, the signals described in the original POSIX.1-1990 standard.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hang-up detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from `abort()` function
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from `alarm()` function
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Next, the signals not in the POSIX.1-1990 standard but described in SUSv2 and POSIX.1-2001.

Signal	Value	Action	Comment
SIGBUS		Term	BUS error (4.2 BSD)
SIGPOLL	SIGIO	Term	Pollable event (Sys V). Synonym of SIGIO
SIGPROF	27,27,29	Term	Profiling timer expired
SIGSYS	12,-,12	Core	Bad argument to routine (SVr4)
SIGTRAP	5	Core	Trace/breakpoint trap
SIGURG	16,23,21	Ign	Urgent condition on socket (4.2BSD)
SIGVTALRM	26,26,28	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	24,24,30	Core	CPU time limit exceeded (4.2BSD)
SIGXFSZ	25,25,31	Core	File size limit exceeded (4.2BSD)

Next, various other signals.

Signal	Value	Action	Comment
SIGIOT			IOT trap (4.2 BSD)
SIGEMT	7,-,7	Term	
SIGSTKFLT	-,16,-	Term	Stack fault on coprocessor (unused)
SIGIO	23,29,22	Term	I/O now possible (4.2BSD)
SIGCLD	-,-,18	Ign	A synonym for SIGCHLD
SIGPWR	29,30,19	Term	Power failure (System V)
SIGINFO	29,-,-		A synonym for SIGPWR
SIGLOST	-,-,-	Term	File lock lost
SIGWINCH	28,28,20	Ign	Window resize signal (4.3BSD, Sun)
SIGUNUSED	-,31,-	Term	Unused signal (will be SIGSYS)

9.4 How a process reacts to a pending signal

When a signal is pending on a process, the process can

1. **Accept the default action of the signal,** which for most signals will terminate the process (exceptions are the SIGCHLD[1] and SIGPWR[2] signals).
2. **Ignore the signal.** The signal will be discarded and it has no effect whatsoever on the recipient process.
3. **Invoke a user-defined function.** The function is known as a *signal handler routine*.

9.5 Some notable points on signals

- **A process may set up *per signal* handling mechanisms**, such that it ignores some signals, catches some other signals, and accepts the default action from the remaining signals. This can be done using the `sigaction` (POSIX) or `signal` (ANSI) APIs.
- **A process may change the handling of certain signals in its course of execution**. For example, a signal may be ignored at the beginning, and then set to be caught, and after being caught, set to accept the default action.
- A signal is said to be **caught** when the signal handler routine is *called*. A signal is said to have been **delivered** if it has been *reacted* by the recipient *process*.
- **The default action for most signals is to terminate the recipient process**. **Exceptions are the SIGCHLD[3] and SIGPWR[4] signals**. The default action for SIGCHLD is to ignore the signal. However, although the default action for SIGPWR is to ignore the signal in *most* UNIX systems, in Linux, its default action is to abnormally terminate the recipient process.
- **Some signals will generate a core file for the aborted process** so that users can trace back the state of the process when it was aborted, and thus debug the program.
- **Most signals can be ignored, caught or blocked except the SIGKILL and SIGSTOP signals.** The SIGKILL signal is sent to a process to cause it to terminate immediately.[5] The SIGSTOP signal halts (suspends) a process execution.[6]
- A process initially inherits the parent's signal mask when it is created, but any pending signals for the parent process are not passed on.
- If there are different signals pending on a process, the order in which they are sent to a recipient process is undefined.
- If multiple instances of the same signal are pending on a process, it is implementation-dependent on whether a single instance or multiple instances of the signal will be delivered to the process.
- If a signal is specified to be ignored and blocked, it is implementation-dependent on whether such a signal will be discarded or left pending when it is sent to the process.

---

[3] Sent to a parent process when its child process has terminated.

[4] Sent to all processes when a power failure is imminent; for example, when the battery is running low on a laptop. Programs would be expected to synchronize their state to permanent storage to ensure that if the system powers off, data is not lost.

[5] The SIGKILL signal cannot be caught, but another signal SIGTERM can be caught. Because SIGKILL gives the process no opportunity to do cleanup operations on terminating, in most system shutdown procedures, an attempt is first made to terminate processes using SIGTERM before resorting to SIGKILL.

The `kill` command in a UNIX shell can be used to send a process the SIGKILL or SIGTERM signals. A process can be sent a SIGTERM signal in three ways (the process ID is '1234' in this case):

- `kill 1234`
- `kill -TERM 1234`
- `kill -15 1234`

The process can be sent a SIGKILL signal in two ways:

- `kill -KILL 1234`
- `kill -9 1234`

[6] A companion signal to SIGSTOP is SIGCONT (which can be generated using the `fg` [foreground] command), which resumes a process execution after it has been stopped. SIGSTOP and SIGCONT signals are used for job control in UNIX.

9.10 Use of some common signals and disambiguation among *seems-to-be-similar* signals

Process terminating signals (SIGKILL, SIGTERM, SIGQUIT[7], SIGABRT, SIGINT[8])

Signal	Generator	Dumps core	Can be caught	Can be ignored	Can be blocked
SIGKILL	Anybody	N	N	N	N
SIGTERM	Anybody	N	Y	Y	Y
SIGQUIT	User	Y	Y	Y	Y
SIGABRT	Process	Y	Y	N	N
SIGINT	User	N	Y	Y	Y

Process suspending signals (SIGSTOP, SIGTSTP[9])

SIGTSTP is sent to a process by its *controlling terminal* when the user presses the special SUSP key combination (usually `<Ctrl-Z>`) or enters the `bg` (background) command. On the other hand, SIGSTOP can be sent to a process by the kernel (for job control) or by another process.

Again, unlike the SIGSTOP signal, a process can register a signal handler for or ignore SIGTSTP.

9.11 How a signal is delivered to a process

When a signal is generated for a process,

1. The kernel sets the corresponding signal flag in the PCB of the recipient process. If the process is asleep, the kernel will awaken the process by scheduling it as well.
2. The kernel consults the array containing signal handling specifications (found in the U-area of a process) to find out how the process will react to the pending signal.
3. The kernel sets up the process to execute that function immediately, and the process will return to its current point of execution if the signal handler function does not terminate the process.

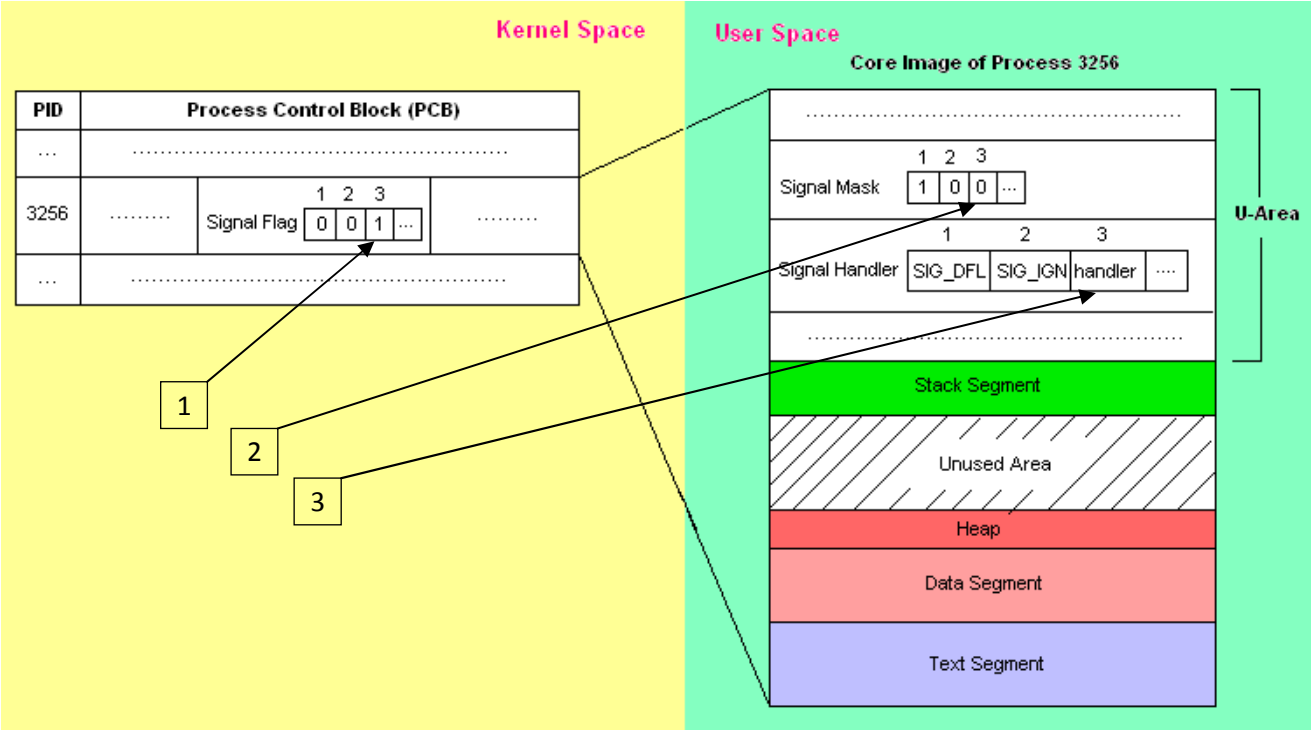


Figure 9.11: How a signal is delivered to a process.

[7] Can be generated by the `<Ctrl-\>` keys.

[8] Can be generated typically by the `<Ctrl-C>` key; and in some systems, by the `<Delete>` or `<Break>` keys.

[9] TSTP in SIGTSTP is an abbreviation for *TTY Stop*, indicating that stop has been typed at the tty. (Tty is itself an abbreviation for *teletypewriter*, an archaic type of computer terminal.)

9.12 **How multiple instances of the *same* signal is delivered to a process**

Each signal flag in PCB records only *whether* a signal is pending, but not *how many* of them are present.

- **UNIX System V.2 and earlier versions**
  - When a signal is caught, the kernel resets the signal handler (for that signal) in the recipient process U-area, and then calls the signal handler. Thus, the process will catch only the first signal and subsequent instances of the signal will be handled in the *default* manner.
- **UNIX System V.4, BSD UNIX 4.2 (and later versions), and POSIX.1**
  - When a signal is caught, the kernel *does not* reset the signal handler. Furthermore, the kernel will block further delivery of the same signal to the process *until* the signal handler function has completed execution.

9.13 **Types of signals**

Signals can be grouped from many different points of views:

- **Synchronous and Asynchronous Signals**

  A *synchronous* signal is a signal that is generated due to some action attributable to the signal receiving process. For example, SIGBUS, SIGFPE, SIGSEGV, SIGILL etc.

  An *asynchronous* signal is a signal that is generated from *outside* the signal receiving process (from an interrupt, another process, OS kernel, user or device). For example, SIGINT, SIGCHLD, SIGKILL, SIGTERM etc.

- **Catchable and Uncatchable Signals**

  Signals for which user-defined handler functions *can* be executed are called *catchable* signals.

  Signals for which user-defined handler functions *cannot* be defined are called *uncatchable* signals.

  Most signals are catchable except the SIGKILL and SIGSTOP signals.

- **Maskable and Non-Maskable Signals**

  Signals that *can* be masked (i.e., blocked) are called maskable signals.

  Signals that *cannot* be masked are called non-maskable signals.

  Most signals are maskable except the SIGKILL, SIGSTOP and SIGCONT signals.

9.14 **The `signal` API (Used to define / install the per-signal handling method) [ANSI defined; used in System V.2 and earlier versions]**

```
#include <signal.h>
void (*signal(int signal_num, void(*handler)(int)))(int);
```

Parameters:

signal_num – A signal identifier (e.g. SIGINT) as defined in the `<bits/signum.h>` header.
handler – The function pointer to a user-defined signal handler function which must take an integer parameter and return nothing.

Returns:

The previous signal handler for a signal on success (can be used to restore the signal handler for a signal after it has been altered). SIG_ERR on failure.

`errno` Conditions:

EINVAL An invalid *signal_num* is specified; or you tried to ignore or provide a handler for SIGKILL or SIGSTOP.

Example:

```
1 #include <stdio.h>
2 #include <signal.h>
3
```

```
/* Signal handler function */
void catch_sig(int sig_num) {
    signal(sig_num, catch_sig);   /* Re-install the signal handler */
    printf("Cought signal: %d", sig_num);
}

/* Main function */
int main() {
    signal(SIGINT, catch_sig);
    signal(SIGSEGV, SIG_DFL);
    void (*old_handler)(int) = signal(SIGINT, SIG_IGN);
    /* .....Do mission-critical task..... */
    signal(SIGINT, old_handler);

    return 0;
}
```

The SIG_IGN (specifies a signal to be ignored) and SIG_DFL (specifies to accept the default action of a signal) are manifest constants defined in the `<signal.h>` header:

```
#define SIG_IGN (void(*)(int))1[10]
#define SIG_DFL (void(*)(int))0
```

9.15 **The `sigset` API (Used to define the per-signal handling method) [Used in System V.3, V.4]**

```
#include <signal.h>
void (*sigset(int signal_num, void(*handler)(int)))(int);
```

This API is similar to the `signal` API. However, whereas `signal` is unreliable, `sigset` is reliable.

9.16 **Masking signals**

- *Why masking out signals?*
  - An application wants to ignore certain signals.
  - Avoid race conditions when another signal happens in the middle of the signal handler's execution.
- *Two ways to mask signals*
  - Affect all signal handlers – using `sigprocmask` API
  - Affect a specific handler – using `sigaction` API

9.17 **The `sigprocmask` API (Used by a process to query or set its signal mask) [POSIX.1]**

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *new_mask, sigset_t *old_mask);
```

Parameters:

new_mask – The set of signals to be set or reset in a calling process signal mask. If this is NULL, the *how* argument will be ignored and the current process signal will not be altered.

how – Indicates the way in which the set is changed. Possible values:

SIG_BLOCK	Adds the signals specified in the *new_mask* argument to the calling process signal mask.
SIG_UNBLOCK	Removes the signals specified in the *new_mask* argument from the calling process signal mask.
SIG_SETMASK	Overrides the calling process signal mask with the value specified in the *new_mask* argument.

old_mask – The address of a `sigset_t`[11] variable that will be assigned the calling process's original signal mask prior to a `sigprocmask` call. If this is NULL, no previous signal mask will be returned.

[10] "`(void(*)(int))1`" means *1* is type-casted to the function pointer `void(*)(int)`.

Returns:

0 on success, -1 on failure.

errno Conditions:

EINVAL Invalid *new_mask* and/or *old_mask* addresses.

Example:

See the example at 9.18.

9.18 **The *sigsetops* APIs (Used to set, reset and query the presence of signals in a signal set) [POSIX.1 and BSD UNIX]**

```
#include <signal.h>
int sigemptyset(sigset_t *sigmask);
```

Clears all signal flags in the *sigmask* argument.

```
int sigfillset(sigset_t *sigmask);
```

Sets all signal flags in the *sigmask* argument.

```
int sigaddset(sigset_t *sigmask, const int signal_mask);
```

Sets the flag corresponding to the *signal_num* signal in the *sigmask* argument.

```
int sigdelset(sigset_t *sigmask, const int signal_mask);
```

Clears the flag corresponding to the *signal_num* signal in the *sigmask* argument.

```
int sigismember(const sigset_t *sigmask, const int signal_num);
```

Returns 1 if the flag corresponding to the *signal_num* signal in the *sigmask* argument is set, 0 if it is not set, and -1 if the call fails.

Return values (for the first four APIs):

0 on success, -1 on failure.

errno Conditions:

EINVAL The *sigmask* and/or *signal_num* arguments are invalid.

Example:

The following example checks whether the SIGINT signal is present in a process signal mask and adds it to the mask if it is not there. Then it clears the SIGSEGV signal from the process signal mask.

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

int main() {
    sigset_t sigmask;
    sigemptyset(&sigmask);                              //initialize set (a must)

    if (sigprocmask(NULL, NULL, &sigmask) != -1) {      //get current signal mask
        sigaddset(&sigmask, SIGINT);                    //set SIGINT flag
    } else {
        perror("sigprocmask API failed.");
    }

    sigdelset(&sigmask, SIGSEGV);                       //clear SIGSEGV flag
    if (sigprocmask(SIG_SETMASK, &sigmask, NULL) == -1) {  //set new signal mask
        perror("sigprocmask API failed.");
    }
}
```

[11] The `sigset_t` is a data type defined in the `<signal.h>` header. It contains a collection of bit-flags, with each bit-flag representing one signal defined in a given system.

```
#include <bits/sigset.h>
#define _SIGSET_NWORDS (1024 / (8 * sizeof(unsigned long int)))
typedef struct {
    unsigned long int __val[_SIGSET_NWORDS];
} __sigset_t;
```

9.19 The `sigpending` API (Used to query which signals are pending for a process) [POSIX.1]

```
#include <signal.h>
int sigpending(sigset_t *sigmask);
```

Parameters:

sigmask – The address of a `sigset_t` variable which is assigned the set of signals pending for the calling process.

Returns:

0 on success, -1 on failure.

9.20 The `sigaction` API (Used to define the per-signal handling method) [POSIX.1]

```
#include <signal.h>
int sigaction(int signal_num, struct sigaction *action, struct sigaction *old_action);
```

Parameters:

signal_num – A signal identifier (e.g. SIGINT) as defined in the `<bits/signum.h>` header.
action – Pointer to a structure containing the signal handling information for *signal_num*.
old_action – Previous signal handling information for *signal_num*.

The `struct sigaction` data type as defined in the `<bits/sigaction.h>` header is:

```
struct sigaction {
     void       (*sa_handler)(int);
     sigset_t   sa_mask;
     int        sa_flag;
};
```

Descriptions of the fields in this structure are as follows:

sa_handler – SIG_IGN, SIG_DFL or a user-defined signal handler function.
sa_mask – Specifies *additional* signals that a process wishes to block (*besides* those signals currently specified in the process' signal mask *and* the *signal_num* signal) when it is handling the *signal_num* signal.
sa_flag – Specifies special handling for certain signals. POSIX.1 defines only two values for this flag:

SA_NOCHLDSTOP – Kernel will generate the SIGCHLD signal to a process when its child process has *terminated*, but *not* when the child process has been *stopped*.
0 – Kernel will send the SIGCHLD signal to the calling process whenever its child process is *either* terminated *or* stopped.

Returns:

0 on success, -1 on failure.

`errno` Conditions:

EINVAL The *signal_num* argument is an invalid or unsupported number.

Example:

In this example, we're going to add the SIGTERM signal to the blocking list. Then, we're going to set up a signal handler for the SIGINT signal. However, while setting up the handler for SIGINT, we'll add SIGSEGV to the masked signals so that SIGSEGV is also blocked (along with SIGTERM) *while* the handler for SIGINT is executing. Lastly, we wait for the SIGINT signal.

```
1 #include <stdio.h>
2 #include <signal.h>
3
4 void callme(int sig_num) {
5     printf("\n\nCought signal: %d\n\n", sig_num);
6 }
```

```c

int main() {
    sigset_t    sigmask;
    struct      sigaction action, old_action;

    /* add SIGINT flag to the signal mask for current process */
    sigemptyset(&sigmask);                          //initialize signal mask set
    if (sigaddset(&sigmask, SIGTERM) == -1 ||     //set SIGINT flag
        sigprocmask(SIG_SETMASK, &sigmask, 0) == -1) {  //set the new signal mask
            perror("Error when setting signal mask.");
    }

    /* set up a signal handler for SIGINT signal */
    sigemptyset(&action.sa_mask);           //initialize set
    sigaddset(&action.sa_mask, SIGSEGV);    //add SIGSEGV to the masked signals
    action.sa_handler = callme;             //set signal handler
    action.sa_flags = 0;                    //leave default
    //set the new signal handler
    if (sigaction(SIGINT, &action, &old_action) == -1) {
        perror("sigaction API failed.");
    }

    /* wait for signal interruption */
    pause();

    printf("Program exiting...");

    return 0;
}
```

9.21 **The `kill` API (Used to send a signal to a related *process* by another *process*) [POSIX.1]**

- The `kill` API is a simple means for interprocess communication or control.
- The sender and recipient processes must be related such that either the sender process real or effective user ID matches that of the recipient process, or the sender process has superuser privileges. For example, a parent and child process can send signals to each other via the `kill` API.

```c
#include <signal.h>
int kill(pid_t pid, int signal_num);
```

Parameters:

pid – The *pid* of the signal recipient process(es). Possible values of *pid* are:

- *a +ve value* – *pid* is a process ID. Sends *signal_num* to that process.
- 0 – Sends *signal_num* to all processes whose process group ID is the same as the calling process.
- -1 – Sends *signal_num* to all processes whose real user ID is the same as the effective user ID of the calling process. If the calling process effective user ID is the superuser user ID, *signal_num* will be sent to all processes in the system (except processes 0 and 1).
- *a -ve value* – Sends *signal_num* to all processes whose process group ID matches the absolute value of *pid*.

signal_num – The signal to be sent to the process(es) designated by the *pid* argument.

Returns:

0 on success, -1 on failure.

`errno` Conditions:

EINVAL	The *signal_num* argument is an invalid or unsupported number.
EPERM	You do not have the privilege to send a signal to the process or any of the processes in the process group named by *pid*.
ESCRH	The *pid* argument does not refer to an existing process or group.

9.22	**The `raise` API (Used to send a signal to the calling process)**
	`#include <signal.h>`
	`int raise(int signal_num);`
	Parameters:
	signal_num – The signal to be sent.
	Returns:
	0 on success, -1 on failure.
	`errno` Conditions:
	EINVAL The *signal_num* argument is an invalid or unsupported number.

Questions

9.1	**What is signal? What are the reasons to occur signal in a program? *[2005, Marks: 2]*** *See Theories & Concepts 9.1 and 9.2.*
9.2	**Write down the tasks of `raise()` and `signal()` functions. *[2005. Marks: 3]*** The `int raise(int sig)` function (system call) sends a signal *sig* to the executing thread or process. If a signal handler is called, the `raise()` function will not return until after the signal handler does.[12] The `signal` API is used to define the per-signal handling method. It takes two arguments – the signal for which a handler is to be defined, and a pointer to the signal handler routine. The `signal` API returns a pointer to the old signal handler routine.
9.3	**What are the purposes of a *signal*? Give at least two examples for each *catchable* and non-*catchable* signal. *[Mid-Term Exam -1, 2004/2006. Marks: 3]*** Purposes of signal – *see Theories & Concepts 9.2*. Examples of catchable signals – SIGINT, SIGTERM, SIGQUIT etc. Examples of non-catchable signals – SIGKILL and SIGSTOP.
9.4	**How is a signal delivered to a process?** *See Theories & Concepts 9.11*.
9.5	**Differentiate between signal and interrupt. *[2007. Marks: 2]***
9.6	**What do you understand by signal handler? Explain SIG_DFL and SIG_IGN. *[2007. Marks: 3]***
9.7	**Why *sigaction* is recommended instead of *signal* system call? *[2006. Marks: 2]***

[12] In case of sending signal to a *process* rather than a *thread*, the effect of the `raise()` function is equivalent to calling:
`kill(getpid(), sig);`

CHAPTER 10
INTERPROCESS COMMUNICATION

Theories and Concepts

10.1 **Interprocess Communication (IPC)**

IPC is a mechanism whereby two or more processes communicate with each-other to perform tasks.

These processes may interact in a *client-server* manner (that is, one or more *client* processes send data to a central server process and the server process responds to each client) or in a *peer-to-peer* fashion (that is, any process may exchange data with others).

Examples of applications that use IPC are database servers and their associated client programs (using the client-server model) and instant messaging systems (using the peer-to-peer model), where a messenger process communicates with other remote messenger processes to send and receive messages over the internet.

10.2 **IPC Methods**

- **Pipes (Named & Unnamed) [*UNIX System V, POSIX*]**

  Allows a communication channel between two *related* processes (for example, between a parent process and a child process, or between two sibling processes with the same parent) [*unnamed pipe*] or *unrelated* processes [*named pipe*] on the same machine to exchange data.

- **Messages [*UNIX System V, POSIX*]**

  Allow multiple processes on the same machine to communicate by sending and receiving messages (formatted data) among themselves.

- **Shared Memory [*UNIX System V, POSIX*]**

  Allows multiple processes on the same machine to share a common region of virtual memory, such that data written to a shared memory can be directly read and modified by other processes.

- **Sockets and TLI (Transport Level Interface) [*UNIX System V*]**

  Allow two processes on *different* machines to set up a direct, two-way communication channel. The *socket* method uses TCP or UDP and the *TLI* method uses STREAMS as the underlying data transport interface.

- **Semaphores [*UNIX System V, POSIX*]**

  Provide a set of system-wide variables that can be modified and used by processes on the same machine to synchronize their execution. Semaphores are commonly used with a shared memory to control the access of data in each shared memory region.

  Note that semaphores are used only to *synchronize* communication among processes, not to transfer data.

10.3 **How messages overcome the deficiencies of pipes (both named and unnamed)**

Problems with pipes

1. Multiple processes can attach to the read and write ends of a pipe, but there are no mechanisms provided to promote *selective* communication between a reader and a writer process.

For example, suppose there are processes, A and B, attached to the write end of a pipe and processes C and D attached to the read end of a pipe. If both A and B write data to the pipe such that A's data is read by C and B's data is read by D, there is no easy way for C and D to selectively read data that are destined for them only.

2. Data stored in a pipe is destroyed when both ends of the pipe have no process attached.

Thus, pipes and their stored data are transient[13] objects. They cannot be used by processes to exchange data reliably if they are not running in the *same* period of time.

How messages overcome the problems of pipes

1. Messages also allow multiple processes to access a central message queue. However, every process that deposits a message in the queue needs to specify a *message type* for the message. Thus, a recipient process can retrieve that message by specifying that same *message type*.

2. Messages stored in a message queue are persistent, even when there is no process referencing the queue. Messages are removed from a queue only when processes explicitly retrieve them.

10.4 Kernel Data Structure for Messages

The implementation of message queues in UNIX System V is analogous to the implementation of UNIX files.

There is a message queue table in a kernel address space that keeps track of all message queues created in a system. Each entry of the message table stores the following data for one message queue:

1. A key ID assigned by the process that created the queue. Other processes may specify this key to *open* the queue and get a descriptor for future access of the queue.
2. The creator and assigned user ID and group ID.
3. Read-write access permission of the queue.
4. The time and process ID of the last process that *sent* a message to the queue.
5. The time and process ID of the last process that *read* a message to the queue.
6. The pointer to a linked list of message records in the queue. Each message record stores one message of data and its assigned message type.

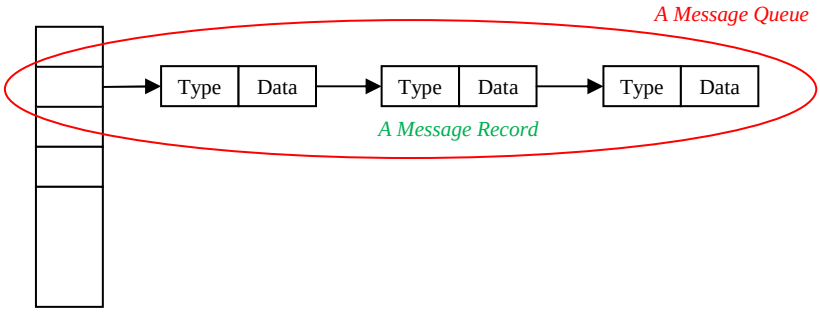


Figure 10.4: Kernel Data Structure for Message Queues.

10.5 How the message passing system works / Message Retrieve Procedure from Message Table

1. When a process sends a message to a queue, the kernel creates a new message record and puts it at the end of the message record linked list for the specified queue.

The message record stores the message type, message data length, and the pointer to another kernel data region where the actual message data is stored.

2. The kernel copies the message data from the sender process's virtual address into this kernel data region so that the sender process is free to terminate, and the message can still be read by another process in the future.

3. When a process retrieves a message from a queue, the kernel copies the message data from a message record to the process's virtual address and discards the message record.

[13] **Transient:** Lasting a very short time.

10.6 System-Imposed Limits on the Manipulation of Messages

System Limit	Indicates	Meaning
MSGMNI [*Max. No. of Instances*]	Message Table Length	The maximum number of message queues that may exist at any given time in a system.
MSGMAX	Message Data Length	The maximum number of bytes of data allowed for a message.
MSGMNB [*Max. No. of Bytes*]	Message Queue Length	The maximum number of bytes of all messages allowed in a queue.
MSGTQL [*Total Queues' Length*]	Number of Messages	The maximum number of messages in all queues allowed in a system.

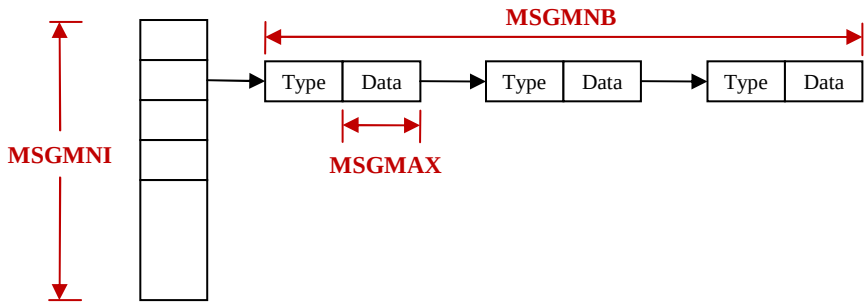


Figure 10.6: Visual representation of the system-imposed limits on the manipulation of messages.

Effects of The System-Imposed Limits on Processes

- If the current number of message queues is MSGMNI, any attempt to create a new message queue by a process will fail, until an existing queue is deleted by a process.
- If a process attempts to send a message whose size is larger than MSGMAX, the system call will fail.
- If a process attempts to send a message to a queue that will cause either the MSGMNB or MSGTQL limit to be exceeded, then the process will be blocked until one or more messages are retrieved from the queue and the message can be inserted into the queue without violating both the MSGMNB and the MSGTQL limits.

10.7 The UNIX APIs for Messages

The `msgget` API

```
#include <sys/types.h>      //for key_t
#include <sys/ipc.h>        //for IPC_CREAT, IPC_PRIVATE
#include <sys/msg.h>        //for msgget()

int msgget(key_t key, int flag);
```

Opens a message queue whose key ID is given in the *key* argument.

Parameters:

key – *Positive Integer*: The API opens a message queue whose key ID matches that value.

IPC_PRIVATE: The API allocates a new message queue to be used exclusively by the calling process.

The *private* message queue is usually allocated by a parent process, which then forks one or more child processes. The parent and child processes then use the shared memory to exchange data.

flag – 0: If there is no message queue whose key ID matches the given *key* value, the API fails. Otherwise, it returns the descriptor for that memory.

IPC_CREAT | *read-write access permission*: A new message queue (if none preexists) is created with the given key ID.

Returns:

Message queue descriptor on success, -1 on failure.

The `msgsnd` API

```
#include <sys/ipc.h>       //for IPC_NOWAIT
#include <sys/msg.h>       //for msgsnd()
int msgsnd(int msgd, const void* msgPtr, int len, int flag);
```

Sends a message (pointed to by *msgPtr*) to a message queue designated by the *msgd* descriptor.

Parameters:

msgd – The message descriptor as obtained from a `msgget` system call.

msgPtr – The actual value of the *msgPtr* argument is the pointer to an object that contains the actual message text and a message type to be sent. The following data type can be used to define an object for such purpose:

```
struct msgbuf {
    long msgtype;           //message type
    char msgtext[MSGMAX];   //buffer to hold the message text
};
```

len – The size (in bytes) of the *msgtext* field of the object pointed to by the *msgPtr* argument.

flag – 0: The process can be blocked, if needed, until the function call completes successfully.

IPC_NOWAIT: The function aborts if the process is to be blocked.

Returns:

0 on success, -1 on failure.

The `msgrcv` API

```
#include <sys/ipc.h>       //for IPC_NOWAIT
#include <sys/msg.h>       //for MSG_NOERROR, msgrcv()
int msgrcv(int msgd, const void* msgPtr, int len, int msgType, int flag);
```

Receives a message of type *msgType* from a message queue designated by *msgd*. The message received is stored in the object pointed to by the *msgPtr* argument.

Parameters:

msgd – The message descriptor as obtained from a `msgget` system call.

msgPtr – The pointer to an object that has the `struct msgbuf`-like data structure as described in the `msgsnd` API.

len – The maximum number of message text bytes that can be received by this call.

msgType – The message type of the message to be received. Possible values:

0: Receive the oldest message of any type in the queue.

Positive Integer: Receive the oldest message of the specified message type.

flag – 0: The process may be blocked if no messages in the queue match the selection criteria specified by *msgType*.

IPC_NOWAIT: The call will be non-blocking.

MSG_NOERROR: A message in the queue will be selectable (even if larger than *len*).The call will return the first *len* bytes of message text to the calling process and discard the rest of the data.

Returns:

The number of bytes written to the *mtext* buffer of the object pointed to by the msgPtr argument on success, -1 on failure.

The `msgctl` API

```
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msgd, int cmd, struct msqid_ds* msgbufptr);
```

This API can be used to query the control data of a message queue designated by the *msgd* argument, to change the information within the control data of the queue, or to delete the queue from the system.

Example of IPC Using Message Passing

The following program creates a new message queue with the key ID of 100 and sets the access permission of the queue to be read-write for owner, read-only for group members and write-only for others. If the *msgget* call succeeds, the process sends the message “Hello” of type 15 to the queue and specifies the call to be nonblocking. After that, the process invokes the *msgrcv* API to wait and retrieve a message of type 20 from the queue. If the call succeeds, the process prints the retrieved message to the standard output. On any error, the process calls *perror* to print out a diagnostic message.

```
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/msg.h>

//data structure of one message
struct mbuf {
    long mtype;
    char mtext[MSGMAX];
} mobj = {15, "Hello"};

int main() {
    int perm = S_IRUSR | S_IWUSR | S_IRGRP | S_IWOTH;
    int fd = msgget(100, IPC_CREAT | IPC_EXCL | perm);
    if (fd == -1) perror("msg");

    if (msgsnd(fd, &mobj, strlen(mobj.mtext) + 1, IPC_NOWAIT))
        perror("msgsnd");

    if (msgrcv(fd, &mobj, MSGMAX, 20, IMSG_NOERROR) > 0)
        printf("%s\n", mobj.mtext);
    else
        perror("msgrcv");

    return 0;
}
```

10.8 Shared Memory

Shared memory allows multiple processes to map a portion of their virtual addresses to a common memory region. Thus, any process can write data to a shared memory region and the data are readily available to be read and modified by other processes.

How shared memory overcomes the performance problem of messages

In case of message passing system, when a message is sent from a process to a message queue, the data are copied from the process virtual address space to a kernel data region. Then when another process receives this message, the kernel copies the message data from the region to the receiving process' virtual address space. Thus, message data are copied twice: From process to kernel and then to another process.

Shared memory, on the other hand, does not have this data transfer overhead. Shared memory is allocated in the kernel virtual address when a process reads or writes data via a shared memory. The data is manipulated directly in the kernel memory region.

Problem with Shared Memory System

Shared memory does not provide any access control method for processes that use it. Therefore, it is a common practice to use semaphores along with shared memory, to implement an IPC media.

10.9 Kernel Data Structure for Shared Memory

In UNIX System V.3 and V.4, there is a shared memory table in the kernel address space that keeps track of all shared memory regions created in the system. Each entry of the table stores the following data for one shared memory region:

1. A key ID assigned by the process that created the shared memory. Other processes may specify this key to *open* the region and get a descriptor for future attachment to or detachment from the region.
2. The creator and assigned user ID and group ID.
3. Read-write access permission of the region.
4. The size, in number of bytes, of the shared memory region.
5. The time when the last process attached to the region.
6. The time when the last process detached from the region.
7. The time when the last process changed control data of the region.

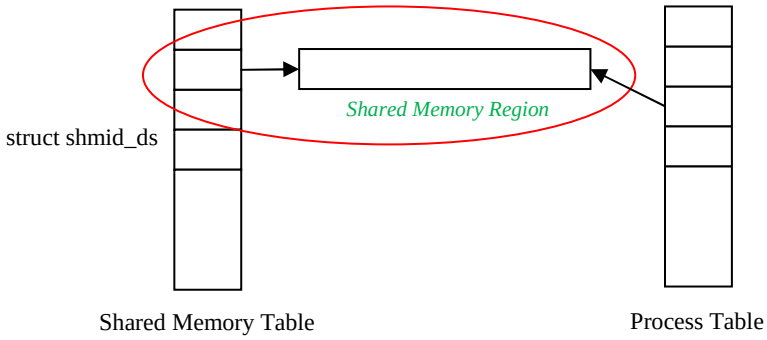


Figure 10.9: Kernel Data Structure for Shared Memory.

10.10 System-Imposed Limits on the Manipulation of Shared Memory

System Limit	Meaning
SHMMNI [*Max. No. of Instances*]	The maximum number of shared memory regions that may exist at any given time in a system.
SHMMAX	The maximum size, in number of bytes, of a shared memory region.
SHMMIN	The minimum size, in number of bytes, of a shared memory region.

Effects of The System-Imposed Limits on Processes

- If a process attempts to create a new shared memory, causing the SHMNI limit to be exceeded, the process will be blocked until an existing region is deleted by another process.
- If a process attempts to create a region whose size is less than SHMMIN or larger than SHMMAX, the system call will fail.

10.11 Shared Memory APIs

The `shmget` (SHared Memory GET) API

```
#include <sys/types.h>      //for key_t
#include <sys/ipc.h>        //for IPC_CREAT, IPC_PRIVATE
#include <sys/shm.h>        //for shmget()
int shmget(key_t key, int size, int flag);
```

Opens a shared memory whose key ID is given in the *key* argument.

Parameters:

key – *Positive Integer*: The API opens a shared memory whose key ID matches that value.

IPC_PRIVATE: The API allocates a new shared memory to be used exclusively by the calling process.

The *private* shared memory is usually allocated by a parent process, which then forks one or more child processes. The parent and child processes then use the shared memory to exchange data.

size – The size of the shared memory region to be attached to the calling process.

flag – 0: If there is no shared memory whose key ID matches the given *key* value, the API fails. Otherwise, it returns the descriptor for that memory.

IPC_CREAT | *read-write access permission*: A new shared memory (if none preexists) is created with the given key ID.

Returns:

Shared memory descriptor on success, -1 on failure.

The `shmat` (SHared Memory ATtach) API

```
#include <sys/shm.h>        //for shmat(), SHM_RDONLY
void *shmat(int shmid, void* addr, int flag);
```

Attaches a shared memory referenced by *shmid* to the calling process virtual address space. The process can then read/write data in that shared memory.

Parameters:

shmid – Descriptor of the shared memory to be attached.

addr – *An address*: The desired starting virtual address in the calling process to which location the shared memory should be mapped.

0: The kernel finds an appropriate virtual address in the calling process to map to the shared memory.

flag – If *addr* is 0, then it should be 0, too. However, the following value can also be applied:

SHM_RDONLY: The calling process attaches to the shared memory for read-only.

Returns:

The mapped virtual address of the shared memory on success, -1 on failure.

The `shmdt` (SHared Memory DeTach) API

```
#include <sys/shm.h>
int shmdt(void* addr);
```

Detaches (or unmaps) a shared memory from the specified *addr* virtual address of the calling process.

Parameters:

addr – The address of the shared memory as obtained from a `shmat` call prior to this API call.

Returns:

0 on success, -1 on failure.

The `shmctl` (SHared Memory ConTroL) API

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds* buf);
```

This API can either query or change the control data of a shared memory designated by *shmid*, or delete the memory altogether.

Example of IPC Using Shared Memory

The following program opens a shared memory with a size of 1024 bytes and the key ID value of 100. After the shared memory is opened, it is attached to the process virtual address. It then writes the message *Hello* to the beginning region of the memory and detaches from the memory. Any other process on the same system can now attach to the shared memory and read the message accordingly.

```
#include <stdio.h>
#include <stdlib.h>     //for EXIT_SUCCESS
#include <string.h>     //for strcpy
#include <sys/shm.h>    //for shmget, shmat, shmdt
#include <sys/stat.h>   //for S_IPWXU, S_IRWXG, S_IRWXO

int main(int argc, char** argv) {
    //set permissions
    int perms = S_IRWXU | S_IRWXG | S_IRWXO;

    //open shared memory
    int fd = shmget(100, 1024, IPC_CREAT | perms);
    if (fd == -1) {
        perror("shmget");
        exit(1);
    }

    //attach shared memory
    char* addr = (char*) shmat(fd, 0, 0);
    if (addr == (char*)-1) {
        perror("shmmat");
        exit(1);
    }
```

```
    //write to shared memory
    strcpy(addr, "Hello");

    //detach shared memory
    if (shmdt(addr) == -1) {
        perror("shmdt");
        exit(1);
    }

    return (EXIT_SUCCESS);
}
```

10.12 **UNIX System V Semaphores**

Semaphores provide a method to synchronize the execution of multiple processes. Semaphores are allocated in sets of one or more. A process can also use multiple semaphore sets. Semaphores are frequently used along with shared memory to establish an elaborate method for IPC.

Functions Provided by UNIX System V Semaphore APIs

1. Create a semaphore set.
2. *Open* a semaphore set and get a descriptor to reference the set.
3. Increase or decrease the integer values of one or more semaphores in a set.
4. Query the values of one or more semaphores in a set.
5. Query or set control data of a semaphore set.

10.13 **Kernel Data Structure for Semaphores**

In UNIX System V.3 and V.4, there is a semaphore table in the kernel address space that keeps track of all semaphore sets created in the system. Each entry of the semaphore table stores the following data for one semaphore set:

1. A key ID assigned by the process that created the queue. Other processes may specify this key to *open* the queue and get a descriptor for future access of the queue.
2. The creator and assigned user ID and group ID.
3. Read-write access permission of the set.
4. The number of semaphores in the set.
5. The time when the last process changed one or more semaphore values.
6. The time when the last process changed the control data of the set.
7. A pointer to an array of semaphores.

Semaphores in a set are referenced by array indices, such that the first semaphore in the set has an index of zero; the second semaphore has an index of 1; and so on. Furthermore, each semaphore stores the following data:

1. The semaphore’s value.
2. The process ID of the last process that operated on the semaphore.
3. The number of processes that are currently blocked pending the increase of semaphore value.
4. The number of processes which are currently blocked pending the semaphore’s value becoming zero.

If a semaphore set is deleted, any processes that are blocked at that time due to the semaphores are awakened by the kernel – the system calls they invoked are aborted and return a -1 failure status.

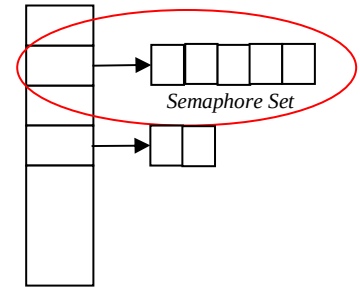


Figure 10.13: Kernel Data Structure for Semaphores.

10.14 System-Imposed Limits on the Manipulation of Semaphores

System Limit	Meaning
SEMMNI [*Max. No. of Instances*]	The maximum number of semaphore sets that may exist at any given time in a system.
SEMMNS [*Max. No. of Semaphores*]	The maximum number of semaphores in all sets that may exist in a system at any one time.
SEMMSL	The maximum number of semaphores allowed per set.
SEMOPL	The maximum number of semaphores in a set that may be operated on at any one time.

Effects of The System-Imposed Limits on Processes

- If a process attempts to create a new semaphore set that causes either the SEMMNI or the SEMMNS limit to be exceeded, the process will be blocked until one or more existing sets are deleted by a process.
- If a process attempts to create a semaphore set with more than SEMMSL semaphores, the system call fails.
- If a process attempts to operate on more than SEMOPM semaphores in a set in one operation, the system call fails.

10.15 The UNIX APIs for Semaphores

The `semget` API

```
#include <sys/types.h>      //for key_t
#include <sys/ipc.h>        //for IPC_CREAT, IPC_PRIVATE
#include <sys/shm.h>        //for semget()
int semget(key_t key, int num, int flag);
```

Opens a semaphore set whose key ID is given in the *key* argument.

Parameters:

key – *Positive Integer*: The API opens a semaphore set whose key ID matches that value.

IPC_PRIVATE: The API allocates a new semaphore set to be used exclusively by the calling process.

The *private* semaphores are usually allocated by a parent process, which then forks one or more child processes. The parent and child processes then use the semaphore to synchronize their operations.

num – The number of semaphores to be allocated when a new set is to be created. The value may be 0 if the IPC_CREAT flag is not specified in the *flag* argument.

flag – 0: If there is no semaphore set whose key ID matches the given *key* value, the API fails. Otherwise, it returns the descriptor for that set.

IPC_CREAT | *read-write access permission*: A new semaphore set (if none preexists) is created with the given key ID.

Returns:

Semaphore descriptor on success, -1 on failure.

The `semop` API

```
#include <sys/ipc.h>       //for IPC_NOWAIT
#include <sys/msg.h>       //for msgsnd()
int semop(int semd, struct sembuf* opPtr, int len);
```

This API may be used to change the value of one or more semaphores in a set (as designated by *semd*) and/or o test whether their values are 0.

Parameters:

semd – The semaphore descriptor as obtained from a `semget` system call.

opPtr – The pointer to any array of *struct sembuf* objects, each of which specifies one operation (query or change value) for a semaphore.

The struct sembuf data type is defined in the <sys/sem.h> header as:

```
struct sembuf {
    short sem_num;     //semaphore index
    short sem_op;      //semaphore operation
    short sem_flg;     //operation flag(s)
};
```

The possible values of *sem_op* and their meanings are:

A positive number – Increase the indexed semaphore value by this amount.
A negative number – Decrease the indexed semaphore value by this amount.
A zero – Test whether the semaphore value is 0.

len – Specifies how many entries are in the array pointed to by *opPtr*.

Returns:

0 on success, -1 on failure.

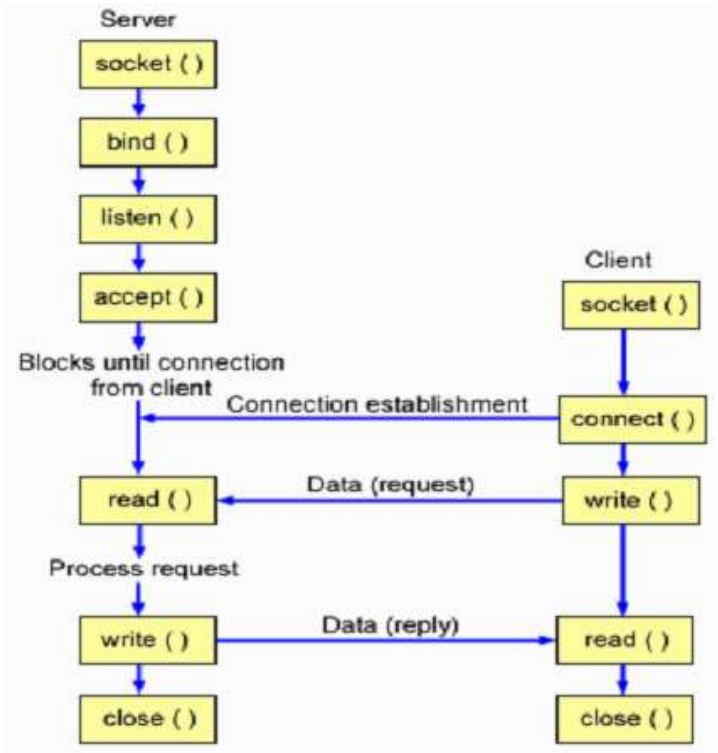
The `semctl` API

```
#include <sys/ipc.h>
#include <sys/msg.h>
int semctl(int semd, int num, int cmd, union semun arg);
```

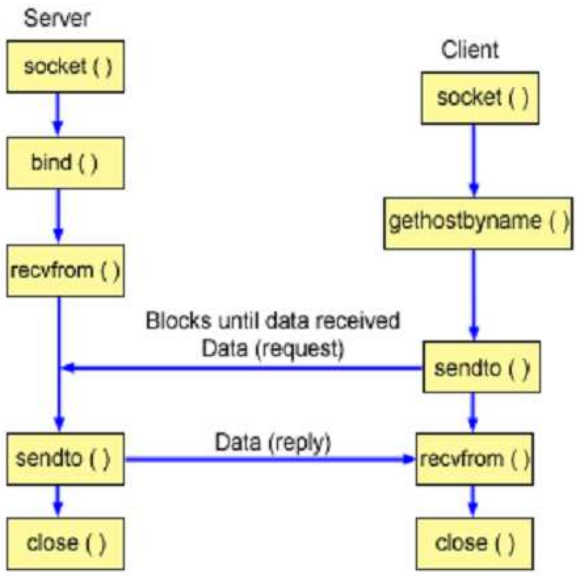
This API can be used to query or change the control data of a semaphore set designated by the *semd* argument or to delete the set altogether.

CHAPTER 11
SOCKET PROGRAMMING

Concepts



A TCP Client-Server Interaction



A UDP Client-Server Interaction

APIs

11.1

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Creates a socket of the given domain, type and protocol.

`domain` – Specifies the socket naming convention and the protocol address format. Possible values:

AF_UNIX	(UNIX Domain)
AF_INET	(DARPA Internet Domain)

`type` – Specifies a socket type. Possible values:

SOCK_STREAM	(for TCP and IPC)
SOCK_DGRAM	(for UDP)
SOCK_SEQPACKET	(for TCP and IPC with a fixed maximum message length)
SOCK_RAW	(Raw socket)

`protocol` – Specifies a particular protocol to be used with the socket. Actual value depends on the *domain* argument. Usually, this is set to 0, and the kernel will choose an appropriate protocol for the specified domain.

Returns: A socket descriptor on success. -1 on failure.

11.2

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sd, struct sockaddr *addr_p, int lenght);
```

Binds a name to a socket.

sd – The socket descriptor as returned by a `socket` function call.
addr_p – The pointer to the structure containing the name to be assigned to the socket. This argument depends on the domain of the socket.

- For internet domain, the name to be bound consists of a machine host name and port number. The structure of the object pointed to by *addr_p* is (as defined in <netinet/in.h> header file):

```
struct sockaddr_in
{
    short           sin_family;
    u_short         sin_port;
    struct in_addr  sin_addr;
}
```

The values of the fields should be as follows:

sin_family – AF_INET
sin_port – The port number to be bound.
sin_addr – The host machine name/ip where the socket resides.

The detailed structure for `struct in_addr` is as follows:

```
struct in_addr
{
    unsigned long  s_addr;
}
```

- For UNIX domain, …

length – Specifies the size of the name structure pointed to by the *addr_p* argument.

Returns: 0 on success. -1 on failure.

11.3

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sd, int size);
```

This is called in a server process to establish a connection-based socket (of type SOCK_STREAM or SOCK_SEQPACKET) for communication.

sd – The socket descriptor as returned by a `socket` function call.
size – Specifies the maximum (*backlog*) number of connection requests that may be queued for the socket. In most UNIX systems, the maximum allowed value for the *size* argument is 5.

Returns: 0 on success. -1 on failure.

11.4

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sd, struct sockaddr *addr_p, int lenght);
```

This is called in a client process in requesting a connection to a server socket.

sd – The socket descriptor as returned by a `socket` function call.
addr_p – The pointer to the object *sockaddr* that holds the name of the server socket to be connected.
length – Specifies the size of the name structure pointed to by the *addr_p* argument.

Returns: 0 on success. -1 on failure.

11.5

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sd, struct sockaddr *addr_p, int* lenght);
```

This is called in a server process to establish a connection-based socket connection with a client socket.

sd – The socket descriptor as returned by a `socket` function call.
addr_p – The pointer to the object *sockaddr* that holds the name of the client socket where the server socket is connected.
length – Initially set to the maximum size of the object pointed to by the *addr_p* argument. On return, it contains the size of the client socket name, as pointed to by the *addr_p* argument.

Returns: A new socket descriptor (which the server process can use to communicate with the client exclusively) on success. -1 on failure.

11.6

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int sd, const char *buf, int len, int flag);
int sendto(int sd, const char *buf, int len, int flag, struct sockaddr *addr_p, int addr_p_len);
```

The *send* function sends a message, contained in *buf*, of size *len* bytes, to a socket that is connected to this socket, as designated by *sd*.

The *sendto* function is the same as the send API, except that the calling process also specifies the address of the recipient socket name via the *addr_p* and *len_p* arguments.

The *send* function is primarily used for TCP and the *sendto* function for UDP.

sd – The socket descriptor as returned by a `socket` function call.
buf – The address of the buffer (of type `char`) where the message to be sent is contained.
len – The length of the message to be sent.
flag – Specifies whether the message is a regular message or an out-of-bound message. Possible values:
0 (Regular message)
MSG_OOB (Out-of-bound message)
addr_p – The pointer to the object that contains the name of a recipient socket.
addr_p_len – The number of bytes in the objet pointed to by *addr_p*.

Returns: Number of data bytes actually sent. -1 on failure.

11.7

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int sd, const char *buf, int len, int flag);
int recvfrom(int sd, const char *buf, int len, int flag, struct sockaddr *addr_p, int* addr_p_len);
```

The *recv* function receives a message via a socket designated by *sid*. The message received is copied to *buf*, and the maximum size of *buf* is specified in the *len* argument.

The *recvfrom* function is the same as the recv API, except that the calling process also specifies the *addr_p* and *len_p* arguments to receive the sender name.

The *send* function is primarily used for TCP and the *sendto* function for UDP.

sd – The socket descriptor as returned by a `socket` function call.
buf – The address of the buffer (of type `char`) where the receiving message is to be copied.
len – The *maximum* length of the buffer.
flag – Specifies whether a regular message or an out-of-bound message is to be received. Possible values:
0 (Regular message)
MSG_OOB (Out-of-bound message)
addr_p – The pointer to the object that will contain the name of the sender socket.
addr_p_len – The number of bytes in the objet pointed to by *addr_p*.

Returns: Number of data bytes actually received. -1 on failure.

11.8

```
int htons(short var);
int htonl(long var);
int ntohs(short var);
```

```
int ntohl(long var);
```

The *htons* (*host to network short*) function converts *short* values from host byte order (little-endian) to network byte order (big-endian).

The *htonl* (*host to network long*) function converts *long* values from host byte order (little-endian) to network byte order (big-endian).

The *ntohs* and *ntohl* functions do the opposites of *htons* and *htonl* functions.

11.9

```
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *addr);
```

Converts the specified string, in the Internet standard dot notation, to an integer value suitable for use as an Internet address. The converted address is in network byte order.

cp	–	The address of the string to be converted.
addr	–	Pointer to the structure where the converted address is to be stored.
Returns:		A non-zero value on success. 0 on failure (e.g., if the specified address is not valid).

```
#include <arpa/inet.h>
char* inet_ntoa(struct in_addr in);
```

Converts the specified Internet host address to a string in the Internet standard dot notation.

in	–	The internet host address to be converted.
Returns:		A pointer to the network address in Internet standard dot notation.

Note: The *inet_aton* and *inet_ntoa* functions are for IPv4 address format. For IPv6, use *inet_pton* and *inet_ntop*.

Examples

TCP Server Example

```
#include <sys/socket.h> //for socket(), send() etc.
#include <netinet/in.h> //for sockaddr_in
#include <stdio.h>      //for close()

char send_bufer[1024] = "Message from server to client";
char rcv_bufer[1024];

main() {
    int sd, new_sd;
    struct sockaddr_in addr_p, client_addr_p;

    sd = socket(AF_INET, SOCK_STREAM, 0);

    addr_p.sin_family = AF_INET;
    addr_p.sin_port = htons(80);
    addr_p.sin_addr.s_addr = htonl(INADDR_ANY);

    bind(sd, (struct sockaddr*) &addr_p, sizeof(addr_p));
    listen(sd, 5);

    while (1) {
        int len = sizeof(client_addr_p);
        new_sd = accept(sd, (struct sockaddr*)&client_addr_p, &len);
        int rcv_bytes = recv(new_sd, rcv_bufer, sizeof(rcv_bufer), 0);
        int send_bytes = send(new_sd, send_bufer, sizeof(send_bufer), 0);
        close(new_sd);
    }
}
```

TCP Client Example

```
#include <sys/socket.h> //for socket(), send() etc.
#include <netinet/in.h> //for sockaddr_in
#include <arpa/inet.h>  //for inet_aton()
#include <stdio.h>      //for close()

char send_bufer[1024] = "Message from client to server";
char rcv_bufer[1024];

main() {
    int sd;
    struct sockaddr_in addr_p;
    sd = socket(AF_INET, SOCK_STREAM, 0);
    addr_p.sin_family = AF_INET;
    addr_p.sin_port = htons(80);
    addr_p.sin_addr.s_addr = inet_aton("50.16.8.1", &addr_p.sin_addr);
    connect(sd, (struct sockaddr*) &addr_p, sizeof(addr_p));
    int send_bytes = send(sd, rcv_bufer, sizeof(rcv_bufer), 0);
    int rcv_bytes = recv(sd, send_bufer, sizeof(send_bufer), 0);
    close(sd);
}
```

UDP Server Example

```
#include <sys/socket.h> //for socket(), send() etc.
#include <netinet/in.h> //for sockaddr_in
#include <stdio.h>      //for close()

char rcv_bufer[1024];

main() {
    int sd;
    struct sockaddr_in addr_p, client_addr_p;
    sd = socket(AF_INET, SOCK_DGRAM, 0);
    addr_p.sin_family = AF_INET;
    addr_p.sin_port = htons(80);
    addr_p.sin_addr.s_addr = htonl(INADDR_ANY);
    bind(sd, (struct sockaddr*) &addr_p, sizeof(addr_p));
    int len = sizeof(client_addr_p);
    int send_bytes = recvfrom(sd, rcv_bufer, sizeof(rcv_bufer), 0,
                      (struct sockaddr*)&client_addr_p, &len);
    close(sd);
}
```

UDP Client Example

```
#include <sys/socket.h> //for socket(), send() etc.
#include <netinet/in.h> //for sockaddr_in
#include <arpa/inet.h>  //for inet_aton()
#include <stdio.h>      //for close()

char send_bufer[1024] = "Message from client to server";

main() {
    int sd;
    struct sockaddr_in addr_p;
    sd = socket(AF_INET, SOCK_DGRAM, 0);
    addr_p.sin_family = AF_INET;
    addr_p.sin_port = htons(80);
    addr_p.sin_addr.s_addr = inet_aton("50.16.8.1", &addr_p.sin_addr);
    int recvd_bytes = sendto(sd, send_bufer, sizeof(send_bufer), 0,
                             (struct sockaddr*) &addr_p, sizeof(addr_p));
    close(sd);
}
```