

System programming

Lecture 1

Part 1: Fundamental Concepts

System programming

- System programming is the practice of writing system software.
- System software lives at a low level, interfacing directly with the kernel and core system libraries.
- System programmer must have a good awareness of the hardware and the operating system on which they work.

Examples

- shell;
- text editor;
- compiler;
- debugger;
- core utilities;
- system daemons;
- web server;
- database;
- etc.

Shell

- A *shell* is a special-purpose program designed to read commands typed by a user and execute appropriate programs in response to those commands.
- Such a program is sometimes known as a *command interpreter*.
- On UNIX systems, the shell is a user process.

System programming

- kernel development
- user-space system-level programming

Kernel

- The central software that manages and allocates computer resources (i.e., the CPU, RAM, and devices).
- The Linux kernel executable typically resides at the pathname /boot/vmlinuz, or something similar.

Tasks performed by the kernel

- Process scheduling
- Memory management (virtual memory)
 - Processes are isolated from one another and from the kernel, so that one process can't read or modify the memory of another process or the kernel.
 - Only part of a process needs to be kept in memory, thereby lowering the memory requirements of each process and allowing more processes to be held in RAM simultaneously.

Tasks performed by the kernel

- Provision of a file system
- Creation and termination of processes
- Access to devices
- Networking
- Provision of a system call application programming interface (API)

Kernel mode and user mode

- When running in **user mode**, the CPU can access only memory in user space; attempts to access memory in kernel space result in a hardware exception.
- When running in **kernel mode**, the CPU can access both user and kernel memory space.
- User processes are not able to access the instructions and data structures of the kernel, or to perform operations that would adversely affect the operation of the system.

Examples of operations in kernel mode

- halt instruction to stop the system,
- accessing the memory-management hardware,
- initiating device I/O operations

Process view and kernel view

- For a process, many things happen asynchronously. A process operates in isolation; it can't directly communicate with another process.
- Kernel knows and controls everything.
- The kernel facilitates the running of all processes on the system.
- The kernel decides which process will next obtain access to the CPU, when it will do so, and for how long.

Linux

- *Linux is a modern Unix like* system, written from scratch by Linus Torvalds.
- Although Linux shares the goals and philosophy of Unix, Linux is not Unix.
- First 7-8 weeks we are going to discuss Linux System Programming.

Cornerstones of System Programming

- System calls
- C library
- C compiler

System call

- System call (*syscall*) is a function invocation made *from user space into the kernel* in order to request some service or resource from the operating system.

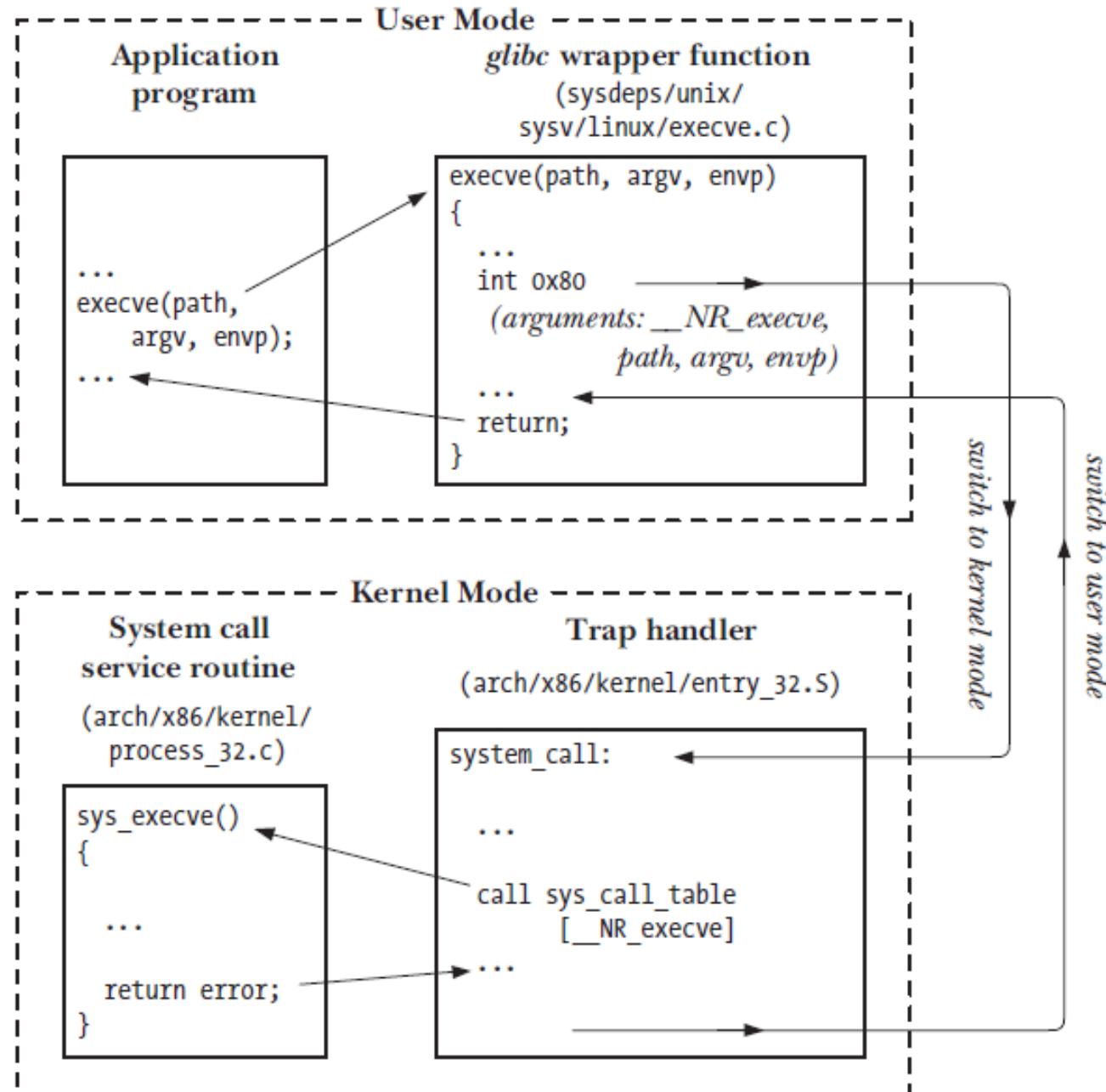
Examples of system calls (Linux)

- `read()`
- `write()`
- `get_thread_area()`
- `set_tid_address()`
- etc.

General points

- A system call changes the processor state from user mode to kernel mode, so that the CPU can access protected kernel memory.
- The set of system calls is fixed. Each system call is identified by a unique number.
- Each system call may have a set of arguments that specify information to be transferred from user space (i.e., the process's virtual address space) to kernel space and vice versa.

Steps in the execution of a system call execve()



Steps in the execution of a system call (x86-32)

1. The application program makes a system call by invoking a wrapper function in the C library.
2. The wrapper function must make all of the system call arguments available to the system call trap-handling routine. These arguments are passed to the wrapper via the stack, but the kernel expects them in specific registers. The wrapper function copies the arguments to these registers.

Steps in the execution of a system call (x86-32)

3. Since all system calls enter the kernel in the same way, the kernel needs some method of identifying the system call. To permit this, the wrapper function copies the **system call number** into a specific CPU register (%eax).
4. The wrapper function executes a trap machine instruction (int 0x80), which causes the processor to switch from user mode to kernel mode and execute code pointed to by location 0x80 of the system's trap vector.

Steps in the execution of a system call (x86-32)

5. In response to the trap to location 0x80, the kernel invokes its `system_call()` routine to handle the trap.
6. If the return value of the system call service routine indicated an error, the wrapper function sets the global variable `errno` using this value. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

Library Functions

- Many library functions don't make any use of system calls (e.g., the string manipulation functions).
- On the other hand, some library functions are layered on top of system calls. For example, the `fopen()` library function uses the `open()` system call to actually open a file.

Examples

- Library functions are designed to provide a more caller-friendly interface than the underlying system call.
- For example, the `printf()` function provides output formatting and data buffering, whereas the `write()` system.

The Standard C Library; The GNU C Library (glibc)

- The most commonly used implementation on Linux is the GNU C library.

The C compiler

- In Linux, the standard C compiler is provided by the *GNU Compiler Collection (gcc)*.

API

- Abstraction that provides a standard **set of interfaces**—usually functions—that one piece of software can invoke from another piece of software;
- Example of an API is the interfaces defined by the C standard and implemented by the standard C library. This API defines a family of basic and essential functions, such as memory management and string-manipulation routines.

ABI

- Whereas an API defines a source interface and ensures source compatibility, an ABI defines the binary interface and binary compatibility between two or more pieces of software on a particular architecture.
- ABIs are concerned with issues such as calling conventions, byte ordering, register use, system call invocation, linking, library behavior, and the binary object format.
- The calling convention, for example, defines how functions are invoked, how arguments are passed to functions, which registers are preserved, and how the caller retrieves the return value.
- The ABI is tied to the architecture; the vast majority of an ABI speaks of machine-specific concepts, such as particular registers or assembly instructions.

Standards

Linux *aims toward* compliance with two of the most important and prevalent standards:

- **POSIX**
- and the Single UNIX Specification (**SUS**).

Programs

- Two forms:
 - **Source code** (human-readable text consisting of a series of statements written in a programming language)
 - Binary **machine-language instructions** that the computer can understand.

Processes

- Process is an **instance** of an executing program.
- From a kernel point of view, processes are the entities among which the kernel must share the various resources of the computer.

Process

- A process is logically divided into the following parts, known as segments:
 - **Text**: the instructions of the program.
 - **Data**: the static variables used by the program.
 - **Heap**: an area from which programs can dynamically allocate extra memory.
 - **Stack**: a piece of memory that grows and shrinks as functions are called and return and that is used to allocate storage for local variables and function call linkage information.

Daemon processes

- A daemon is a special-purpose process that is created and handled by the system in the same way as other processes, but which is distinguished by the following characteristics:
 - It is long-lived. A daemon process is often started at system boot and remains in existence until the system is shut down.
 - It runs in the background, and has no controlling terminal from which it can read input or to which it can write output.
- Examples of daemon processes include
 - syslogd, which records messages in the system log,
 - and httpd, which serves web pages via the Hypertext Transfer Protocol (HTTP).

Threads

- In modern UNIX implementations, each process can have multiple threads of execution.
- Each thread is executing the same program code and shares the same data area and heap.
- However, each thread has its own stack containing local variables and function call linkage information.

Threads

- Threads can communicate with each other via the global variables that they share.
- The threading API provides condition variables and mutexes, which are primitives that enable the threads of a process to communicate and synchronize their actions.
- Threads can also communicate with one another using the IPC and synchronization mechanisms.

Handling Errors from System Calls and Library Functions

- Almost every system call and library function returns some type of status value indicating whether the call succeeded or failed.
- This status value should always be checked to see whether the call succeeded.

Examples

- A few system calls never fail. For example, `getpid()` always successfully returns the ID of a process, and `_exit()` always terminates a process.
- It is not necessary to check the return values from such system calls.

Handling system call errors

- The manual page for each system call documents the possible return values of the call, showing which value(s) indicate an error.
- Usually, an error is indicated by a return of `-1`.

Checking a system call

```
fd = open(pathname, flags, mode);      /* system call to open a file */
if (fd == -1) {
    /* Code to handle the error */
}
...
if (close(fd) == -1) {
    /* Code to handle the error */
}
```

errno

- When a system call fails, it sets the global integer variable *errno* to a positive value that identifies the specific error.
- Including the <errno.h> header file provides a declaration of *errno*, as well as a set of constants for the various error numbers.

Example

```
cnt = read(fd, buf, numbytes);
if (cnt == -1) {
    if (errno == EINTR)
        fprintf(stderr, "read was interrupted by a signal\n");
    else {
        /* Some other error occurred */
    }
}
```

- Successful system calls and library functions never reset *errno* to 0, so this variable may have a nonzero value as a consequence of an error from a previous call.
- Therefore, when checking for an error, we should always first check if the function return value indicates an error, and only then examine *errno* to determine the cause of the error.

Some cases

- A few system calls (e.g., *getpriority()*) can legitimately return -1 on success.
- To determine whether an error occurs in such calls, we set `errno` to 0 before the call, and then check it afterward.
- If the call returns -1 and `errno` is nonzero, an error occurred.

Printing an error message

- A common course of action after a failed system call is to print an error message based on the *errno* value.
- The *perror()* and *strerror()* library functions are provided for this purpose.

Example of handling errors from system calls

```
fd = open(pathname, flags, mode);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

perror()

- The *perror()* function prints the string pointed to by its *msg* argument, followed by a message corresponding to the current value of *errno*.

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

```
void perror(const char *msg);
```

strerror()

- The *strerror()* function returns the error string corresponding to the error number given in its *errnum* argument.

```
#include <string.h>
```

```
char *strerror(int errnum);
```

Handling errors from **library functions**

- Some library functions return error information in exactly the same way as system calls: a `-1` return value, with `errno` indicating the specific error.
- An example of such a function is `remove()`, which removes a file (using the `unlink()` system call) or a directory (using the `rmdir()` system call).
- Errors from these functions can be diagnosed in the same way as errors from system calls.

- Some library functions return a value other than `-1` on error, but nevertheless set `errno` to indicate the specific error condition.
- For example, `fopen()` returns a NULL pointer on error, and the setting of `errno` depends on which underlying system call failed.
- The `perror()` and `strerror()` functions can be used to diagnose these errors.

- Other library functions don't use *errno* at all.
- The method for determining the existence and cause of errors depends on the particular function and is documented in the function's manual page.
- For these functions, it is a mistake to use *errno*, *perror()*, or *strerror()* to diagnose errors.

System programming

Lecture 1

Part 2: File I/O

Everything-is-a-file philosophy

- In Linux, everything is a file.

File descriptor

- Nonnegative integer used to refer to all types of open files, including pipes, FIFOs, sockets, terminals, devices, and regular files.

Standard file descriptors

File descriptor	Purpose	POSIX name	<i>stdio</i> stream
0	standard input	STDIN_FILENO	<i>stdin</i>
1	standard output	STDOUT_FILENO	<i>stdout</i>
2	standard error	STDERR_FILENO	<i>stderr</i>

- When referring to these file descriptors in a program, we can use either the numbers (0, 1, or 2) or, preferably, the POSIX standard names defined in <unistd.h>.

Key system calls for performing file I/O

- fd = open(pathname, flags, mode)
- numread = read(fd, buffer, count)
- numwritten = write(fd, buffer, count)
- status = close(fd)

Using I/O system calls (copy.c)

```
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#ifndef BUF_SIZE
#define BUF_SIZE 1024
#endif
```

Using I/O system calls (copy.c)

```
int inputFd, outputFd, openFlags;
mode_t filePerms;
ssize_t numRead;
char buf[BUF_SIZE];

/* Open input and output files */

inputFd = open(argv[1], O_RDONLY);
if (inputFd == -1)
    errExit("opening file %s", argv[1]);

openFlags = O_CREAT | O_WRONLY | O_TRUNC;
filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
            S_IROTH | S_IWOTH;      /* RW-RW-RW- */
outputFd = open(argv[2], openFlags, filePerms);
if (outputFd == -1)
    errExit("opening file %s", argv[2]);
```

Using I/O system calls (copy.c)

```
/* Transfer data until we encounter end of input or an error */

while ((numRead = read(inputFd, buf, BUF_SIZE)) > 0)
    if (write(outputFd, buf, numRead) != numRead)
        fatal("couldn't write whole buffer");
if (numRead == -1)
    errExit("read");

if (close(inputFd) == -1)
    errExit("close input");
if (close(outputFd) == -1)
    errExit("close output");
```

Universality of I/O

```
$ ./copy test test.old  
$ ./copy a.txt /dev/tty  
$ ./copy /dev/tty b.txt  
$ ./copy /dev/pts/16 /dev/tty
```

Copy a regular file

Copy a regular file to this terminal

Copy input from this terminal to a regular file

Copy input from another terminal

Opening a File: open()

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

int open(const char * pathname, int flags, ... /* mode_t mode */);
```

Returns file descriptor on success, or -1 on error

The open() system call maps the file given by the pathname to a file descriptor, which it returns on success.

The file position is to the start of the file (zero) and the file is opened for access according to the flags given by flags.

File access modes

Access mode	Description
<code>O_RDONLY</code>	Open the file for reading only
<code>O_WRONLY</code>	Open the file for writing only
<code>O_RDWR</code>	Open the file for both reading and writing

Open existing file for reading

```
fd = open("startup", O_RDONLY);
if (fd == -1)
    errExit("open");
```

Open new or existing file for reading and writing, truncating to zero bytes; file permissions read + write for owner, nothing for all others

```
fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");
```

Open new or existing file for writing;
writes should always append to end of
file

```
fd = open("w.log", O_WRONLY | O_CREAT | O_TRUNC | O_APPEND,  
          S_IRUSR | S_IWUSR);  
if (fd == -1)  
    errExit("open");
```

Permissions of new files

- The *mode* argument is ignored unless a file is created; it is required if `O_CREAT` is given.
- If you forget to provide the mode argument when using `O_CREAT`, the results are undefined.
- When a file is created, the mode argument provides the permissions of the newly created file. The mode is not checked on this particular open of the file, so you can perform operations that contradict the assigned permissions, such as opening the file for writing but assigning the file read-only permissions.

S_IRWXU

Owner has read, write, and execute permission.

S_IRUSR

Owner has read permission.

S_IWUSR

Owner has write permission.

S_IXUSR

Owner has execute permission.

S_IRWXG

Group has read, write, and execute permission.

S_IRGRP

Group has read permission.

S_IWGRP

Group has write permission.

S_IXGRP

Group has execute permission.

S_IRWXO

Everyone else has read, write, and execute permission.

S_IROTH

Everyone else has read permission.

S_IWOTH

Everyone else has write permission.

S_IXOTH

Everyone else has execute permission.

open() flags

Flag	Purpose	SUS?
O_RDONLY	Open for reading only	v3
O_WRONLY	Open for writing only	v3
O_RDWR	Open for reading and writing	v3
O_CLOEXEC	Set the close-on-exec flag (since Linux 2.6.23)	v4
O_CREAT	Create file if it doesn't already exist	v3
O_DIRECT	File I/O bypasses buffer cache	
O_DIRECTORY	Fail if <i>pathname</i> is not a directory	v4
O_EXCL	With O_CREAT: create file exclusively	v3
O_LARGEFILE	Used on 32-bit systems to open large files	
O_NOATIME	Don't update file last access time on <i>read()</i> (since Linux 2.6.8)	
O_NOCTTY	Don't let <i>pathname</i> become the controlling terminal	v3
O_NOFOLLOW	Don't dereference symbolic links	v4
O_TRUNC	Truncate existing file to zero length	v3
O_APPEND	Writes are always appended to end of file	v3
O_ASYNC	Generate a signal when I/O is possible	
O_DSYNC	Provide synchronized I/O data integrity (since Linux 2.6.33)	v3
O_NONBLOCK	Open in nonblocking mode	v3
O_SYNC	Make file writes synchronous	v3

Example

```
int fd;

fd = open (file, O_WRONLY | O_CREAT | O_TRUNC,
           S_IWUSR | S_IRUSR | S_IWGRP | S_IRGRP | S_IROTH);
if (fd == -1)
    /* error */
```

The **creat()** System Call

```
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

Returns file descriptor, or -1 on error

Calling **creat()** is equivalent to the following
open() call:

```
fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Example

The following typical `creat()` call,

```
int fd;  
  
fd = creat (filename, 0644);  
if (fd == -1)  
    /* error */
```

is identical to

```
int fd;  
  
fd = open (filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);  
if (fd == -1)  
    /* error */
```

Reading from a File: read()

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
```

Returns number of bytes read, 0 on EOF, or -1 on error

Each call reads up to *count* bytes into the memory pointed at by *buffer* from the current file offset of the file referenced by *fd*. On success, the number of bytes written into *buffer* is returned. On error, the call returns -1 and sets *errno*.

The file position is advanced by the number of bytes read from *fd*. If the object represented by *fd* is not capable of seeking (for example, a character device file), the read always occurs from the “current” position.

Using `read()`

```
#define MAX_READ 20
char buffer[MAX_READ];

if (read(STDIN_FILENO, buffer, MAX_READ) == -1)
    errExit("read");
printf("The input data was: %s\n", buffer);
```

read() can result in many possibilities

- The call returns a value equal to len. All len read bytes are stored in buf. The results are as intended.
- The call returns a value less than len, but greater than zero. The read bytes are stored in buf. This can occur because a signal interrupted the read midway; an error occurred in the middle of the read; more than zero, but less than len bytes' worth of data was available; or EOF was reached before len bytes were read. Reissuing the read will read the remaining bytes into the rest of the buffer or indicate the cause of the problem.

- The call returns 0. This indicates EOF. There is nothing to read.
- The call blocks because no data is currently available. This won't happen in nonblocking mode.
- The call returns -1, and errno is set to EINTR. This indicates that a signal was received before any bytes were read. The call can be reissued.
- The call returns -1, and errno is set to EAGAIN. This indicates that the read would block because no data is currently available, and that the request should be reissued later. This happens only in nonblocking mode.
- The call returns -1, and errno is set to a value other than EINTR or EAGAIN. This indicates a more serious error. Simply reissuing the read is unlikely to succeed.

Reading All the Bytes

```
ssize_t ret;

while (len != 0 && (ret = read (fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR)
            continue;
        perror ("read");
        break;
    }

    len -= ret;
    buf += ret;
}
```

Nonblocking Reads

- *Nonblocking I/O* allows applications to perform I/O, potentially on multiple files, without ever blocking, and thus missing data available in another file.
- Consequently, an additional `errno` value is worth checking: `EAGAIN`. If the given file descriptor was opened in nonblocking mode (if `O_NONBLOCK` was given to `open()`) and there is no data to read, the `read()` call will return `-1` and set `errno` to `EAGAIN` instead of blocking.

Nonblocking Reads

```
char buf[BUFSIZ];
ssize_t nr;

start:
nr = read (fd, buf, BUFSIZ);
if (nr == -1) {
    if (errno == EINTR)
        goto start; /* oh shush */
    if (errno == EAGAIN)
        /* resubmit later */
    else
        /* error */
}
```

Writing to a File: write()

```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t count);
```

Returns number of bytes written, or -1 on error

On success, the number of bytes written is returned, and the file position is updated in kind. On error, -1 is returned and `errno` is set appropriately. A call to `write()` can return 0, but this return value does not have any special meaning; it simply implies that zero bytes were written.

Example

```
const char *buf = "My ship is solid!";
ssize_t nr;

/* write the string in 'buf' to 'fd' */
nr = write (fd, buf, strlen (buf));
if (nr == -1)
    /* error */

        unsigned long word = 1720;
        size_t count;
        ssize_t nr;

        count = sizeof (word);
        nr = write (fd, &word, count);
        if (nr == -1)
            /* error, check errno */
        else if (nr != count)
            /* possible error, but 'errno' not set */
```

Remark

When performing I/O on a disk file, a successful return from `write()` doesn't guarantee that the data has been transferred to disk, because the kernel performs **buffering** of disk I/O in order to reduce disk activity and expedite `write()` calls.

Closing a File: close()

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

Returns 0 on success, or -1 on error

```
if (close(fd) == -1)  
    errExit("close");
```

Remark

- File descriptors are a consumable resource, so failure to close a file descriptor could result in a process running out of descriptors.
- This is a particularly important issue when writing long-lived programs that deal with multiple files, such as shells or network servers.

Changing the File Offset: lseek()

- For each open file, the kernel records a file **offset**, sometimes also called the read-write offset or **pointer**.
- This is the location in the file at which the next read() or write() will commence.
- The file offset is expressed as an ordinal byte position relative to the start of the file.
- The first byte of the file is at offset 0.

Changing the File Offset: lseek()

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Returns new file offset if successful, or -1 on error

whence argument

The whence argument indicates the base point from which offset is to be interpreted, and is one of the following values:

- **SEEK_SET**

The file offset is set offset bytes from the beginning of the file.

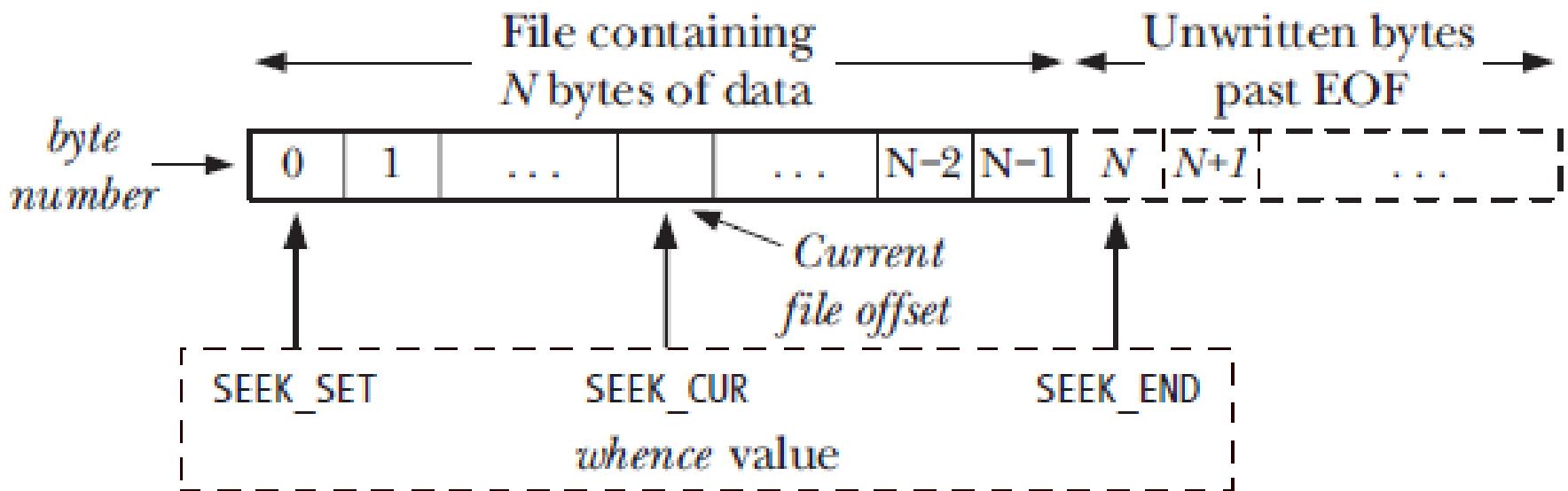
- **SEEK_CUR**

The file offset is adjusted by offset bytes relative to the current file offset.

- **SEEK_END**

The file offset is set to the size of the file plus offset. In other words, offset is interpreted with respect to the **next byte after the last byte** of the file.

Interpreting the **whence** argument of **Iseek()**



Remark

- If whence is SEEK_CUR or SEEK_END, offset may be negative or positive; for SEEK_SET, offset must be nonnegative.

Retrieving the current location of the
file offset without changing it

```
curr = lseek(fd, 0, SEEK_CUR);
```

Examples

```
lseek(fd, 0, SEEK_SET);          /* Start of file */  
lseek(fd, 0, SEEK_END);          /* Next byte after the end of the file */  
lseek(fd, -1, SEEK_END);         /* Last byte of file */  
lseek(fd, -10, SEEK_CUR);        /* Ten bytes prior to current location */  
lseek(fd, 10000, SEEK_END);       /* 10001 bytes past last byte of file */
```

Remark

- We can't apply lseek() to all types of files. Applying lseek() to a pipe, FIFO, socket, or terminal **is not permitted**; lseek() fails, with errno set to ESPIPE.
- It is **possible** to apply lseek() to devices where it is sensible to do so. For example, it is possible to seek to a specified location on a disk or tape device.

File holes

- What happens if a program seeks past the end of a file, and then performs I/O?
- A call to **read()** will return 0, indicating end-of-file.
- It is possible to **write** bytes at an arbitrary point past the end of the file.
- The space in between the previous end of the file and the newly written bytes is referred to as a **file hole**.

Remark

- File holes don't take up any disk space.
- The file system doesn't allocate any disk blocks for a hole until, at some later point, data is written into it.

- The existence of holes means that a file's nominal size may be larger than the amount of disk storage it utilizes.
- Writing bytes into the middle of the file hole will decrease the amount of free disk space as the kernel allocates blocks to fill the hole, even though the file's size doesn't change.

System programming

Lecture 2 Buffered I/O

Blocks

- *Block* is an abstraction representing the smallest unit of storage on a filesystem.
- No I/O operation may execute on an amount of data less than the block size or that is not an integer multiple of the block size.
- If you only want to read a byte, too bad: you'll have to read a whole block.

Blocks

- Partial block operations are inefficient.
- Everything occurs on block-aligned boundaries and rounding up to the next largest block.

Buffered I/O

- **Problem:** reading or writing a single byte 1,024 times rather than a single 1,024-byte block all at once.
- **Solution:** read and write data in whatever amounts but have the actual I/O occur in units of the filesystem block size.

Compare:

```
dd bs=1 count=2097152 if=/dev/zero of=pirate
```

```
dd bs=1024 count=2048 if=/dev/zero of=pirate
```

Effects of block size on performance

Block size	Real time	User time	System time
1 byte	18.707 seconds	1.118 seconds	17.549 seconds
1,024 bytes	0.025 seconds	0.002 seconds	0.023 seconds
1,130 bytes	0.035 seconds	0.002 seconds	0.027 seconds

Real time is the total elapsed wall clock time

User time is the time spent executing the program's code in user space

System time is the time spent executing system calls in kernel space on the process's behalf.

Block Size

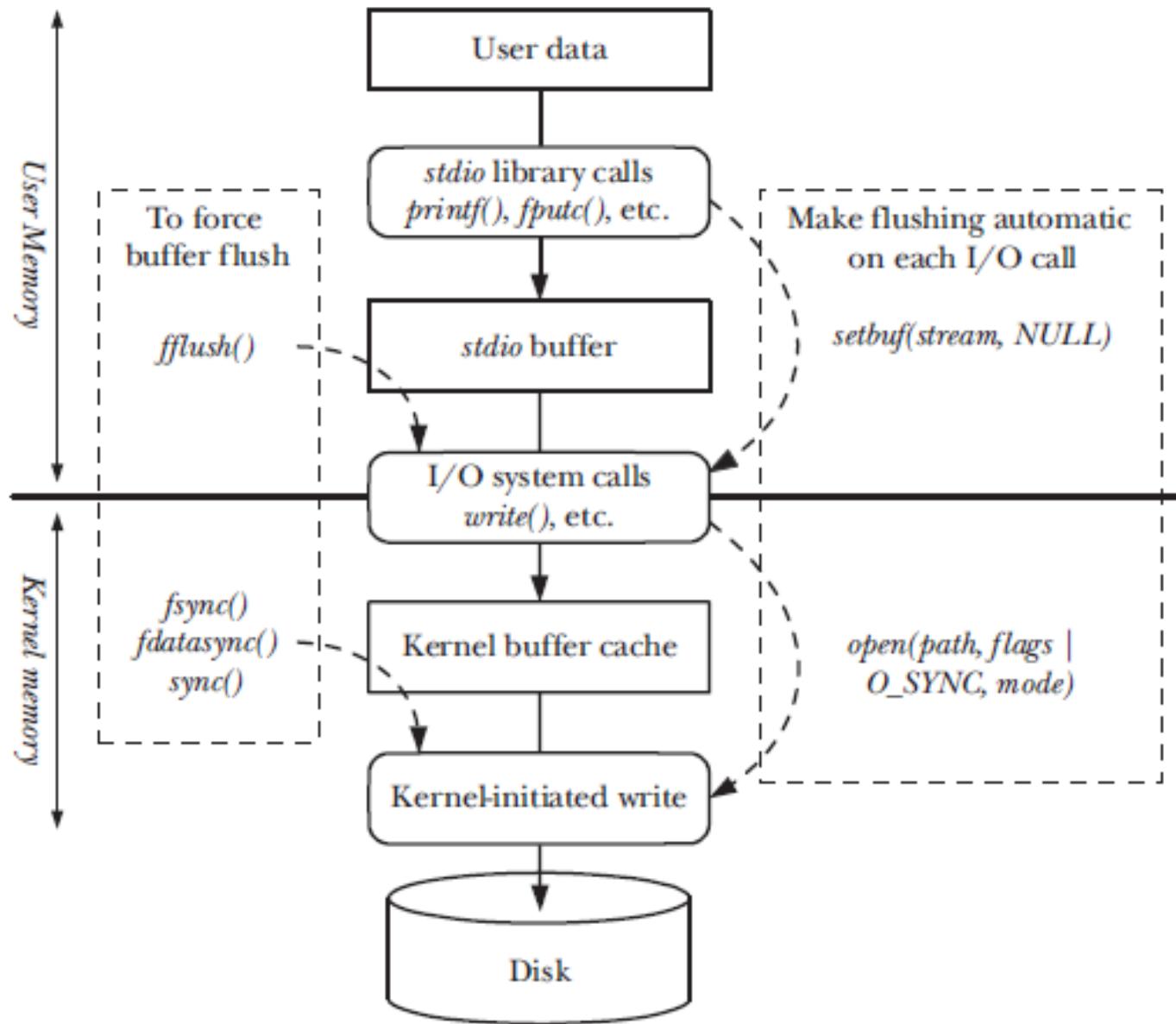
- In practice, blocks are usually 512, 1,024, 2,048, 4,096, or 8,192 bytes in size.
- A large performance gain is realized by performing operations in chunks that are integer multiples or divisors of the block size.
- Figuring out the block size for a given device is easy using the *stat()* system call or the *stat(1)* command.
- Larger multiples will simply result in fewer system calls.

Problem

- Programs work with fields, lines, and single characters, not abstractions such as blocks.
- User-buffered I/O closes the gap between the filesystem, which speaks in blocks, and the application, which talks in its own abstractions.
- How it works is simple, yet very powerful: as data is written, it is stored in a buffer inside the program's address space.

- When the buffer reaches a set size, called the *buffer size*, the entire buffer is written out in a single write operation.
- Likewise, data is read using buffer-sized, block-aligned chunks.

I/O buffering



The Buffer Cache

- The **read()** and **write()** system calls don't directly initiate disk access.
- They simply copy data between a user-space buffer and a buffer in the kernel buffer cache.

```
write(fd, "abc", 3);
```

The aim of this design is to allow `read()` and `write()` to be fast, since they don't need to wait on a (slow) disk operation.

Time required to copy a file of 100 million bytes

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	107.43	107.32	8.20	99.12
2	54.16	53.89	4.13	49.76
4	31.72	30.96	2.30	28.66
8	15.59	14.34	1.08	13.26
16	7.50	7.14	0.51	6.63
32	3.76	3.68	0.26	3.41
64	2.19	2.04	0.13	1.91
128	2.16	1.59	0.11	1.48
256	2.06	1.75	0.10	1.65
512	2.06	1.03	0.05	0.98
1024	2.05	0.65	0.02	0.63
4096	2.05	0.38	0.01	0.38
16384	2.05	0.34	0.00	0.33
65536	2.06	0.32	0.00	0.32

Remark

- With a buffer size of 1 byte, 100 million calls are made to read() and write().
- With a buffer size of 4096 bytes, the number of invocations of each system call falls to around 24,000, and near optimal performance is reached.
- Beyond this point, there is no significant performance improvement, because the cost of making read() and write() system calls becomes negligible compared to the time required to copy data between user space and kernel space, and to perform actual disk I/O.

Main result

If we are transferring a large amount of data to or from a file, then by buffering data in large blocks, and thus performing fewer system calls, we can greatly improve I/O performance.

Time required to write a file of 100 million bytes (table 2)

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	72.13	72.11	5.00	67.11
2	36.19	36.17	2.47	33.70
4	20.01	19.99	1.26	18.73
8	9.35	9.32	0.62	8.70
16	4.70	4.68	0.31	4.37
32	2.39	2.39	0.16	2.23
64	1.24	1.24	0.07	1.16
128	0.67	0.67	0.04	0.63
256	0.38	0.38	0.02	0.36
512	0.24	0.24	0.01	0.23
1024	0.17	0.17	0.01	0.16
4096	0.11	0.11	0.00	0.11
16384	0.10	0.10	0.00	0.10
65536	0.09	0.09	0.00	0.09

Main result

- The majority of the time required for the large buffer cases in the first table is due to the disk **reads**.

Standard I/O (stdio)

File pointers

- Standard I/O routines do not operate directly on file descriptors.
- Instead, they use their own unique identifier, known as the *file pointer*.
- Inside the C library, the file pointer maps to a file descriptor. The file pointer is represented by a pointer to the FILE typedef, which is defined in <stdio.h>.

Standard I/O (stdio)

Opening files

```
#include <stdio.h>

FILE * fopen (const char *path, const char *mode);
```

Modes

r

Open the file for reading. The stream is positioned at the start of the file.

r+

Open the file for both reading and writing. The stream is positioned at the start of the file.

w

Open the file for writing. If the file exists, it is truncated to zero length. If the file does not exist, it is created. The stream is positioned at the start of the file.

w+

Open the file for both reading and writing. If the file exists, it is truncated to zero length. If the file does not exist, it is created. The stream is positioned at the start of the file.

a

Open the file for writing in append mode. The file is created if it does not exist. The stream is positioned at the end of the file. All writes will append to the file.

a+

Open the file for both reading and writing in append mode. The file is created if it does not exist. The stream is positioned at the end of the file. All writes will append to the file.

Example

```
FILE *stream;  
  
stream = fopen ("/etc/manifest", "r");  
if (!stream)  
    /* error */
```

Opening a Stream via File Descriptor

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

```
FILE * fdopen (int fd, const char *mode);
```

Using the file descriptor to create an associated stream

```
FILE *stream;  
int fd;  
  
fd = open ("/home/kidd/map.txt", O_RDONLY);  
if (fd == -1)  
    /* error */  
  
stream = fdopen (fd, "r");  
if (!stream)  
    /* error */
```

Closing Streams

```
#include <stdio.h>

int fclose (FILE *stream);
```

```
#include <stdio.h>

int fcloseall (void);
```

Reading a Character at a Time

```
#include <stdio.h>

int fgetc (FILE *stream);

int c;

c = fgetc (stream);
if (c == EOF)
    /* error */
else
    printf ("c=%c\n", (char) c);
```

Putting the character back

```
#include <stdio.h>

int ungetc (int c, FILE *stream);
```

Reading an Entire Line

The function `fgets()` reads a string from a given stream:

```
#include <stdio.h>

char * fgets (char *str, int size, FILE *stream);
```

This function reads up to *one less* than `size` bytes from `stream` and stores the results in `str`. A null character (`\0`) is stored in the buffer after the last byte read in. Reading stops after an EOF or a newline character is reached. If a newline is read, the `\n` is stored in `str`.

On success, `str` is returned; on failure, `NULL` is returned.

```
char buf[LINE_MAX];

if (!fgets (buf, LINE_MAX, stream))
    /* error */
```

Reading Binary Data

```
#include <stdio.h>

size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

A call to `fread()` will read up to `nr` elements of data, each of `size` bytes, from `stream` into the buffer pointed at by `buf`. The file pointer is advanced by the number of bytes read.

Writing a Single Character

```
#include <stdio.h>

int fputc (int c, FILE *stream);
```

The `fputc()` function writes the byte specified by `c` (cast to an `unsigned char`) to the stream pointed at by `stream`. Upon successful completion, the function returns `c`. Otherwise, it returns `EOF`, and `errno` is set appropriately.

Use is simple:

```
if (fputc ('p', stream) == EOF)
    /* error */
```

This example writes the character `p` to `stream`, which must be open for writing.

Writing a String of Characters

```
#include <stdio.h>

int fputs (const char *str, FILE *stream);
```

A call to `fputs()` writes all of the null-terminated string pointed at by `str` to the stream pointed at by `stream`. On success, `fputs()` returns a nonnegative number. On failure, it returns EOF.

Example

```
FILE *stream;

stream = fopen ("journal.txt", "a");
if (!stream)
    /* error */

if (fputs ("The ship is made of wood.\n", stream) == EOF)
    /* error */

if (fclose (stream) == EOF)
    /* error */
```

Writing Binary Data

```
#include <stdio.h>

size_t fwrite (void *buf,
               size_t size,
               size_t nr,
               FILE *stream);
```

A call to `fwrite()` will write to `stream` up to `nr` elements, each `size` bytes in length, from the data pointed at by `buf`. The file pointer will be advanced by the total number of bytes written.

The number of elements (not the number of bytes!) successfully written will be returned.
A return value less than `nr` denotes error.

Sample Program Using Buffered I/O

```
#include <stdio.h>

int main (void)
{
    FILE *in, *out;
    struct pirate {
        char          name[100]; /* real name */
        unsigned long booty;     /* in pounds sterling */
        unsigned int   beard_len; /* in inches */
    } p, blackbeard = { "Edward Teach", 950, 48 };

    out = fopen ("data", "w");
    if (!out) {
        perror ("fopen");
        return 1;
    }

    if (!fwrite (&blackbeard, sizeof (struct pirate), 1, out)) {
        perror ("fwrite");
        return 1;
    }
}
```

```
}

if (fclose (out)) {
    perror ("fclose");
    return 1;
}

in = fopen ("data", "r");
if (!in) {
    perror ("fopen");
    return 1;
}

if (!fread (&p, sizeof (struct pirate), 1, in)) {
    perror ("fread");
    return 1;
}

if (fclose (in)) {
    perror ("fclose");
    return 1;
}

printf ("name=\"%s\" booty=%lu beard_len=%u\n",
       p.name, p.booty, p.beard_len);

return 0;
}
```

Seeking a Stream

```
#include <stdio.h>

int fseek (FILE *stream, long offset, int whence);
```

If whence is set to SEEK_SET, the file position is set to offset. If whence is set to SEEK_CUR, the file position is set to the current position plus offset. If whence is set to SEEK_END, the file position is set to the end of the file plus offset.

Obtaining the Current Stream Position

```
#include <stdio.h>

long ftell (FILE *stream);
```

On error, it returns `-1` and `errno` is set appropriately.

Flushing a Stream

```
#include <stdio.h>

int fflush (FILE *stream);
```

On invocation, any unwritten data in the stream pointed to by `stream` is flushed to the kernel. If `stream` is `NULL`, *all* open input streams in the process are flushed. On success, `fflush()` returns `0`. On failure, it returns `EOF`, and `errno` is set appropriately.

ferror()

- The function `ferror()` tests whether the error indicator is set on stream:

```
#include <stdio.h>

int ferror (FILE *stream);
```

The error indicator is set by other standard I/O interfaces in response to an error condition. The function returns a nonzero value if the indicator is set, and `0` otherwise.

feof()

- The function `feof()` tests whether the EOF indicator is set on stream:

```
#include <stdio.h>

int feof (FILE *stream);
```

The EOF indicator is set by other standard I/O interfaces when the end of a file is reached. This function returns a nonzero value if the indicator is set, and 0 otherwise.

clearerr()

- The clearerr() function clears the error and the EOF indicators for stream:

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

```
void clearerr (FILE *stream);
```

Obtaining the Associated File Descriptor

- Sometimes it is advantageous to obtain the file descriptor backing a given stream.
- For example, it might be useful to perform a system call on a stream, via its file descriptor, when an associated standard I/O function does not exist.

```
#include <stdio.h>  
  
int fileno (FILE *stream);
```

Controlling the Buffering

- *Unbuffered*
 - No user buffering is performed. Data is submitted directly to the kernel. As this disables user buffering, negating any benefit, this option is not commonly used, with a lone exception: standard error, by default, is unbuffered.

- *Line-buffered*
 - Buffering is performed on a per-line basis. With each newline character, the buffer is submitted to the kernel. Line buffering makes sense for streams being output to the screen, since messages printed to the screen are delimited with newlines. Consequently, this is the default buffering used for streams connected to terminals, such as standard out.
- *Block-buffered*
 - Buffering is performed on a per-block basis, where a block is a fixed number of bytes.

Controlling the Buffering

```
#include <stdio.h>

int setvbuf (FILE *stream, char *buf, int mode, size_t size);
```

The `setvbuf()` function sets the buffering type of `stream` to `mode`, which must be one of the following:

`_IONBF`

Unbuffered

`_IOLBF`

Line-buffered

`_IOFBF`

Block-buffered

System programming

Lecture 3

Processes

Part 1

Process

- A process is an instance of an executing program.

Program

- A program is a file containing **a range of information** that describes how to construct a process at run time.

Structure of a program file

- Machine-language instructions
- Program entry-point address
- Data
- Symbol and relocation tables
- Shared-library and dynamic-linking information
- etc.

Structure of a process

A process consists of :

- **user-space** memory containing program code and variables,
- and a range of **kernel** data structures that maintain information about the state of the process
 - Ids
 - Virtual memory tables
 - Table of open file descriptors
 - Information relating to signal handling
 - Process resource usages and limits
 - etc.

Process ID (PID)

- a positive integer that uniquely identifies the process on the system

getpid() system call

```
#include <unistd.h>  
  
pid_t getpid(void);
```

Always successfully returns process ID of caller

Parent process

- Each process has a parent—the process that created it.

getppid() system call

```
#include <unistd.h>

pid_t getppid(void);
```

Always successfully returns process ID of parent of caller

init

- the ancestor of all processes.
- If a child process becomes **orphaned** because its “birth” parent terminates, then the child is adopted by the init process, and subsequent calls to `getppid()` in the child return 1.

Memory Layout of a Process

- **text segment** (machine-language instructions of the program)
- **initialized data segment** (global and static variables that are explicitly initialized)
- **uninitialized data segment** (global and static variables that are not explicitly initialized)
- **stack** (stack frames, one for each currently called function)
- **heap** (area from which memory can be dynamically allocated at run time)

Mapping between C variables and the segments of a process

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

char globBuf[65536];          /* Uninitialized data segment */
int primes[] = { 2, 3, 5, 7 }; /* Initialized data segment */

static int
square(int x)                  /* Allocated in frame for square() */
{
    int result;                /* Allocated in frame for square() */

    result = x * x;
    return result;              /* Return value passed via register */
}
```

```
static void
doCalc(int val) /* Allocated in frame for doCalc() */
{
    printf("The square of %d is %d\n", val, square(val));

    if (val < 1000) {
        int t; /* Allocated in frame for doCalc() */

        t = val * val * val;
        printf("The cube of %d is %d\n", val, t);
    }
}
```

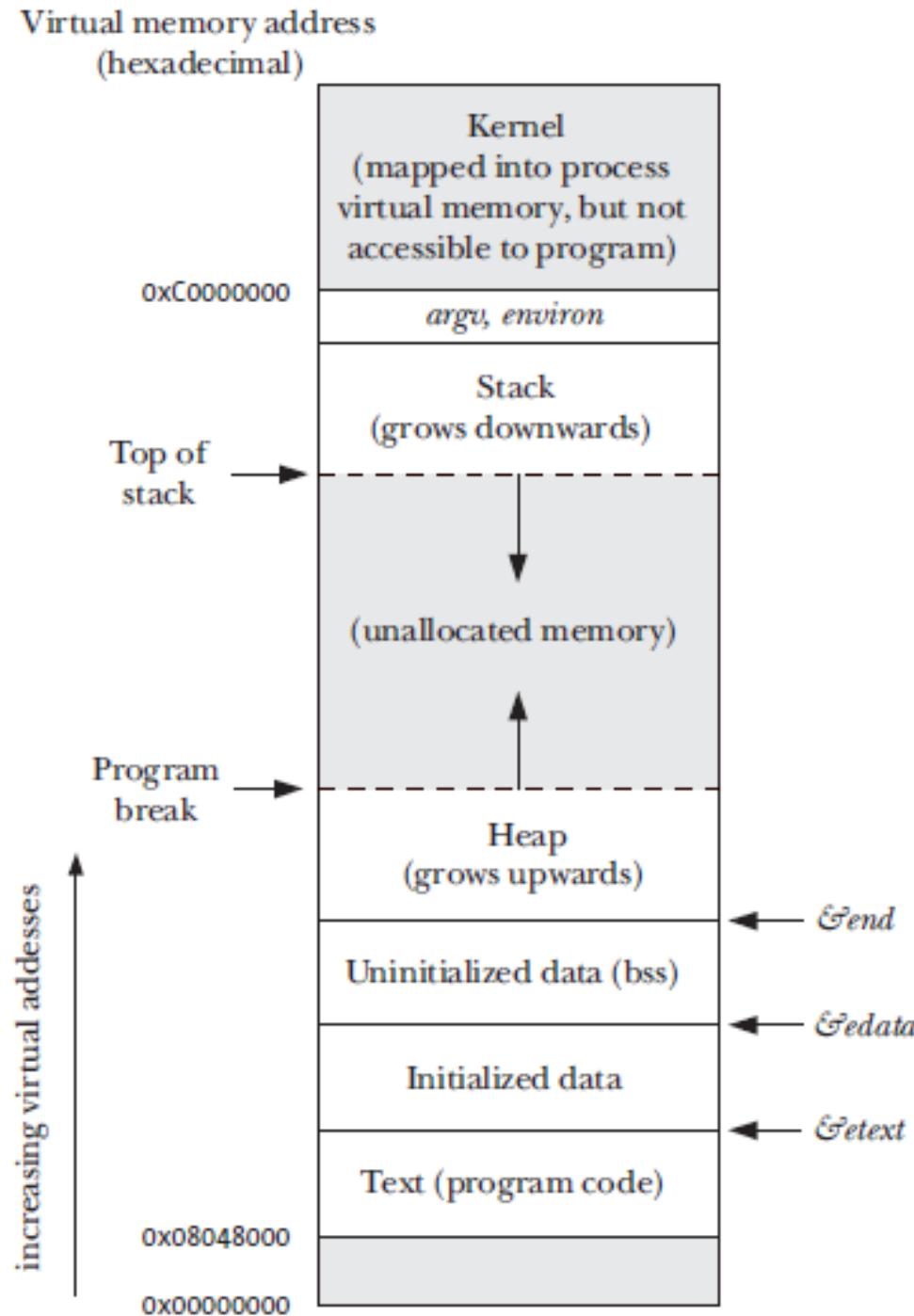
```
int
main(int argc, char *argv[]) /* Allocated in frame for main() */
{
    static int key = 9973;      /* Initialized data segment */
    static char mbuf[10240000]; /* Uninitialized data segment */
    char *p;                  /* Allocated in frame for main() */

    p = malloc(1024);         /* Points to memory in heap segment */

    doCalc(key);

    exit(EXIT_SUCCESS);
}
```

Memory layout of a process on Linux x86-32 architecture



Locality of reference

- **Spatial locality** is the tendency of a program to reference memory addresses that are near those that were recently accessed (because of sequential processing of instructions, and, sometimes, sequential processing of data structures).
- **Temporal locality** is the tendency of a program to access the same memory addresses in the near future that it accessed in the recent past (e.g., because of loops).

Virtual Memory

- The term *virtual memory* refers to a combination of hardware and operating system software that solves several computing problems. It receives a single name because it is a single mechanism, but it meets several goals:
- To simplify memory management and program loading by providing *virtual addresses*.
- To allow multiple large programs to be run without the need for large amounts of RAM, by providing virtual storage.

Virtual memory management

- The aim of the virtual memory management technique is to make efficient use of both the CPU and RAM (physical memory) by exploiting a locality of reference.

Pages and frames

- A virtual memory scheme splits the memory used by each program into small, fixed-size units called **pages**.
- Correspondingly, RAM is divided into a series of **page frames** of the same size.

Resident set

- At any one time, only some of the pages of a program need to be resident in physical memory page frames; these pages form the so-called resident set.

Swap area

- Copies of the unused pages of a program are maintained in the swap area—a reserved area of disk space loaded into physical memory only as required.

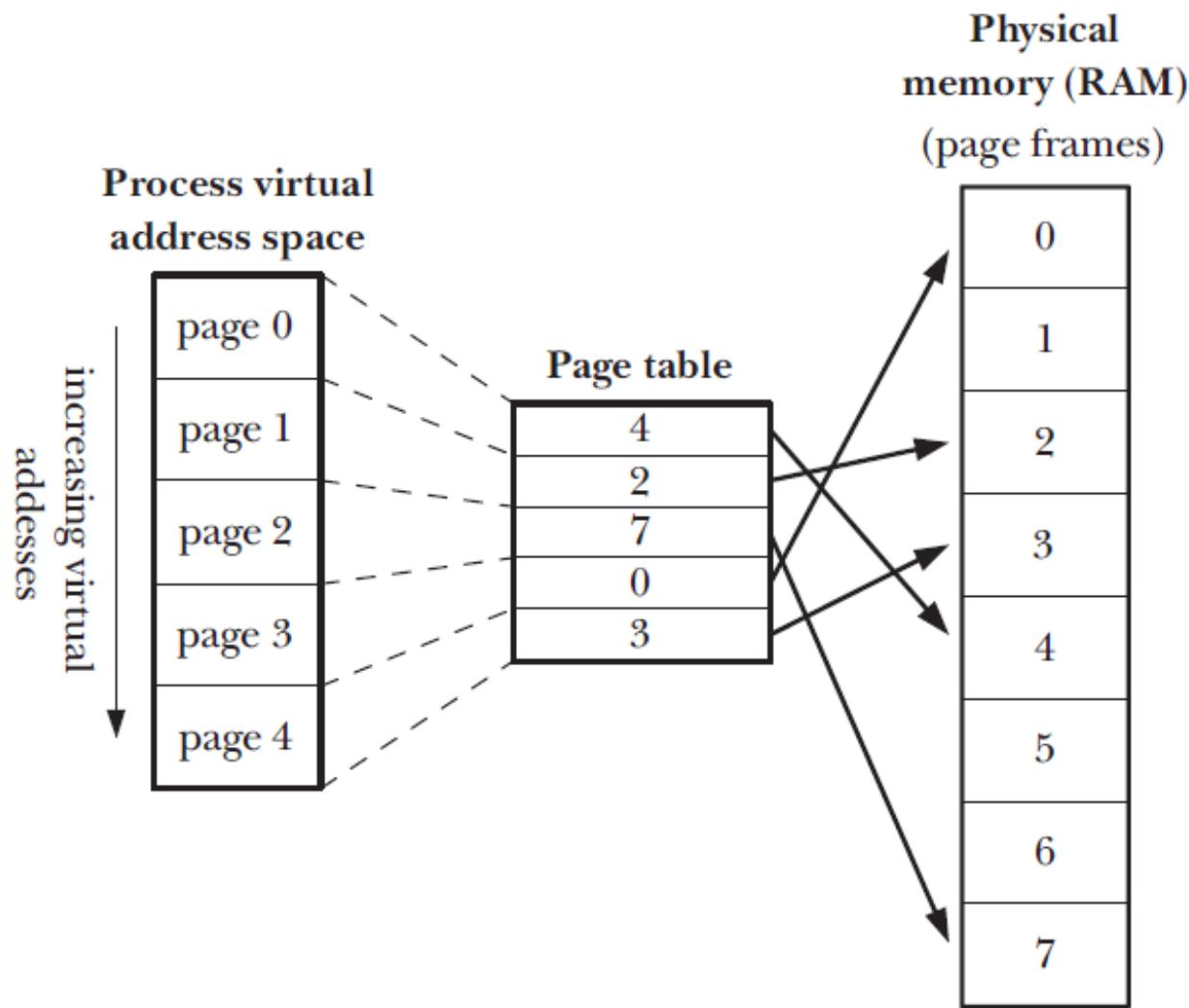
Page fault

- When a process references a page that is not currently resident in physical memory, a page fault occurs, at which point the kernel suspends execution of the process while the page is loaded from disk into memory.

Page table

- The page table describes the location of each page in the process's virtual address space.
- Each entry in the page table either indicates the location of a virtual page in RAM or indicates that it currently resides on disk.

Virtual memory



Paged memory management unit (PMMU)

- The PMMU translates each virtual memory address reference into the corresponding physical memory address and advises the kernel of a page fault when a particular virtual memory address corresponds to a page that is not resident in RAM.

Advantages of using VM

- Processes are **isolated** from one another and from the kernel, so that one process can't read or modify the memory of another process or the kernel.
- Where appropriate, two or more processes can **share** memory. The kernel makes this possible by having page-table entries in different processes refer to the same pages of RAM.

Advantages of using VM

- Programmers, and tools such as the compiler and linker, don't need to be concerned with the **physical layout** of the program in RAM.
- Because only a part of a program needs to reside in memory, the program loads and runs **faster**.
- Since each process uses less RAM, **more processes** can simultaneously be held in RAM.

Stack

- The stack grows and shrinks as functions are called and return.
- Stack holds parameter values, local variables, and the address of calling function.
- For Linux on the x86-32 architecture, the stack resides at the **high end** of memory and grows **downward**.
- A special-purpose register, the **stack pointer**, tracks the current top of the stack.
- Each time a function is called, an additional **stack frame** (or **activation record**) is allocated on the stack, and this frame is removed when the function returns.

Example: Simple functions

```
int foo()
{
    int b;          int bar()
    b = bar();      {
                    return b;
    }              int b = 0;
                  b = baz(b)
                    int baz(int b)
                    return b;
    }              {
                    if (b < 1) return baz(b + 1);
                    else return b;
    }
```

A call to `foo()`

`foo()` calls `bar()`.

`bar()` calls `baz(0)`.

In `baz()`, `b` is 0, and `b < 1`, so `baz()` calls `baz(b + 1)`, or `baz(1)`.

In this second instance of `baz()`, `b` is 1, the test (`b < 1`) fails, so we return `b`, which is 1.

Back in the first instance of `baz()`, we return `baz(b + 1)`, which just returned 1.

Returning from the first instance of `baz()` with 1, we are in `bar()`. `b` is assigned 1.

Returning `b` (which is 1) from `bar()`, we arrive at `foo()`, where `b` is assigned 1.

`foo()` returns `b`, which is 1.

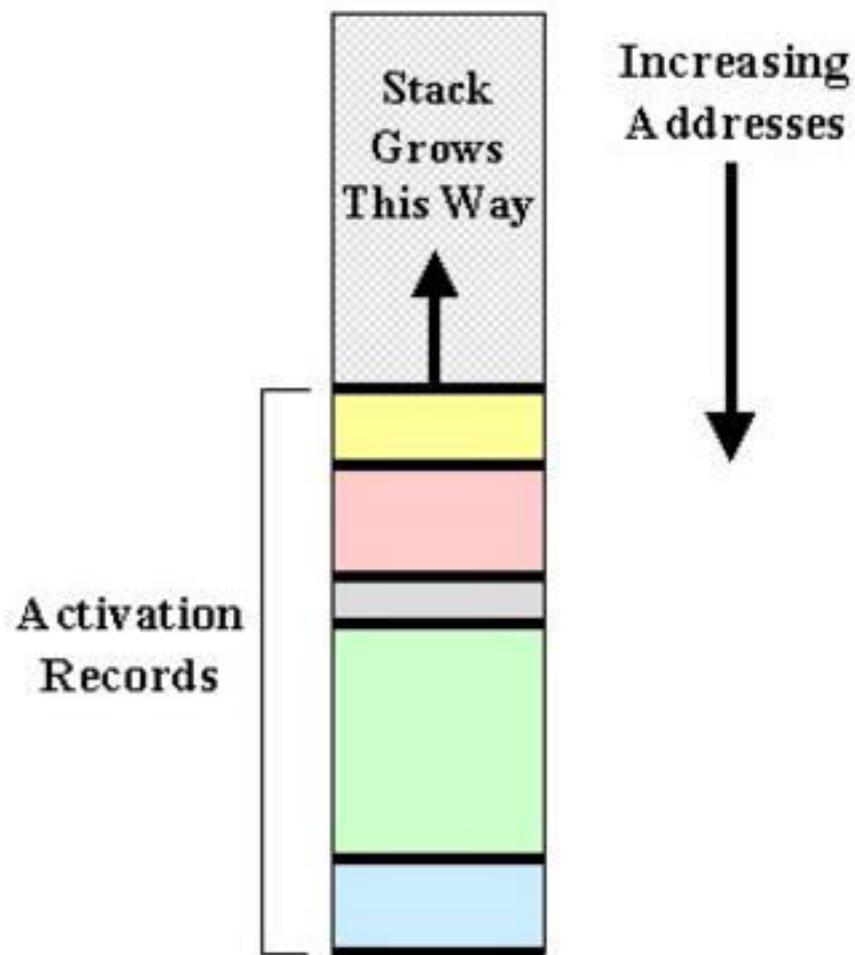
Problems

- variables in different functions can have the same name but still represent different variables;
- each instance of a recursive function can have its own set of private variables;
- recursive functions can create arbitrarily many instances of functions

Stack frame (activation record)

- The information for one function is called an *activation record* or *stack frame*.
- Function arguments and local variables
- Call linkage information
- When the function returns, the activation record will be removed

Stack Frame

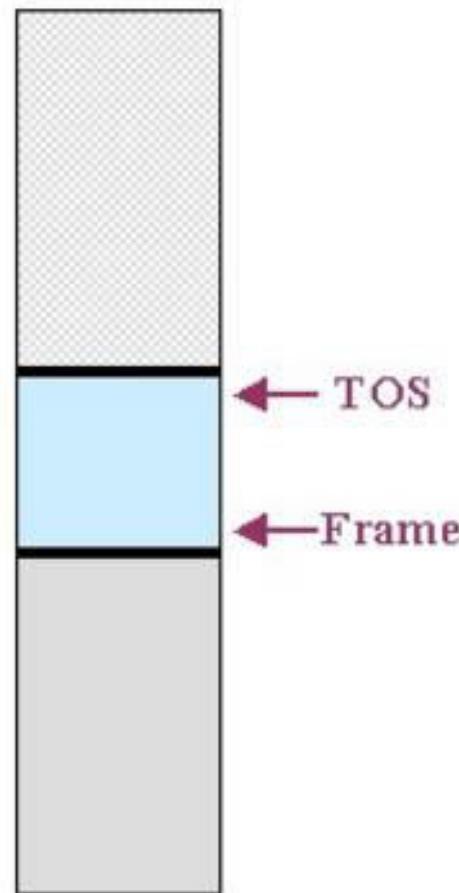


Call stack: push b

TOS = Top of Stack
Frame = Activation Record Base
PC = Program Counter

bar:

```
...
b = baz(b)
push b ← PC
call baz
pop b
...
```



Call stack: call baz

TOS = Top of Stack

Frame = Activation Record Base

PC = Program Counter

bar:

...

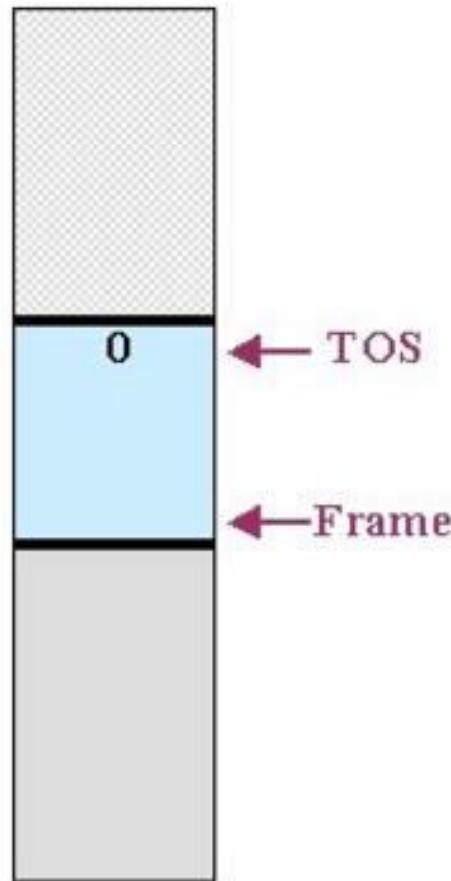
b = *baz*(*b*)

push *b*

call *baz* ← PC

pop *b*

...



Call stack: save state of bar

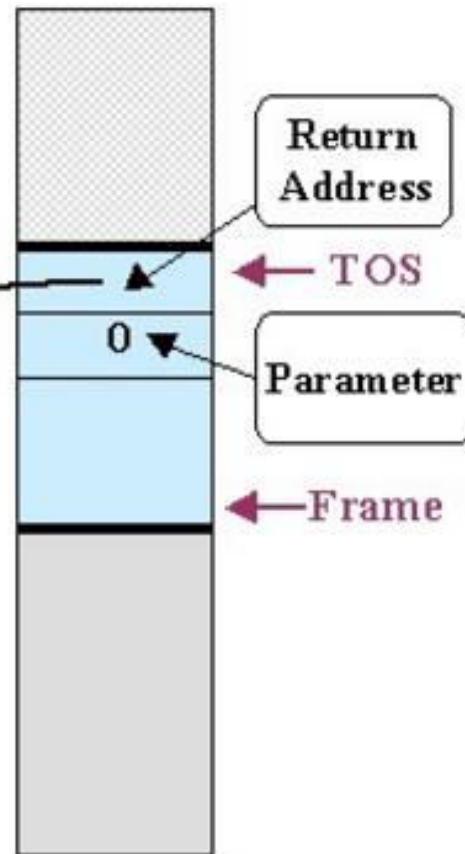
TOS = Top of Stack
Frame = Activation Record Base
PC = Program Counter

bar:

```
...
b = baz(b)
push b
call baz
pop b
...
```

baz:

```
push Frame ← PC
Frame = TOS
...
```



Call stack: execute baz

TOS = Top of Stack

Frame = Activation Record Base

PC = Program Counter

bar:

```
...
    b = baz(b)
    push b
    call baz
    pop b
...

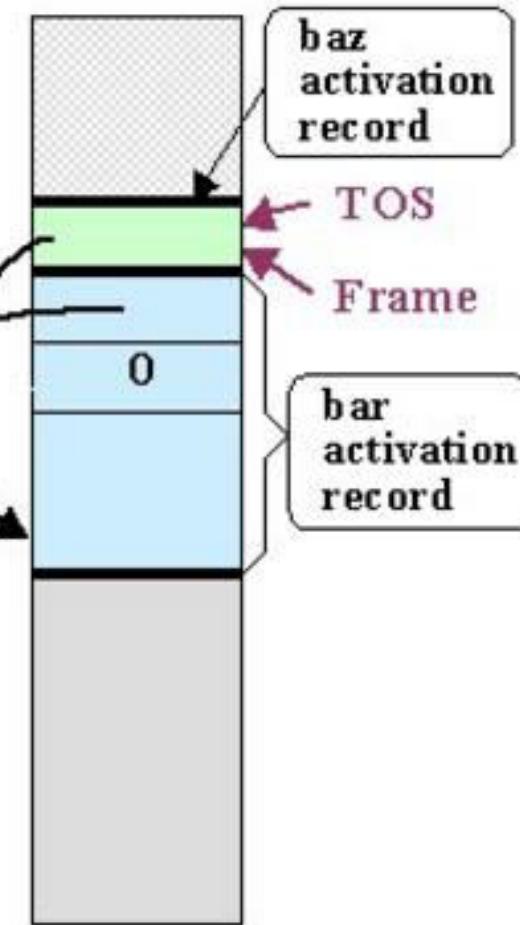
```

baz:

```
push Frame
```

```
Frame = TOS
```

```
...
```



- Notice that `baz()` will access its parameter `b` from the stack by using an offset of eight bytes from the *Frame* register. (The pointer to the caller's activation record takes four bytes, and the return address takes another four bytes; the parameter is the next location in memory.)

Returning From a Function Call

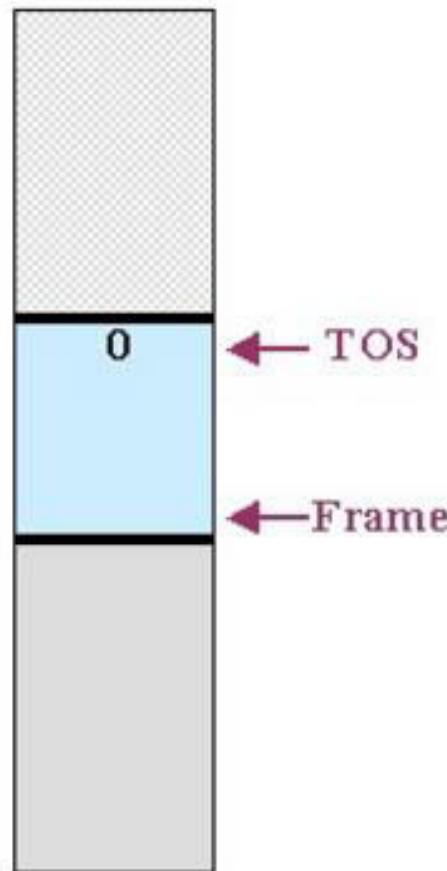
- `TOS = Frame // restore top of stack from Frame base pointer`
- `pop Frame // pop the top of stack (previous Frame base) into Frame`
- `return // pop the PC from the top of the stack`

Call stack: pop b

TOS = Top of Stack
Frame = Activation Record Base
PC = Program Counter

bar:

```
...
b = baz(b)
push b
call baz
pop b ← PC
...
```

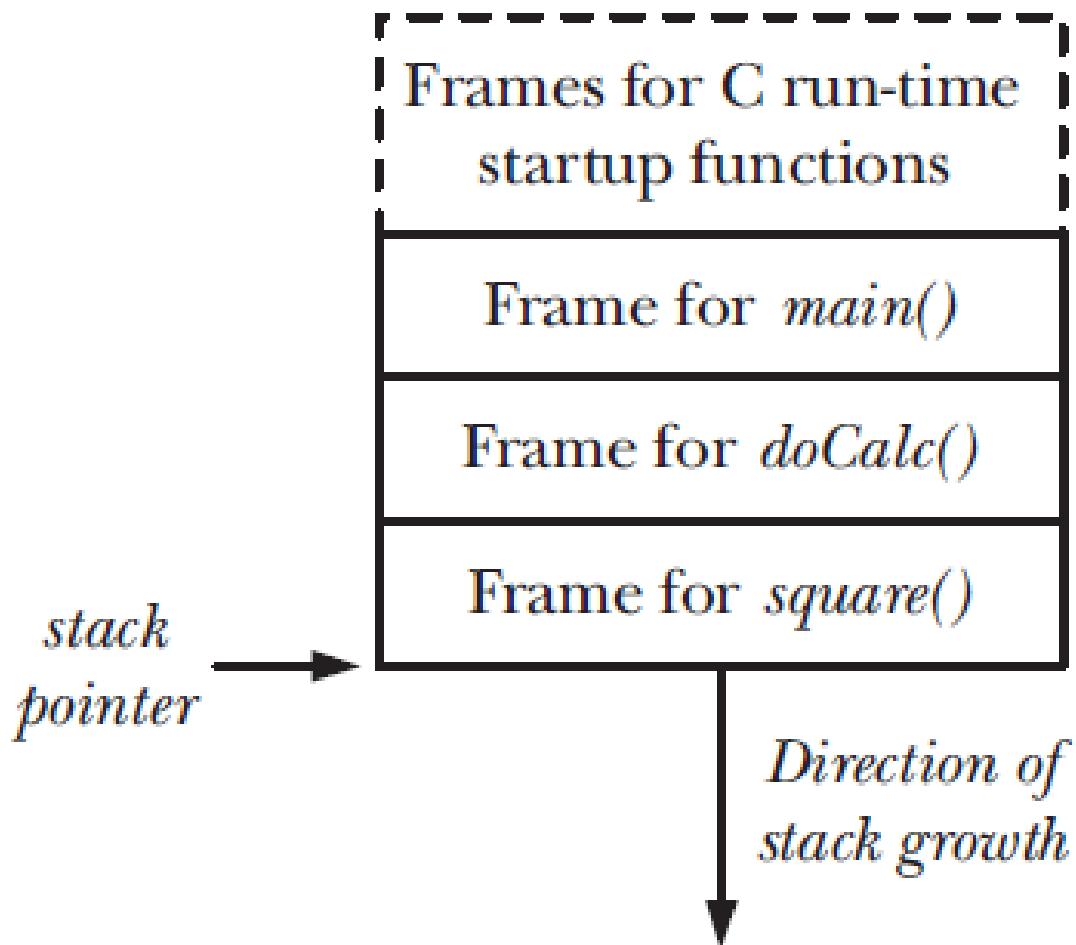


Activation record

In general, an activation record may contain:

- **Parameters.** For example, the parameter 0 passed by bar() to baz()
- **The return address.** The address used to restore the PC. (Again, this is shown in bar()'s activation record.)
- **A pointer to the caller's activation record.** The pointer used to restore the *Frame* pointer.
- **Saved machine registers.** In addition to saving the *Frame* pointer, the callee may need to save and restore other machine registers, allowing them to be used as fast, intermediate storage while executing instructions in the callee.
- **Local variables.** Any variables declared within the function are allocated on the stack.

Example of a process stack



System programming

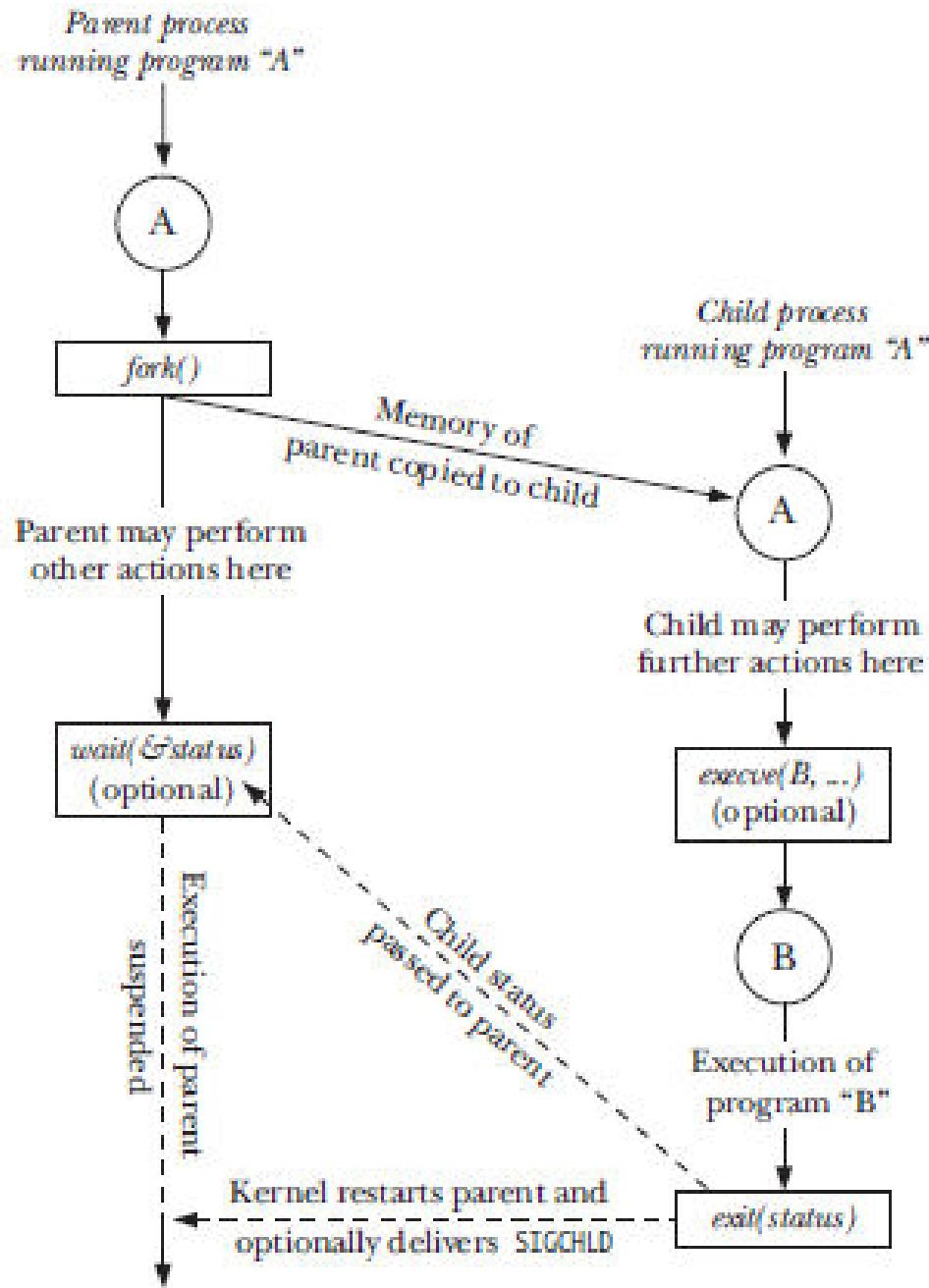
Lecture 3

Processes

Part 2

Main system calls

- **fork()**
- **wait()**
- **exit()** (library function layered on top of `_exit()` system call)
- **execve()**



PROCESS CREATION

Creating a New Process: `fork()`

```
#include <unistd.h>

pid_t fork(void);
```

In parent: returns process ID of child on success, or `-1` on error;
in successfully created child: always returns `0`

The two processes are executing the same program text, but they have separate copies of the stack, data, and heap segments.

How to distinguish two processes?

- For the parent, **fork()** returns the process ID of the newly created child. This is useful because the parent may create, and thus need to track, several children (via `wait()` or one of its relatives).
- For the child, **fork()** returns 0.

General structure

```
pid_t childPid;           /* Used in parent after successful fork()  
                           to record PID of child */  
  
switch (childPid = fork()) {  
case -1:                 /* fork() failed */  
    /* Handle error */  
  
case 0:                  /* Child of successful fork() comes here */  
    /* Perform actions specific to child */  
  
default:                 /* Parent comes here after successful fork() */  
    /* Perform actions specific to parent */  
}
```

Using fork()

```
static int idata = 111;           /* Allocated in data segment */

int
main(int argc, char *argv[])
{
    int istack = 222;             /* Allocated in stack segment */
    pid_t childPid;

    switch (childPid = fork()) {
    case -1:
        errExit("fork");

    case 0:
        idata *= 3;
        istack *= 3;
        break;

    default:
        sleep(3);                 /* Give child a chance to execute */
        break;
    }
}
```

```
/* Both parent and child come here */

printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
       (childPid == 0) ? "(child)" : "(parent)", idata, istack);

exit(EXIT_SUCCESS);
}
```

Output

```
$ ./t_fork
PID=28557 (child)  idata=333  istack=666
PID=28556 (parent) idata=111  istack=222
```

File sharing between parent and child

(sharing of file offset and open file status flags)

```
printf("File offset before fork(): %lld\n",
       (long long) lseek(fd, 0, SEEK_CUR));

flags = fcntl(fd, F_GETFL);
if (flags == -1)
    errExit("fcntl - F_GETFL");
printf("O_APPEND flag before fork() is: %s\n",
       (flags & O_APPEND) ? "on" : "off");
```

```
switch (fork()) {  
case -1:  
    errExit("fork");  
  
case 0: /* Child: change file offset and status flags */  
    if (lseek(fd, 1000, SEEK_SET) == -1)  
        errExit("lseek");  
  
    flags = fcntl(fd, F_GETFL);           /* Fetch current flags */  
    if (flags == -1)  
        errExit("fcntl - F_GETFL");  
    flags |= O_APPEND;                  /* Turn O_APPEND on */  
    if (fcntl(fd, F_SETFL, flags) == -1)  
        errExit("fcntl - F_SETFL");  
    _exit(EXIT_SUCCESS);
```

```
default: /* Parent: can see file changes made by child */
    if (wait(NULL) == -1)
        errExit("wait"); /* Wait for child exit */
    printf("Child has exited\n");

    printf("File offset in parent: %lld\n",
           (long long) lseek(fd, 0, SEEK_CUR));

    flags = fcntl(fd, F_GETFL);
    if (flags == -1)
        errExit("fcntl - F_GETFL");
    printf("O_APPEND flag in parent is: %s\n",
           (flags & O_APPEND) ? "on" : "off");
    exit(EXIT_SUCCESS);
}
```

Output

```
$ ./fork_file_sharing
File offset before fork(): 0
O_APPEND flag before fork() is: off
Child has exited
File offset in parent: 1000
O_APPEND flag in parent is: on
```

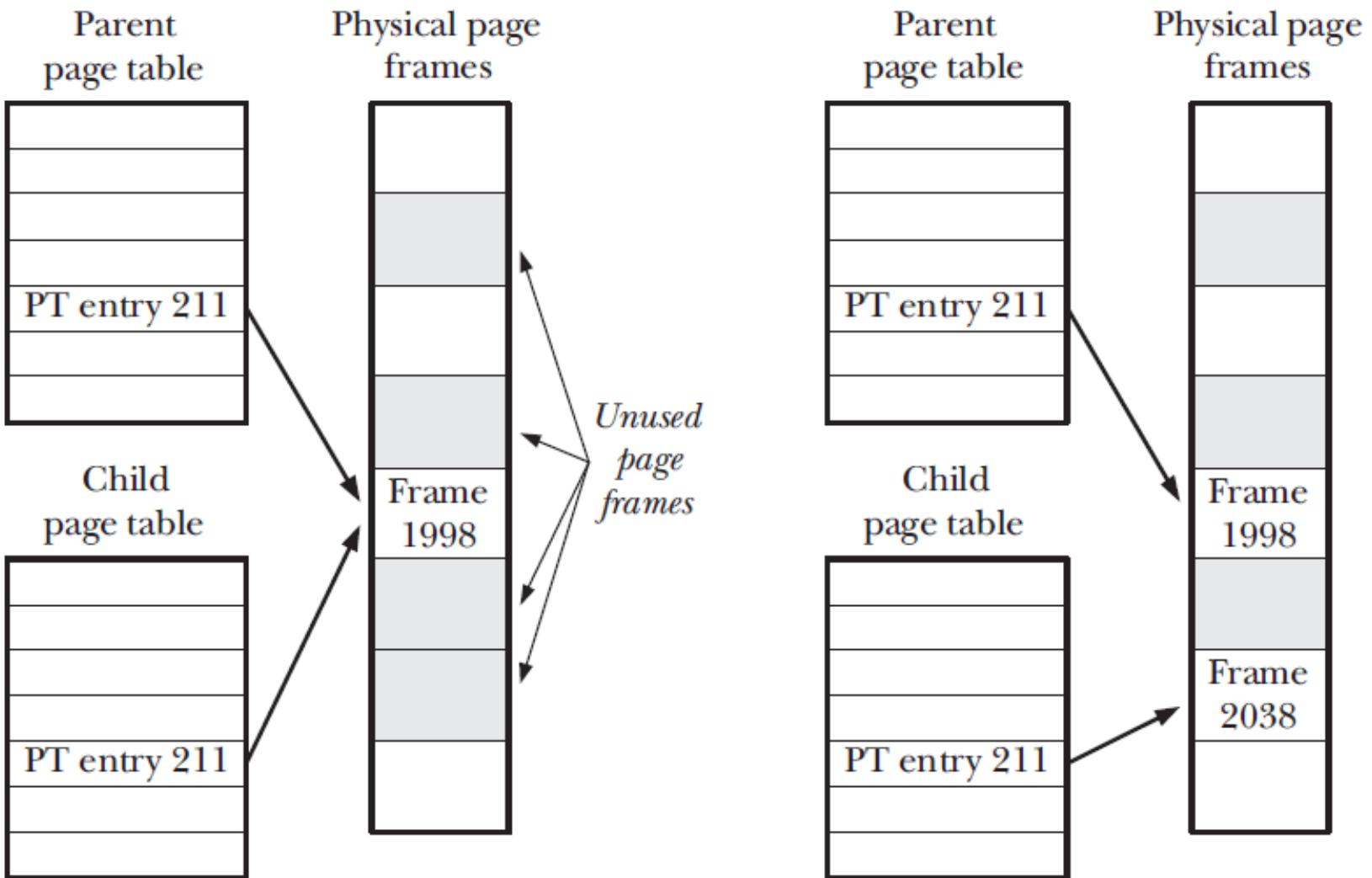
Main result

attributes are shared by the parent and child after a fork()

Memory Semantics of `fork()`

- The kernel marks the **text** segment of each process as read-only, so that a process can't modify its own code.
- For the pages in the **data**, **heap**, and **stack** segments of the parent process, the kernel employs a technique known as *copy-on-write*.

Copy-on-write technique



Process's memory footprint

- The process's memory footprint is the range of virtual memory pages used by the process.

Calling a function without changing the parent process's memory footprint

```
pid_t childPid;
int status;

childPid = fork();
if (childPid == -1)
    errExit("fork");

if (childPid == 0)                  /* Child calls func() and */
    exit(func(arg));               /* uses return value as exit status */

/* Parent waits for child to terminate. It can determine the
   result of func() by inspecting 'status'. */

if (wait(&status) == -1)
    errExit("wait");
```

Main idea

- If we know that `func()` causes memory leaks or excessive fragmentation of the heap, this technique eliminates the problem.

Race conditions after **fork()**

- After a `fork()`, it is indeterminate which process—the parent or the child—next has access to the CPU.

Parent and child race to write a message after fork()

```
for (j = 0; j < numChildren; j++) {  
    switch (childPid = fork()) {  
        case -1:  
            errExit("fork");  
  
        case 0:  
            printf("%d child\n", j);  
            _exit(EXIT_SUCCESS);  
  
        default:  
            printf("%d parent\n", j);  
            wait(NULL); /* Wait for child to terminate */  
            break;  
    }  
}
```

Main idea

- We can't assume a particular order of execution for the parent and child after a `fork()`.
- If we need to guarantee a particular order, we must use some kind of synchronization technique.

PROCESS TERMINATION

Two ways of process termination

- **Abnormal** (delivery of a signal whose default action is to terminate the process)
- **Normal** (using the `_exit()` system call)

_exit() system call

```
#include <unistd.h>

void _exit(int status);
```

exit() library function

```
#include <stdlib.h>  
  
void exit(int status);
```

Actions are performed by **exit()**:

- Exit handlers are called.
- The stdio stream buffers are flushed.
- The **_exit()** system call is invoked, using the value supplied in *status*.

Details of Process Termination

- Open file descriptors are closed.
- Any file locks held by this process are released.
- Any POSIX named semaphores that are open in the calling process are closed.
- Any POSIX message queues that are open in the calling process are closed.
- Any memory locks established by this process are removed.
- Any memory mappings established by this process are unmapped.
- etc.

Exit Handlers

- An exit handler is a programmer-supplied function that is registered at some point during the life of the process and is then automatically called during **normal** process termination via **exit()**.
- Exit handlers **are not** called if a program calls **_exit()** directly or if the process is terminated **abnormally** by a signal.

Registering exit handlers with *atexit()*

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Returns 0 on success, or nonzero on error

- The **atexit()** function adds **func** to a list of functions that are called when the process terminates.
- It is possible to register multiple exit handlers.
- When the program invokes `exit()`, these functions are called in **reverse** order of registration.
- A child process created via `fork()` inherits a copy of its parent's exit handler registrations. When a process performs an **exec()**, all exit handler registrations are removed.

Limitations of *atexit()*

- when called, an exit handler doesn't know what status was passed to `exit()`;
- we can't specify an argument to the exit handler when it is called

Registering exit handlers with *on_exit()*

```
#define _BSD_SOURCE           /* Or: #define _SVID_SOURCE */  
#include <stdlib.h>  
  
int on_exit(void (*func)(int, void *), void *arg);
```

Returns 0 on success, or nonzero on error

Using exit handlers

```
static void  
atexitFunc1(void)  
{  
    printf("atexit function 1 called\n");  
}  
  
static void  
atexitFunc2(void)  
{  
    printf("atexit function 2 called\n");  
}  
  
static void  
onexitFunc(int exitStatus, void *arg)  
{  
    printf("on_exit function called: status=%d, arg=%ld\n",  
          exitStatus, (long) arg);  
}
```

```
int
main(int argc, char *argv[])
{
    if (on_exit(onexitFunc, (void *) 10) != 0)
        fatal("on_exit 1");
    if (atexit(atexitFunc1) != 0)
        fatal("atexit 1");
    if (atexit(atexitFunc2) != 0)
        fatal("atexit 2");
    if (on_exit(onexitFunc, (void *) 20) != 0)
        fatal("on_exit 2");

    exit(2);
}
```

Output

```
$ ./exit_handlers
on_exit function called: status=2, arg=20
atexit function 2 called
atexit function 1 called
on_exit function called: status=2, arg=10
```

Interaction of `fork()` and stdio buffering

```
int
main(int argc, char *argv[])
{
    printf("Hello world\n");
    write(STDOUT_FILENO, "Ciao\n", 5);

    if (fork() == -1)
        errExit("fork");

    /* Both child and parent continue execution here */

    exit(EXIT_SUCCESS);
}
```

Standard output to the terminal

```
$ ./fork_stdio_buf
```

```
Hello world
```

```
Ciao
```

Standard output to a file

```
$ ./fork_stdio_buf > a
```

```
$ cat a
```

```
Ciao
```

```
Hello world
```

```
Hello world
```

Results

- The stdio buffers are maintained in a process's user-space memory. Therefore, these buffers are duplicated in the child by fork().
- When standard output is directed to a terminal, it is **line-buffered** by default, with the result that the newline-terminated string written by printf() appears immediately.
- When standard output is directed to a file, it is **block-buffered** by default.
- The output of the write() doesn't appear twice, because write() transfers data directly to a kernel buffer, and this buffer is not duplicated during a fork().

Possible solutions

- We can use fflush() to flush the stdio buffer prior to a fork() call.
- Alternatively, we can use setvbuf() or setbuf() to disable buffering on the stdio stream.
- Instead of calling exit(), the child can call _exit(), so that it doesn't flush stdio buffers.

CHILD PROCESSES

Two techniques used to monitor child processes

- **wait()** system call (and its variants)
- **SIGCHLD** signal (will be discussed later)

wait() system call

- The wait() system call waits for one of the children of the calling process to terminate
- and returns the termination status of that child in the buffer pointed to by status.

```
#include <sys/wait.h>  
  
pid_t wait(int *status);
```

Returns process ID of terminated child, or -1 on error

Possible error

- One possible error is that the calling process has no (previously unwaited-for) children, which is indicated by the errno value **ECHILD**.

```
while ((childPid = wait(NULL)) != -1)
    continue;
if (errno != ECHILD)                      /* An unexpected error... */
    errExit("wait");
```

Creating and waiting for multiple children

```
for (j = 1; j < argc; j++) {      /* Create one child for each argument */
    switch (fork()) {
        case -1:
            errExit("fork");

        case 0:                      /* Child sleeps for a while then exits */
            printf("[%s] child %d started with PID %ld, sleeping %s "
                   "seconds\n", currTime("%T"), j, (long) getpid(), argv[j]);
            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
            _exit(EXIT_SUCCESS);

        default:                     /* Parent just continues around loop */
            break;
    }
}
```

```
numDead = 0;
for (;;) {                                /* Parent waits for each child to exit */
    childPid = wait(NULL);
    if (childPid == -1) {
        if (errno == ECHILD) {
            printf("No more children - bye!\n");
            exit(EXIT_SUCCESS);
        } else {                      /* Some other (unexpected) error */
            errExit("wait");
        }
    }

    numDead++;
    printf("[%s] wait() returned child PID %ld (numDead=%d)\n",
           currTime("%T"), (long) childPid, numDead);
}
```

Output

```
$ ./multi_wait 7 1 4
[13:41:00] child 1 started with PID 21835, sleeping 7 seconds
[13:41:00] child 2 started with PID 21836, sleeping 1 seconds
[13:41:00] child 3 started with PID 21837, sleeping 4 seconds
[13:41:01] wait() returned child PID 21836 (numDead=1)
[13:41:04] wait() returned child PID 21837 (numDead=2)
[13:41:07] wait() returned child PID 21835 (numDead=3)
No more children - bye!
```

Limitations of wait()

- If a parent process has created multiple children, it is not possible to wait() for the completion of a specific child; we can only wait for the next child that terminates.
- If no child has yet terminated, wait() always blocks. Sometimes, it would be preferable to perform a nonblocking wait so that if no child has yet terminated, we obtain an immediate indication of this fact.
- Using wait(), we can find out only about children that have terminated. It is not possible to be notified when a child is stopped by a signal (such as SIGSTOP or SIGTTIN) or when a stopped child is resumed by delivery of a SIGCONT signal.

waitpid() system call

```
#include <sys/wait.h>
```

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

Returns process ID of child, 0 (see text), or -1 on error

- If pid is greater than 0, wait for the child whose process ID equals pid.
- If pid equals 0, wait for any child in the same process group as the caller (parent).
- If pid is less than -1, wait for any child whose process group identifier equals the absolute value of pid.
- If pid equals -1, wait for any child. The call `wait(&status)` is equivalent to the call `waitpid(-1, &status, 0)`.

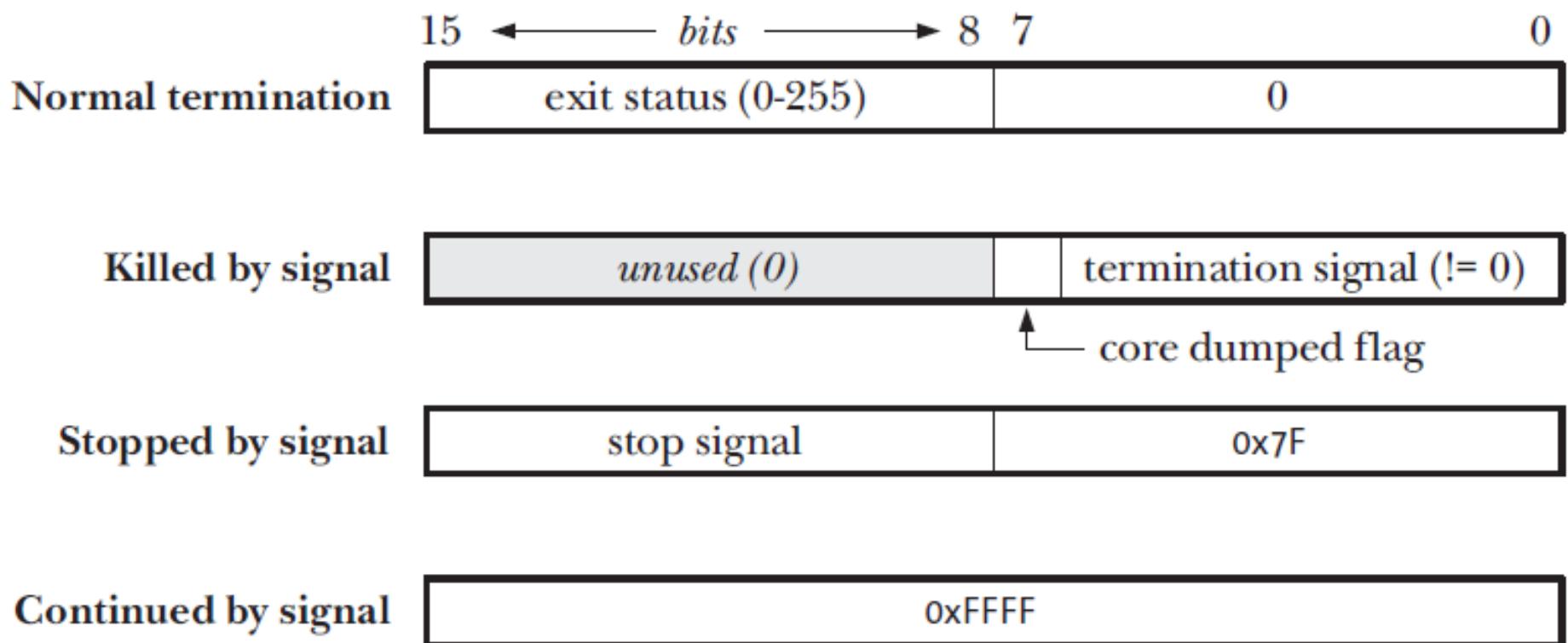
options argument

- **WUNTRACED** - In addition to returning information about terminated children, also return information when a child is stopped by a signal.
- **WCONTINUED** - Also return status information about stopped children that have been resumed by delivery of a SIGCONT signal.
- **WNOHANG** - If no child specified by pid has yet changed state, then return immediately, instead of blocking

status value

- The child terminated by calling `_exit()` (or `exit()`), specifying an integer exit status.
- The child was terminated by the delivery of an unhandled signal.
- The child was stopped by a signal, and `waitpid()` was called with the `WUNTRACED` flag.
- The child was resumed by a `SIGCONT` signal, and `waitpid()` was called with the `WCONTINUED` flag.

Value returned in the status argument of wait() and waitpid()



Set of macros for *status* value

- **WIFEXITED(status)** - This macro returns true if the child process exited normally. In this case, the macro WEXITSTATUS(status) returns the exit status of the child process.
- **WIFSIGNALED(status)** - This macro returns true if the child process was killed by a signal.
- **WIFSTOPPED(status)** - This macro returns true if the child process was stopped by a signal.
- **WIFCONTINUED(status)** - This macro returns true if the child was resumed by delivery of SIGCONT.

Displaying the status value returned by wait()

```
void * Examine a wait() status using the W* macros */  
printWaitStatus(const char *msg, int status)  
{  
    if (msg != NULL)  
        printf("%s", msg);  
  
    if (WIFEXITED(status)) {  
        printf("child exited, status=%d\n", WEXITSTATUS(status));  
  
    } else if (WIFSIGNALED(status)) {  
        printf("child killed by signal %d (%s)",  
               WTERMSIG(status), strsignal(WTERMSIG(status)));  
    }  
}
```

```
    printf("\n");

} else if (WIFSTOPPED(status)) {
    printf("child stopped by signal %d (%s)\n",
           WSTOPSIG(status), strsignal(WSTOPSIG(status)));

#ifndef WIFCONTINUED      /* SUSv3 has this, but older Linux versions and
                           some other UNIX implementations don't */
} else if (WIFCONTINUED(status)) {
    printf("child continued\n");
#endif

} else {                  /* Should never happen */
    printf("what happened to this child? (status=%x)\n",
           (unsigned int) status);
}
}
```

wait3() and wait4() system calls

- Return resource usage information about the terminated child in the structure pointed to by the `rusage` argument.
- This information includes the amount of CPU time used by the process and memory management statistics.

```
#define _BSD_SOURCE      /* Or #define _XOPEN_SOURCE 500 for wait3() */
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

Both return process ID of child, or -1 on error

wait3(), wait4(), and waitpid()

- **wait3()** is equivalent to the following
waitpid() call:

```
waitpid(-1, &status, options);
```

- **wait4()** is equivalent to the following:

```
waitpid(pid, &status, options);
```

Orphans

- **Who becomes the parent of an orphaned child?** The orphaned child is adopted by init, the ancestor of all processes, whose process ID is 1. In other words, after a child's parent terminates, a call to getppid() will return the value 1.

Zombies

- **What happens to a child that terminates before its parent has had a chance to perform a wait()?**
- The point here is that, although the child has finished its work, the parent should still be permitted to perform a `wait()` at some later time to determine how the child terminated. The kernel deals with this situation by turning the child into a zombie.
- This means that most of the resources held by the child are released back to the system to be reused by other processes.
- The only part of the process that remains is an entry in the kernel's process table recording (among other things) the child's process ID, termination status, and resource usage statistics

- A zombie process can't be killed by a signal, not even the (silver bullet) SIGKILL. This ensures that the parent can always eventually perform a wait().
- When the parent does perform a wait(), the kernel removes the zombie.
- If the parent terminates without doing a wait(), then the init process adopts the child and automatically performs a wait(), thus removing the zombie process from the system.
- If a parent creates a child, but fails to perform a wait(), then an entry for the zombie child will be maintained indefinitely in the kernel's process table.
- If a large number of such zombie children are created, they will eventually fill the kernel process table, preventing the creation of new processes.

Creating a zombie child process

```
int
main(int argc, char *argv[])
{
    char cmd[CMD_SIZE];
    pid_t childPid;

    setbuf(stdout, NULL);          /* Disable buffering of stdout */

    printf("Parent PID=%ld\n", (long) getpid());

    switch (childPid = fork()) {
    case -1:
        errExit("fork");

    case 0:      /* Child: immediately exits to become zombie */
        printf("Child (PID=%ld) exiting\n", (long) getpid());
        _exit(EXIT_SUCCESS);
    }
```

```
default: /* Parent */
    sleep(3);           /* Give child a chance to start and exit */
    snprintf(cmd, CMD_SIZE, "ps | grep %s", basename(argv[0]));
    cmd[CMD_SIZE - 1] = '\0'; /* Ensure string is null-terminated */
    system(cmd);         /* View zombie child */

    /* Now send the "sure kill" signal to the zombie */

    if (kill(childPid, SIGKILL) == -1)
        errMsg("kill");
    sleep(3);           /* Give child a chance to react to signal */
    printf("After sending SIGKILL to zombie (PID=%ld):\n", (long) childPid);
    system(cmd);         /* View zombie child again */

    exit(EXIT_SUCCESS);
}
```

Output

```
$ ./make_zombie
Parent PID=1013
Child (PID=1014) exiting
1013 pts/4    00:00:00 make_zombie
1014 pts/4    00:00:00 make_zombie <defunct>
After sending SIGKILL to make_zombie (PID=1014):
1013 pts/4    00:00:00 make_zombie
1014 pts/4    00:00:00 make_zombie <defunct>
```

Output from ps(1)

Output from ps(1)

System programming

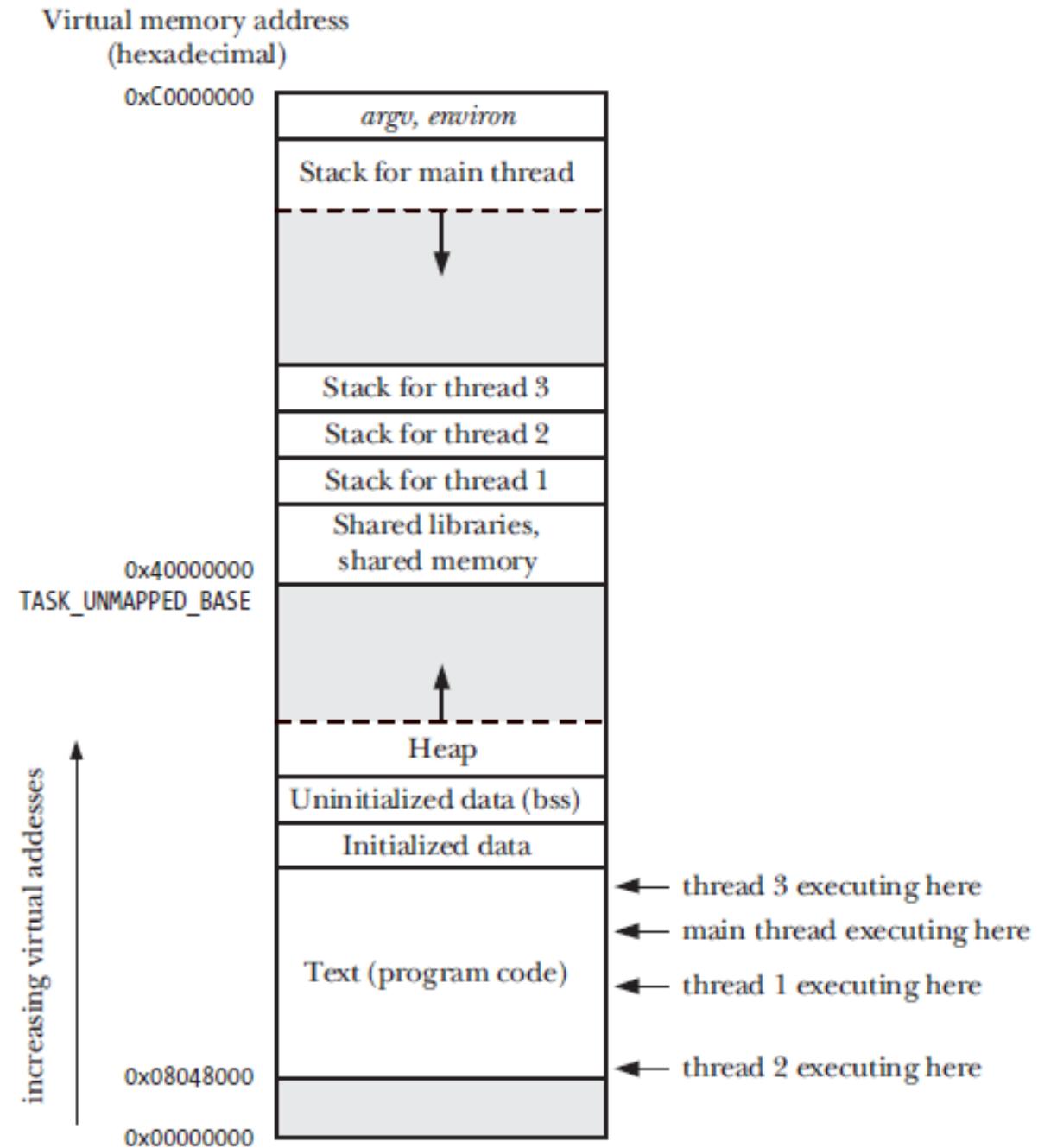
Lecture 4

POSIX Threads

Overview

- Threads are a mechanism that permits an application to perform multiple tasks concurrently.

Single process with multiple threads



Limitations of processes

- It is **difficult to share information** between processes. Since the parent and child don't share memory (other than the read-only text segment), we must use some form of interprocess communication in order to exchange information between processes.
- Process creation with **fork() is relatively expensive**. The need to duplicate various process attributes such as page tables and file descriptor tables means that a fork() call is still time-consuming.

How threads solve these problems

- Sharing information between threads is **easy and fast**. It is just a matter of copying data into shared (global or heap) variables. In order to avoid the problems that can occur when multiple threads try to update the same information, we must employ the synchronization techniques
- Thread creation is faster because many of the attributes that must be duplicated in a child created by `fork()` are instead **shared between threads**. In particular, copy-on-write duplication of pages of memory is not required, nor is duplication of page tables.

Threads **share** a number of attributes

- process ID and parent process ID;
- process credentials (user and group IDs);
- open file descriptors;
- record locks created using `fctl()`;
- signal dispositions;
- etc.

Threads **do not share**

- thread ID;
- signal mask;
- thread-specific data;
- the errno variable;
- stack (local variables and function call linkage information);
- etc.

Pthreads data types

Data type	Description
<i>pthread_t</i>	Thread identifier
<i>pthread_mutex_t</i>	Mutex
<i>pthread_mutexattr_t</i>	Mutex attributes object
<i>pthread_cond_t</i>	Condition variable
<i>pthread_condattr_t</i>	Condition variable attributes object
<i>pthread_key_t</i>	Key for thread-specific data
<i>pthread_once_t</i>	One-time initialization control context
<i>pthread_attr_t</i>	Thread attributes object

Return value from Pthreads functions

- All Pthreads functions return 0 on success or a positive value on failure.
- The failure value is one of the same values that can be placed in errno by traditional UNIX system calls.

Compiling Pthreads programs

- On Linux, programs that use the Pthreads API must be compiled with the `cc -pthread` option.

Thread Creation: pthread_create()

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start)(void *), void *arg);
```

Returns 0 on success, or a positive error number on error

- The new thread commences execution by calling the function identified by **start** with the argument **arg** (i.e., `start(arg)`).
- The thread that calls `pthread_create()` continues execution with the next statement that follows the call.
- The **thread** argument points to a buffer of type `pthread_t` into which the unique identifier for this thread is copied before `pthread_create()` returns. This identifier can be used in later Pthreads calls to refer to the thread.
- The **attr** argument is a pointer to a `pthread_attr_t` object that specifies various attributes for the new thread.

Thread Termination: pthread_exit()

```
include <pthread.h>  
  
void pthread_exit(void *retval);
```

- The thread's start function performs a return specifying a return value for the thread.
- The thread calls **pthread_exit()**.
- The thread is canceled using **pthread_cancel()**.
- Any of the threads calls **exit()**, or the main thread performs a return (in the `main()` function), which causes all threads in the process to terminate immediately.
- The **retval** argument specifies the return value for the thread.

Thread IDs

- Each thread within a process is uniquely identified by a thread ID. This ID is returned to the caller of `pthread_create()`, and a thread can obtain its own ID using `pthread_self()`.

```
include <pthread.h>

pthread_t pthread_self(void);
```

Returns the thread ID of the calling thread

pthread_equal()

- The pthread_equal() function allows us check whether two thread IDs are the same.

```
include <pthread.h>
```

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

Returns nonzero value if *t1* and *t2* are equal, otherwise 0

Joining with a terminated thread

- The `pthread_join()` function waits for the thread identified by `thread` to terminate.

```
include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **retval);
```

Returns 0 on success, or a positive error number on error

pthread_join() for threads and waitpid() for processes

- Threads are peers. Any thread in a process can use `pthread_join()` to join with any other thread in the process.
- For example, if thread A creates thread B, which creates thread C, then it is possible for thread A to join with thread C, or vice versa.
- This differs from the hierarchical relationship between processes. When a parent process creates a child using `fork()`, it is the only process that can `wait()` on that child.

A simple program using Pthreads

```
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    char *s = (char *) arg;
    printf("%s", s);

    return (void *) strlen(s);
}

int
main(int argc, char *argv[])
{
    pthread_t t1;
    void *res;
    int s;

    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n");
    if (s != 0)
        errExitEN(s, "pthread_create");

    printf("Message from main()\n");
    s = pthread_join(t1, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("Thread returned %ld\n", (long) res);

    exit(EXIT_SUCCESS);
}
```

Output

```
$ ./simple_thread
Message from main()
Hello world
Thread returned 12
```

Detaching a thread

- By default, a thread is **joinable**, meaning that when it terminates, another thread can obtain its return status using `pthread_join()`.
- Sometimes, we don't care about the thread's return status; we simply want the system to automatically clean up and remove the thread when it terminates.
- In this case, we can mark the thread as **detached**, by making a call to `pthread_detach()` specifying the thread's identifier in `thread`.

pthread_detach()

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Returns 0 on success, or a positive error number on error

Once a thread has been detached, it is no longer possible to use `pthread_join()` to obtain its return status, and the thread can't be made joinable again.

Creating a thread with the detached attribute

```
pthread_t thr;
pthread_attr_t attr;
int s;

s = pthread_attr_init(&attr);                  /* Assigns default values */
if (s != 0)
    errExitEN(s, "pthread_attr_init");

s = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
if (s != 0)
    errExitEN(s, "pthread_attr_setdetachstate");

s = pthread_create(&thr, &attr, threadFunc, (void *) 1);
if (s != 0)
    errExitEN(s, "pthread_create");

s = pthread_attr_destroy();                    /* No longer needed */
if (s != 0)
    errExitEN(s, "pthread_attr_destroy");
```

Threads vs. Processes

- Sharing data between threads is easy. By contrast, sharing data between processes requires more work (e.g., creating a shared memory segment or using a pipe).
- Thread creation is faster than process creation; context-switch time may be lower for threads than for processes.

Threads vs. Processes

- When programming with threads, we need to ensure that the functions we call are thread-safe or are called in a thread-safe manner.
- A bug in one thread (e.g., modifying memory via an incorrect pointer) can damage all of the threads in the process, since they share the same address space and other attributes. By contrast, processes are more isolated from one another.
- Each thread is competing for use of the finite virtual address space of the host process. In particular, each thread's stack and thread-specific data (or thread local storage) consumes a part of the process virtual address space, which is consequently unavailable for other threads.

System programming

Lecture 4

Thread synchronization

Critical section

- The term **critical section** is used to refer to a section of code that accesses a shared resource and whose execution should be **atomic**;
- that is, its execution should not be interrupted by another thread that simultaneously accesses the same shared resource.

Incorrectly incrementing a global variable from two threads

```
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;

static void *                         /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j;

    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }

    return NULL;
}
```

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

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

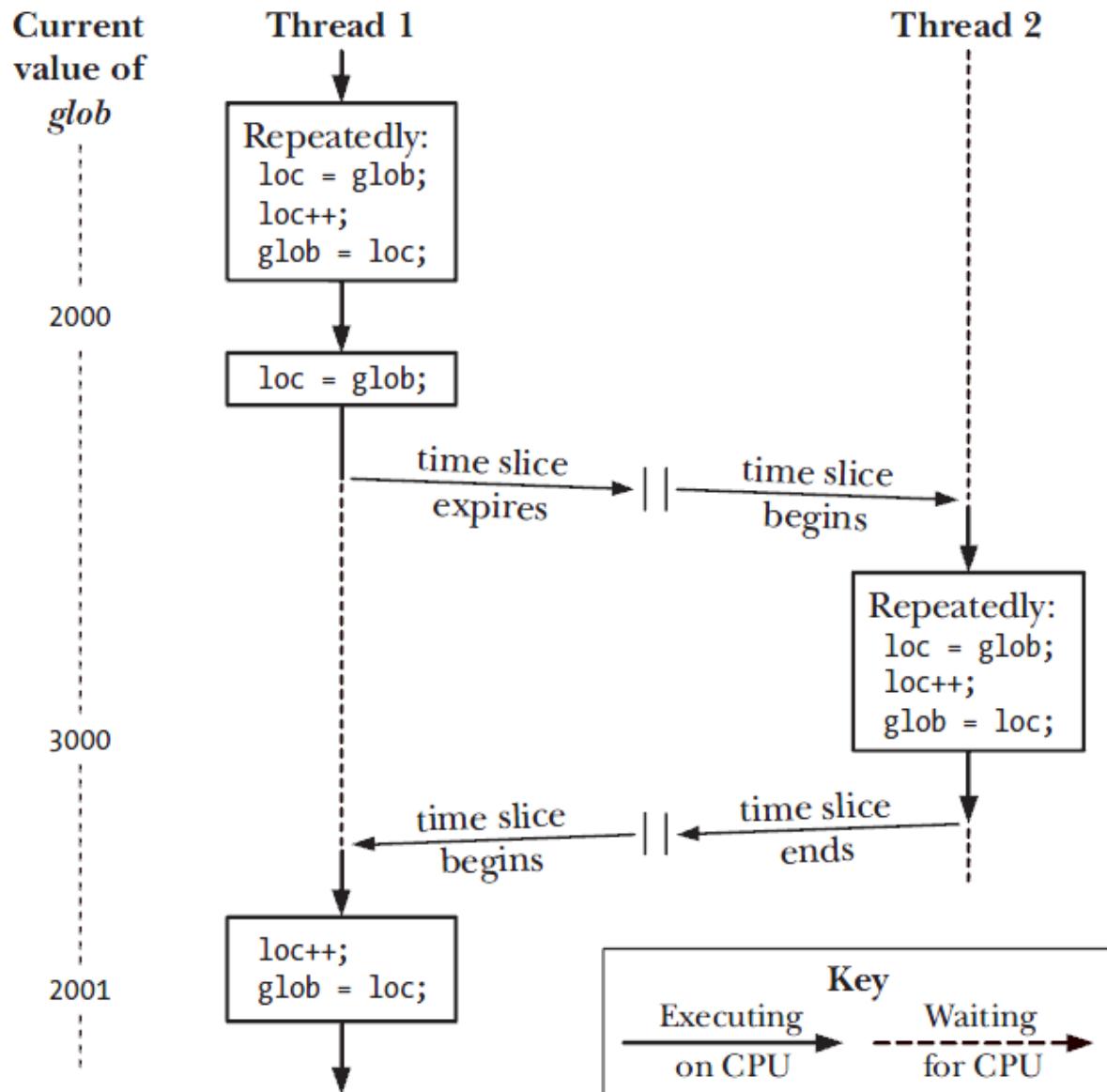
    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

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

Some comments

- This program creates two threads, each of which executes the same function.
- The function executes a loop that repeatedly increments a global variable, **glob**, by copying **glob** into the local variable **loc**, incrementing **loc**, and copying **loc** back to **glob**.
- Since **loc** is an automatic variable allocated on the per-thread stack, each thread has its own copy of this variable.

Two threads incrementing a global variable without synchronization



Nondeterministic behavior

```
$ ./thread_incr 10000000
glob = 10880429
$ ./thread_incr 10000000
glob = 13493953
```

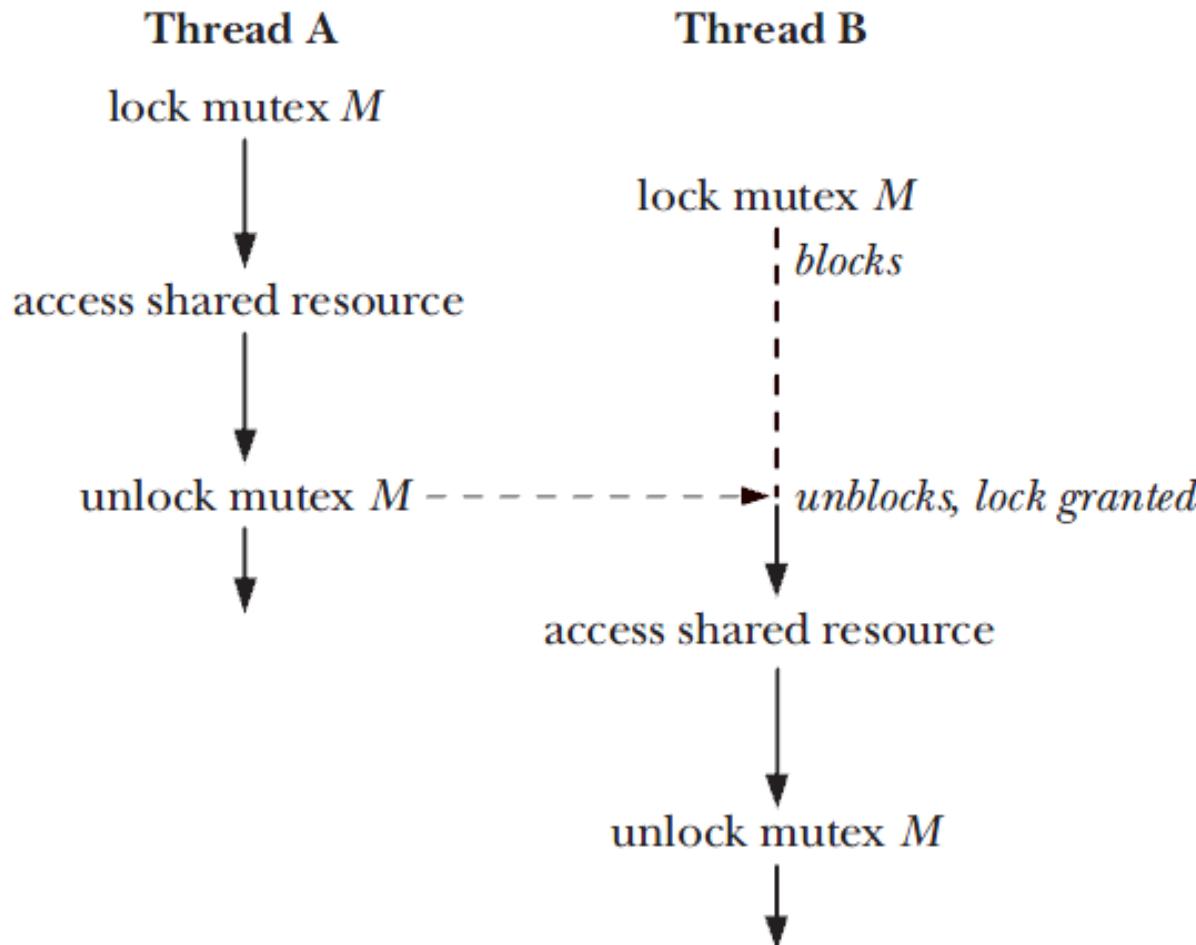
Mutex

- Mutexes can be used to ensure atomic access to any shared resource, but protecting shared variables is the most common use.
- A mutex has two states: **locked** and **unlocked**. At any moment, at most one thread may hold the lock on a mutex.
- When a thread locks a mutex, it becomes the owner of that mutex. Only the mutex owner can unlock the mutex.

Protocol for accessing a resource

- lock the mutex for the shared resource;
- access the shared resource;
- unlock the mutex.

Using a mutex to protect a critical section



Statically allocated mutexes

- A mutex is a variable of the type **`pthread_mutex_t`**.
- `pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;`

Locking and unlocking a mutex

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Both return 0 on success, or a positive error number on error

- If the mutex is **currently unlocked**, this call locks the mutex and returns immediately.
- If the mutex is **currently locked** by another thread, then `pthread_mutex_lock()` blocks until the mutex is unlocked, at which point it locks the mutex and returns.
- The `pthread_mutex_unlock()` function unlocks a mutex previously locked by the calling thread. It is an error to unlock a mutex that is not currently locked, or to unlock a mutex that is locked by another thread.

Using a mutex to protect access to a global variable

```
#include <pthread.h>
#include "tlpi_hdr.h"

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

static void *                  /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j, s;

    for (j = 0; j < loops; j++) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");
        loc = glob;
        glob++;
        if (loc != glob)
            errExitEN(s, "glob inconsistency");
        pthread_mutex_unlock(&mtx);
    }
}
```

```
    loc = glob;
    loc++;
    glob = loc;

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");
}

return NULL;
}
```

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

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

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

pthread_mutex_trylock()

- The pthread_mutex_trylock() function is the same as pthread_mutex_lock(), except that if the mutex is currently locked, pthread_mutex_trylock() fails, returning the error EBUSY.

pthread_mutex_timedlock()

- The pthread_mutex_timedlock() function is the same as pthread_mutex_lock(), except that the caller can specify an additional argument, abstime, that places a limit on the time that the thread will sleep while waiting to acquire the mutex.
- If the time interval specified by its abstime argument expires without the caller becoming the owner of the mutex, pthread_mutex_timedlock() returns the error ETIMEDOUT.

A deadlock when two threads lock two mutexes

Thread A

1. *pthread_mutex_lock(mutex1);*
2. *pthread_mutex_lock(mutex2);*
blocks

Thread B

1. *pthread_mutex_lock(mutex2);*
2. *pthread_mutex_lock(mutex1);*
blocks

Signaling changes of state: condition variables

- A **condition variable** allows one thread to inform other threads about changes in the state of a shared resource and allows the other threads to wait (block) for such notification.

Example: Producer-Consumer (without condition variables)

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;  
  
static int avail = 0;
```

Producer

```
/* Code to produce a unit omitted */

s = pthread_mutex_lock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_lock");

avail++; /* Let consumer know another unit is available */

s = pthread_mutex_unlock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_unlock");
```

Consumer

```
for (;;) {
    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    while (avail > 0) {          /* Consume all available units */
        /* Do something with produced unit */
        avail--;
    }

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");
}
```

Some comments

- This code works, but it wastes CPU time, because the main thread continually loops, checking the state of the variable **avail**.
- Condition variable allows a thread to sleep (wait) until another thread notifies (signals) it that it must do something.

Condition variable and mutex

- A condition variable is always used in conjunction with a mutex.
- The mutex provides mutual exclusion for accessing the shared variable, while the condition variable is used to signal changes in the variable's state.

Statically allocated condition variables:

`pthread_cond_t`

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Signaling and waiting on condition variables

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

All return 0 on success, or a positive error number on error

Using a condition variable in the producer-consumer example

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;  
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
  
static int avail = 0;
```

pthread_cond_signal() and pthread_cond_broadcast()

- The pthread_cond_signal() and pthread_cond_broadcast() functions both signal the condition variable specified by cond.

pthread_cond_wait()

- The pthread_cond_wait() function blocks a thread until the condition variable cond is signaled.
- pthread_cond_wait() performs the following steps:
 - unlock the mutex specified by mutex;
 - block the calling thread until another thread signals the condition variable cond; and
 - relock mutex.

Main idea

1. The thread locks the mutex in preparation for checking the state of the shared variable.
2. The state of the shared variable is checked.
3. If the shared variable is not in the desired state, then the thread must unlock the mutex (so that other threads can access the shared variable) before it goes to sleep on the condition variable.
4. When the thread is reawakened because the condition variable has been signaled, the mutex must once more be locked, since, typically, the thread then immediately accesses the shared variable.

Producer (with cond. var.)

```
s = pthread_mutex_lock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_lock");

avail++;                      /* Let consumer know another unit is available */

s = pthread_mutex_unlock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_unlock");

s = pthread_cond_signal(&cond);      /* Wake sleeping consumer */
if (s != 0)
    errExitEN(s, "pthread_cond_signal");
```

Consumer (with cond. var.)

```
for (;;) {
    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    while (avail == 0) /* Wait for something to consume */
        s = pthread_cond_wait(&cond, &mtx);
        if (s != 0)
            errExitEN(s, "pthread_cond_wait");
}

while (avail > 0) /* Consume all available units */
    /* Do something with produced unit */
    avail--;
}

s = pthread_mutex_unlock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_unlock");

/* Perhaps do other work here that doesn't require mutex lock */
}
```

System programming

Lecture 5

Signals

Signal

- A **signal** is a notification to a process that an event has occurred.
- Kernel → Process
- Process → Process

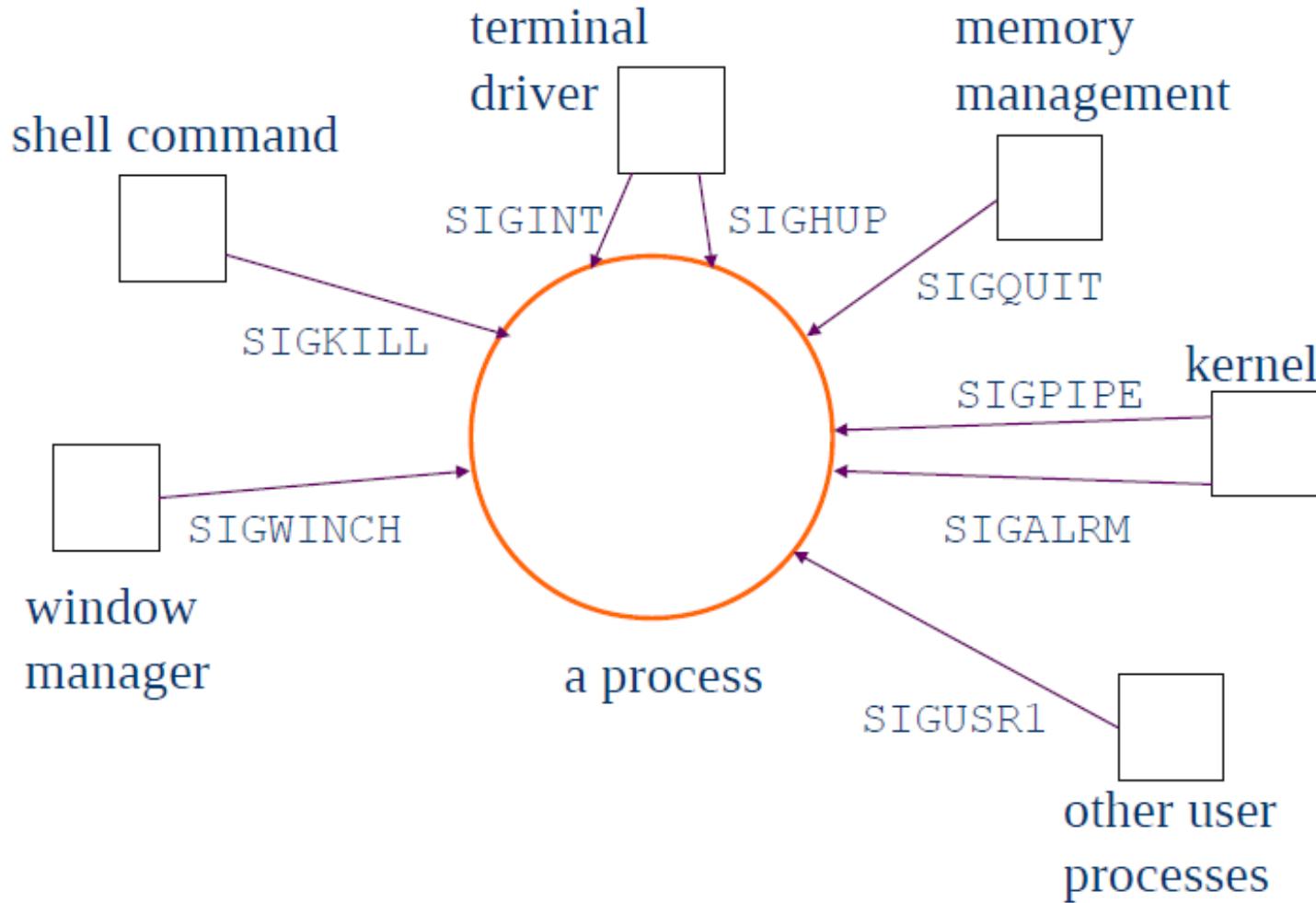
Examples of events

- **Hardware exception:**
 - executing a malformed machine-language instruction,
 - dividing by 0,
 - referencing a part of memory that is inaccessible,
 - etc.
- Terminal **special characters:**
 - interrupt character (usually Control-C)
 - suspend character (usually Control-Z)
 - etc.
- **Software event:**
 - input became available on a file descriptor,
 - the terminal window was resized,
 - a timer went off,
 - the process's CPU time limit was exceeded,
 - a child of this process terminated
 - etc.

Categories of signals

- Standard (are used by the kernel to notify processes and events, 1-31)
- Realtime

Signal sources



POSIX predefined signals

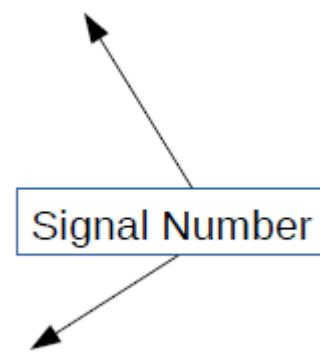
- SIGALRM: Alarm timer time-out. Generated by alarm() API.
- SIGABRT: Abort process execution. Generated by abort() API.
- SIGFPE: Illegal mathematical operation.
- SIGHUP: Controlling terminal hang-up.
- SIGILL: Execution of an illegal machine instruction.
- SIGINT: Process interruption. Can be generated by <Delete> or <ctrl_C> keys.
- SIGKILL: Sure kill a process. Can be generated by
 - “kill -9 <process_id>“ command.
- SIGPIPE: Illegal write to a pipe.
- SIGQUIT: Process quit. Generated by <crtl_\> keys.
- SIGSEGV: Segmentation fault. generated by de-referencing a NULL pointer.

POSIX predefined signals

- SIGTERM: process termination. Can be generated by
 - “kill <process_id>” command.
- SIGUSR1: Reserved to be defined by user.
- SIGUSR2: Reserved to be defined by user.
- SIGCHLD: Sent to a parent process when its child process has terminated.
- SIGCONT: Resume execution of a stopped process.
- SIGSTOP: Stop a process execution.
- SIGTTIN: Stop a background process when it tries to read from its controlling terminal.
- SIGTSTP: Stop a process execution by the control_Z keys.
- SIGTTOOUT: Stop a background process when it tries to write to its controlling terminal.

Example of signals

- **User types Ctrl-c**
 - Event gains attention of OS
 - OS stops the application process immediately, sending it a 2/SIGINT signal
 - Signal handler for 2/SIGINT signal executes to completion
 - Default signal handler for 2/SIGINT signal exits process
 - **Process makes illegal memory reference**
 - Event gains attention of OS
 - OS stops application process immediately, sending it a 11/SIGSEGV signal
 - Signal handler for 11/SIGSEGV signal executes to completion
 - Default signal handler for 11/SIGSEGV signal prints “segmentation fault” and exits process



Send signals via commands

kill Command

–kill signal

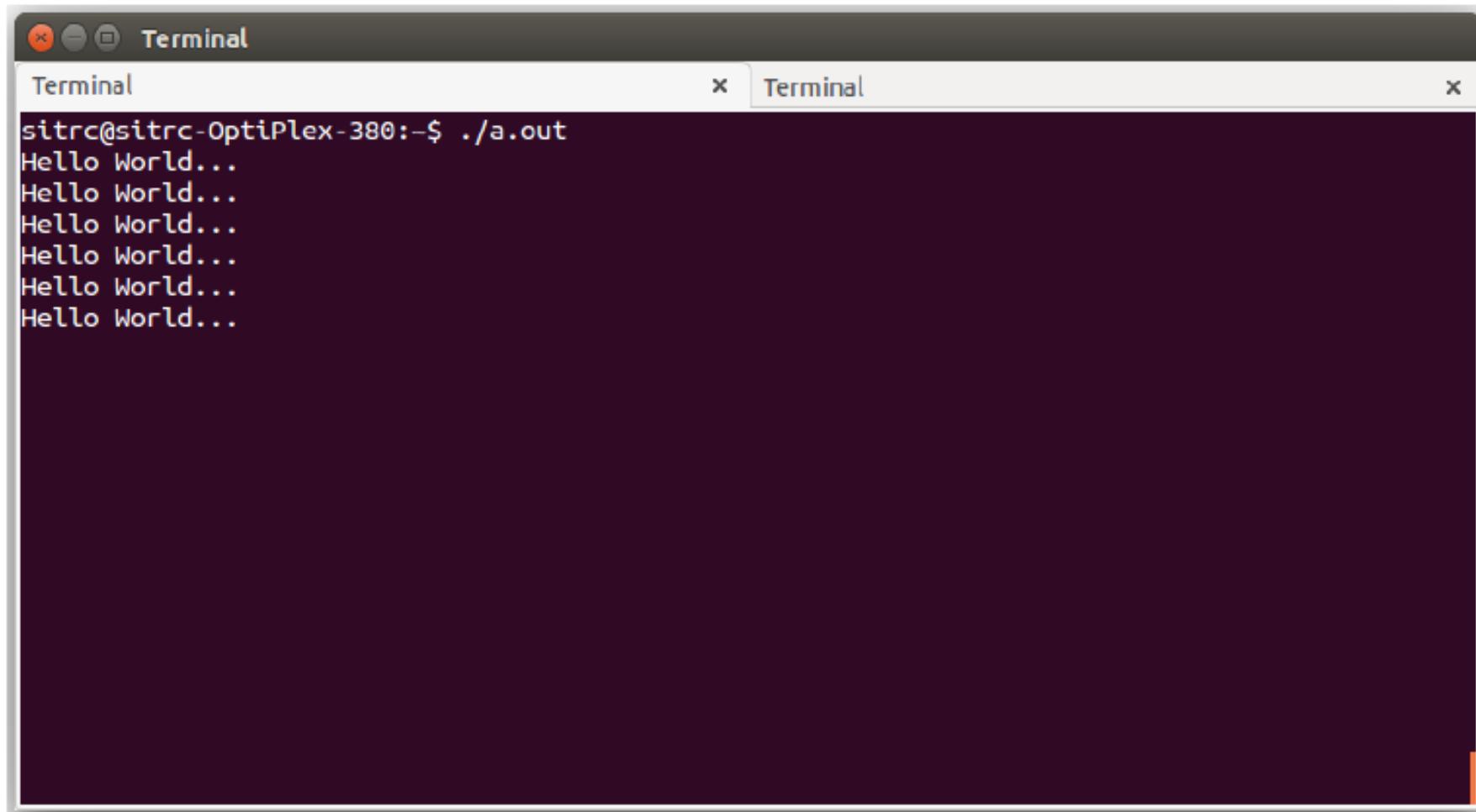
pid

- Send a signal of type signal to the process with id pid
 - Can specify either signal type name (-SIGINT) or number (-2)
- No signal type name or number specified => sends 15/SIGTERM signal*
- Default 15/SIGTERM handler exits process
 - Better command name would be sendsig
 - Examples
- kill –2 1234**
- kill SIGINT**
- 1234**
- Same as pressing Ctrl-c if process 1234 is running in foreground

Demonstration

```
#include<stdio.h>
int main()
{
    while(1)
        printf("Hello World...\\n");
    return 0;
}
```

Output

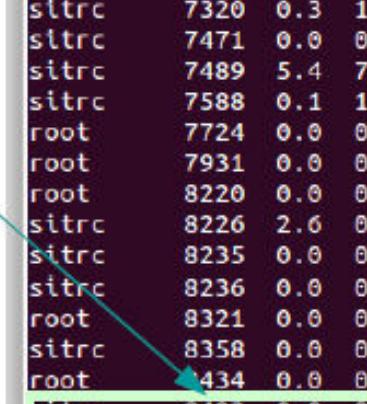


A screenshot of a terminal window titled "Terminal". The window has a dark background and contains the following text:

```
sitrc@sitrc-OptiPlex-380:~/Desktop$ ./a.out
Hello World...
```

Output

- Go to new terminal and check the process list
(ps -aux)

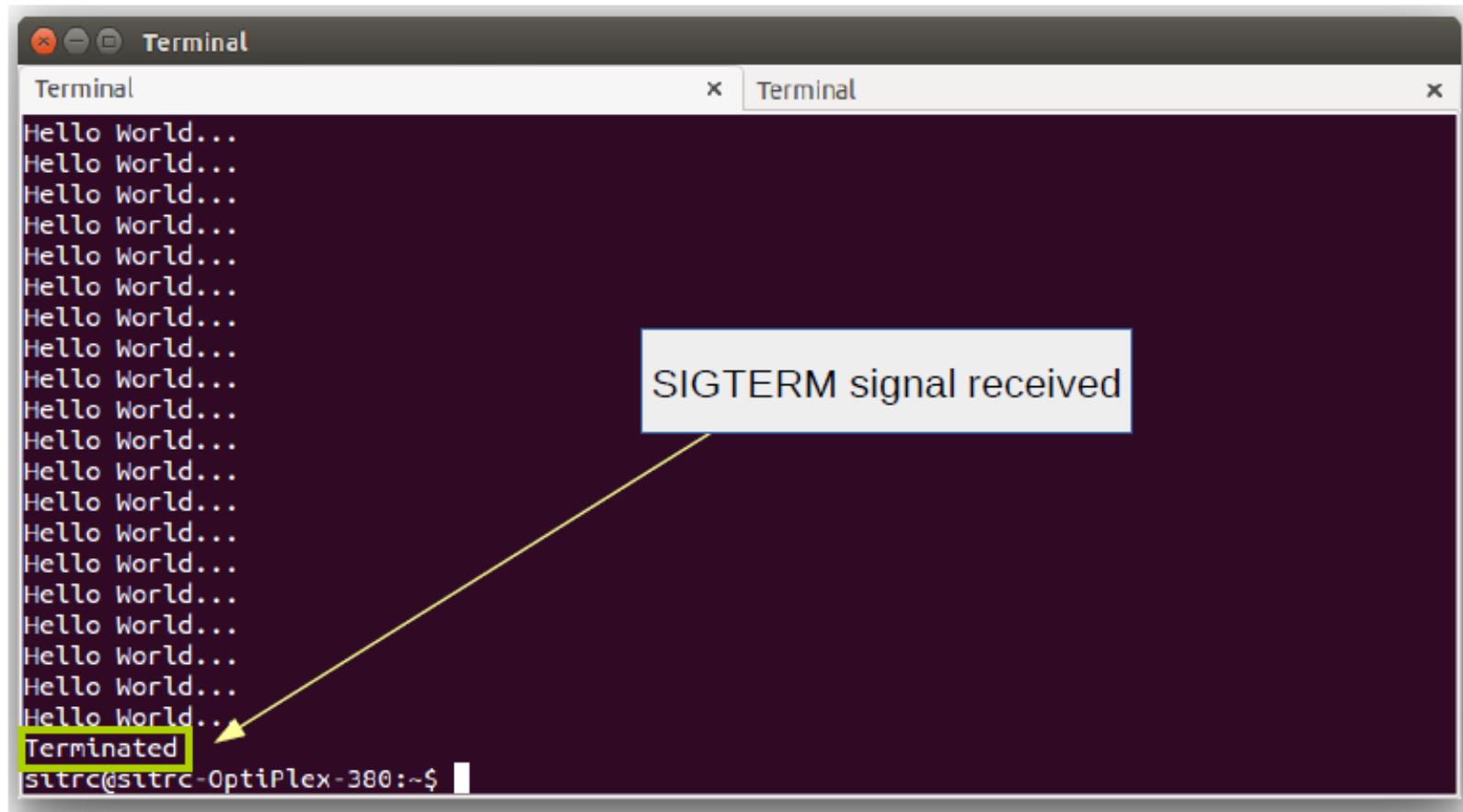


```
Terminal
pid
sitrc 5961 0.0 0.3 76788 6088 ?
sitrc 5976 0.0 0.5 60084 11184 ?
root 7270 0.0 0.0 0 0 ?
root 7271 0.0 0.0 0 0 ?
root 7273 0.0 0.0 0 0 ?
root 7274 0.0 0.0 0 0 ?
sitrc 7320 0.3 1.2 248892 25184 ?
sitrc 7471 0.0 0.2 35400 5380 ?
sitrc 7489 5.4 7.5 512800 153396 ?
sitrc 7588 0.1 1.2 275096 26024 ?
root 7724 0.0 0.0 3348 1160 ?
root 7931 0.0 0.0 0 0 ?
root 8220 0.0 0.0 0 0 ?
sitrc 8226 2.6 0.9 230648 20180 ?
sitrc 8235 0.0 0.0 2428 1332 ?
sitrc 8236 0.0 0.0 5728 1788 pts/1
root 8321 0.0 0.0 3092 1748 ?
sitrc 8358 0.0 0.1 5732 2692 pts/5
root 434 0.0 0.0 0 0 ?
sitrc 8493 0.0 0.0 2036 428 pts/1
sitrc 8558 0.0 0.1 5232 2248 pts/5
sitrc@sitrc-OptiPlex-380:~$
```

The screenshot shows a terminal window titled "Terminal" displaying the output of the "ps -aux" command. The output lists various processes running on the system, including system daemons like kworker and processes like chrome and libreoffice. A green arrow points from a purple box labeled "pid" to the first column of the process list, which contains the process IDs (PIDs). The last two lines of the output are highlighted with a green box: "sitrc 8493 0.0 0.0 2036 428 pts/1" and "sitrc 8558 0.0 0.1 5232 2248 pts/5". The prompt at the bottom of the terminal is "sitrc@sitrc-OptiPlex-380:~\$".

Kill the process

- kill 8493



Killing process by different signals

- kill -SIGSEGV 8493

The screenshot shows a terminal window with two tabs, both titled "Terminal". The left tab contains the output of a "Hello World" program, with "Hello World..." printed multiple times. The right tab is active and shows the same output. A yellow arrow points from the text "Segmentation fault (core dumped)" at the bottom of the right tab to a callout box containing the text "SIGSEGV signal received".

```
Hello World...
Segmentation fault (core dumped)
sitrc@sitrc-OptiPlex-380:~$
```

Signal concepts

Signals are defined in <signal.h>

- **man 7 signal for complete list of signals** and their numeric values.
- **kill -l for full list of signals on a system.**
- 64 signals. The first 32 are traditional signals, the rest are for real time applications

Some useful terms

- A signal is said to be **generated** by some event.
- Once generated, a signal is later **delivered** to a process, which then takes some action in response to the signal.
- Between the time it is generated and the time it is delivered, a signal is said to be **pending**.

Signal mask

- Sometimes we need to ensure that a segment of code is not interrupted by the delivery of a signal.
- To do this, we can add a signal to the process's **signal mask**—a set of signals whose delivery is currently **blocked**.

Different (default) actions can be taken upon delivery of a signal

- The signal is **ignored** and has no effect on the process.
- The process is **terminated** (killed). (Abnormal termination)
- A **core dump file** is generated, and the process is terminated. (For debugging purposes)
- The process is **stopped**—execution of the process is suspended.
- Execution of the process is **resumed** after previously being stopped.

Disposition

- Changing the action that occurs when the signal is delivered instead of accepting the default for a particular signal.

Dispositions

- The default action should occur.
- The signal is ignored.
- A signal handler is executed.

Signal handler

- A signal handler is a function, written by the programmer, that performs appropriate tasks in response to the delivery of a signal.

Linux signals

Name	Signal number	Description	SUSv3	Default
SIGABRT	6	Abort process	•	core
SIGALRM	14	Real-time timer expired	•	term
SIGBUS	7 (SAMP=10)	Memory access error	•	core
SIGCHLD	17 (SA=20, MP=18)	Child terminated or stopped	•	ignore
SIGCONT	18 (SA=19, M=25, P=26)	Continue if stopped	•	cont
SIGEMT	undef (SAMP=7)	Hardware fault		term
SIGFPE	8	Arithmetic exception	•	core
SIGHUP	1	Hangup	•	term
SIGILL	4	Illegal instruction	•	core
SIGINT	2	Terminal interrupt	•	term
SIGIO / SIGPOLL	29 (SA=23, MP=22)	I/O possible	•	term

Linux signals

SIGKILL	9	Sure kill	•	term
SIGPIPE	13	Broken pipe	•	term
SIGPROF	27 (M=29, P=21)	Profiling timer expired	•	term
SIGPWR	30 (SA=29, MP=19)	Power about to fail	•	term
SIGQUIT	3	Terminal quit	•	core
SIGSEGV	11	Invalid memory reference	•	core
SIGSTKFLT	16 (SAM=undef, P=36)	Stack fault on coprocessor	•	term
SIGSTOP	19 (SA=17, M=23, P=24)	Sure stop	•	stop
SIGSYS	31 (SAMP=12)	Invalid system call	•	core
SIGTERM	15	Terminate process	•	term
SIGTRAP	5	Trace/breakpoint trap	•	core

Linux signals

SIGTSTP	20 (SA=18, M=24, P=25)	Terminal stop	•	stop
SIGTTIN	21 (M=26, P=27)	Terminal read from BG	•	stop
SIGTTOU	22 (M=27, P=28)	Terminal write from BG	•	stop
SIGURG	23 (SA=16, M=21, P=29)	Urgent data on socket	•	ignore
SIGUSR1	10 (SA=30, MP=16)	User-defined signal 1	•	term
SIGUSR2	12 (SA=31, MP=17)	User-defined signal 2	•	term
SIGVTALRM	26 (M=28, P=20)	Virtual timer expired	•	term
SIGWINCH	28 (M=20, P=23)	Terminal window size change	•	ignore
SIGXCPU	24 (M=30, P=33)	CPU time limit exceeded	•	core
SIGXFSZ	25 (M=31, P=34)	File size limit exceeded	•	core

Changing Signal Dispositions: signal()

```
#include <signal.h>

void ( *signal(int sig, void (*handler)(int)) ) (int);
```

Returns previous signal disposition on success, or SIG_ERR on error

```
void
handler(int sig)
{
    /* Code for the handler */
}
```

`signal()`

- The first argument, **sig**, identifies the signal whose disposition we wish to change.
- The second argument, **handler**, is the address of the function that should be called when this signal is delivered.
- The **return value** of `signal()` is the previous disposition of the signal.

Establishing a handler for a signal, and resetting the disposition of the signal to whatever it was previously

```
void (*oldHandler)(int);

oldHandler = signal(SIGINT, newHandler);
if (oldHandler == SIG_ERR)
    errExit("signal");

/* Do something else here. During this time, if SIGINT is
   delivered, newHandler will be used to handle the signal. */

if (signal(SIGINT, oldHandler) == SIG_ERR)
    errExit("signal");
```

SIG_DFL and **SIG_IGN**

- **SIG_DFL** - Reset the disposition of the signal to its default.
- **SIG_IGN** - Ignore the signal.

Example 1

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ohh(int sig)
{
    printf("Ohh! - I got signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}
int main()
{
    (void) signal(SIGINT, ohh);
    while(1)
    {
        printf("Hello World!\n");
        sleep(1);
    }
    return 0;
}
```

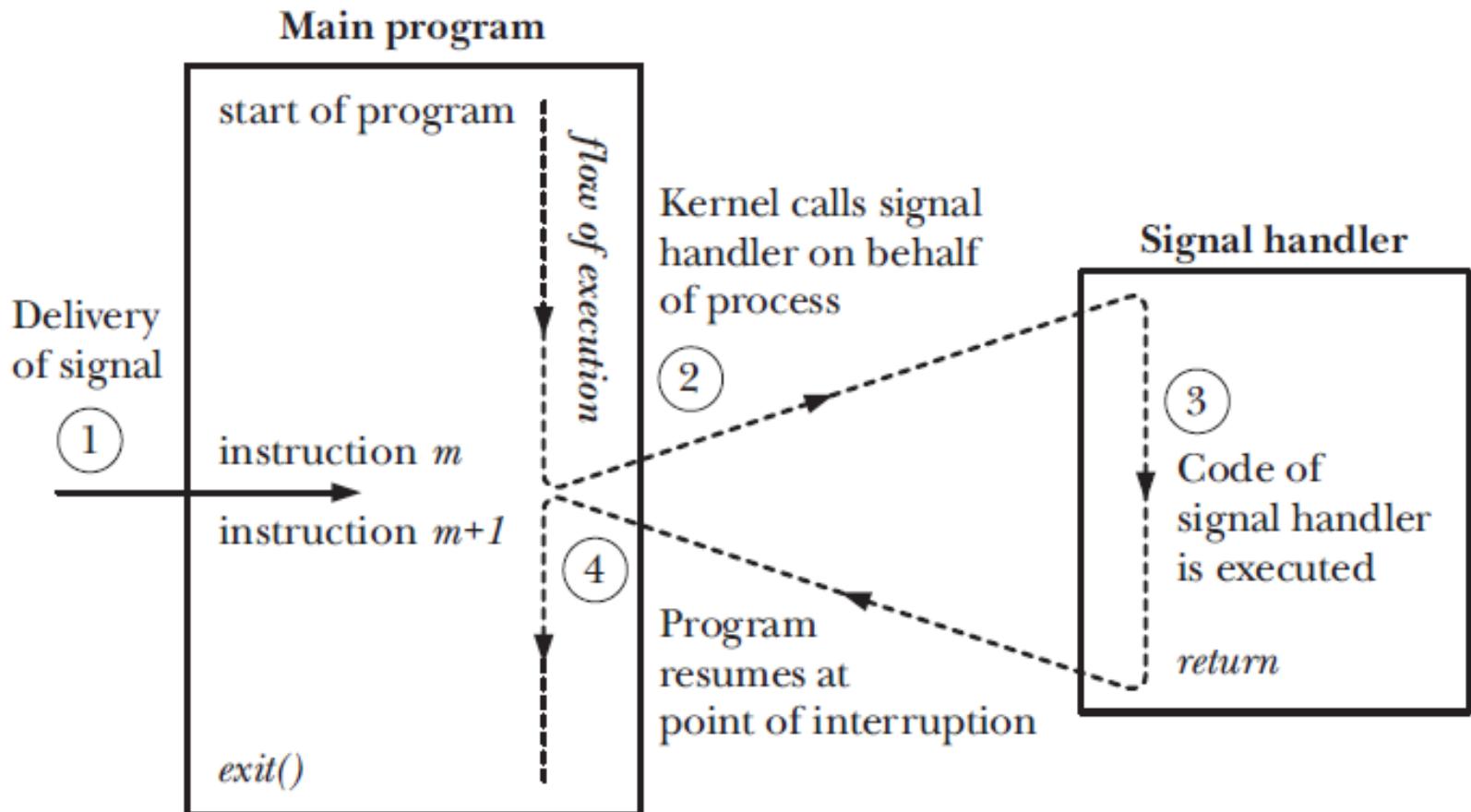
Example 2

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void error(int sig)
{
    printf("Ohh! its a floating point error...\n");
    (void) signal(SIGFPE, SIG_DFL);
}
int main()
{
    (void) signal(SIGFPE, error);
    int a = 12, b = 0, result;
    result = a / b;
    printf("Result is : %d\n", result);
    return 0;
}
```

Signal handlers

- A **signal handler** is a function that is called when a specified signal is delivered to a process.

Signal delivery and handler execution



Installing a handler for SIGINT (example)

```
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigHandler(int sig)
{
    printf("Ouch!\n");
}

int
main(int argc, char *argv[])
{
    int j;

    if (signal(SIGINT, sigHandler) == SIG_ERR)
        errExit("signal");

    for (j = 0; ; j++) {
        printf("%d\n", j);
        sleep(3);
    }
}
```

Output

```
$ ./ouch
```

```
0
```

Main program loops, displaying successive integers

Type Control-C

```
Ouch!
```

*Signal handler is executed, and returns
Control has returned to main program*

```
1
```

```
2
```

Type Control-C again

```
Ouch!
```

```
3
```

Type Control-\ (the terminal quit character)

Quit (core dumped)

Establishing the same handler for two different signals (example)

```
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigHandler(int sig)
{
    static int count = 0;

    if (sig == SIGINT) {
        count++;
        printf("Caught SIGINT (%d)\n", count);
        return;                  /* Resume execution at point of interruption */
    }

    /* Must be SIGQUIT - print a message and terminate the process */

    printf("Caught SIGQUIT - that's all folks!\n");
    exit(EXIT_SUCCESS);
}
```


Output

```
$ ./intquit
```

Type Control-C

Caught SIGINT (1)

Type Control-C again

Caught SIGINT (2)

and again

Caught SIGINT (3)

Type Control-

Caught SIGQUIT - that's all folks!

Sending signals: kill()

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

Returns 0 on success, or -1 on error

pid

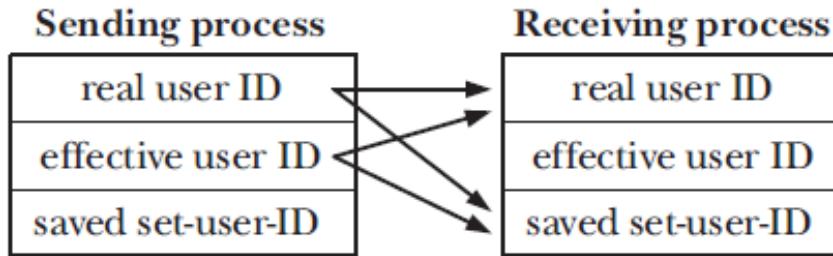
- If pid is greater than 0, the signal is sent to the process with the process ID specified by pid.
- If pid equals 0, the signal is sent to every process in the same process group as the calling process, including the calling process itself.
- If pid is less than –1, the signal is sent to all of the processes in the process group whose ID equals the absolute value of pid.
- If pid equals –1, the signal is sent to every process for which the calling process has permission to send a signal, except init (process ID 1) and the calling process.

Process permissions

- A privileged (CAP_KILL) process may send a signal to any process.
- The init process (process ID 1), which runs with user and group of root, is a special case. It can be sent only signals for which it has a handler installed.

Process permissions

An unprivileged process can send a signal to another process if the real or effective user ID of the sending process matches the real user ID or saved setuser-ID of the receiving process



→ indicates that if IDs match,
then sender has permission
to send a signal to receiver

- The SIGCONT signal is treated specially. An unprivileged process may send this signal to any other process in the same session, regardless of user ID checks.

Checking for the Existence of a Process

- If the sig argument is specified as 0 (the so-called **null signal**), then no signal is sent.
- Instead, kill() performs error checking to see if the process can be signaled.
- If sending a null signal fails with the error **ESRCH**, then we know the process doesn't exist.
- If the call fails with the error **EPERM** (meaning the process exists, but we don't have permission to send a signal to it) or succeeds (meaning we do have permission to send a signal to the process), then we know that the process exists.

Using the kill() system call (example)

```
int
main(int argc, char *argv[])
{
    int s, sig;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sig-num pid\n", argv[0]);

    sig = getInt(argv[2], 0, "sig-num");

    s = kill(getLong(argv[1], 0, "pid"), sig);

    if (sig != 0) {
        if (s == -1)
            errExit("kill");
```

```
    } else {                                /* Null signal: process existence check */
        if (s == 0) {
            printf("Process exists and we can send it a signal\n");
        } else {
            if (errno == EPERM)
                printf("Process exists, but we don't have "
                        "permission to send it a signal\n");
            else if (errno == ESRCH)
                printf("Process does not exist\n");
            else
                errExit("kill");
        }
    }

    exit(EXIT_SUCCESS);
}
```

Sending a signal to itself: raise()

```
#include <signal.h>  
  
int raise(int sig);
```

Returns 0 on success, or nonzero on error

- In a single-threaded program, a call to `raise()` is equivalent to the following call to `kill()`:

```
kill (getpid() ,  sig) ;
```

- On a system that supports threads, `raise(sig)` is implemented as:

```
pthread_kill (pthread_self() ,  sig)
```

Displaying Signal Descriptions

- `char *strsignal(int sig);`
- `void psignal(int sig, const char *msg);`

Signal Sets

- Multiple signals are represented using a data structure called a signal set, provided by the system data type `sigset_t`.

sigemptyset() and sigfillset() functions

- The **sigemptyset()** function initializes a signal set to contain no members.
- The **sigfillset()** function initializes a set to contain all signals (including all realtime signals).

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
```

Both return 0 on success, or -1 on error

sigaddset() and sigdelset() functions

- After initialization, individual signals can be added to a set using **sigaddset()** and removed using **sigdelset()**.

```
#include <signal.h>

int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
```

Both return 0 on success, or -1 on error

sigismember() function

- The **sigismember()** function is used to test for membership of a set.

```
#include <signal.h>

int sigismember(const sigset_t *set, int sig);
```

Returns 1 if *sig* is a member of *set*, otherwise 0

Print list of signals within a signal set

```
void /* Print list of signals within a signal set */
printSigset(FILE *of, const char *prefix, const sigset_t *sigset)
{
    int sig, cnt;

    cnt = 0;
    for (sig = 1; sig < NSIG; sig++) {
        if (sigismember(sigset, sig)) {
            cnt++;
            fprintf(of, "%s%d (%s)\n", prefix, sig, strsignal(sig));
        }
    }

    if (cnt == 0)
        fprintf(of, "%s<empty signal set>\n", prefix);
}
```

Print mask of blocked signals for this process

```
int /* Print mask of blocked signals for this process */
printSigMask(FILE *of, const char *msg)
{
    sigset_t currMask;

    if (msg != NULL)
        fprintf(of, "%s", msg);

    if (sigprocmask(SIG_BLOCK, NULL, &currMask) == -1)
        return -1;

    printSigset(of, "\t\t", &currMask);

    return 0;
}
```

Print signals currently pending for this process

```
int /* Print signals currently pending for this process */
printPendingSigs(FILE *of, const char *msg)
{
    sigset_t pendingSigs;

    if (msg != NULL)
        fprintf(of, "%s", msg);

    if (sigpending(&pendingSigs) == -1)
        return -1;

    printSigset(of, "\t\t", &pendingSigs);

    return 0;
}
```

The Signal Mask

- For each process, the kernel maintains a **signal mask**—a set of signals whose delivery to the process is currently blocked.
- If a signal that is blocked is sent to a process, delivery of that signal is delayed until it is unblocked by being removed from the process signal mask.

Adding a signal to the signal mask

- When a signal handler is invoked, the signal that caused its invocation can be automatically added to the signal mask.
- When a signal handler is established with `sigaction()`, it is possible to specify an additional set of signals that are to be blocked when the handler is invoked.
- The `sigprocmask()` system call can be used at any time to explicitly add signals to, and remove signals from, the signal mask.

sigprocmask()

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Returns 0 on success, or -1 on error

The **how** argument determines the changes that `sigprocmask()` makes to the signal mask:

- **SIG_BLOCK** - The signals specified in the signal set pointed to by `set` are added to the signal mask.
- **SIG_UNBLOCK** - The signals in the signal set pointed to by `set` are removed from the signal mask.
- **SIG_SETMASK** - The signal set pointed to by `set` is assigned to the signal mask.

Temporarily blocking delivery of a signal

```
sigset_t blockSet, prevMask;

/* Initialize a signal set to contain SIGINT */

sigemptyset(&blockSet);
sigaddset(&blockSet, SIGINT);

/* Block SIGINT, save previous signal mask */

if (sigprocmask(SIG_BLOCK, &blockSet, &prevMask) == -1)
    errExit("sigprocmask1");

/* ... Code that should not be interrupted by SIGINT ... */

/* Restore previous signal mask, unblocking SIGINT */

if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
    errExit("sigprocmask2");
```

Pending Signals

- If a process receives a signal that it is currently blocking, that signal is added to the process's set of pending signals.
- When (and if) the signal is later unblocked, it is then delivered to the process.

sigpending()

- To determine which signals are pending for a process, we can call `sigpending()`.

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

Returns 0 on success, or -1 on error

Changing disposition of a pending signal

- If we change the disposition of a pending signal, then, when the signal is later unblocked, it is handled according to its new disposition.

Signals are not queued

- If the same signal is generated multiple times while it is blocked, then it is recorded in the set of pending signals, and later delivered, just **once**.

Changing signal dispositions: sigaction()

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);
```

Returns 0 on success, or -1 on error

- The **sig** argument identifies the signal whose disposition we want to retrieve or change. This argument can be any signal except SIGKILL or SIGSTOP.
- The **act** argument is a pointer to a structure specifying a new disposition for the signal.
- The **oldact** argument is a pointer to a structure of the same type, and is used to return information about the signal's previous disposition.

Waiting for a signal: pause()

```
#include <unistd.h>  
  
int pause(void);
```

Always returns -1 with *errno* set to EINTR

Calling `pause()` suspends execution of the process until the call is interrupted by a signal handler (or until an unhandled signal terminates the process).