CSCI 210 Systems Programming

Week 10

Processes

Overview

- How processes are run
  - Kernel Space versus User Space
  - Interrupts, signals
  - File descriptors
  - Priorities
- Process creation
  - fork() and exec()
  - posix_spawn()
  - exit()
  - wait()

Recap - What happens when you run a compiled program?

- Bash sees that you are trying to execute a program – it finds the file and checks, and learns that it is an executable (and remembers this, for quicker future responses)

- Bash uses Linux to prepare an address space and then load and execute the program in the new address space. This is done with the **fork** and **exec** systems calls. (Both have several variants).

Keeping track of processes

- A process in the system is represented using:
  - Process Identification Elements
  - Process State Information
  - Process Control Information
  - User Stack
  - Private User Address Space, Programs and Data
  - Shared Address Space
- This information is stored in what is known as the Process Control Block, aka, PCB.
  - https://en.wikipedia.org/wiki/Process_control_block

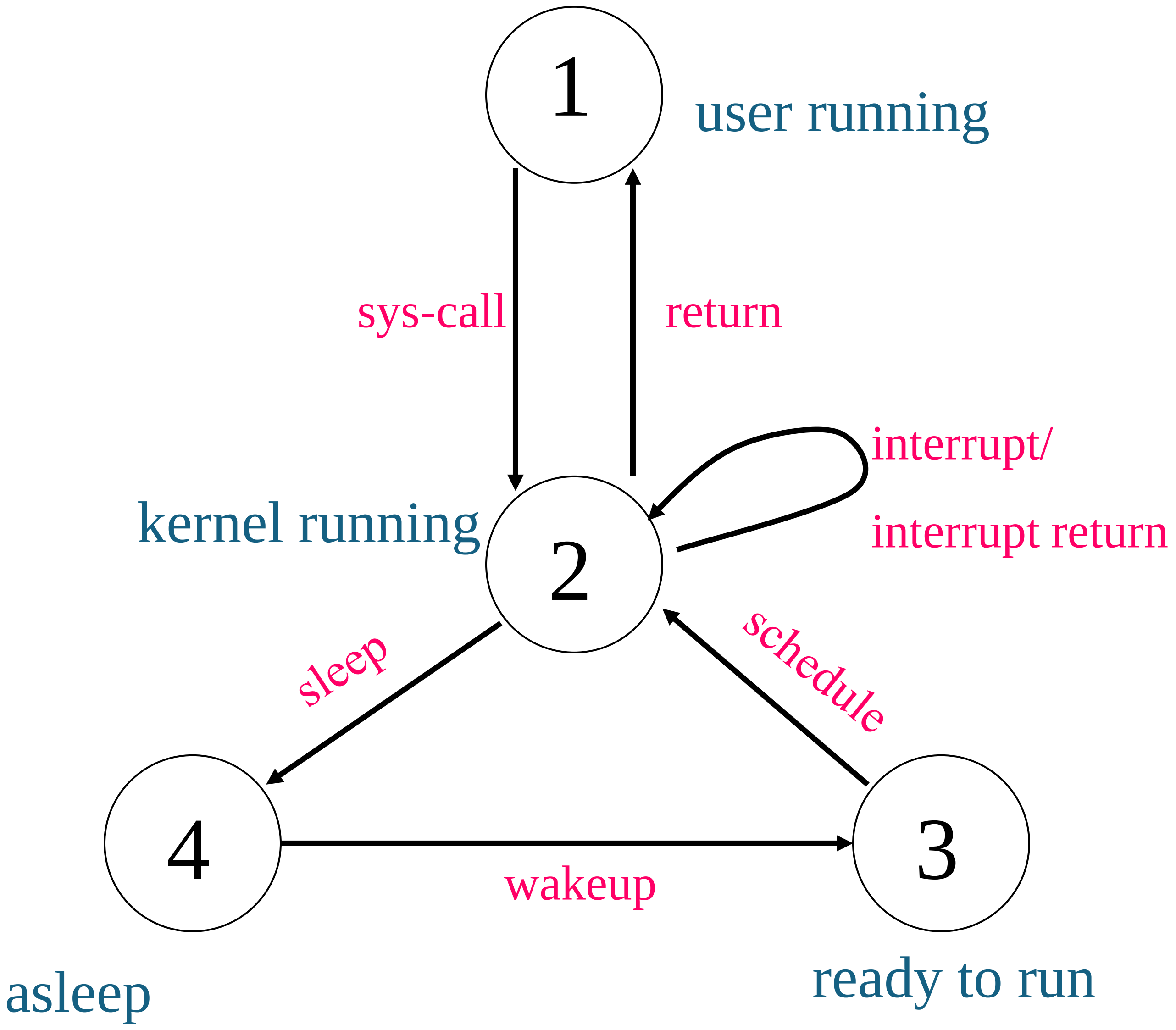
Process Control Block

- Process Information, Process State Information, and Process Control Information constitute the PCB.
- All Process State Information is stored in the Process Status Word (PSW).
- All information needed by the OS to manage the process is contained in the PCB.
- A UNIX process can be in a variety of states

States of a process

- User running: Process executes in user mode
- Kernel running: Process executes in kernel mode
- Ready to run in memory: process is waiting to be scheduled
- Asleep in memory: waiting for an event
- Ready to run swapped: ready to run but requires swapping in
- Preempted: Process is returning from kernel to user-mode but the system has scheduled another process instead
- Created: Process is newly created and not ready to run
- Zombie: Process no longer exists, but it leaves a record for its parent process to collect.

Process state diagram



Kernel Mode versus User Mode

- User mode and kernel mode are two process modes in an OS that differ in the level of access and privileges granted to the code running in each mode
  - A user program runs in User mode, but several switches from the User mode to the Kernel mode occur while the process is handled by the OS.
  - Kernel mode is an elevated, higher privilege mode with direct control over the hardware and system resources.
  - A process may switch to the kernel mode due to these exceptions:
    - Interrupts
    - Traps
    - System calls

Interrupts

- Interrupts are signals from external devices to the CPU, requesting for CPU service.
- While executing in user mode, the CPU's interrupts are enabled so that it will respond to any interrupts.
- When an interrupt occurs, the CPU will enter the kernel mode to handle the interrupt, which also causes the process to enter the kernel mode.

Traps

- Traps are error conditions, such as invalid address, illegal instruction, divide by 0, etc., which are recognized by the CPU as exceptions, causing it to enter the kernel mode to deal with the error.
- In Unix/Linux, the kernel trap handler converts the trap reason to a signal number and delivers the signal to the process.
- For most signals, the default action of a process is to terminate.

System calls

- System call, or syscall for short, is a mechanism which allows a user mode process to enter the kernel mode to execute Kernel functions.
- When a process finishes executing Kernel functions, it returns to the user mode with the desired results and a return value, which is normally 0 for success or -1 for error.
- In case of error, the external global variable errno (in errno.h) contains an ERROR code which identifies the error. The user may use the library function `perror("error message");` to print an error message, which is followed by a string describing the error.

Another view of exceptions

	intentional happens every time	**unintentional** contributing factors
synchronous caused by an instruction	**trap: system call** open, close, read, write, fork, exec, exit, wait, kill, etc.	**fault** invalid or protected address or opcode, page fault, overflow, etc.
asynchronous caused by some other event	"software interrupt" software requests an interrupt to be delivered at a later time	**interrupt** caused by an external event: I/O op completed, clock tick, power fail, etc.

Creating a new process

- In UNIX, a new process is created by means of the fork() - system call. The OS performs the following functions:
    - It allocates a slot in the process table for the new process
    - It assigns a unique ID to the new process
    - It makes a copy of process image of the parent (except shared memory)
    - It assigns the child process to the Ready to Run State
    - It returns the ID of the child to the parent process, and 0 to the child.

- Note, the fork() is called once but returns twice - namely in the parent and the child process.

fork()

- `pid_t fork(void)` is the prototype of the fork() call.
- Remember that fork() returns twice
  - in the newly created (child) process with return value 0
  - in the calling process (parent) with return value = pid of the new process.
  - A negative return value (-1) indicates that the call has failed

- Different return values are the key for distinguishing parent process from child process!
- The child process is an exact copy of the parent, yet, it is a copy, i.e., an identical but separate process image.

A fork() Example

```
#include <unistd.h>
main()
{
    pid_t pid              /* process id */
    printf("just one process before the fork()\n");
    pid = fork();
    if(pid == 0)
            printf("I am the child process\n");
    else if(pid > 0)
            printf("I am the parent process\n");
    else
            printf("the fork() has failed\n")
}
```

Basic Process Coordination

- The `exit()` call is used to terminate a process.
  - Its prototype is: `void exit(int status)`, where status is used as the return value of the process.
  - `exit(i)` can be used to announce success and failure to the calling process.

- The `wait()` call is used to temporarily suspend the parent process until one of the child processes terminates.
  - The prototype is: `pid_t wait(int *status)`, where status is a pointer to an integer to which the child's status information is being assigned.
  - `wait()` will return with a pid when any one of the children terminates or with -1 when no children exist.

More coordination

- To wait for a particular child process to terminate, we can use the `waitpid()` call.
  - Prototype: `pid_t waitpid(pid_t pid, int *status, int opt)`

- To get information about the process or its parent:
  - `getpid()` returns the process id
  - `getppid()` returns the parent’s process id
  - `getuid()` returns the user’s user id

Orphans and Zombies

- A child process whose parent has terminated is referred to as orphan.

- When a child exits when its parent is not currently executing a `wait()`, a zombie emerges.
  - A zombie is not really a process as it has terminated but the system retains an entry in the process table for the non-existing child process.
  - A zombie is put to rest when the parent finally executes a `wait()`.

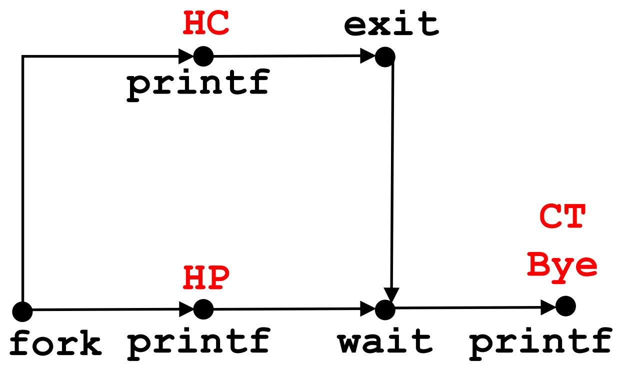
- When a parent terminates, orphans and zombies are adopted by the `init` process (prosess id 1) of the system.

Reaping Children

- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process

`wait` Example

```
int main() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    return 0;
}
```



Feasible output:
HC
HP
CT
Bye

Infeasible output:
HP
CT
Bye
HC

Reaping Children

- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by **`init`** process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Returning from the `main` routine
  - Calling the `exit` function
  - Receiving a signal whose default action is to terminate

- `void exit(int status)`
  - Terminates with an **exit status** of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- `exit` is called once but never returns.

But how do we run a **new** program?

- The child, or any process really, can **replace its program** in midstream.
- **exec*** system call: “forget everything in my address space and reinitialize my entire address space with stuff from a named program file.”
- The exec system call **never returns**: the new program executes in the calling process until it dies (exits).

exec (original concept) from Thompson & Ritchie 1974 article

5.3 Execution of Programs

Another major system primitive is invoked by

execute(file, arg_1, arg_2, ..., arg_n)

which requests the system to read in and execute the program named by *file*, passing it string arguments arg_1, arg_2, ..., arg_n. Ordinarily, arg_1 should be the same string as *file*, so that the program may determine the name by which it was invoked. All the code and data in the process using *execute* is replaced from the file, but open files, current directory, and interprocess relationships are unaltered. Only if the call fails, for example because *file* could not be found or because its execute-permission bit was not set, does a return take place from the *execute* primitive; it resembles a "jump" machine instruction rather than a subroutine call.

notable *exec* properties

- an *exec* call transforms the calling process by loading a new program in its memory space.

- the *exec* does not create a new sub-process.

- unlike the *fork* there is no return from a successful *exec*.

- all types of *exec* calls perform in principle the same task.
    - Find the program (i.e., code) from a reference to a file, set up command line arguments and the environment variables, and run.

exec()* calls

- *See man exec*

exec() example:

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

int main()
{
	printf("executing ls\n");
	execl("/bin/ls", "ls", "-l", (char *)0);

	/* if execl returns, then the call has failed.... */
	perror("execl failed to run ls");
	exit(1);
}
```

argv[0] convention

- Consider *int execl (const char *path, const char *arg0 …)*

- Here, the parameter *arg0* is, by convention, the name of the program or command without any path-information.

- Example:
  - *execl(“/bin/ls”, “ls”, “-l”, (char *)0);*

- In general, the first argument in a command-line argument list is the name of the command or program itself!

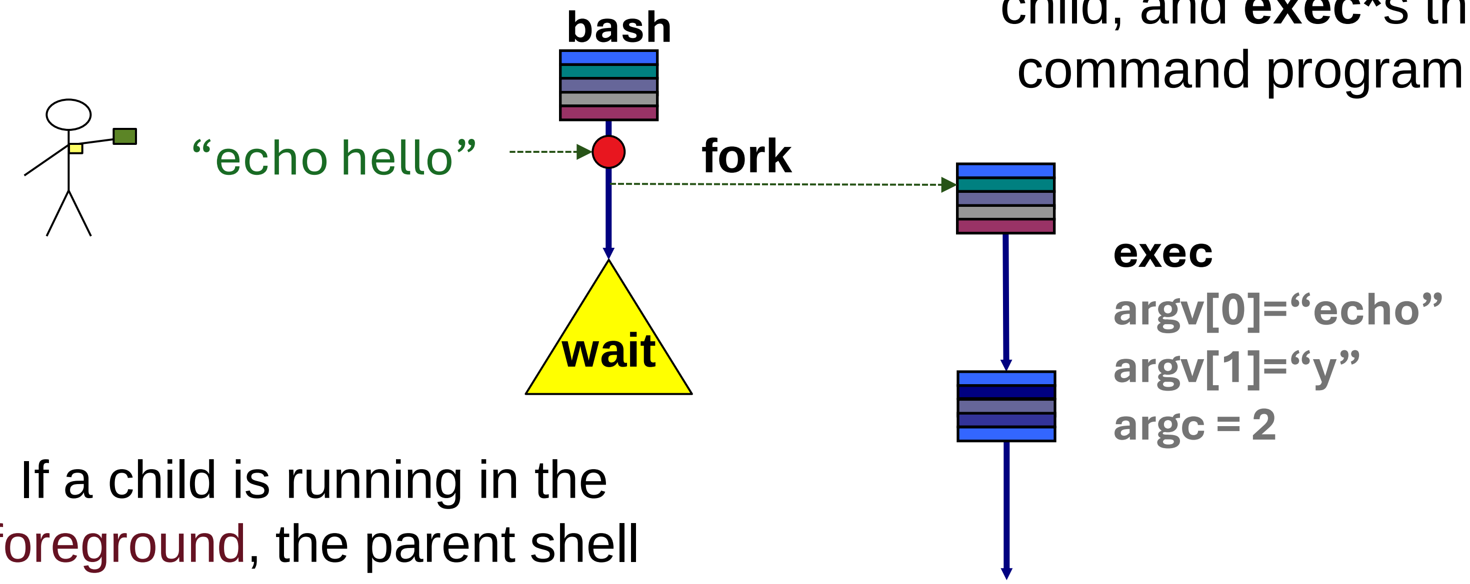
An *execvp()* example

```
#include <unistd.h>

main()
 {
   char * const av[] = {“myecho”,”hello”,”world”, (char *) 0};

   execvp(av[0], av);

 }
```

where myecho is name of a program that lists all its command line arguments.

Shell: command execution

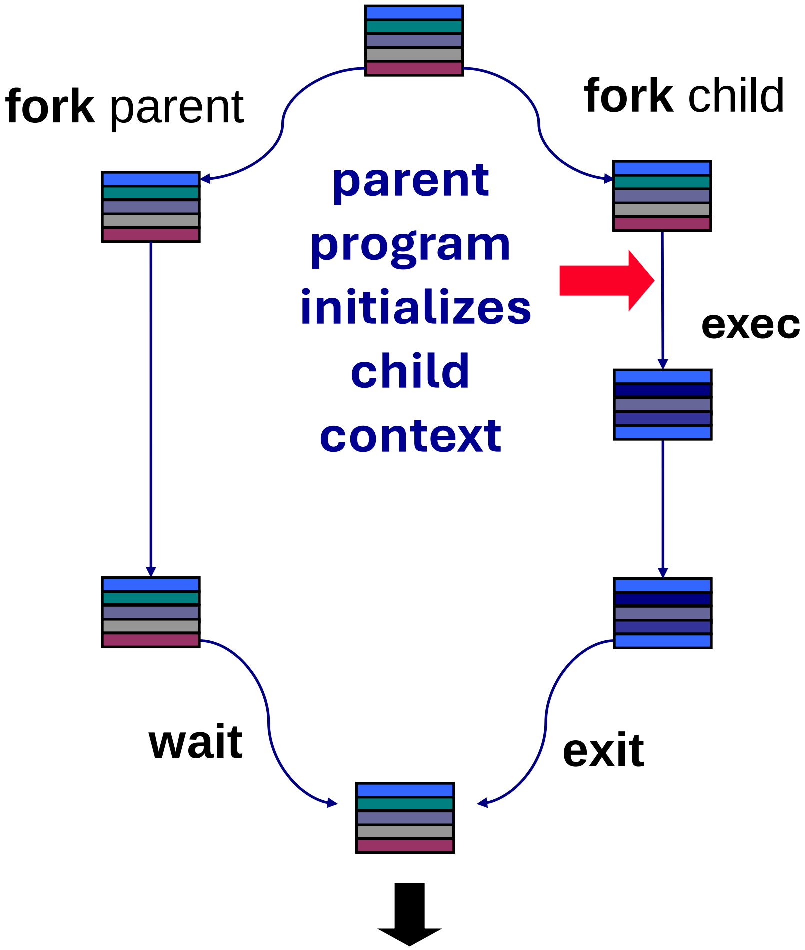
Shell waits in **read** call for next command from tty, **forks** child, and **exec***s the named command program in child.



If a child is running in the foreground, the parent shell blocks in a **wait*** system call until the child changes state (e.g., **exits**).

Child process continues on to execute "echo" program independently of parent.

Summary of fork/exec/exit/wait syscalls



int pid = fork();
Create a new process that is a clone of its parent.

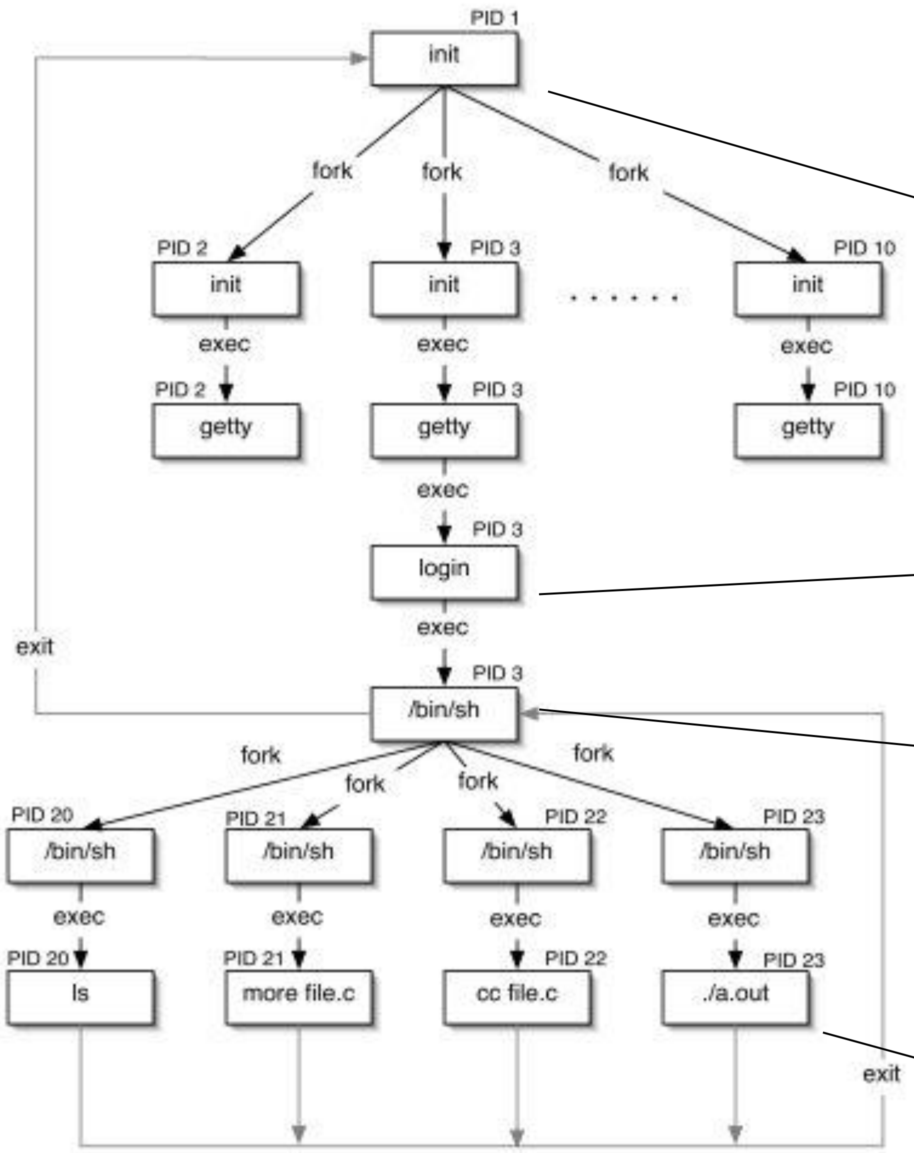
exec*("program" [argvp, envp]);
Overlay the calling process with a new program, and transfer control to it, passing arguments and environment.

exit(status);
Exit with status, destroying the process.

int pid = wait*(&status);
Wait for exit (or other status change) of a child, and "reap" its exit status.
<u>Recommended</u>: use **waitpid()**.

How is the first process created?

Init and Descendents



Kernel "handcrafts" initial process to run "init" program.

Other processes descend from init, including **login** program.

Login runs user **shell** in a child process after user authenticates.

User shell runs user commands as child processes.

Using posix_spawn instead of fork()/exec()

- See the man page
- Take a look at the simple example at:
  - https://buffaloitservices.com/an-in-depth-look-at-child-process-creation-in-c

Summary of Process Creation

Processes

- Four basic process control function families:
  - fork()
  - exec()
    - And other variants such as execve()
  - exit()
  - wait()
    - And variants like waitpid()
- Standard on all UNIX-based systems
- **F**ork(), **E**xit(), **W**ait() are all wrappers to the syscall() function
  - man syscall()

Processes

- **`int fork(void)`**
  - creates a new process (child process) that is identical to the calling process (parent process)
  - OS creates an exact duplicate of parent's state:
    - Virtual address space (memory), including heap and stack
    - Registers, except for the return value (%eax/%rax)
    - File descriptors of files are copied into child process
  - **Result → Equal but separate state**

  - Fork is interesting (and often confusing) because it is called *once* but returns *twice*

Processes

- **`int fork(void)`**
  - returns 0 to the child process
  - returns child's **`pid`** (process id) to the parent process
  - Usually used like:

```
pid_t pid = fork();

if (pid == 0) {
   // pid is 0 so we can detect child
   printf("hello from child\n");
}

else {
   // pid = child's assigned pid
   printf("hello from parent\n");
}
```

Processes

- `int exec()`
  - Replaces the current process's state and context
    - But keeps PID, open files, and signal context
  - Provides a way to load and run **another** program
    - Replaces the current running memory image with that of new program
    - Set up stack with arguments and environment variables
    - Start execution at the entry point
  - Never returns on successful execution
  - The newly loaded program's perspective: as if the previous program has not been run before
  - More useful variant is **`int execve()`**
  - More information? man 3 exec

Processes

- `void exit(int status)`
  - Normally return with status 0 (other numbers indicate an error)
  - Terminates the current process
  - OS frees resources such as heap memory and open file descriptors and so on…
  - Reduce to a zombie state
    - Must wait to be reaped by the parent process (or the init process if the parent died)
    - Signal is sent to the parent process notifying of death
    - Reaper can inspect the exit status

Processes

- `int wait(int *child_status)`
  - suspends current process until one of its children terminates
  - return value is the pid of the child process that terminated
    - When wait returns a pid > 0, child process has been reaped
    - All child resources freed
  - if child_status != NULL, then the object it points to will be set to a status indicating why the child process terminated
  - More useful variant is `int waitpid()`
  - For details: man 2 wait

Process Examples

```
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf("Child!\n");

   exit(0);
}
else{


   printf("Parent!\n");
}
```

- What are the possible output (assuming fork succeeds) ?
  - `Child!`
`Parent!`
  - `Parent!`
`Child!`
- How to get the child to always print first?

Process Examples

```
int status;
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf("Child!\n");

   exit(0);
}
else{
   waitpid(child_pid, &status, 0);

   printf("Parent!\n");
}
```

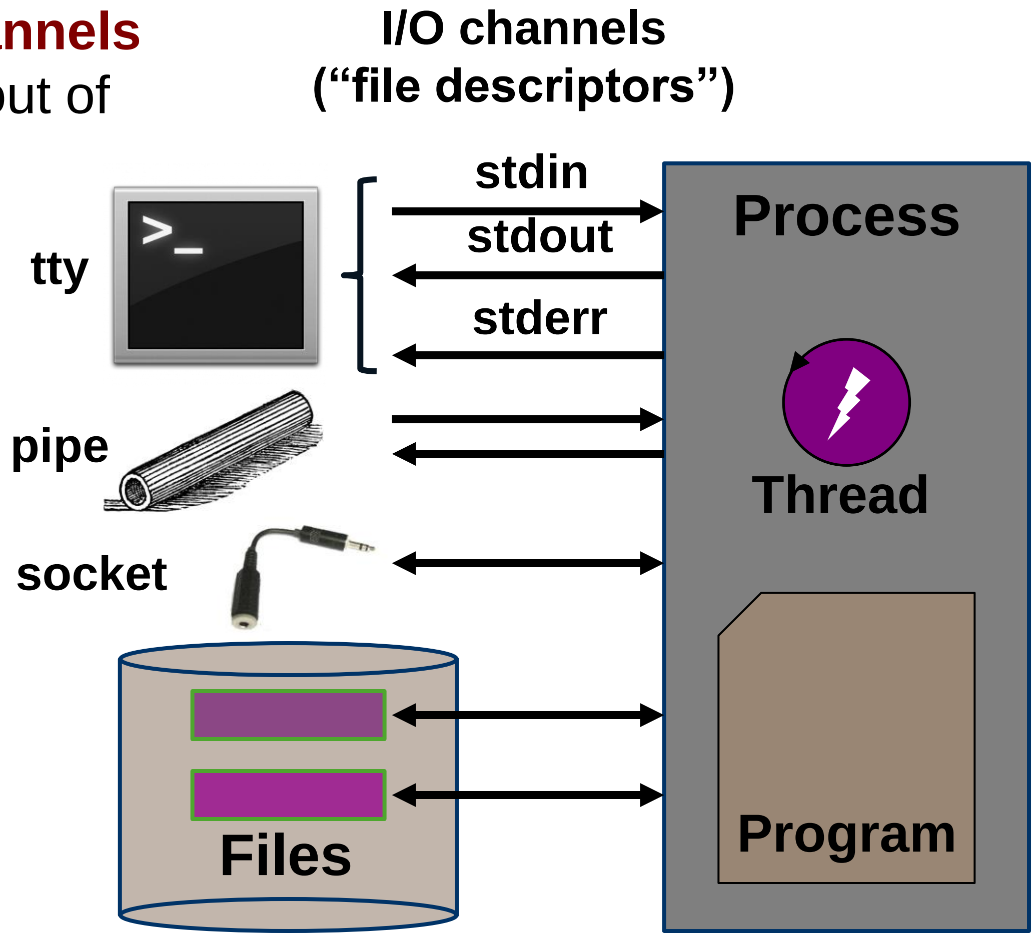
- Waits until the child has terminated.
  Parent can inspect exit status of child using 'status'
  - WEXITSTATUS(status)

- Output always:
  Child!
  Parent!

Data versus Processes

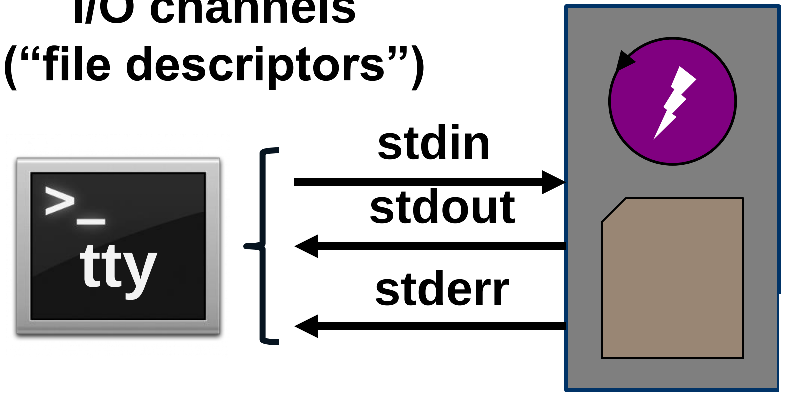
A process has multiple **channels** for data movement in and out of the process (I/O).

The parent process and parent program set up and control the channels for a child (until **exec**).



Standard I/O descriptors

I/O channels
(“file descriptors”)



Open files or other I/O channels are named within the process by an integer **file descriptor** value.

```
count = read(0, buf, count);
if (count == -1)  {
        perror(“read failed”);  /* writes to stderr */

        exit(1);
}
count = write(1, buf, count);
if (count == -1) {
        perror(“write failed”);  /* writes to stderr */
        exit(1);
}
```

Standard descriptors for primary input (stdin=0), primary output (stdout=1), error/status (stderr=2).

These are inherited from the parent process and/or set by the parent program.

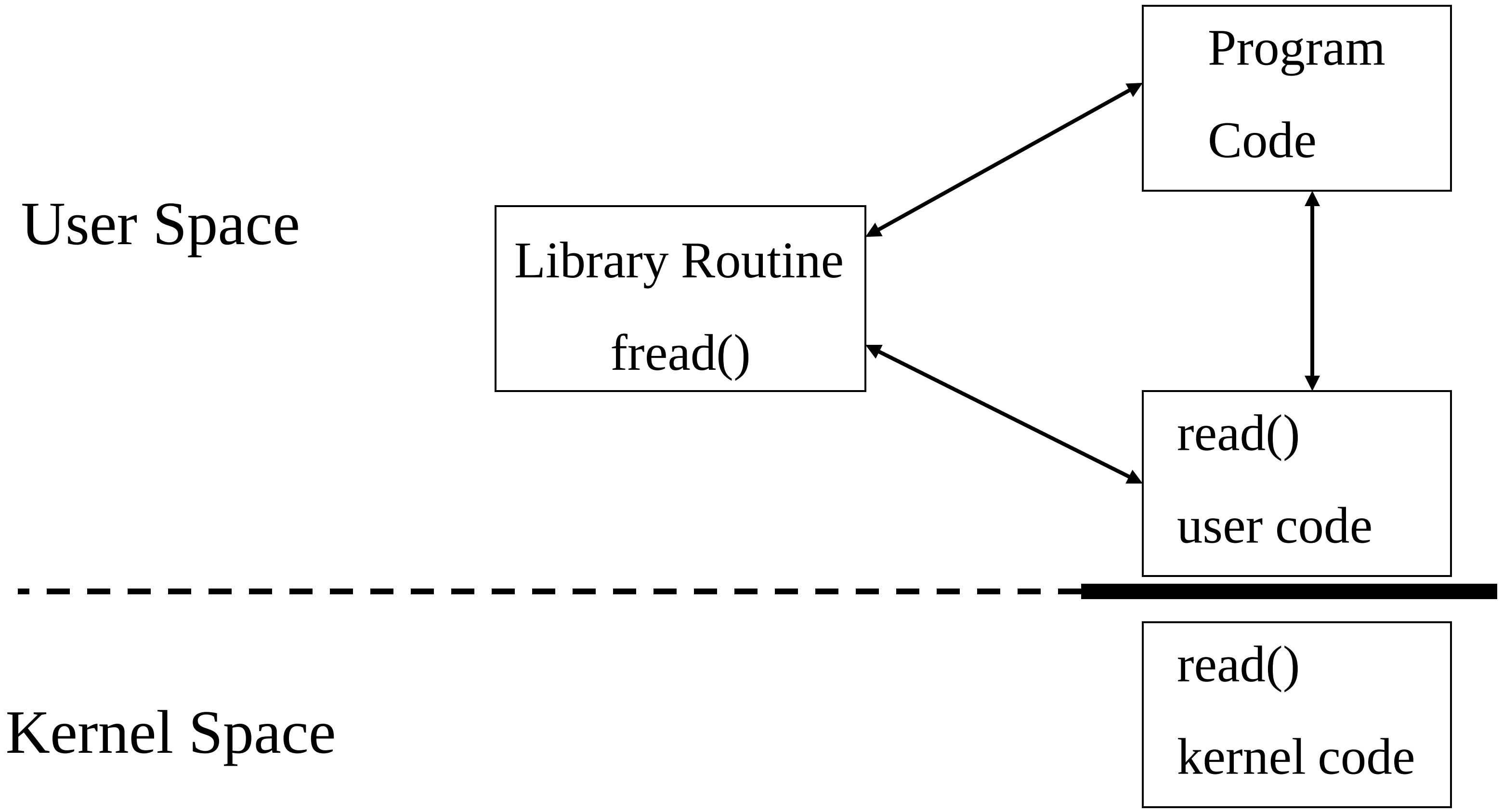
By default, they are bound to the controlling terminal.

Files

- UNIX Input/Output operations are based on the concept of files.
- Files are an abstraction of specific I/O devices.
- A very small set of system calls provide the primitives that give direct access to I/O facilities of the UNIX kernel.
- Most I/O operations rely on the use of these primitives.

- We must remember that the basic I/O primitives are system calls, executed by the kernel.
  - Which means the file I/O calls we make cause a switch to the Kernel Mode

User and System Space



Different types of files

- UNIX deals with two different classes of files:
  - Special Files
  - Regular Files

- Regular files are just ordinary data files on disk - something you have used all along when you studied programming!

- Special files are abstractions of devices. UNIX deals with devices as if they were regular files.
- The interface between the file system and the device is implemented through a device driver - a program that hides the details of the actual device.

Special files

- UNIX distinguishes two types of special files:
  - Block Special Files represent a device with characteristics similar to a disk. The device driver transfers chunks or blocks of data between the operating system and the device.
  - Character Special Files represent devices with characteristics similar to a keyboard. The device is abstracted by a stream of bytes that can only be accessed in sequential order.

Access Primitives

- UNIX provides access to files and devices through a (very) small set of basic system calls (primitives)

  - create()
  - open()
  - close()
  - read()
  - write()
  - ioctl()
  - fcntl()

open()

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

*int open(const char *path, int flags, [mode_t mode]);*

char *path: is a string that contains the fully qualified filename of the file to be opened.

int flags: specifies the method of access i.e. read_only, write_only read_and_write.

mode_t mode: optional parameter used to set the access permissions upon file creation.

read() and *write()*

#include <unistd.h>

*ssize_t read(int filedes, void *buffer, size_t n);*

*ssize_t write(int filedes, const void *buffer, size_t n);*

int filedes: file descriptor that has been obtained though an open() or create() call.

*void *buffer: pointer to an array that will hold the data that is read or holds the data to be written.*

size_t n: the number of bytes that are to be read or written from/to the file.

close()

- Although all open files are closed by the OS upon completion of the program, it is *good programming style* to “clean up” after you are done with any system resource.

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

A simple example:

```
#include <fcntl.h> /* controls file attributes */
#include<unistd.h> /* defines symbolic constants */
main()
{
    int fd;   /* a file descriptor */
    ssize_t nread;      /* number of bytes read */
    char buf[1024];   /* data buffer */
    /* open the file “data” for reading */
    fd = open(“data”, O_RDONLY);
    /* read in the data */
    nread = read(fd, buf, 1024);
    /* close the file */
    close(fd);
}
```

Buffered vs unbuffered I/O

- What happens when we write to a file?
  - The write call forces a switch to Kernel Mode.
  - The OS copies the specified number of bytes from user space into kernel space
  - The system wakes up the device driver to write these bytes to the physical device (if the file-system is in synchronous mode).
  - The system selects a new process to run (remember there are other processes, too)
  - Finally, control is returned to the process that executed the write call.
- Discuss the effects on the performance of your program!

Un-buffered I/O

- Every read and write is executed by the kernel.
- Hence, every read and write will cause a context switch in order for the system routines to execute.
- Poor performance

Buffered I/O

- Explicit versus implicit buffering:
  - Explicit - collect as many bytes as you can before writing to file and read more than a single byte at a time.
  - However, use the basic UNIX I/O primitives
    - Careful !! Your program my behave differently on different systems.
    - Here, the programmer is explicitly controlling the buffer-size

  - Implicit - use the Stream facility provided by <stdio.h>
  - FILE *fd, fopen, fprintf, fflush, fclose, … etc.
  - A FILE structure contains a buffer (in user space) that is usually the size of the disk blocking factor (512 or 1024)

The *fcntl()* system call

- The *fcntl()* system call provides some control over already open files. fcntl() can be used to execute a function on a file descriptor.
- The prototype is: *int fcntl(int fd, int cmd, …….) where*
  - fd is the corresponding file descriptor
  - cmd is a “pre-defined” command (integer const)
  - …. are additional parameters that depend on what cmd is.

The *fcntl()* system call

Two important commands are: *F_GETFL and F_SETFL*

- F_GETFL is used to instruct fcntl() to return the current status flags
- F_SETFL instructs fcntl() to reset the file status flags
- Example: int i = fcntl(fd, F_GETFL); or
- int i = fcntl(fd, F_SETFL, …

Using *fcntl()* to change to non-blocking I/O

- We can use the *fcntl()* system call to change the blocking behavior of the *read()* and *write():*

- *Example:*

```
#include <fcntl.h>
.....
if ( fcntl(filedes, F_SETFL, O_NONBLOCK) == -1)
   perror("fcntl")
```