OPerating Systems Laboratory
OPS*Lab*

Programming in the UNIX Environment

(Part 4.2.2)

Prof. Marco Autili

University of L'Aquila

What we have done so far

Chapter 1
Chapter 3
Chapter 4

- Introduction
- File I/O
- Files and Directories

Chapter 5

- Standard I/O Library

What we are going to do

Chapter 7

Process Environment

- main function
- Process termination
- Environment list

Chapter 8

Process Control

- Creation of new processes
- Program execution
- Process termination

Chapter 10

Signals

- kill
- raise
- alarm
- pause

What we are going to do

Chapter 7

Process Environment

- main function
- Process termination
- Environment list

Chapter 8

Process Control

- Creation of new processes
- Program execution
- Process termination

Chapter 10

Signals

- kill
- raise
- alarm
- pause

Basic ingredient: user Identification

The *user ID* from our entry in the password file (/etc/passwd - Directory Services on macOS) is a numeric value that identifies us to the system (see also Section 6.6)

- This user ID is assigned by "the system administrator" when our login name is assigned, and we cannot change it
- The user ID is normally assigned to be unique for every user
- We'll see how the kernel uses the user ID to check whether we have the appropriate permissions to perform certain operations
- We call the user whose user ID is 0 either the *root* or the *superuser*.
  The entry in the password file normally has a login name of root, and we refer to the special privileges of this user as superuser privileges
- As explained in Chapter 4 of the book, if a process has superuser privileges, most file permission checks are bypassed
- The superuser has free rein over the system, though some operating system functions are restricted to the superuser also

Basic ingredient: user Identification

Group ID

- Our entry in the password file also specifies our numeric *group ID* (see also Section 6.6)
- It is also assigned by "the system administrator" when our login name is assigned
- Groups are normally used to collect users together into, e.g., projects or departments
- This allows the sharing of resources, such as files, among members of the same group
- We can set the permissions on a file so that all members of a group can access the file, whereas others outside the group cannot (Section 4.5)
- There is also a group file that maps group names into numeric group IDs. The group file is usually /etc/group (Directory Services: called Apple Open Directory on macOS)

Basic ingredient: user Identification

Supplementary Group IDs

- In addition to the group ID specified in the password file for a login name, most versions of the UNIX System allow a user to belong to more than one group

- This practice started with 4.2 BSD, which allowed a user to belong to up to 16 additional groups

- These supplementary group IDs are obtained at login time by reading the file /etc/group and finding the first 16 entries that list the user as a member
    - e.g., try with cat /etc/group | egrep root

- POSIX requires that a system support at least 8 supplementary groups per process, but most systems support at least 16 (more on Chapter 2)

Processes and Process ID (intro)

An executing instance of a program file is called a process

- Some operating systems use the term task to refer to a program that is being executed ☹
- UNIX-like systems guarantee that every process has a unique numeric identifier called the Process ID (PID), it is always a non-negative integer
- Process ID 0 is usually the scheduler process
  (often known as the swapper)
  - No program on disk corresponds to this process, which is part of the kernel and is known as a system process
- Process ID 1 is usually the init process and is invoked by the kernel at the end of the bootstrap procedure
  - The program file for this process was /etc/init in older versions of the UNIX System and it is /sbin/init in newer versions
  - It is launchd under macOS systems (see next slide)

The init process

- The init process is responsible for bringing up a UNIX system after the kernel has been bootstrapped
- The init process usually reads the system-dependent initialization files (the /etc/rc* files or /etc/inittab and the files in /etc/init.d) and brings the system to a certain state, such as multiuser
- The init process never dies
- It is a normal user process, not a system process within the kernel, like the swapper, although it does run with superuser privileges

In Mac OS X 10.4, the `init` process was replaced with the `launchd` process, which performs the same set of tasks as `init`, but has expanded functionality. See Section 5.10 in Singh [2006] for a discussion of how `launchd` operates.

```
marcoautili@iMac ~ %
marcoautili@iMac ~ % ls -la /bin/launchctl
-rwxr-xr-x  1 root  wheel  329344 Jan  1  2020 /bin/launchctl
marcoautili@iMac ~ %
marcoautili@iMac ~ %
marcoautili@iMac ~ % man launchctl
```

The launch control command launchctl interfaces with launchd

Processes privileges and file access permissions

Before diving into details, let's consider the following:

a user program (i.e., an executable file) needs access a resource (a file)

- Usually, a user program can be executed by the owner or (if "x" permissions are granted) by another user
- Thus, beyond considering the owner of the executable file, we need to consider the user that actually executes it and, hence, the "owner" of the corresponding running process

(1) the owner of the executable file
(2) the owner of the corresponding running process (i.e., the user that runs the executable file)
(3) the owner of the resource

- the simplest case
  if (1), (2) and (3) are the same "person" → the access is granted

- a more complicated case
  if (1) and (3) are the same "person" but (2) is not (i.e., the user that runs the executable file is a "persona" different from the owner of the executable file and the resource) → the access is **not** granted
  However, intuitively, if there was a mechanism for changing the "persona" of a running process to that of the owner of the corresponding executable file → the access would be granted… … next slides

Processes privileges and file access permissions

Every process has six or more IDs associated with it (see Figure 4.5)

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by **exec** functions

Figure 4.5 User IDs and group IDs associated with each process

- The real user ID identifies the user who started the process, and the real group ID identifies that user's default group
  - these IDs are the ones taken from our entry in the password file when we log in
  - these values don't change during a login session, although there are ways for a superuser process to change them (see Section 8.11)

Processes privileges and file access permissions

- The effective user ID, effective group ID, and supplementary group IDs determine the privileges of the process
  - They are collectively called the *persona* of the process, because they determine "who it is" for purposes of access control
  - Normally, a process inherits its persona from the parent process, but under special circumstances a process can change its persona and thus change its access permissions

    → more on that later…

- The saved set-user-ID and saved set-group-ID contain copies of the effective user ID and the effective group ID, respectively, when a program is executed (see Section 8.11)
  - Remember that every file (and program file as well) has the user ID of the owner and the group ID of the owner that are specified by the st_uid and st_gid members of the stat structure, resp. (see next slides)

Processes privileges and file access permissions

- The real user ID and the real group ID do not play a role in access control, so they are not considered part of the persona
- The operating system normally decides access permission for a file based on:
  - the effective user and group IDs of the process and its supplementary group IDs, together with
  - the owner, group and permission bits of the file
- The effective user ID of a process also controls permissions for sending signals using the kill function (later on these slides)
- There are many operations which can only be performed by a process whose effective user ID is 0

  A process with this user ID is a *privileged process* (remember that also the username root is associated with user ID 0)

Processes privileges and file access permissions

- Normally, the effective user ID equals the real user ID, and the effective group ID equals the real group ID
- However, we can also set a special flag (i.e., a bit within the part of st_mode dedicated to permissions) that says:
  - "Upon execution, set the effective user ID of the process to be the owner of the executable file (st_uid)"
  - Similarly, we can set another bit in the file's mode word that causes the effective group ID to be the group owner of the file (st_gid)
- These two bits in the file's mode word are called the *set-user-ID* bit and the *set-group-ID* bit
  - These two bits can be tested against the symbolic constants S_ISUID and S_ISGID, resp., (see the example chap4-filedir/file-isuid-isgid.c)

```
struct stat {
  mode_t          st_mode;    /* file type & mode (permissions) */
  ino_t           st_ino;     /* i-node number (serial number) */
  dev_t           st_dev;     /* device number (file system) */
  dev_t           st_rdev;    /* device number for special files */
  nlink_t         st_nlink;   /* number of links */
  uid_t           st_uid;     /* user ID of owner */
  gid_t           st_gid;     /* group ID of owner */
  off_t           st_size;    /* size in bytes, for regular files */
  struct timespec st_atim;    /* time of last access */
  struct timespec st_mtim;    /* time of last modification */
  struct timespec st_ctim;    /* time of last file status change */
  blksize_t       st_blksize; /* best I/O block size */
  blkcnt_t        st_blocks;  /* number of disk blocks allocated */
};
```

Changing User IDs and Group IDs

In fact, upon execution, since the bit is set, the effective user ID of the process is set to be the owner of the file (st_uid)

For example, if the owner of the file is the superuser and if the file's set-user-ID bit is set, then while that program file is running as a process, it has superuser privileges. This happens regardless of the real user ID of the process that executes the file. As an example, the UNIX System program that allows anyone to change his or her password, `passwd`(1), is a set-user-ID program. This is required so that the program can write the new password to the password file, typically either `/etc/passwd` or `/etc/shadow`, files that should be writable only by the superuser. Because a process that is running set-user-ID to some other user usually assumes extra permissions, it must be written carefully. We'll discuss these types of programs in more detail in Chapter 8.

Indeed, in my macOS, the program /usr/bin/passwd is NOT a set-user-ID program… see next slide

Changing User IDs and Group IDs

```
bash-3.2$
bash-3.2$ ./file-isuid-isgid dirhardlink.c

dirhardlink.c:
The "set-user-ID" bit is NOT set
The "set-group-ID" bit is NOT set

bash-3.2$
bash-3.2$
bash-3.2$ ./file-isuid-isgid /usr/bin/login

/usr/bin/login:
The "set-user-ID" bit is set
The "set-group-ID" bit is NOT set

bash-3.2$
bash-3.2$ ./file-isuid-isgid /usr/bin/passwd

/usr/bin/passwd:
The "set-user-ID" bit is NOT set
The "set-group-ID" bit is NOT set

bash-3.2$
```

See next slide

In macOS, /usr/bin/passwd does not write directly to the password file

As already said, it uses Apple Open Directory

Changing User IDs and Group IDs

- When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access

- Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource

- It is recommended to use the least-privilege model when we design our applications

- According to the least-privilege model, our programs should use the least privilege necessary to accomplish any given task. This reduces the risk that security might be compromised by a malicious user trying to trick our programs into using their privileges in unintended ways

Changing User IDs and Group IDs

- The more common case of changing persona is when an ordinary user program needs access a resource that wouldn't ordinarily be accessible to the user actually running it
  - For example, you may have a file that is controlled by your program but that shouldn't be read or modified directly by other users, either because it implements some kind of locking protocol, or because you want to preserve the integrity or privacy of the information it contains
  - This kind of restricted access can be implemented by having the program change its effective user or group ID to match that of the resource

Both the real and effective user ID can be changed during a process execution

- The most obvious situation where it is necessary for a process to change its user and/or group IDs is the login program
  - when login starts running, its user ID is root
  - its job is to start a shell whose user and group IDs are those of the user who is logging in
  - to accomplish this fully, the login process must set the real user and group IDs as well as its persona

More on Changing User IDs and Group IDs

The _POSIX_SAVED_IDS symbol

- If the symbol _POSIX_SAVED_IDS is defined, it indicates that
  - the system remembers the effective user and group IDs of a process before it executes an executable file with the *set-user-ID* (SETUID) or *set-group-ID* (SETGID) bits set, and
  - that explicitly changing the effective user or group IDs back to these values is permitted

- If this option is not defined, then if a non privileged process changes its effective user or group ID to the real user or group ID of the process, it can't change it back again

> The saved IDs are required as of the 2001 version of POSIX.1. They were optional in older versions of POSIX. An application can test for the constant `_POSIX_SAVED_IDS` at compile time or can call `sysconf` with the `_SC_SAVED_IDS` argument at runtime, to see whether the implementation supports this feature.

How an Application Can Change Persona

SUMMING UP

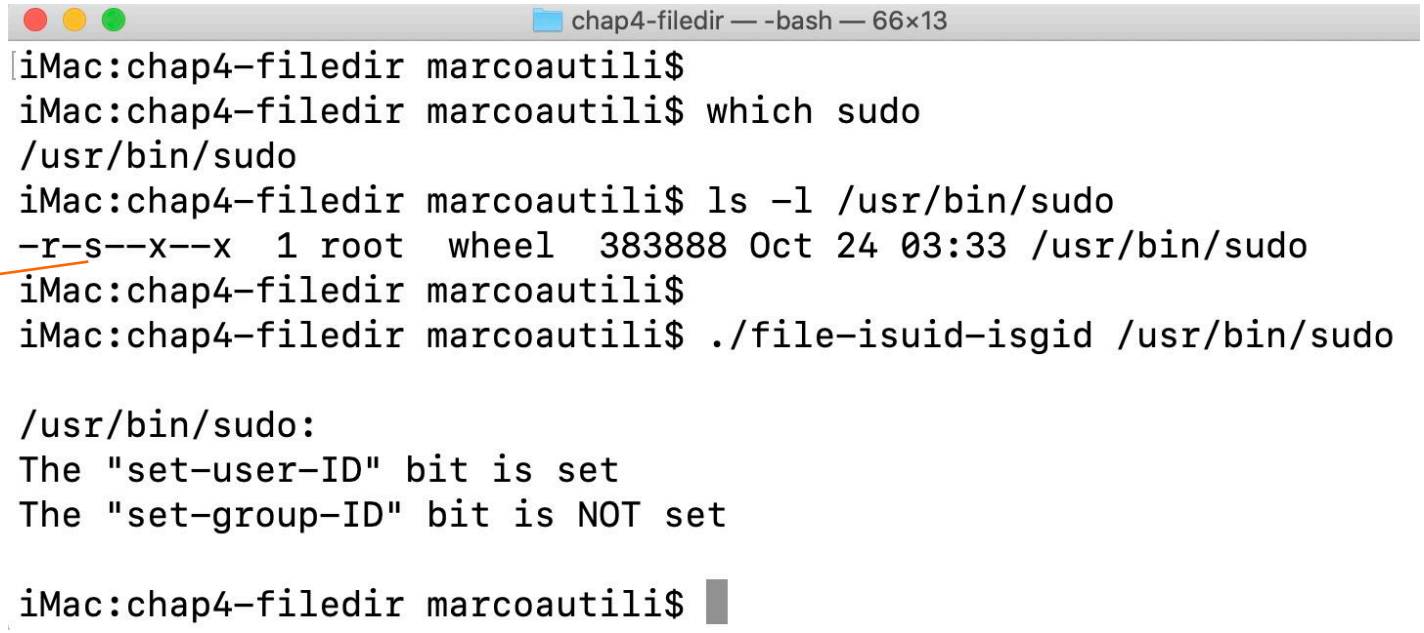
- SETUID or SETGID represents special permission attributes in Unix and Unix-like systems, they allow unprivileged users to run programs with elevated privileges (the privileges of who created the program: the owner)
  - With SETUID we can run programs as the user who created them
  - With SETGID we can run programs as if we were in the group of who created them

- Another (and most notable) example of setuid programs is sudo
- Since root created sudo, if we execute sudo (if we are able to do that) we gain root privileges

next slide…

How SETUID/SETGID works

The 's' (in the place of 'x') in the user permission means that the binary sudo has the SUID bit enabled



How to set SUID/GUID from the SHELL

```
chap8-proc — bash — 82×24
bash-3.2$
bash-3.2$ chmod 755 file-isuid-isgidcopy.c
bash-3.2$
bash-3.2$ chmod 755 file-isuid-isgidcopy.c
bash-3.2$ ls -l file-isuid-isgidcopy.c
-rwxr-xr-x@ 1 marcoautili  staff  1026 Dec 11  2020 file-isuid-isgidcopy.c
```

421 401 401

7 5 5

Try also chmod u+s <filename> or <dirname>

and similarly, chmod g+s <filename> or <dirname>

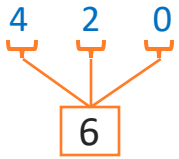
```
bash-3.2$
bash-3.2$ chmod 4755 file-isuid-isgidcopy.c
bash-3.2$ ls -l file-isuid-isgidcopy.c
-rwsr-xr-x@ 1 marcoautili  staff  1026 Dec 11  2020 file-isuid-isgidcopy.c
```

4 0 0

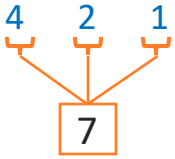
4

How to set SUID/GUID from the SHELL

```
bash-3.2$
bash-3.2$ chmod 6755 file-isuid-isgidcopy.c
bash-3.2$ ls -l file-isuid-isgidcopy.c
-rwsr-sr-x@ 1 marcoautili  staff  1026 Dec 11  2020 file-isuid-isgidcopy.c
```




```
bash-3.2$
bash-3.2$ chmod 7755 file-isuid-isgidcopy.c
bash-3.2$ ls -l file-isuid-isgidcopy.c
-rwsr-sr-t@ 1 marcoautili  staff  1026 Dec 11  2020 file-isuid-isgidcopy.c
```

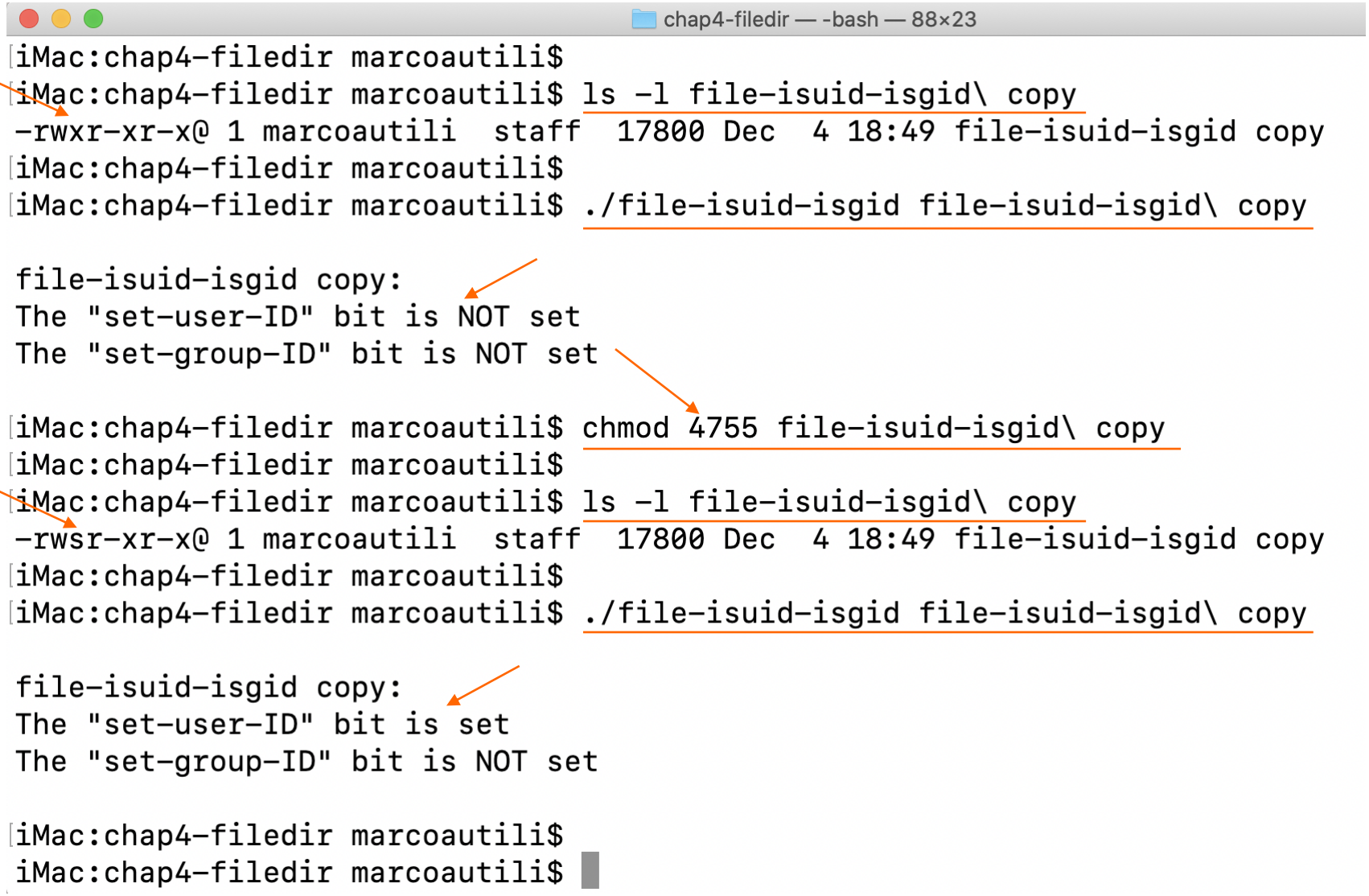


See also chmod +t <filename> or <dirname>

A Sticky bit is a permission bit that can be set on a file or a directory.

The modern function of the sticky bit refers to directories and protects directories and their content from being hijacked by non-owners; this is found in most modern Unix-like systems. Files in a shared directory such as /tmp belong to individual owners, and non-owners may not delete, overwrite , or rename them. See https://en.wikipedia.org/wiki/Sticky_bit for more

How to set SUID/GUID from the SHELL



Summing up: File Access Permissions

- The file access tests that the kernel performs each time a process opens, creates, or deletes a file depend on the
  - owners of the file
    (i.e., the user and the group in st_uid and st_gid, resp., within the stat structure)
  - effective IDs of the process
    i.e., effective user ID and effective group ID of the process
  - supplementary group IDs of the process (if supported)
    - i.e., other groups to which the user belongs
- Note that
  - the two owner IDs are properties of the file, whereas
  - the two effective IDs and the supplementary group IDs are properties of the process
- Remember that
  - the st_mode value also encodes the access permission bits for the file (which in turn include the set-user-ID bit and the set-group-ID bit)

Summing up: File Access Permissions

The tests performed by the kernel are as follows (in sequence)

1. If the effective user ID of the process is 0 (i.e., the superuser), access is allowed. This gives the superuser free rein throughout the entire file system
2. If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the file), access is allowed if the appropriate user access permission bit is set. Otherwise, permission is denied

   By *appropriate access permission bit*, we mean that if the process is opening the file for reading, the user-read bit must be on. If the process is opening the file for writing, the user-write bit must be on. If the process is executing the file, the user-execute bit must be on
3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied
4. If the appropriate "other" access permission bit is set, access is allowed. Otherwise, permission is denied

Reading the IDs of a Process

```
#include <unistd.h>

pid_t getpid(void);
```

Returns: process ID of calling process

```
pid_t getppid(void);
```

Returns: parent process ID of calling process

```
uid_t getuid(void);
```

Returns: real user ID of calling process

```
uid_t geteuid(void);
```

Returns: effective user ID of calling process

```
gid_t getgid(void);
```

Returns: real group ID of calling process

```
gid_t getegid(void);
```

Returns: effective group ID of calling process

Reading the IDs of a Process

- E.g., getpid returns a pid_t data type
  - we don't know its size; all we know is that the standards guarantee that it will fit in a long integer
  - because we have to tell printf the size of each argument to be printed, we have to cast the value to the largest data type that it might use (in this case, a long integer, thus %ld)
  - although most process IDs will fit in an int, using a long promotes portability

hellofrompidbis.c 〉 No Selection

```c
#include "apue.h"

int
main(void)
{
    printf("\nHello World from the Process having ID: %ld\n", (long)getpid());

    printf("My Parent Process ID is: %ld\n", (long)getppid());

    printf("My Real User ID is: %ld\n", (long)getuid());

    printf("My Effective User ID is: %ld\n", (long)geteuid());

    printf("My Real Group ID is: %ld\n", (long)getgid());

    printf("My Effective Group ID is: %ld\n\n", (long)getegid());

    exit(0);
}
```

Setting the real user ID and effective user ID

- We can set both the real user ID and effective user ID with the setuid() function

  (same for group with the setgid() function)

```
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

Both return: 0 if OK, –1 on error

uid_t and gid_t are integer data types used to represent user and group IDs, resp.

There are rules for who can change the IDs

- If the process has superuser privileges,
  - the setuid function sets the real user ID, effective user ID, and saved set-user-ID to uid.
- If the process does not have superuser privileges, but uid equals either the real user ID or the saved set-user-ID,
  - setuid sets only the effective user ID to uid. The real user ID and the saved set-user-ID are not changed.
- If neither of these two conditions is true,
  - errno is set to EPERM and –1 is returned.

See `#define EPERM 1  /* Operation not permitted */` into the header file `/usr/include/asm/errno.h`

Here, it is assumed that _POSIX_SAVED_IDS is true. If this feature isn't provided (as in old system versions), then delete all preceding references to the saved set-user-ID

Setting the real user ID and effective user ID

We can make a few statements about the three user IDs that the kernel maintains.

1. Only a superuser process can change the real user ID. Normally, the real user ID is set by the `login`(1) program when we log in and never changes. Because `login` is a superuser process, it sets all three user IDs when it calls `setuid`.
2. The effective user ID is set by the `exec` functions only if the set-user-ID bit is set for the program file. If the set-user-ID bit is not set, the `exec` functions leave the effective user ID as its current value. We can call `setuid` at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we can't set the effective user ID to any random value.
3. The saved set-user-ID is copied from the effective user ID by `exec`. If the file's set-user-ID bit is set, this copy is saved after `exec` stores the effective user ID from the file's user ID.

Figure 8.18 summarizes the various ways these three user IDs can be changed.

ID	`exec`		`setuid(`*uid*`)`	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged user
real user ID	unchanged	unchanged	set to *uid*	unchanged
effective user ID	unchanged	set from user ID of program file	set to *uid*	set to *uid*
saved set-user ID	copied from effective user ID	copied from effective user ID	set to *uid*	unchanged

Figure 8.18 Ways to change the three user IDs

Setting the real user ID and effective user ID

setreuid and setregid functions

- Historically, BSD supported the swapping of the real user ID and the effective user ID with the setreuid function

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);
```

Both return: 0 if OK, –1 on error

seteuid and setegid functions

- POSIX.1 includes the two functions seteuid and setegid. These functions are similar to setuid and setgid, but only the effective user ID or effective group ID is changed

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
```

Both return: 0 if OK, –1 on error

Summing up (see the example in Sec. 8.11, page 259)

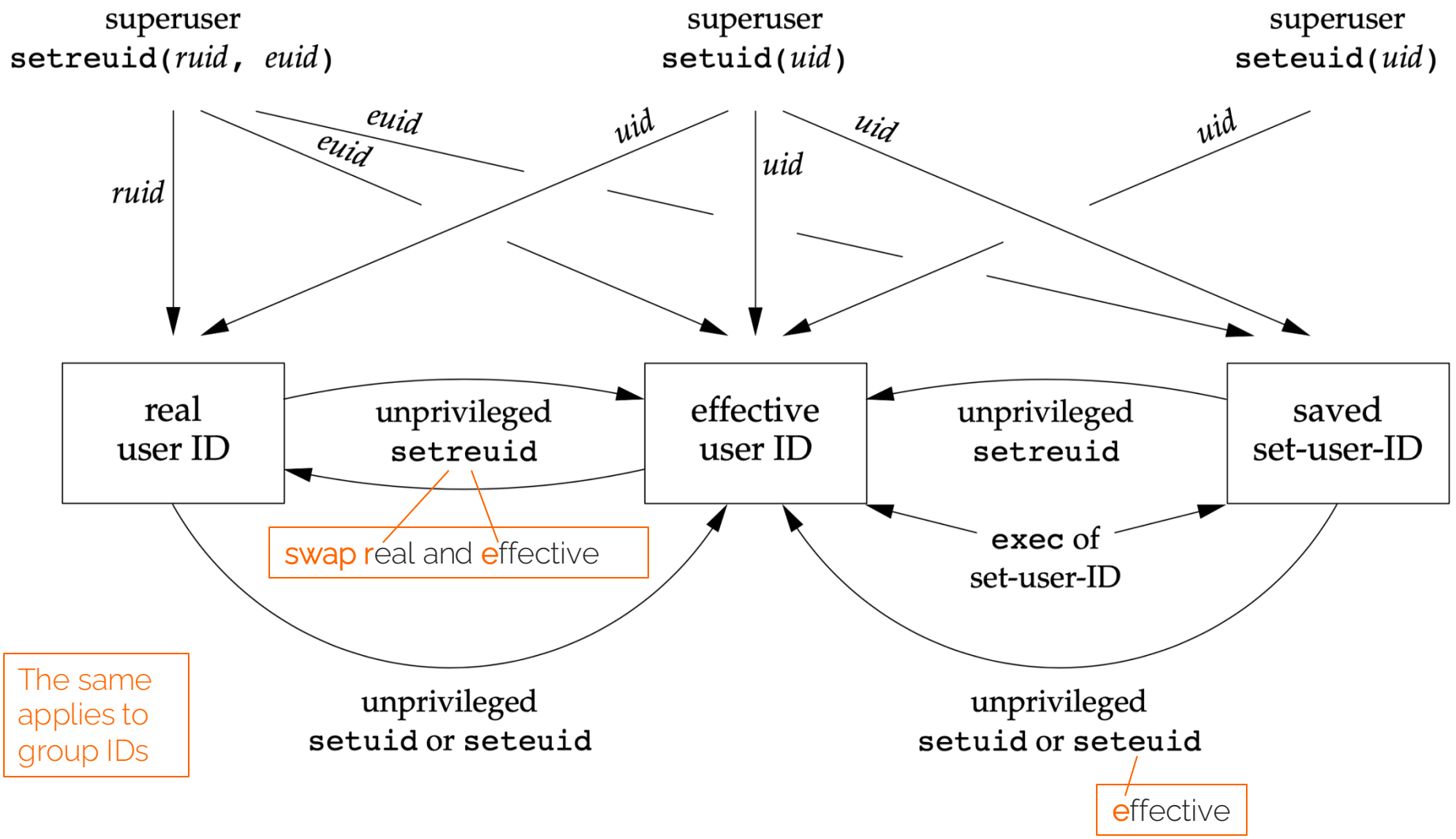


Figure 8.19 Summary of all the functions that set the various user IDs

Change the Persona of a Process: an example

Imagine a game program (say mygame) that saves scores in a file (say scores)

- The game program itself needs to be able to update this file no matter who is running it, but if players can write the file without going through the game program, they can give themselves any scores they like

- This can be prevented by creating a new user ID and login name (say, games) to own the scores file, and make the file writable only by this user (thus, the scores file is given mode 644)

```
-rw-r--r-- 1 games 0 Jul 31 15:33 scores
```

- The executable file mygame will be owned by games too, and it will have the setuid bit set (thus, it will be given mode 4755)

```
-rwsr-xr-x   1 games    184422 Jul 30 15:17 mygame
```

- Then, when during execution the game program mygame wants to update the scores file, it can change its effective user ID to be that of games
- In effect, whoever player is running it, the game program mygame must adopt the persona of games so it can write the scores file

Game program example

A typical setuid program does not need its special access all of the time. It is a good idea to turn off this access when it is not needed, so it cannot possibly give unintended access

- If the system supports the _POSIX_SAVED_IDS feature, you can accomplish this with seteuid
- When the game program mygame starts, its real user ID is pippo, its effective user ID is games, and its saved user ID is also games. The program should record both user ID values once at the beginning, like this:

```
user_user_id = getuid ();
game_user_id = geteuid ();
```

- Then, it can turn off game file access with

```
seteuid (user_user_id);
```

- and turn it on with

```
seteuid (game_user_id);
```

- Throughout this process, the real user ID remains pippo and the file user ID remains games, so the program can always set its effective user ID to either one

Game program example

- If the system DOES NOT support the _POSIX_SAVED_IDS feature, you can turn the access on and off by using setreuid() to swap the real and effective user IDs of the process, as follows:

```
setreuid ( geteuid(), getuid() );
```

- This special case is always allowed - it cannot fail
- Why does this have the effect of toggling the SETUID access? Suppose a game program has just started, and its real user ID is pippo while its effective user ID is games. In this state, the game can write the scores file. If it **swaps** the two uids, the real becomes games and the effective becomes pippo; now the program has only pippo access. Another swap brings games back to the effective user ID and restores access to the scores file
- In order to handle both kinds of systems, test for the saved user ID feature with a preprocessor conditional, like this:

```
#ifdef  _POSIX_SAVED_IDS
   seteuid (user_user_id);
#else
   setreuid (geteuid (), getuid ());
#endif
```

See chap8-proc/mygame/mygame.c example and next slide

For the example to be more meaningful, you would need to have at least two user accounts in your PC and then put the files in a shared folder (e.g., under macOS, the folders are /Users/marcoautili/Public and /Users/Shared

Game program example @work

- As already said, the program assumes that its executable file will be “installed” with the setuid bit set and owned by the same user as the scores file

- Typically, a system administrator will set up an ad-hoc account like games for this purpose

- Then you may also create another user account in your PC and then put the files in a shared folder

- As you will see in next slide, to simplify matter, I used the root account and my marcoautili account… you can do better by creating the ad-hoc account games ☺

Game program example @work

```
mygame — -zsh — 89×20
marcoautili@Marcos-MacBook-Pro-2020 mygame %
marcoautili@Marcos-MacBook-Pro-2020 mygame %
marcoautili@Marcos-MacBook-Pro-2020 mygame % ls -l
total 120
-rwxr-xr-x  1 marcoautili  staff  50040 Dec  3 08:51 mygame
-rwxr-xr-x@ 1 marcoautili  staff   2902 Dec 17  2020 mygame.c
-rwxr-xr-x  1 marcoautili  staff    304 Dec  3 12:48 scores
marcoautili@Marcos-MacBook-Pro-2020 mygame %
marcoautili@Marcos-MacBook-Pro-2020 mygame %
marcoautili@Marcos-MacBook-Pro-2020 mygame % ./mygame
The Real User ID is: 501 and is name is: marcoautili
The Effective User ID is: 501 and is name is: marcoautili

Inside the function record_score() the effective user name is: marcoautili

Inside the function record_score() the effective user name is: marcoautili

Inside the function record_score() the effective user name is: marcoautili
marcoautili@Marcos-MacBook-Pro-2020 mygame %
marcoautili@Marcos-MacBook-Pro-2020 mygame %
```

This is NOT the way to go → see next slide

Game program example @work

```
bash-3.2$ make
gcc -I../include -Wall -DMACOS -D_DARWIN_C_SOURCE  mygame/mygame.c -o mygame/mygame  -L../lib -lapue
bash-3.2$
bash-3.2$ ls -l mygame
total 120
-rwxr-xr-x  1 marcoautili  staff  50040 Dec  3 08:51 mygame
-rwxr-xr-x@ 1 marcoautili  staff   2902 Dec 17  2020 mygame.c
-rw-r--r--@ 1 marcoautili  staff    152 Dec  2 22:39 scores
bash-3.2$
bash-3.2$ sudo cp ./mygame/mygame ./mygame-installation/mygame-inst
Password:
bash-3.2$
bash-3.2$ cd mygame-installation/
bash-3.2$ sudo touch scores
Password:
bash-3.2$
bash-3.2$
bash-3.2$ ls -l
total 104
-rwxr-xr-x  1 root  staff  50040 Dec  3 08:53 mygame-inst
-rw-r--r--  1 root  staff      0 Dec  3 08:58 scores
bash-3.2$
```

In alternative, simply copy mygame into mygame-installation and then change its owner: sudo chown root mygame

Now root is the owner of both mygame and scores

Game program example @work

```
bash-3.2$
bash-3.2$ whoami
marcoautili
bash-3.2$
bash-3.2$ chmod 4755 mygame-inst
chmod: Unable to change file mode on mygame-inst: Operation not permitted
bash-3.2$ sudo chmod 4755 mygame-inst
bash-3.2$
bash-3.2$ ls -l
total 104
-rwsr-xr-x  1 root  staff  50040 Dec  3 08:53 mygame-inst
-rw-r--r--  1 root  staff      0 Dec  3 08:58 scores
bash-3.2$
bash-3.2$ ./mygame-inst
The Real User ID is: 501 and is name is: marcoautili
The Effective User ID is: 0 and is name is: root

Inside the function record_score() the effective user name is: root

Inside the function record_score() the effective user name is: root

Inside the function record_score() the effective user name is: root
bash-3.2$
bash-3.2$
```

Do not forget that I am marcoautili ☺

root is the owner… not marcoautili!

Now, mygame-inst is a set-user-ID program, thus fine

This is the way to go ☺

Finding out the login name (more on)

```
#include <unistd.h>

char *getlogin(void);
```

Returns: pointer to string giving login name if OK, `NULL` on error

The mygame.c example uses the getpwuid(getuid()) function to find out the login name of the user who is running the program

- but what if a single user has multiple login names, each with the same user ID?
  - A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry
  - The system normally keeps track of the name we log in under (see Section 6.8), and the getlogin function provides a way to fetch that login name
- Given the login name, we can then use it to look up the user in the password file (for example, to determine the login shell) using getpwnam() function (see next slide)

See Section 8.15 User Identification

User identification (more on)

Given the login name, we can then use it to look up the user in the password file—to determine the login shell, for example—using `getpwnam`.

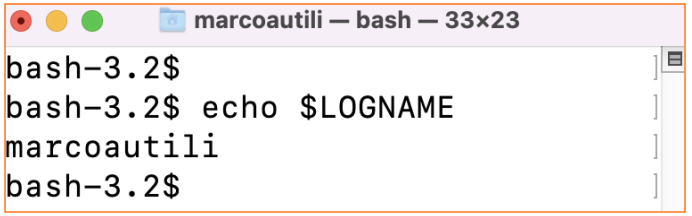
To find the login name, UNIX systems have historically called the `ttyname` function (Section 18.9) and then tried to find a matching entry in the `utmp` file (Section 6.8). FreeBSD and Mac OS X store the login name in the session structure associated with the process table entry and provide system calls to fetch and store this name.

System V provided the `cuserid` function to return the login name. This function called `getlogin` and, if that failed, did a `getpwuid(getuid())`. The IEEE Standard 1003.1-1988 specified `cuserid`, but it called for the effective user ID to be used, instead of the real user ID. The 1990 version of POSIX.1 dropped the `cuserid` function.

The environment variable `LOGNAME` is usually initialized with the user's login name by `login`(1) and inherited by the login shell. Realize, however, that a user can modify an environment variable, so we shouldn't use `LOGNAME` to validate the user in any way. Instead, we should use `getlogin`.

System V is one of the first commercial versions of the Unix operating system

Check if cuserid() is still available under Linux...



Process Control

Process control concerns

- program execution
- the creation of new processes
- process termination

Process control primitives

- exec functions to initiate new programs
- fork to create new processes
- wait functions waiting for termination
- exit function to handle termination

Fork function

An existing process can create a new one by calling the `fork` function.

```
#include <unistd.h>

pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, –1 on error

- The new process created by fork is called the child process
- This function is called once but returns twice
  - the return value in the child is 0
  - the return value in the parent is the process ID of the new child
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children
- The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent (since the process ID 0 is reserved for use by the kernel to the scheduler process, it is not possible for 0 to be the process ID of a child)

Execution after a fork

- Both the child and the parent continue executing with the instruction that follows the call to fork
  - The child is a copy of the parent - the child gets a copy of the parent's data space, heap, and stack
  - Important Note – the data space is a copy for the child; thus, the parent and the child DO NOT share these portions of memory

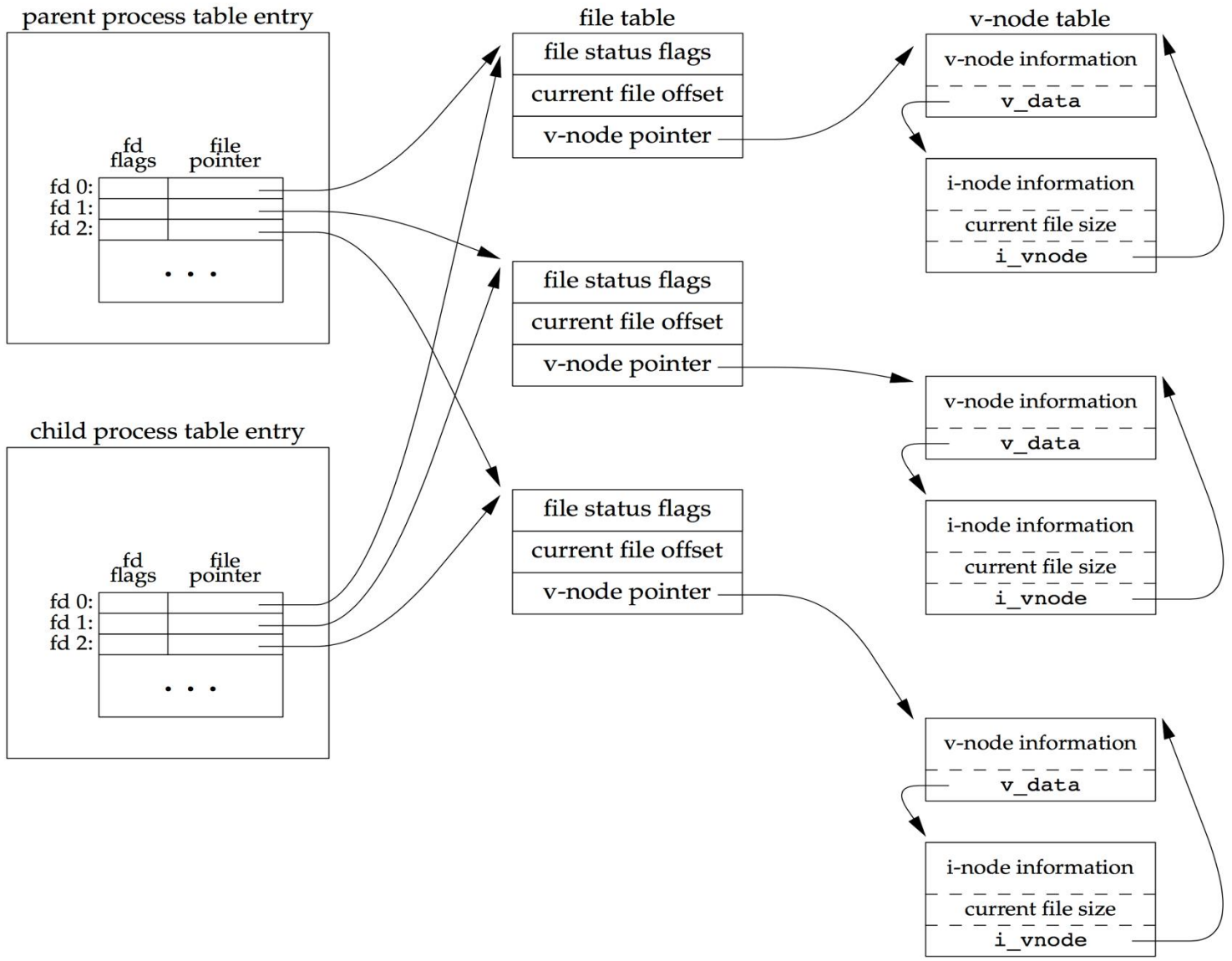
- In general, we never know whether the child starts executing before the parent, or vice versa
  - The order depends on the scheduling algorithm used by the kernel. If it is required that the child and parent synchronize their actions, some form of Signaling and/or Inter Process Communication (IPC) might be required
  - In very simple cases, might be enough to put the parent to sleep for few seconds, to let the child execute… however, there is NO guarantee that the length of this delay is adequate

- CHILD_MAX specifies the maximum number of simultaneous processes per real user ID

File sharing between parent and child

- One characteristic of the fork is that all file descriptors that are open in the parent are duplicated in the child
  - We say "duplicated" because it is as if the dup function had been called for each descriptor
- The parent and the child share a file table entry for every open descriptor
- Consider a process that has three different files opened for standard input, standard output, and standard error
  - On return from fork, we have the arrangement shown in next slide

File sharing between parent and child



Example fork1.c

```
chap8-proc — -bash — 68×15
desktop-jisqlks:chap8-proc marcoautili$ ./fork1
A write to stdout
before fork
I am the parent, my pid is 6799  and I go to sleep for a while!
I am the child and my pid is 6800
pid = 6800, glob = 7, var = 89
pid = 6799, glob = 6, var = 88
desktop-jisqlks:chap8-proc marcoautili$
```

NO redirection

write to standard out

→ Line buffered

- Note that changes to variables in a child process do not affect the value of the variables in the parent process

See the example fork1.c into the folder chap8-proc

```
chap8-proc — -bash — 68×18
desktop-jisqlks:chap8-proc marcoautili$ ./fork1 > output-file.txt
desktop-jisqlks:chap8-proc marcoautili$ cat output-file.txt
A write to stdout
before fork
I am the child and my pid is 6814
pid = 6814, glob = 7, var = 89
before fork
I am the parent, my pid is 6813  and I go to sleep for a while!
pid = 6813, glob = 6, var = 88
desktop-jisqlks:chap8-proc marcoautili$
```

Once again! Why? ☹

WITH redirection:

write to a file

→ Fully buffered

Fork uses

1. When a process wants to duplicate itself so that the parent and the child can each execute different sections of code "at the same time"

   - This is common for network servers -- the parent waits for a service request from a client
   - When the request arrives, the parent calls fork and lets the child handle the request
   - The parent goes back to waiting for the next service request to arrive

2. When a process wants to execute a different program

   - This is common for shells. In this case, the child does an exec right after it returns from the fork
   - Some operating systems combine the fork and exec operations (a fork followed by an exec) into a single operation called a spawn

A curiosity: fork VS vfork

8.4 `vfork` Function

The function `vfork` has the same calling sequence and same return values as `fork`, but the semantics of the two functions differ.

> The `vfork` function originated with 2.9BSD. Some consider the function a blemish, but all the platforms covered in this book support it. In fact, the BSD developers removed it from the 4.4BSD release, but all the open source BSD distributions that derive from 4.4BSD added support for it back into their own releases. The `vfork` function was marked as an obsolescent interface in Version 3 of the Single UNIX Specification and was removed entirely in Version 4. We include it here for historical reasons only. Portable applications should not use it.

The `vfork` function was intended to create a new process for the purpose of executing a new program (step 2 at the end of the previous section), similar to the method used by the bare-bones shell from Figure 1.7. The `vfork` function creates the new process, just like `fork`, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls `exec` (or `exit`) right after the `vfork`. Instead, the child runs in the address space of the parent until it calls either `exec` or `exit`. This optimization is more efficient on some implementations of the UNIX System, but leads to undefined results if the child modifies any data (except the variable used to hold the return value from `vfork`), makes function calls, or returns without calling `exec` or `exit`. (As we mentioned in the previous section, implementations use copy-on-write to improve the efficiency of a `fork` followed by an `exec`, but no copying is still faster than some copying.)

Another difference between the two functions is that `vfork` guarantees that the child runs first, until the child calls `exec` or `exit`. When the child calls either of these functions, the parent resumes. (This can lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.)

We will see the bare-bones shell in later slides

Program execution

The `main()` function

- When a C program is executed by the kernel (by one of the exec functions) a special start-up routine is called before the main() function is called

- The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler

- This start-up routine takes values from the kernel (the command-line arguments and the environment) and sets things up so that the main() function is called

Command-Line Arguments

- When a program is executed, the process that does the exec can pass command-line arguments to the new program

- This is part of the normal operation of the UNIX system shells

See the source code echoarg.c into the folder chap8-proc

```
#include "apue.h"

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

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

Figure 7.4 Echo all command-line arguments to standard output

Environment List

- Each program is thus passed an *environment list*

- Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string

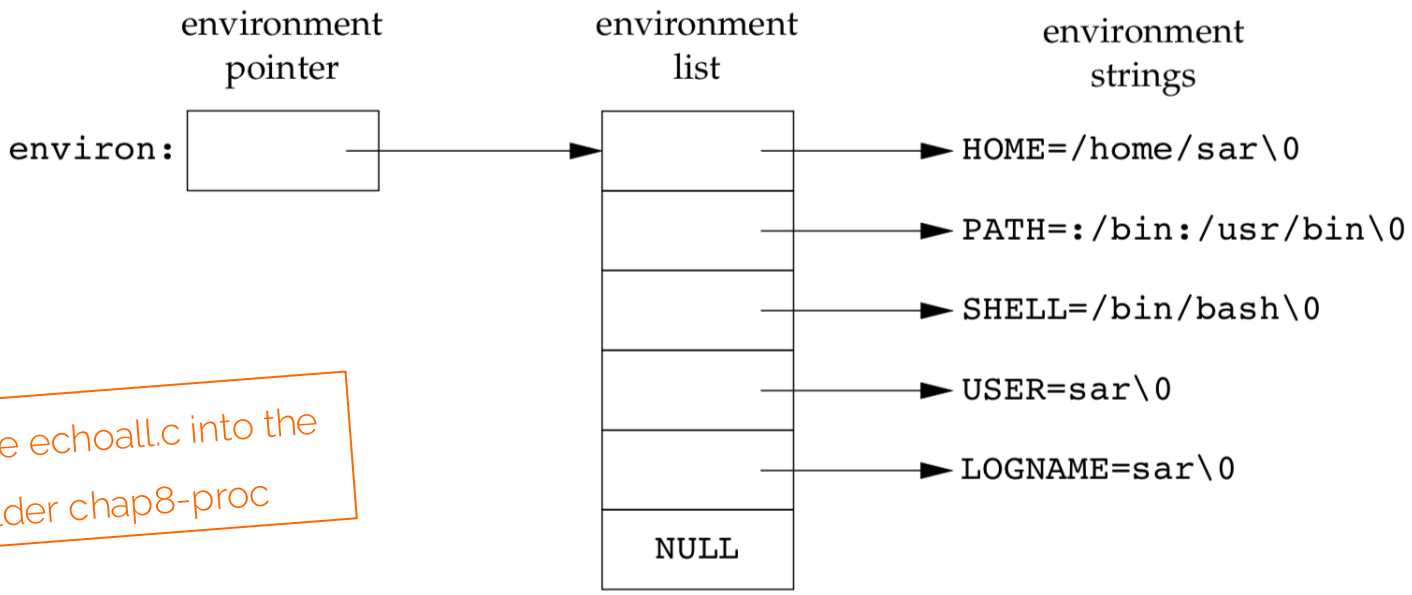
- The address of the array of pointers is contained in the global variable `environ`:

```
extern char **environ;
```

- By convention, the environment consists of *name =value* strings,
  as shown in Figure 7.5 (next slide)

Accessing the Environment List

- Access to specific environment variables is normally done through the getenv and putenv functions, described in Section 7.9



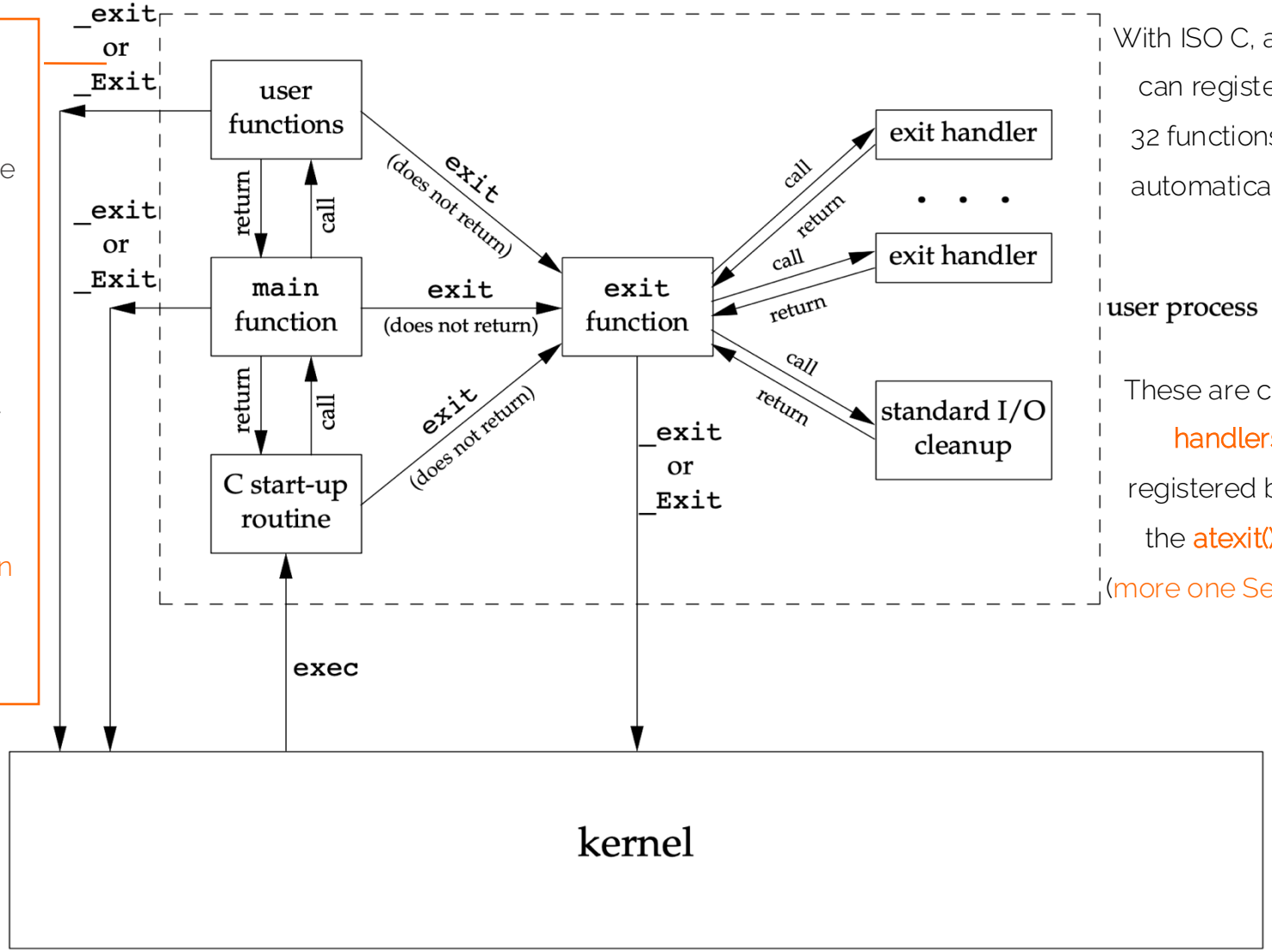
See echoall.c into the folder chap8-proc

Figure 7.5 Environment consisting of five C character strings

Program execution and termination schema

ISO C defines _Exit (or equiv. _exit) to provide a way for a process to terminate without running exit handlers or signal handlers

More on Section "8.5 exit Functions"



With ISO C, a process can register at least 32 functions that are automatically called by exit()

These are called exit handlers and are registered by calling the atexit() function (more one Section 7.3)

Figure 7.2 How a C program is started and how it terminates

The exec function(s)

- exec is a functionality of an operating system that runs an executable file in the context of an already existing process, replacing the previous executable
  - this act is also referred to as an overlay

- It is especially important in Unix-like systems, although exists elsewhere
  - as a new process is not created, the process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program

Let's see a concrete example as first... →

Process Control (see chap1-intro)

shell1.c

```c
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

Very basic implementation of a shell-like program

There are three primary functions for process control: fork, exec, and waitpid

The exec function has seven variants, but we often refer to them collectively as simply the exec function(s)

Process Control (see chap1-intro)

shell1.c

```c
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

The standard I/O function fgets reads one line at a time from the standard input

When we type the end-of-file character (which is often Control-D) as the first character of a line, fgets returns a null pointer, the loop stops, and the process terminates

Process Control (see chap1-intro)

shell1.c

```c
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

(char *)0 guarantees to produce a *null-pointer value* of type char *

Because each line returned by fgets is terminated with a newline character, we use the standard C function strlen to calculate the length of the string, and then replace the newline with a null byte

We do this because the execlp function wants a null-terminated argument, not a newline-terminated argument

More on execlp later…

Process Control (see chap1-intro)

shell1.c

```c
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

In the child, we call execlp to execute the command that was read from the standard input. This replaces the child process with the new program file (→ next slides)

As already said, on some operating systems, the combination of fork followed by exec is called spawning a new process

More on the exec functions in later slides

Process Control (see chap1-intro)

shell1.c

```c
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

← 127
command not found

Because the child calls execlp to execute the new program file, the parent wants to wait for the child to terminate

This is done by calling waitpid, specifying which process to wait for: the pid argument, which is the process ID of the child

More on waitpid later...

Process Control (see chap1-intro)

shell1.c

```c
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

The most fundamental limitation of this program is that we can't pass arguments to the command we execute

Since we are using the execlp function, to allow arguments would require that we parse the input line, separating the arguments by some convention, probably spaces or tabs, and then pass each argument as a separate parameter to the execlp function

The exec function(s)

- In many UNIX system implementations, only one of these seven functions, i.e., execve, is a system call within the kernel
- The other six are "just" library functions that eventually invoke this system call

Mark the end of the arguments with a null pointer

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

See Section 8.10

All seven return: –1 on error, no return on success

Not that difficult to remember…

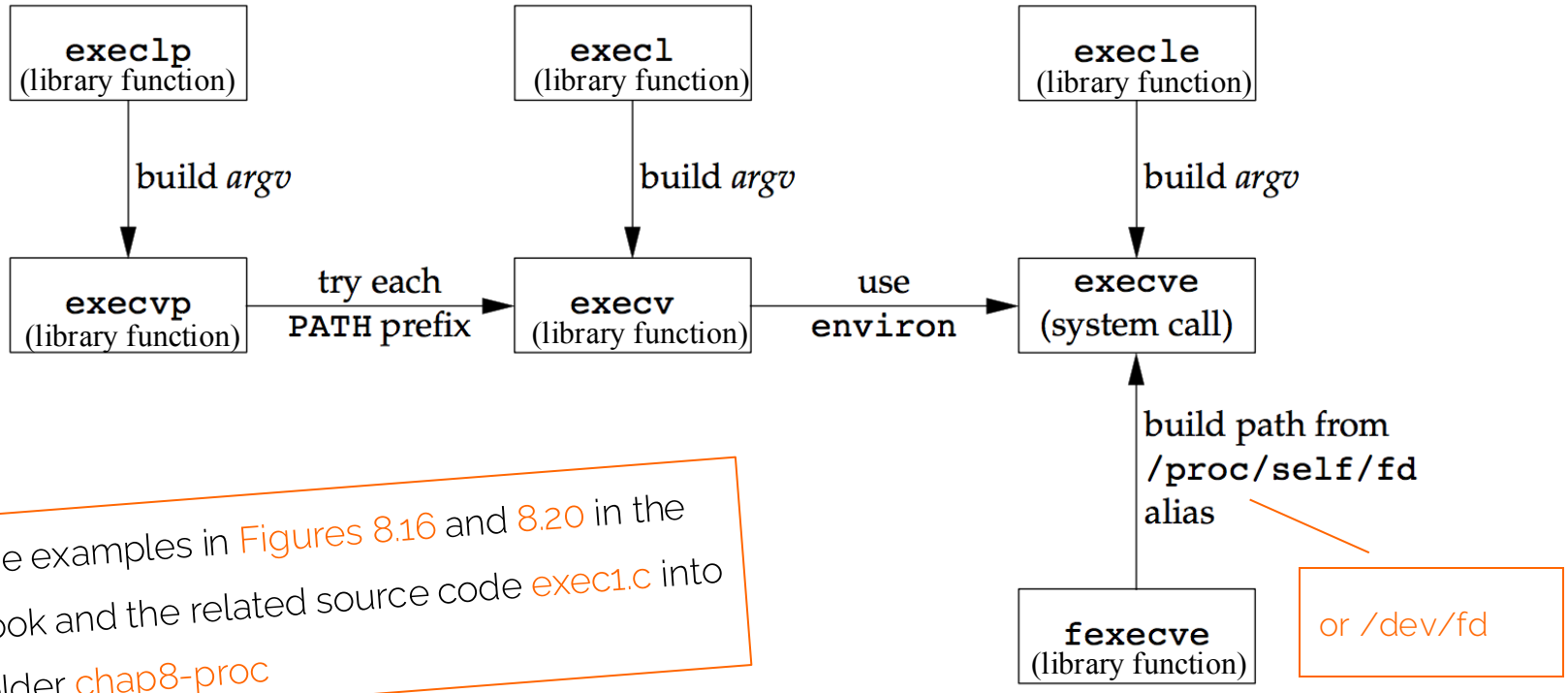
The arguments for these seven exec functions are difficult to remember

- The letter p means that the function takes a *filename* argument and uses the *PATH environment variable* to find the executable file
- The letter l means that the function takes a *list of* arguments and is mutually exclusive with the letter v, which means that it takes an *argv* [] vector
- Finally, the letter e means that the function takes an *envp* [] array instead of using the current environment

Function	*pathname*	*filename*	*fd*	Arg list	*argv*[]	environ	*envp*[]
execl	•			•		•	
execlp		•		•		•	
execle	•			•			•
execv	•				•	•	
execvp		•			•	•	
execve	•				•		•
fexecve			•		•		•
(letter in name)		p	f	l	v		e

Figure 8.14 Differences among the seven exec functions

More on the exec function(s)



See the examples in Figures 8.16 and 8.20 in the textbook and the related source code exec1.c into the folder chap8-proc

Figure 8.15 Relationship of the seven **exec** functions

Back to the shebang

A shebang (also called sha-bang, hashbang, pound-bang, or hash-pling) is the character sequence consisting of the characters number sign and exclamation mark (#!) at the beginning of a script

Interpreter Files

- All contemporary UNIX systems support interpreter files
- These files are text files that begin with a line of the form

```
#!pathname [ optional-argument ]
```

  (the space between the exclamation point and the pathname is optional)

- The most common of these interpreter files begin with the line

```
#!/bin/sh
```

See exec2.c into the folder chap8-proc

Process control primitives @work

- The `system` function is for executing a command string from within a program (SECTION 8.13)
  - e.g., `system("date > file");`

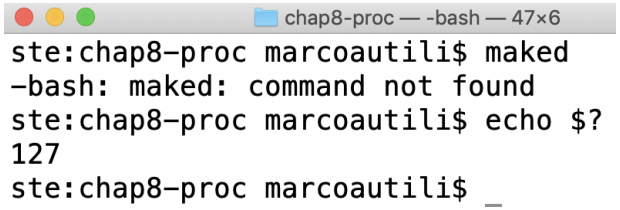
```
#include <stdlib.h>

int system(const char *cmdstring);

                                        Returns: (see below)
```

- ISO C defines the system function, but its operation is strongly system dependent
- POSIX.1 includes the system interface, expanding on the ISO C definition to describe its behavior in a POSIX environment

- If *cmdstring* is a null pointer, the function returns nonzero only if a command processor is available
  - this characteristic can be used to determine whether the system function is supported on a given operating system (under the UNIX System, system is always available)
- Because the function is implemented by calling fork, exec, and waitpid, there are three types of return values

man errno

1. If either the fork fails or waitpid returns an error other than EINTR, system returns –1 with errno set to indicate the error
2. If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127)



3. Otherwise, all three functions fork, exec, and waitpid succeed, and the return value from system is the termination status of the shell, in the format specified for waitpid

Process control primitives @work

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

int
system(const char *cmdstring)    /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

- This implementation doesn't handle signals
- The version with signal handling can be found in Section 10.18

On UNIX systems, _Exit and _exit are synonymous and do not flush standard I/O streams

That's why we call _exit instead of exit

We do this to prevent any standard I/O buffers (which would have been copied from the parent to the child across the fork) from being flushed in the child

If the -c option is present, then commands are read from string. If there are arguments, they are assigned to the positional parameters, starting with $0

Macro: int EINTR "Interrupted system call"

An asynchronous signal occurred and prevented the completion of the call. When this happens, you should try the call again

Figure 8.22 The system function, without signal handling

Process termination

- A process can terminate normally or abnormally in different way, e.g., through exit, return, abort
  - Returning an integer value from the main() function is equivalent to calling exit with the same value:

    for example, exit(0); is the same as return(0); from the main function

- Regardless of how a process terminates, the "same" code in the kernel is eventually executed
  1. This kernel code closes all the open descriptors for the process, releases the memory that it was using, and so on
  2. In the case of an abnormal termination, however, the kernel (not the process) generates a termination status to indicate the reason for the abnormal termination (whether a core file is generated is left up to the implementation)
  3. In any case, the parent of the process can obtain the termination status from either the wait or the waitpid function (see next slides)

Process termination

1. If the parent terminates before the child

- the child is said to be an orphan process
    - i.e., a process that is still executing, but whose parent has died

- The init process becomes the parent process of any process whose parent terminates, i.e., the process has been inherited by init

- Whenever a process terminates, the kernel scans all active processes to see whether the terminating process is the parent of any process that still survives
    - if so, the parent process ID of the surviving process is changed to be 1 (the process ID of init)
    - this way, we are guaranteed that every process has a parent

Process termination

2. If the child terminates before the parent, and the parent is not (yet) waiting for it (or is not programmed to wait for it)
   - the child is said to be a zombie process (or defunct process), i.e., a process that has completed its execution (e.g., via the exit system call) but still has an entry in the process table (as created and kept by the Kernel)
     - it is a process in the "Terminated state"
     - this occurs for child processes, where the entry is still needed to allow the parent process to read its child's exit status
     - once the exit status is read via the wait system call, the zombie's entry is removed from the process table, and it is said to be "reaped"
   - A child process always first becomes a zombie before being removed from the resource table
     - in most cases, under normal system operation, zombies are immediately waited on by their parents and then reaped by the system
     - processes that stay zombies for a long time are generally an error and cause a resource leak

- As a result, a process that is both a zombie and an orphan will be reaped automatically

Process termination

3. What happens when a process that has been inherited by init terminates?

   Does it become a zombie?

   - The answer is "NO", because init is written so that whenever one of its children terminates, init calls one of the wait functions to fetch the termination status
   - By doing this, init prevents the system from being clogged by zombies

Later, we will see how to avoid zombie processes

wait and waitpid

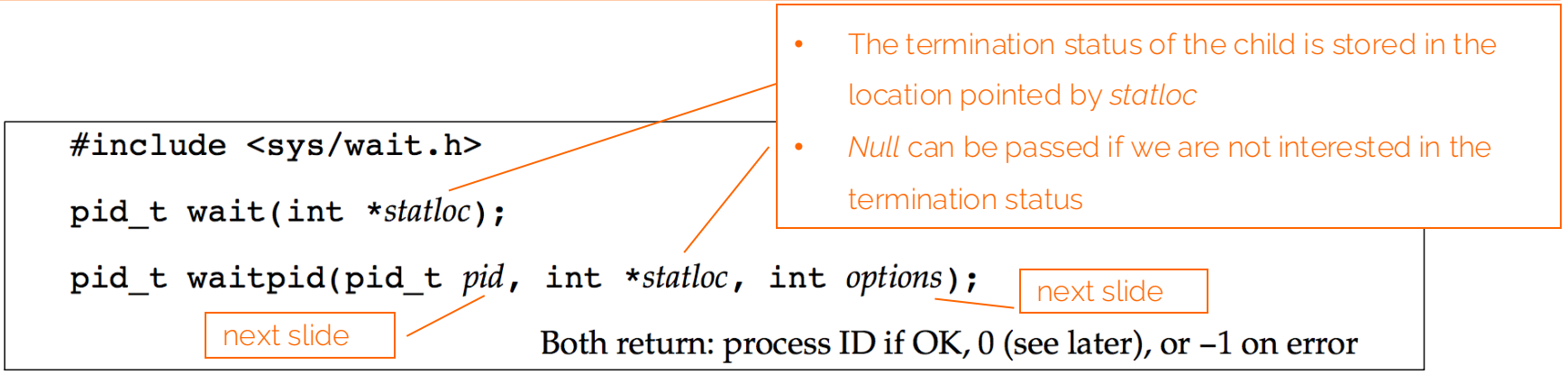
- When a process terminates, either normally or abnormally, the kernel notifies the parent process by sending the SIGCHLD signal to it
- Because the termination of a child is an asynchronous event - it can happen at any time while the parent is running
- This signal is the asynchronous notification from the kernel to the parent
- The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler (the default action for the SIGCHLD signal is to be ignored)

- We will see signals later on these slides. For now, let's focus on wait and waitpid

wait and waitpid

- A process that calls wait or waitpid can
  - Block, if all of its children are still running
  - Return immediately with the termination status of the child that terminates first
    (if a child has terminated and is waiting for its termination status to be fetched)
  - Return immediately with an error, if it does not have any child processes
- If the process is calling wait because it received the SIGCHLD signal, we expect wait to return immediately. But if we call it at any random point in time, it can block

wait and waitpid



- The wait function can block the caller until one of the child process terminates, whereas waitpid has an option that prevents it from blocking
  - If a child has already terminated and it is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates
  - If the caller blocks and has multiple children, wait returns when one terminates
- The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for
- We can always tell which child terminated because the function returns the process ID

waitpid

- The waitpid function returns the process ID of the child that terminated and stores the child's termination status in the memory location pointed to by *statloc*

- The interpretation of the *pid* argument for waitpid depends on its value

pid == −1	Waits for any child process. In this respect, `waitpid` is equivalent to `wait`.
pid > 0	Waits for the child whose process ID equals *pid*.
pid == 0	Waits for any child whose process group ID equals that of the calling process. (We discuss process groups in Section 9.4.)
pid < −1	Waits for any child whose process group ID equals the absolute value of *pid*.

More on waitpid (not part of the course program)

- The *options* argument lets us further control the operation of waitpid
- This argument either is 0 or is constructed from the bitwise OR of the following constants

FreeBSD 8.0 and Solaris 10 support one additional, but nonstandard, *option* constant. `WNOWAIT` has the system keep the process whose termination status is returned by `waitpid` in a wait state, so that it may be waited for again.

See Section "10.21 Job-Control Signals" for more on jb control

Constant	Description
`WCONTINUED`	If the implementation supports job control, the status of any child specified by *pid* that has been continued after being stopped, but whose status has not yet been reported, is returned (XSI option).
`WNOHANG`	The `waitpid` function will not block if a child specified by *pid* is not immediately available. In this case, the return value is 0.
`WUNTRACED`	If the implementation supports job control, the status of any child specified by *pid* that has stopped, and whose status has not been reported since it has stopped, is returned. The `WIFSTOPPED` macro determines whether the return value corresponds to a stopped child process.

XSI System Interfaces and Extensions

The base specifications of the Single UNIX Specification (published by The Open Group, formerly X/Open) have been merged with POSIX.1

The X/Open System Interface is the core application programming interface for systems conforming to the Single UNIX Specification. This is a superset of the mandatory requirements for conformance to IEEE Std 1003.1-2001

waitid

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

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
                                                    Returns: 0 if OK, –1 on error
```

- It is similar to waitpid, but provides extra flexibility
- Like waitpid, waitid allows a process to specify which children to wait for. Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used.
- The *id* parameter is interpreted based on the value of *idtype*
- The types supported are summarized in Figure 8.9

Constant	Description
P_PID	Wait for a particular process: *id* contains the process ID of the child to wait for.
P_PGID	Wait for any child process in a particular process group: *id* contains the process group ID of the children to wait for.
P_ALL	Wait for any child process: *id* is ignored.

Figure 8.9 The *idtype* constants for `waitid`

More on waitid (not part of the course program)

- The *options* argument is a bitwise OR of the flags shown in Figure 8.10
- These flags indicate which state changes the caller is interested in
- At least one of WCONTINUED, WEXITED, or WSTOPPED must be specified in the *options* argument.
- The *infop* argument is a pointer to a siginfo structure. This structure contains detailed information about the signal generated that caused the state change in the child process. The siginfo structure is discussed further in Section 10.14

Constant	Description
WCONTINUED	Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported.
WEXITED	Wait for processes that have exited.
WNOHANG	Return immediately instead of blocking if there is no child exit status available.
WNOWAIT	Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to wait, waitid, or waitpid.
WSTOPPED	Wait for a process that has stopped and whose status has not yet been reported.

Figure 8.10 The *options* constants for waitid

There are also additional versions of the wait function: see Section 8.8 for wait3 and wait4

How to avoid zombie processes

- If we want to write a process that forks a child, but we do not want to wait for the child to complete and we do not want the child to become a zombie until we terminate, a possible trick could be to call fork twice. It often works, but not always... for the purpose of teaching, it is however useful ☺
- Let's play with the fork2.c example in the folder chap8-proc

```
chap8-proc — -zsh — 107×19

marcoautili@iMac-di-User chap8-proc %
marcoautili@iMac-di-User chap8-proc %
marcoautili@iMac-di-User chap8-proc % ./fork2

 After the "first" fork, I am the "first child": my pid is 7500


 I will immediately execute a second fork...

 After the "second fork", I am still the "first child", hence my pid is still 7500


 I am the "parent" (i.e., the "original process"): my pid is 7499 and my parent pid is 6923

marcoautili@iMac-di-User chap8-proc %
 I am the "second child": my pid is 7501 and my parent pid is 1
```

Read the last comment in the file fork2.c and observe this behavior: It seems that the process is not yet terminated... indeed, it is!

Race conditions

- A race condition can occur when multiple processes are trying to do something with shared data/resources and the final outcome depends on the order in which the processes run

- The fork function is a lively breeding ground for race conditions, if any of the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork

- In general, we cannot predict (and we should not assume) which process runs first!
  - play with the previous example and try to uncomment the various sleeps within the code

Race conditions

tellwait1.c

```c
#include "apue.h"

static void charatatime(char *);

int main(void) {
    pid_t   pid;
    int i,j;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        for (i = 0; i < 1000; i++) {
            charatatime("\n output from child \n");
        }
    } else {
        for (j = 0; j < 1000; j++) {
            charatatime("\n output from parent \n");
        }
    }
    exit(0);
}

static void charatatime(char *str) {
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

Race condition on *stdout* between the parent process and the child process

chap8-proc — -bash — 41×49

```
output from parent

output from parent

output from parent

output from parent

output from parent

output from parent

output from parent

outpu
t  foruotmp upta rfernotm
c
h ioludt p
u
t  ofurtopmu tp afrreonmt  c
h
i lodu t
p
u to uftrpoumt  pfarroemn tc h
i
l do u
t
p uotu tfpruotm  fpraorme ncth i
l
d  o
u
t pouutt pfurto mf rpoamr ecnhti l
d

o
u topuuttp uftr ofmr opma rcehnitl d


o uotuptuptu tf rformo mp acrheinltd


  oouuttppuutt  ffrroomm  cphairledn t
```

Avoiding race conditions

- A parent process that wants to wait for a child to terminate must call one of the wait functions

- If a child process wants to wait for its parent to terminate, as in the previous example, a loop of the following form could be used 

```
while (getppid() != 1)
        sleep(1);
```

Until my father becomes init … I "wait by sleeping"

- The problem with this type of loop, called *polling*, is that it wastes CPU time, as the caller is awakened every second to test the condition

- To avoid race conditions and to avoid polling:
  - some form of signaling can be used between multiple processes
  - various forms of InterProcess Communication (IPC) can be used

Avoiding race conditions

```
TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) {            /* child */

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid());     /* tell parent we're done */
    WAIT_PARENT();              /* and wait for parent */

    /* and the child continues on its way ... */

    exit(0);
}

/* parent does whatever is necessary ... */

TELL_CHILD(pid);            /* tell child we're done */
WAIT_CHILD();               /* and wait for child */

/* and the parent continues on its way ... */

exit(0);
```

Later we'll study a possible implementation of TELL and WAIT that uses pipes

(in the textbook, see Figure 10.24, you also have another implementation that uses signals)

What we are going to do

Chapter 7

Process Environment

- main function
- Process termination
- Environment list

Chapter 8

Process Control

- Creation of new processes
- Program execution
- Process termination

Chapter 10

Signals

- kill
- raise
- alarm
- pause

Signals (intro)

- Signals are a technique used to notify a process that some condition (hardware or software) has occurred, e.g.,
  - if a process divides by zero, the SIGFPE (floating-point exception) signal is sent to the process
  - if a process executes an invalid memory is reference, SIGSEGV is generated

  Specifically, these are hardware conditions usually detected by the hardware, and the kernel is notified
- Signals are software interrupts
- Signals provide a way of handling asynchronous events
  - for example, a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely
- Many other conditions can generate signals, e.g., two terminal keys, called
  - SIGINT → the *interrupt key* (often the DELETE key or Control-C)
  - the *quit key* (often Control-backslash)

  are used to interrupt the currently running process

Signals (intro)

- The process has three choices for dealing with the signal

1. Ignore the signal - This option is not recommended for signals that denote a hardware exception, such as dividing by zero or referencing memory outside the address space of the process, as the results are undefined

2. Let the default action occur - For a divide-by-zero condition, the default is to terminate the process

3. Provide a function that is called when the signal occurs - (this is called ‘‘catching’’ the signal). By providing a function of our own, we’ll know when the signal occurs, and we can handle it as we wish

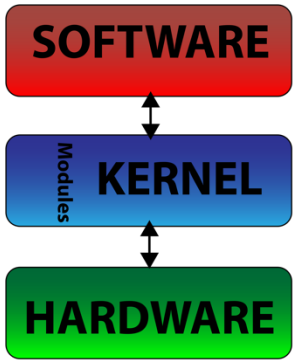
Signals (more on from PART 2)

- Signals are a limited form of inter-process communication (IPC), typically used in Unix, Unix-like, and other POSIX-compliant operating systems

- A signal is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred

- When a signal is sent, the operating system interrupts the target process' normal flow of execution to deliver the signal. Execution can be interrupted during any non-atomic instruction
  - If the process has previously registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed

- Signals are similar to interrupts, the difference being that
  - interrupts are mediated by the CPU and handled by the kernel
  - signals are mediated by the kernel (possibly via system calls) and handled by processes



https://en.wikipedia.org/wiki/Signal_(IPC)
https://en.wikipedia.org/wiki/Interrupt

Signals (more on from PART 2)

- The kernel may pass an "interrupt" as a "signal" to the process that caused it (typical examples are SIGSEGV, SIGBUS, SIGILL and SIGFPE)

- A process's execution may result in the generation of a hardware exception, for instance, if the process attempts to divide by zero or incurs a page fault

- Thus, if a process attempts to divide an integer by zero, a Floating-Point Exception is generated, and the kernel sends the SIGFPE signal to the process

- Also, if process makes an invalid virtual memory reference, or segmentation fault, i.e., it performs a SEGmentation Violation, the kernel sends the SIGSEGV signal to the process

- The exact mapping between signal names and exceptions is obviously dependent upon the CPU, since exception types differ between architectures

https://en.wikipedia.org/wiki/Signal_(IPC)

Signals

Every signal has a name

- These names all begin with the three characters SIG, e.g., (see Figure 10.1)
  - SIGABRT is the abort signal that is generated when a process calls the abort function
  - SIGALRM is the alarm signal that is generated when the timer set by the alarm function goes off
  - SIGPIPE is generated when a process writes to a pipe that has no reader

  Specifically, these are software conditions

- Signal names are all defined by positive integer constants (the signal number) in the header <signal.h>

> Implementations actually define the individual signals in a different header file, but this header file is included by `<signal.h>`. It is considered bad form for the kernel to include header files meant for user-level applications, so if the applications and the kernel both need the same definitions, the information is placed in a kernel header file that is then included by the user-level header file. Thus both FreeBSD 8.0 and Mac OS X 10.6.8 define the signals in `<sys/signal.h>`. Linux 3.2.0 defines the signals in `<bits/signum.h>`, and Solaris 10 defines them in `<sys/iso/signal_iso.h>`.

Signals

- Another way to generate a signal is by calling the kill function
- We can call this function from a process to send a signal to another process
    - Naturally, there are limitations: we have to be the owner of the other process (or the superuser) to be able to send it a signal
- No signal has a signal number of 0
- The kill function uses the signal number of 0 for a special case.
    - POSIX.1 calls this value the *null signal* (see in Section 10.9)

kill and *raise* functions (Section 10.9)

- The kill function sends a signal to a process or a group of processes

- The raise function allows a process to send a signal to itself

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

Both return: 0 if OK, –1 on error

The call

```
raise(signo);
```

is equivalent to the call

```
kill(getpid(), signo);
```

The *pid* argument to *kill*

There are four different conditions for the *pid* argument to `kill`.

pid > 0	The signal is sent to the process whose process ID is *pid*.
pid == 0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal. Note that the term *all processes* excludes an implementation-defined set of system processes. For most UNIX systems, this set of system processes includes the kernel processes and `init` (pid 1).
pid < 0	The signal is sent to all processes whose process group ID equals the absolute value of *pid* and for which the sender has permission to send the signal. Again, the set of all processes excludes certain system processes, as described earlier.
pid == –1	The signal is sent to all processes on the system for which the sender has permission to send the signal. As before, the set of processes excludes certain system processes.

The *alarm* function

- The *alarm* function allows us to set a timer that will expire at a specified time in the future

- When the timer expires, the SIGALRM signal is generated. If we ignore or don't catch this signal, its default action is to terminate the process.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Returns: 0 or number of seconds until previously set alarm

The *seconds* value is the number of clock seconds in the future when the signal should be generated. When that time occurs, the signal is generated by the kernel, although additional time could elapse before the process gets control to handle the signal, because of processor scheduling delays.

Earlier UNIX System implementations warned that the signal could also be sent up to 1 second early. POSIX.1 does not allow this behavior.

The *pause* function

```
#include <unistd.h>

int pause(void);
```

Returns: –1 with `errno` set to `EINTR`

- The pause function suspends the calling process until a signal is caught
  - The only time pause returns is if a signal handler is executed and that handler returns
  - In that case, pause returns –1 with errno set to EINTR

Signal function

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
```

Returns: previous disposition of signal (see following) if OK, `SIG_ERR` on error

See man signal

The `signal` function is defined by ISO C, which doesn't involve multiple processes, process groups, terminal I/O, and the like. Therefore, its definition of signals is vague enough to be almost useless for UNIX systems.

Implementations derived from UNIX System V support the `signal` function, but it provides the old unreliable-signal semantics. (We describe these older semantics in Section 10.4.) The `signal` function provides backward compatibility for applications that require the older semantics. New applications should not use these unreliable signals.

4.4BSD also provides the `signal` function, but it is defined in terms of the `sigaction` function (which we describe in Section 10.14), so using it under 4.4BSD provides the newer reliable-signal semantics. Most current systems follow this strategy, but Solaris 10 follows the System V semantics for the `signal` function.

Because the semantics of `signal` differ among implementations, we must use the `sigaction` function instead. We provide an implementation of `signal` that uses `sigaction` in Section 10.14. All the examples in this text use the `signal` function from Figure 10.18 to give us consistent semantics regardless of which particular platform we use.

In my examples, I'm using the "official" signal() and not the one from the book in Figure 10.18... Thus,

Signals

In chap-intro/shell1.c, the default action for this signal, named SIGINT (i.e., the interrupt key, e.g., Control-C), was to terminate the process

Now, in chap10-signals/shell2.c, SIGINT is caught by the function signal(), which in turn calls the user-defined function sig_int(), which simply prints "interrupt" without exiting

Calling the signal() function the process is basically registering its own signal handler (as opposed to the default signal handler)

The signal handler function is also called the signal-catching function

(see <signal.h>)

shell2.c

```c
#include "apue.h"
#include <sys/wait.h>

static void sig_int(int);       /* our signal-catching function */

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal error");

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}

void
sig_int(int signo)
{
    printf("interrupt\n%% ");
}
```