MODULE – 4
Process Control

5.1 Introduction:

Process control is concerned about creation of new processes, program execution, and process termination

5.1.1 Process Identifiers:

Every process has a unique process ID, a non-negative integer. Because the process ID is the only well-known identifier of a process that is always unique, it is often used as a piece of other identifiers, to guarantee uniqueness.

For example, applications sometimes include the process ID as part of a filename in an attempt to generate unique filenames.

Although unique, process IDs are reused.

As processes terminate, their IDs become candidates for reuse. Most UNIX systems implement algorithms to delay reuse, however, so that newly created processes are assigned IDs different from those used by processes that terminated recently. This prevents a new process from being mistaken for the previous process to have used the same ID.

Process ID 0 is usually the *scheduler* process and is 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. This process is responsible for bringing up a UNIX system after the kernel has been bootstrapped. The init process never dies.

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 only difference in the returns is that the return value in the child is 0, whereas 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. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

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.
For example, the child gets a copy of the parent's data space, heap, and stack.

Note that this is a copy for the child; the parent and the child do not share these portions of memory.

The parent and the child share the text segment .

Example program: Program to demonstrate fork function Program name – fork_1.c

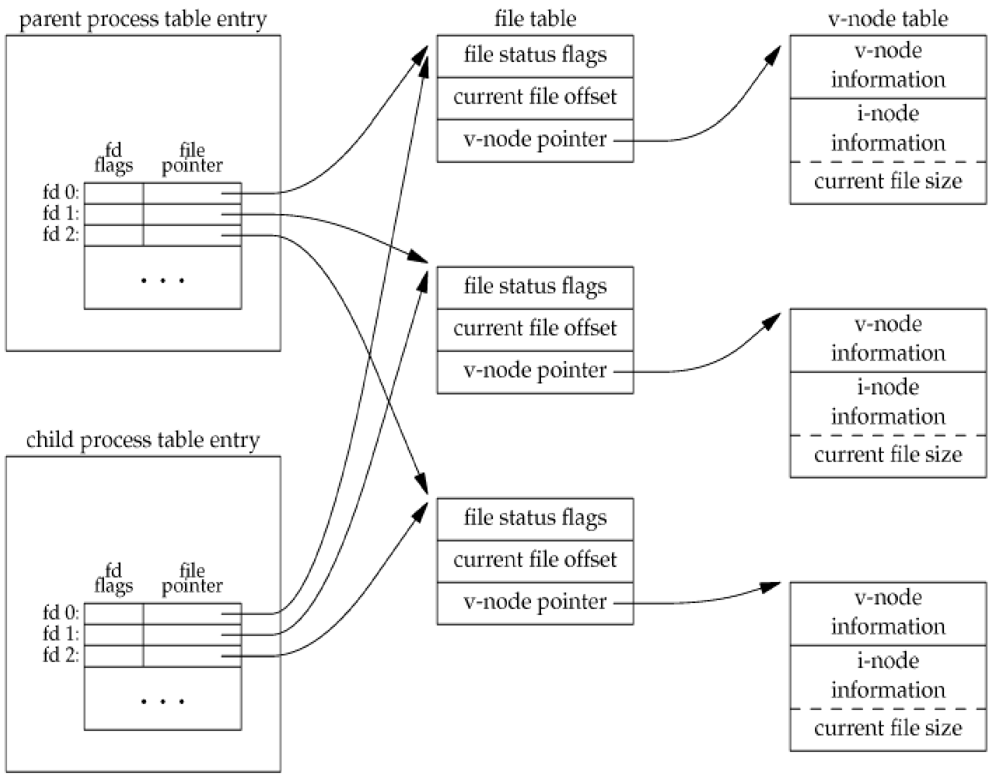
```
#include<sys/types.h>
#include<unistd.h>

int main( )
{
fork( );
printf("\n hello USP");
}
```

```
Output: $ cc fork1.c
$ ./a.out hello USP hello USP
```

Note: The statement hello USP is executed twice as both the child and parent have executed that instruction.

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 Figure



Sharing of open files between parent and child after fork

- It is important that the parent and the child share the same file offset.
- Consider a process that forks a child, then waits for the child to complete.
- Assume that both processes write to standard output as part of their normal processing.
- If the parent has its standard output redirected (by a shell, perhaps) it is essential that the parent's file offset be updated by the child when the child writes to standard output.
- In this case, the child can write to standard output while the parent is waiting for it; on completion of the child, the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote.
- If the parent and the child did not share the same file offset, this type of interaction would be more difficult to accomplish and would require explicit actions by the parent.

Two ways of handling the descriptors after a fork:

- The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors. When the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.
- Both the parent and the child go their own ways. Here, after the fork, the parent closes the descriptors that it doesn't need, and the child does the same thing. This way, neither interferes with the other's open descriptors. This scenario is often the case with network servers.

Properties of the parent inherited by the child:

- Real user ID, real group ID, effective user ID, effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments

- Memory mappings
- Resource limits

Differences between the parent and child:

- The return value from fork
- The process IDs are different
- The two processes have different parent process IDs: the parent process ID of the child is the parent; the parent process ID of the parent doesn't change
- The child's tms_utime, tms_stime, tms_cutime, and tms_cstime values are set to 0
- File locks set by the parent are not inherited by the child
- Pending alarms are cleared for the child
- The set of pending signals for the child is set to the empty set

Two main reasons for fork to fail are:

- If too many processes are already in the system, which usually means that something else is wrong
- If the total number of processes for this real user ID exceeds the system's limit

Two uses for fork:

- When a process wants to duplicate itself so that the parent and 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.
- 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.

5.1.2 vfork:

- The function vfork has the same calling sequence and same return values as fork.
- The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.
- 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, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System.
- 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.

5.1.3 exit:

A process can terminate normally in five ways:

- Executing a return from the main function.
- Calling the exit function.
- Calling the _exit or _Exit function.

In most UNIX system implementations, exit is a function in the standard C library, whereas _exit is a system call.

- Executing a return from the start routine of the last thread in the process. When the last thread returns from its start routine, the process exits with a termination status of 0.
- Calling the pthread_exit function from the last thread in the process.

The three forms of abnormal termination are as follows:

- Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.
- When the process receives certain signals. Examples of signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.
- The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later, the target thread terminates.

5.1.4 <u>wait and waitpid functions:</u>

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. 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

A process that calls wait or waitpid can:

- Block, if all of its children are still running
- Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- Return immediately with an error, if it doesn't have any child processes.

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0 (see later), or 1 on error.

The differences between these two functions are as follows:

- The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
- 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.

If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates.

For both functions, the argument statloc is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

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.

pid < 1 Waits for any child whose process group ID equals the absolute value of pid.

<u>Print a description of the exit status :</u>

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

void pr_exit(int status)
{
if (WIFEXITED(status))
    printf("normal termination, exit status = %d\n", WEXITSTATUS(status));
else if (WIFSIGNALED(status))
    printf("abnormal termination, signal number = %d%s\n", WTERMSIG(status),
#ifdef WCOREDUMP WCOREDUMP(status) ? " (core file generated)" : "");
    #else "");
#endif
else if (WIFSTOPPED(status))
    printf("child stopped, signal number = %d\n", WSTOPSIG(status));
}
```

<u>**Macros to examine the termination status returned by** wait **and** waitpid</u>

Macro	**Description**
WIFEXITED(status)	True if status was returned for a child that terminated normally. In this case, we can execute WEXITSTATUS (status) to fetch the low-order 8 bits of the argument that the child passed to exit, _exit,or _Exit
WIFSIGNALED (status)	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute WTERMSIG (status) to fetch the signal number that caused the termination. Additionally, some implementations (but not the Single UNIX Specification) define the macro WCOREDUMP (status) that returns true if a core file of the terminated process was generated
WIFSTOPPED (status)	True if status was returned for a child that is currently stopped. In this case, we can execute WSTOPSIG (status) to fetch the signal number that caused the child to stop
WIFCONTINUED (status)	True if status was returned for a child that has been continued after a job control stop
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
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

Features provided by “waitpid” function that are not provided by the “wait” function”

1. The waitpid function lets us wait for one particular process, whereas the wait function returns the status of any terminated child.
2. The waitpid function provides a nonblocking version of wait. There are times when we want to fetch a child's status, but we don't want to block
3. The waitpid function provides support for job control with the WUNTRACED and WCONTINUED options

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

int main(void)
{
pid_t pid;
((pid = fork()) < 0)
{
   err_sys("fork error");
}
else if (pid == 0)
{                              /* first child */
if ((pid = fork()) < 0)
   err_sys("fork error");
else if (pid > 0) exit(0);
    /* parent from second fork == first child */
    /* * We're the second child; our parent becomes init as soon * as our real parent
calls exit() in the statement above. * Here's where we'd continue executing, knowing
that when * we're done, init will reap our status. */

   sleep(2);
   printf("second child, parent pid = %d\n", getppid());
   exit(0);
}
if (waitpid(pid, NULL, 0) != pid) /* wait for first child */
   err_sys("waitpid error");
   /* * We're the parent (the original process); we continue executing, * knowing that
we're not the parent of the second child. */
exit(0);
}
```

Output:

```
$ ./a.out
$ second child, parent pid = 1
```

5.1.5 wait3 and wait4 functions:

The only feature provided by these two functions that isn't provided by the wait, waitid, and waitpid functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.

The prototype of these functions are:

```
#include<sys/types.h>
#include<sys/wait.h>
#include<sys/time.h>
#include<sys/resource.h>
pid_t wait3(*statloc, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options, struct rusage
*rusage);
```

Both return: process ID if OK, -1 on error

5.2**Functions:**

5.2.1 **Race Conditions:**

A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.

Example: The program below outputs two strings: one from the child and one from the parent. The program contains a race condition because the output depends on the order in which the processes are run by the kernel and for how long each process runs.

```
#include “apue.h”

static void charatatime(char *);

int main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("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);
}
```

Output:

```
$ ./a.out
output from child
output from parent
$ ./a.out
output from child
output from parent
$ ./a.out
output from child
output from parent
```

5.2.2 exec Functions:

When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk
There are 6 exec functions:

```
#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 []);
```

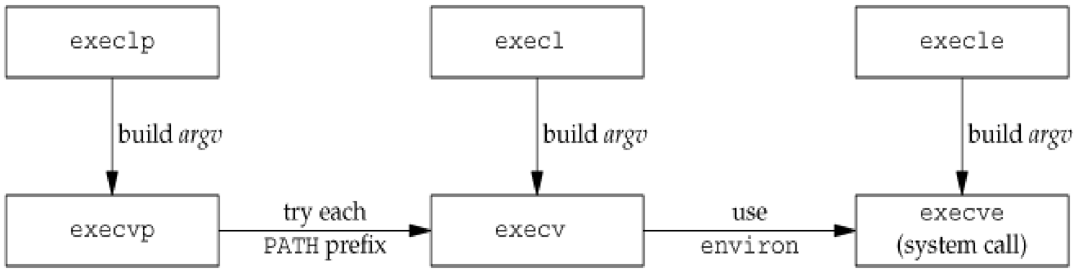
All six return: -1 on error, no return on success

Differences in these functions:

- First four take a pathname argument, whereas the last two take a filename argument. When a filename argument is specified
  - If filename contains a slash, it is taken as a pathname
  - Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.
- Next difference concerns the passing of the argument list (lstands for list and v stands for vector). The functions execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments. For the other three functions (execv, execvp, and execve), we have

to build an array of pointers to the arguments, and the address of this array is the argument to these three functions

- Final difference is the passing of the environment list to the new program. The two functions whose names end in an e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program



Relationship of the six exec functions

5.3 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

```
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);

Both return: 0 if OK, 1 on error
```

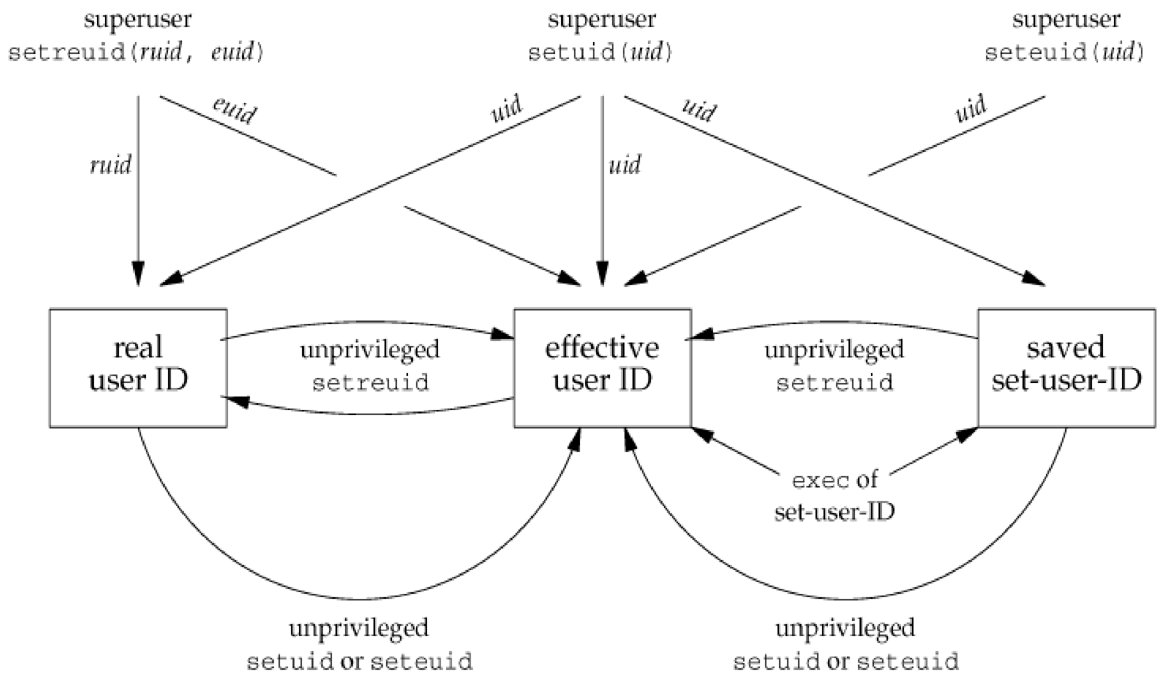
There are rules for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

- 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

Important points about the 3 user IDs that the kernel maintains:

- 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
- 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
- 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



Summary of all the functions that set the various user Ids

5.4 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

The pathname is normally an absolute pathname, since no special operations are performed on it (i.e., PATH is not used). The recognition of these files is done within the kernel as part of processing the exec system call. The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter file. Be sure to differentiate between the interpreter filea text file that begins with #!and the interpreter, which is specified by the pathname on the first line of the interpreter file. Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the #!, the pathname, the optional argument, the terminating newline, and any spaces

5.5 System Function:

```
#include <stdlib.h>

int system(const char *cmdstring);
```

If cmdstring is a null pointer, system returns nonzero only if a command processor is available. This feature determines whether the system function is supported on a given operating system. Under the UNIX System, system is always available. Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.

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

Program: Calling the system function

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

int main(void)
{
    int status;

    if ((status = system("date")) < 0) err_sys("system() error");
        pr_exit(status);
    if ((status = system("nosuchcommand")) < 0) err_sys("system() error");
        pr_exit(status);
    if ((status = system("who; exit 44")) < 0) err_sys("system() error");
        pr_exit(status); exit(0);
}
```

5.6**Process Accounting:**

- Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates.
- These accounting records are typically a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on.
- A superuser executes accton with a pathname argument to enable accounting.
- The accounting records are written to the specified file, which is usually /var/account/acct. Accounting is turned off by executing accton without any arguments.
- The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork.
- Each accounting record is written when the process terminates.
- This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started.
- The accounting records correspond to processes, not programs.
- A new record is initialized by the kernel for the child after a fork, not when a new program is executed.

5.6.1 <u>User Identification:</u>

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call getpwuid(getuid()), 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 and the getloginfunction provides a way to fetch that login name.

#include <unistd.h>
char *getlogin(void);
Returns: pointer to string giving login name if OK, NULL on error

This function can fail if the process is not attached to a terminal that a user logged in to.

5.7<u>Process Times:</u>

We describe three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the times function to obtain these values for itself and any terminated children.

#include <sys/times.h>
clock_t times(struct tms *buf);
Returns: elapsed wall clock time in clock ticks if OK, 1 on error

This function fills in the tmsstructure pointed to by buf:

```
  struct tms {
    clock_t   tms_utime;  /* user CPU time */
    clock_t   tms_stime;  /* system CPU time */
    clock_t   tms_cutime; /* user CPU time, terminated children */
    clock_t   tms_cstime; /* system CPU time, terminated children */
 };
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value.