Process Control

Prepared By
Chandrika Prasad
Dept of CSE
RIT

Reference: Advanced Programming in the UNIX Environment by W. Richard Stevens.

Process Identifiers

Objectives of this chapter:

1. Creation of new processes, program execution and process termination.

2. Understand the various IDs that are the property of the process — real, effective, and saved; user and group IDs—and how they're affected by the process control primitives.

3. To implement Interpreter files.

4. Understand the process accounting provided by most UNIX systems.

Process Identifier

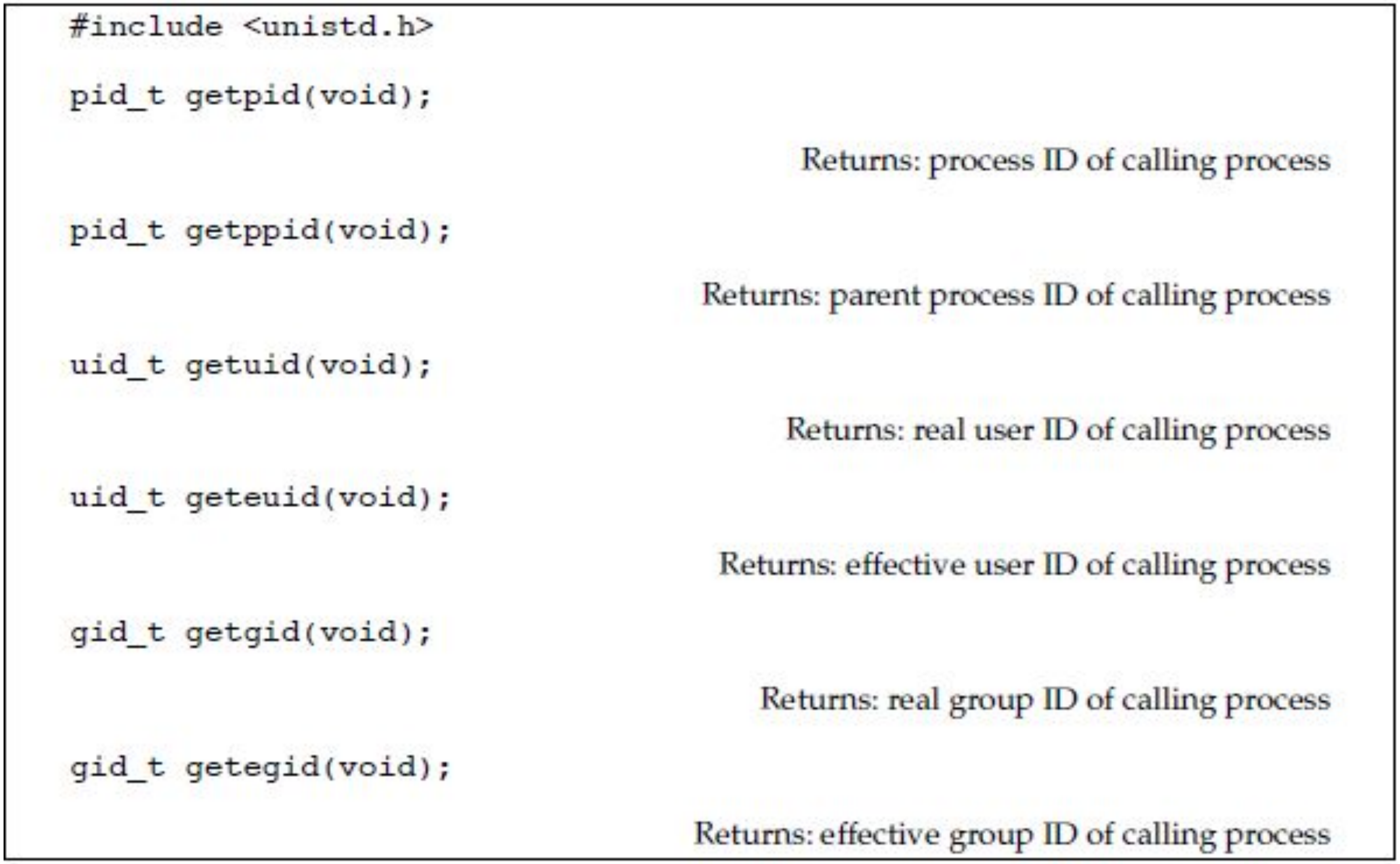
- Every process has a unique process ID, a non-negative integer.
- Sometimes applications includes the process ID as part of a filename in an attempt to generate unique filenames.
- Process Ids can be reused.

Note: 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.

Special processes:

1. Process ID 0 is usually the scheduler process and is often known as the *swapper.*
2. Process ID 1 is usually the init process and is invoked by the kernel at the end of the bootstrap procedure.
3. process ID 2 is the pagedaemon. This process is responsible for supporting the paging of the virtual memory system.

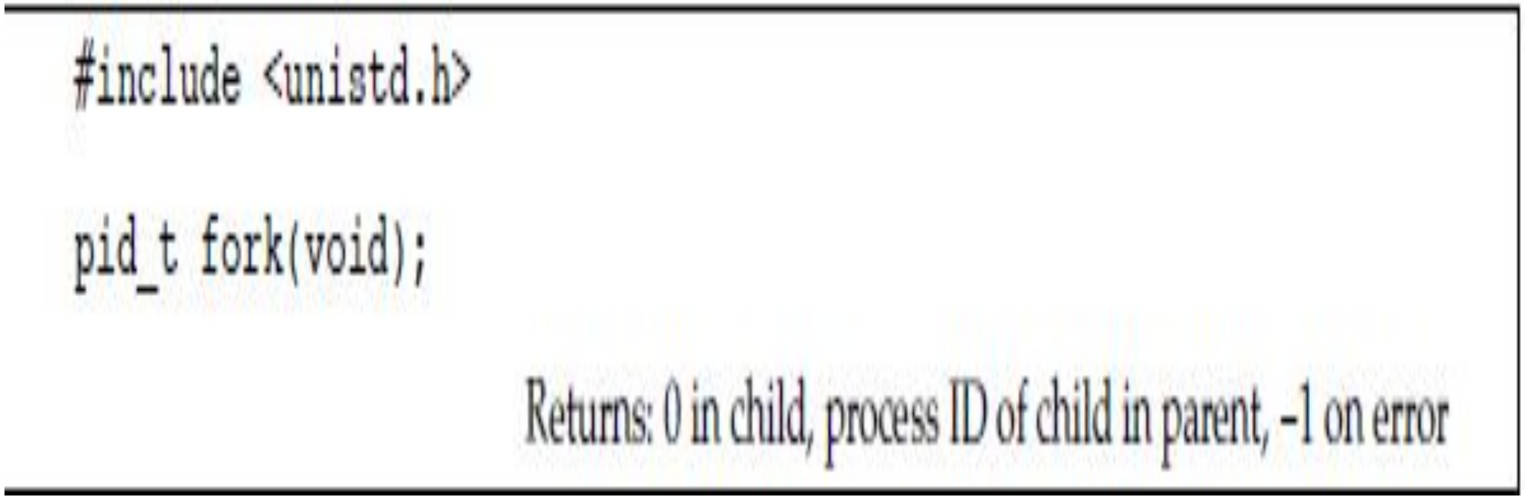
In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers.



```
#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
```

fork( )

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



```
#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.
    - The return value in the parent is the process ID of the new child.
- The child can always call getppid to obtain the process ID of its parent.

- 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.

- 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.

- Modern implementations don’t perform a complete copy of the parent’s data, stack, and heap. Instead, a technique called copy-on-write (COW) is used.
- These regions are shared by the parent and the child and have their protection changed by the kernel to read-only.
- If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only, typically a ‘‘page’’ in a virtual memory system.

Example: Implementing fork()

```
int globvar = 6; /* external variable in initialized data */
char buf[] = "a write to stdout\n";
int main(void)
{
int var; /* automatic variable on the stack */
pid_t pid;
var = 88;
if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
    err_sys("write error");
printf("before fork\n"); /* we don’t flush stdout */
if ((pid = fork()) < 0)
  {
        err_sys("fork error");
   }
   else if (pid == 0)
   { /* child */
        globvar++; /* modify variables */
         var++;
   }
   else
   {
        sleep(2); /* parent */
    }
    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar, var);
    exit(0);
}
```

output

If we execute this program, we get

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89        child's variables were changed
pid = 429, glob = 6, var = 88           parent's copy was not changed
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

About the output

- 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.

- The interaction of fork with the I/O functions in the program – notice that the write function is not buffered.

- Because write is called before the fork, its data is written once to standard output.
- The standard I/O library is buffered.

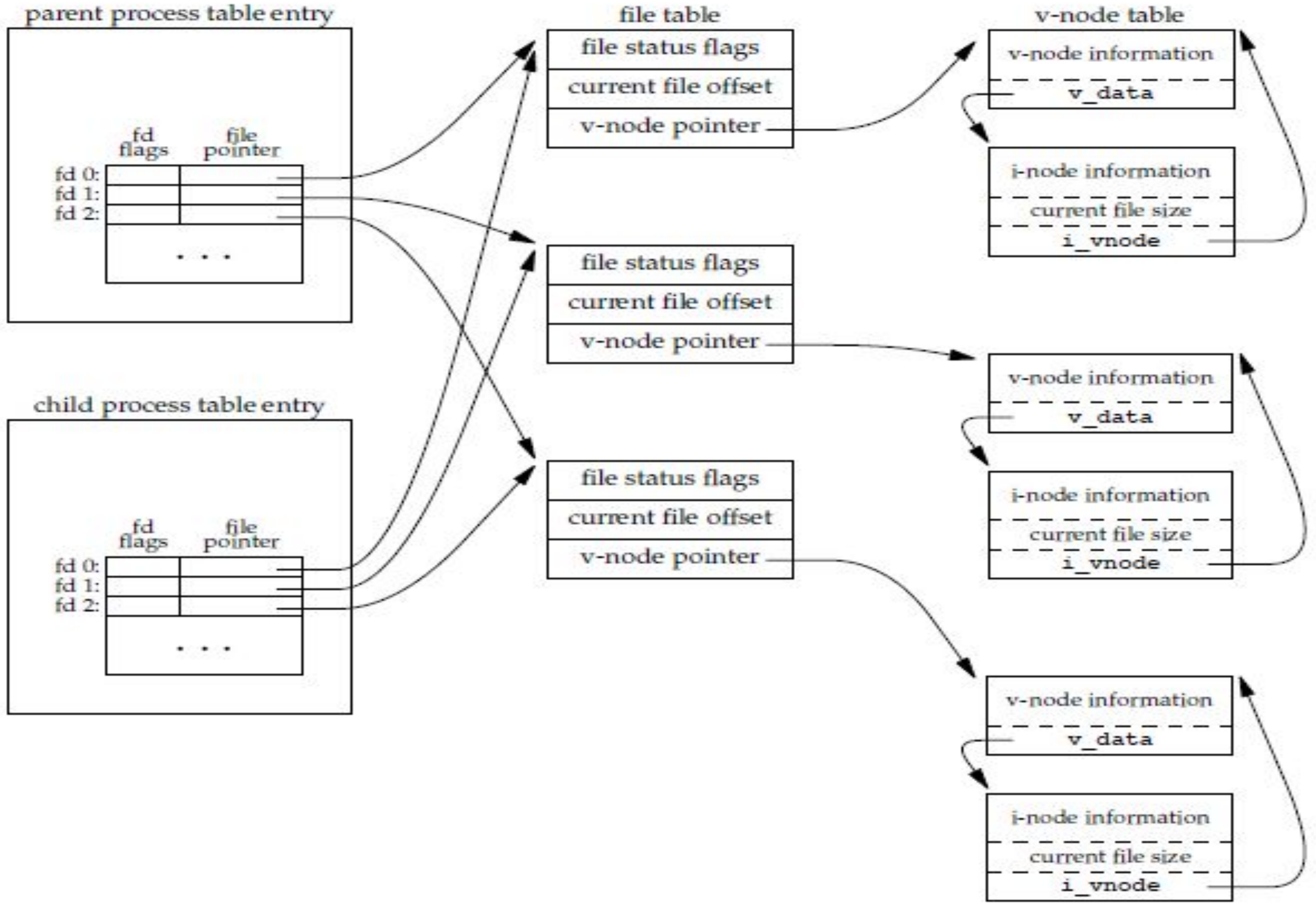
- we run the program interactively, we get only a single copy of the first printf line, because the standard output buffer is **flushed by the newline**.
- When we redirect standard output to a file, however, we get two copies of the printf line because

→ The printf before the fork is called once, but the line remains in the buffer when fork is called.

→ This buffer is then copied into the child when the parent's data space is copied to the child.

- Both the parent and the child now have a standard I/O buffer with this line in it.
- The second printf, right before the exit, just appends its data to the existing buffer.
- When each process terminates, its copy of the buffer is finally flushed.

Sharing of open files between parent and child after fork



- one characteristic of fork is that all file descriptors that are open in the parent are duplicated in the child.
- It is important that the parent and the child share the same file offset.

Besides the open files, numerous other properties of the parent are inherited by the child:

- Real user ID, real group ID, effective user ID, and 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

The differences between the parent and child are

- The return values from fork are different.
- 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.

There are two normal cases for handling the descriptors after a fork.

1. The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors.
2. Both the parent and the child go their own ways. This scenario is often found with network servers.

- The two main reasons for fork to fail are

(a) if too many processes are already in the system, which usually means that something else is wrong.

(b) if the total number of processes for this real user ID exceeds the system's limit.

There are two uses for fork:

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.
2. When a process wants to execute a different program. This is common for shells. In this case, the child does an exec

vfork()

- The function vfork has the same calling sequence and same return values as fork, but

the semantics of the two functions differ.

```
int globvar = 6; /* external variable in initialized
   data */
int main(void)
{
int var; /* automatic variable on the stack */
pid_t pid;
var = 88;
printf("before vfork\n"); /* we don’t flush stdio */
if ((pid = vfork()) < 0) {
printf("vfork error");
} else if (pid == 0) { /* child */
globvar++; /* modify parent’s variables */
var++;
_exit(0); /* child terminates */
}
printf("pid = %ld, glob = %d, var = %d\n",
    (long)getpid(), globvar, var); exit(0); }
```

output

- $ **./a.out**
- before vfork
- pid = 29039, glob = 7, var = 89

- _exit does not perform any flushing of standard I/O buffers.
- If we call exit instead, the results are indeterminate.
- Depending on the implementation of the standard I/O library, we might see no difference in the output, or we might find that the output from the first printf in the parent has disappeared.

exit()

A process can terminate normally in five ways:

1. Executing a return from the main function. This is equivalent to calling exit.
2. Calling the exit function. This function is calls all exit handlers that have been registered by calling atexit and closing all standard I/O streams.
3. Calling the _exit provide a way for a process to terminate without running exit handlers or signal handlers.

4. Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.
5. When the process receives certain signals.

- Regardless of how a process terminates, the kernel code closes all the open descriptors for the process, releases the memory that it was using, and so on.
- For any of the preceding cases, we want the terminating process to be able to notify its parent how it terminated.
- For the three exit functions (exit and _exit), this is done by passing an exit status as the argument to the function.

- In the case of an abnormal termination, however, the kernel generates a termination status to indicate the reason for the abnormal termination.

- In any case, the parent of the process can obtain the termination status from either the wait or the waitpid function

- The exit status of a process is converted into a termination status by the kernel when _exit function is called.

From fork function’s description, it was obvious that the child has a parent process after the call to fork and about returning a termination status to the parent.

- But what happens if the parent terminates before the child?
  - The answer is that the init process becomes the parent process of any process whose parent terminates.
  - We say that the process has been inherited by init.

Note: What normally happens is that whenever a process terminates, the kernel goes through all active processes to see whether the terminating process is the parent of any process that still exists.

- If so, the parent process ID of the surviving process is changed to be 1 (the process ID of init).
- This way, we're guaranteed that every process has a parent.

Another condition - when a child terminates before its parent.

- If the child completely disappeared, the parent wouldn't be able to fetch its termination status when and if the parent were finally ready to check if the child had terminated.
- The kernel keeps a small amount of information for every terminating process, so that the information is available when the parent of the terminating process calls wait or waitpid.
- Minimally, this information consists of the process ID, the termination status of the process, and the amount of CPU time taken by the process.
- The kernel can discard all the memory used by the process and close its open files.
- In UNIX System terminology, a process that has terminated, but whose parent has not yet waited for it, is called a *zombie.*
- *The ps(1)* command prints the state of a zombie process as *Z.*

The final condition to consider is this: 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.

When we say "one of init's children," we mean either a process that init generates directly (such as getty) or a process whose parent has terminated and has been subsequently inherited by init.

1. What happens when parent process terminates before a child?

Ans: init will become parent for those child processes.

2. What happens when a child terminates before its parent without notification?

Ans: The kernel keeps a small amount of information for every terminating process, so that the information is available when the parent of the terminating process calls wait or waitpid.

3. Identify the information given by a termination status of a process

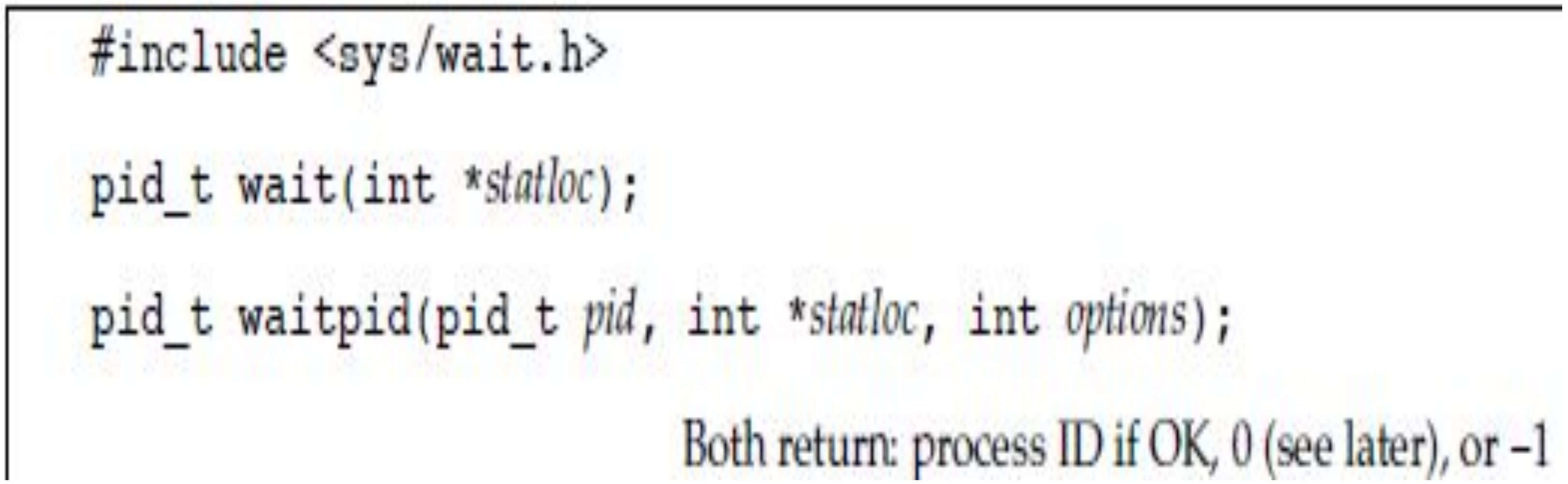
Ans: Minimally, this information consists of the process ID, the termination status of the process, and the amount of CPU time taken by the process.

4. Define Zombie process.

Ans: When a process has already terminated, having received a signal to do so, it normally takes some time to finish all tasks (such as closing open files) before ending itself. In that normally very short time frame, the process is a zombie.

Wait() and waitpid() :

we called the wait function to handle the terminated child.



```
#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

wait and ***waitpid*** both return two values: the return value of
the function is the process ID of the terminated child,
and the termination status of the child (an integer) is
returned through the *statloc* pointer.

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.
- waitpid supports job control.
- waitpid lets us wait for one particular process.

If we have more than one child, wait returns on termination of any of the children.

5. what if we want to wait for a specific process to terminate (assuming we know which process ID we want to wait for)?

Ans: In older versions of the UNIX System, we would have to call wait and compare the returned process ID with the one we're interested in.

If the terminated process wasn't the one we wanted, we would have to save the process ID and termination status and call wait again.

```
Int main(void)
{
pid_t pid;
int status;
if ((pid = fork()) < 0)
err_sys("fork error");
else if (pid == 0) /* child */
exit(7);
if (wait(&status) != pid) /* wait for child */
err_sys("wait error");
pr_exit(status); /* and print its status */
```

```
if ((pid = fork()) < 0)
err_sys("fork error");
else if (pid == 0) /* child */
abort(); /* generates SIGABRT */
if (wait(&status) != pid) /* wait for child */
err_sys("wait error");
pr_exit(status); /* and print its status */
```

```
if ((pid = fork()) < 0)
err_sys("fork error");
else if (pid == 0) /* child */
status /= 0; /* divide by 0 generates SIGFPE */
if (wait(&status) != pid) /* wait for child */
err_sys("wait error");
pr_exit(status); /* and print its status */
exit(0);
}
```

```
$ ./a.out
normal termination, exit status = 7
abnormal termination, signal number = 6
    (core file generated)
abnormal termination, signal number = 8
    (core file generated
```

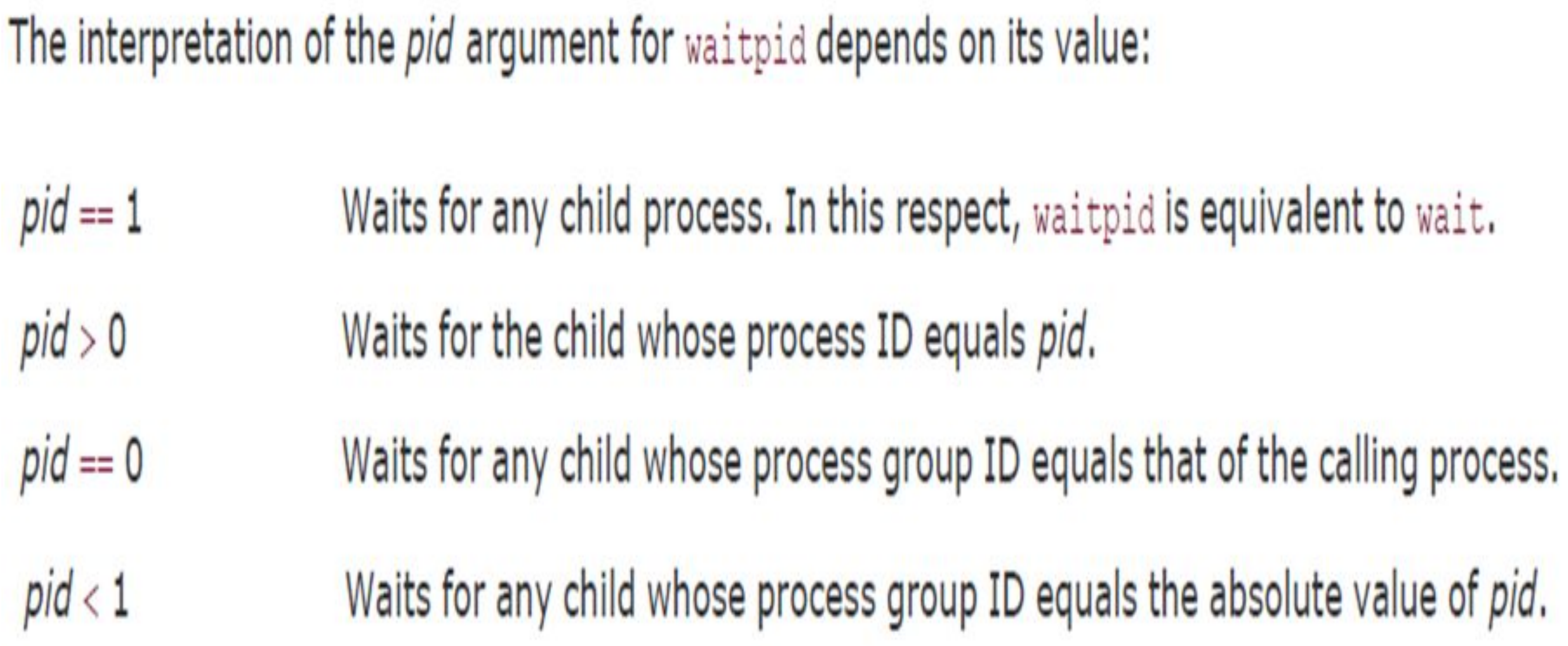
Print a description of the exit status

```
#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));
}
```

Macros to examine the termination status returned by `wait` and `waitpid`

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 (XSI extension to POSIX.1; `waitpid` only).

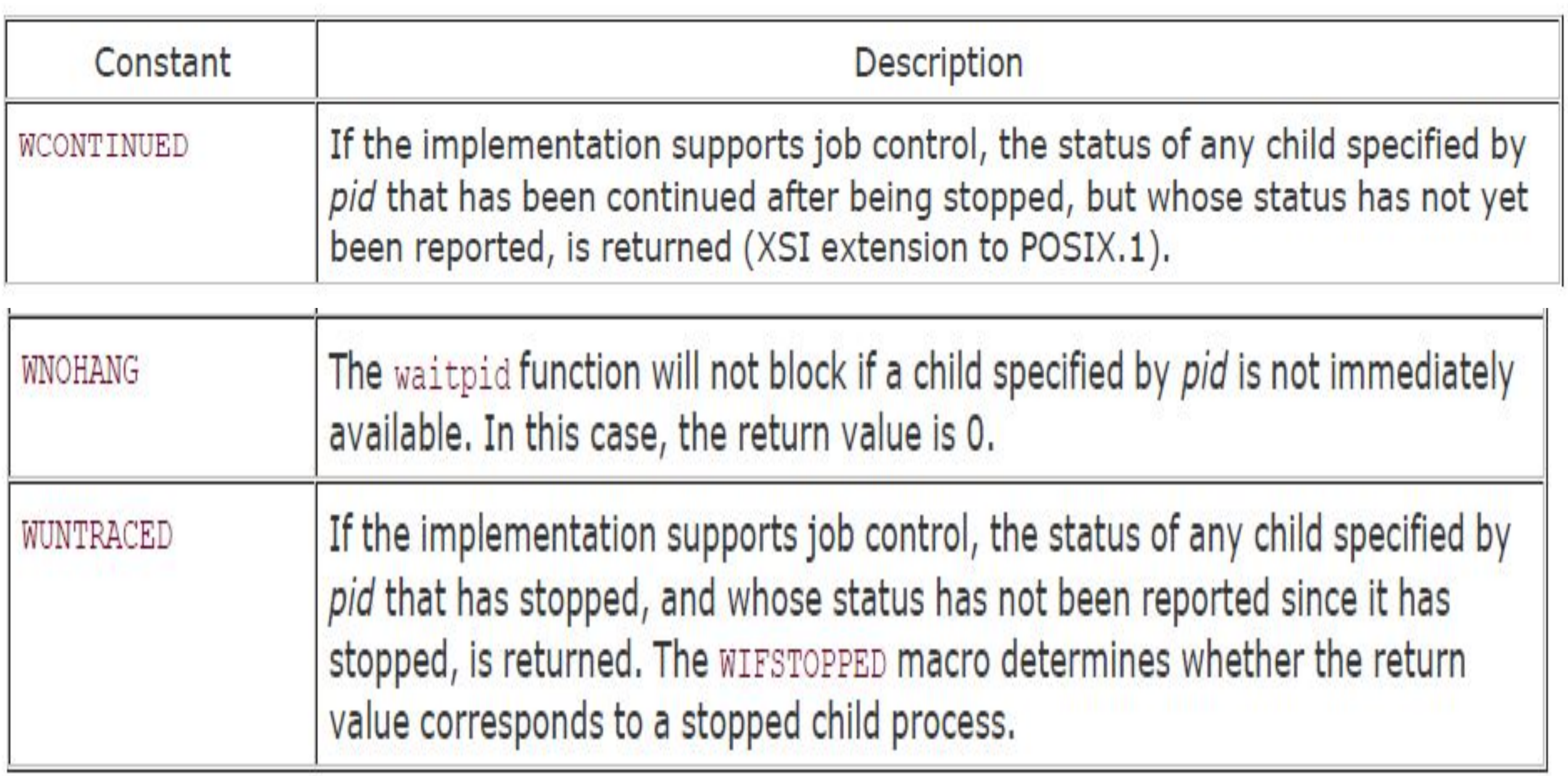
<u>Note</u>: if we have more than one child, wait returns on termination of any of the children. This limitation can be overcome by using waitpid.



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*.

The *options constants for waitpid*



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 extension to POSIX.1).
`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.

Calculate number of times hello is printed.

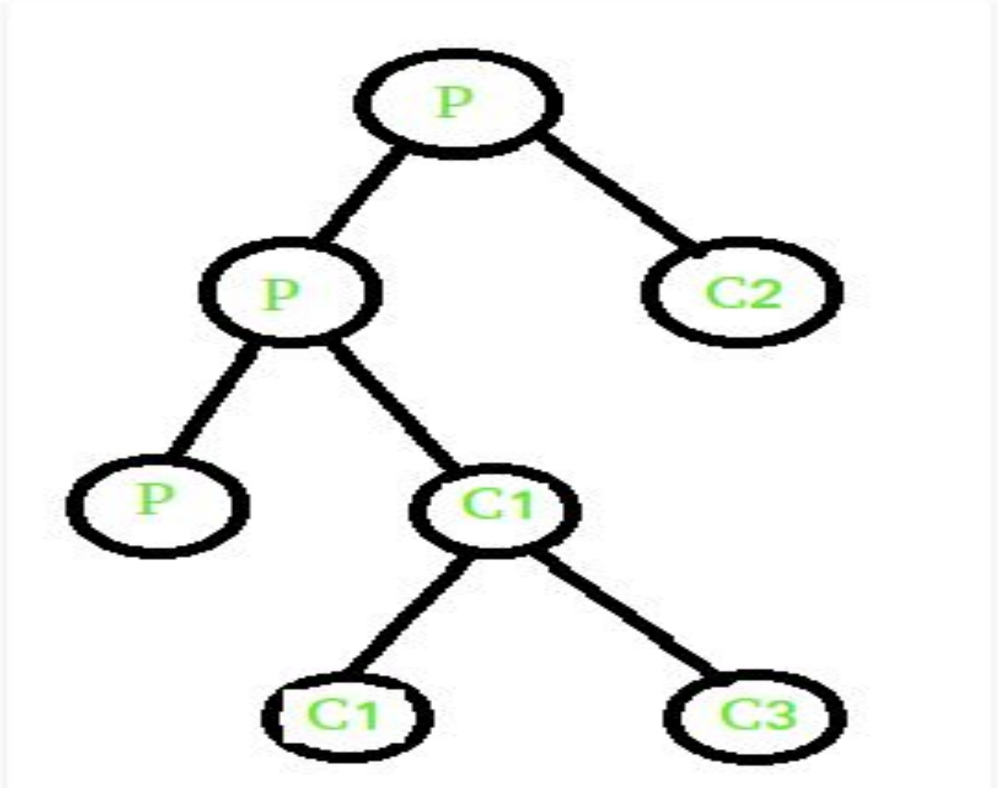
```
#include  <stdio.h>
#include  <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

TODO

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

int main()
{        if (fork()) {
             if (!fork()) {
                 fork();
             printf("1 ");
             } else {
                 printf("2 ");
             }
         }  else   {
             printf("3 ");
         }
    printf("4 ");
    return 0;
}
```

Output:

```
2 4 2 4 1 4 1 4 3 4 1 4 1 4
```



Explanation:

1. It will create two process one parent P (has process ID of child process) and other is child C1 (process ID = 0).

2. When condition is true parent P executes if statement and child C1 executes else statement and print 3. Parent P again check if statement and create two process (one parent P and child C2). In if statement we are using not operator (i.e, !), it executes for child process C2 and parent P executes else part and print value 2. Child C2 further creates two new processes (one parent C2 and other is child C3).

How to avoid zombie status for a process

- If we want to write a process so that it forks a child but we don’t want to wait for the child to complete and we don’t want the child to become a zombie until we terminate,
    - the trick is to call fork twice.

```
#include "apue.h"
#include <sys/wait.h>
int
main(void)
{
pid_t pid;
if ((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);
sleep(2);
printf("second child, parent pid = %ld\n", (long)getppid());
exit(0);
}
if (waitpid(pid, NULL, 0) != pid)
    err_sys("waitpid error");
exit(0);
}
```

About the program

- We call sleep in the second child to ensure that the first child terminates before printing the parent process ID.
- After a fork, either the parent or the child can continue executing; we never know which will resume execution first
- Output

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

Race condition

- 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.
- A process that wants to wait for a child to terminate must call one of the wait functions. If a process wants to wait for its parent to terminate, a loop of the following form could be used:

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

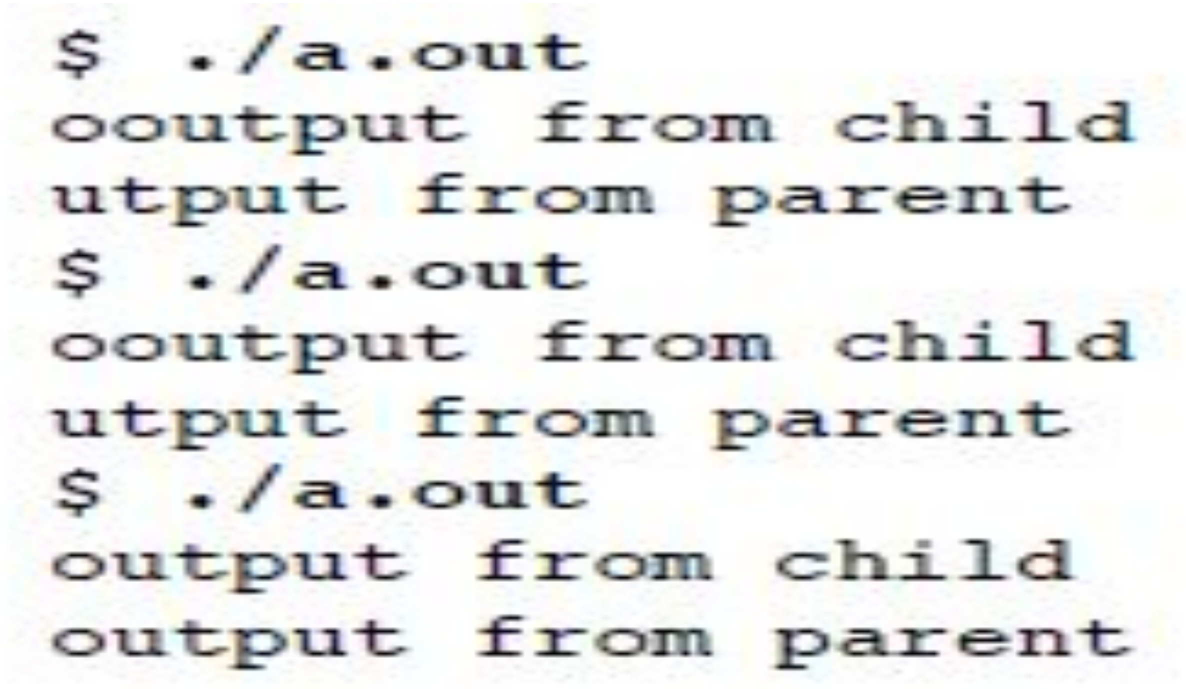
The problem with this type of loop, called ***polling**, is that it wastes CPU time, as the caller is* awakened every second to test the condition.

To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes. Various forms of interprocess communication (IPC) can also be used.

```
#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);
}
```

- The program 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 the length of time for which each process runs.
- We set the standard output unbuffered, so every character output generates a write.
- The goal in this example is to allow the kernel to switch between the two processes as often as possible to demonstrate the race condition.

output



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

```
arl@arl-Lenovo-H30-50:~/usp_2018_arl$ ./race_condition
output froomu tppaurte nftr
om child
```

exec()

In computing, **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.

- 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 six different exec functions, but we’ll often simply refer to ‘‘the exec function’’.

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 functions return: −1 on error, no return on success

- The first difference in these functions is that the first four take a pathname argument, the next 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.

The PATH variable contains a list of directories, called path prefixes, that are separated by colons.
For example, the name=value environment string
PATH=/bin:/usr/bin:/usr/local/bin/:.

specifies four directories to search. The last path prefix specifies the current directory. (A zero-length prefix also means the current directory. It can be specified as a colon at the beginning of the *value,* two colons in a row, or a colon at the end of the *value.)*

If either execlp or execvp finds an executable file using one of the path prefixes, but the file isn't a machine executable that was generated by the link editor, the function assumes that the file is a shell script and tries to invoke /bin/sh with the filename as input to the shell.

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.
We mark the end of the arguments with a null pointer. 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.
*char *arg0, char *arg1, ..., char *argn, (char *)0*

This specifically shows that the final command-line argument is followed by a null pointer. If this null pointer is specified by the constant 0, we must explicitly cast it to a pointer; if we don't, it's interpreted as an integer argument.

The 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.
Normally, a process allows its environment to be propagated to its children, but in some cases, a process wants to specify a certain environment for a child.

The arguments to execle were shown as
char *pathname, char *arg0, ..., char *argn, (char *)0, char *envp[]

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

The process ID does not change after an exec, but the new program inherits additional properties from the calling process:

- Process ID and parent process ID
- Real user ID and real group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- Time left until alarm clock
- Current working directory
- Root directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals
- Resource limits
- Values for tms_utime, tms_stime, tms_cutime, and tms_cstime

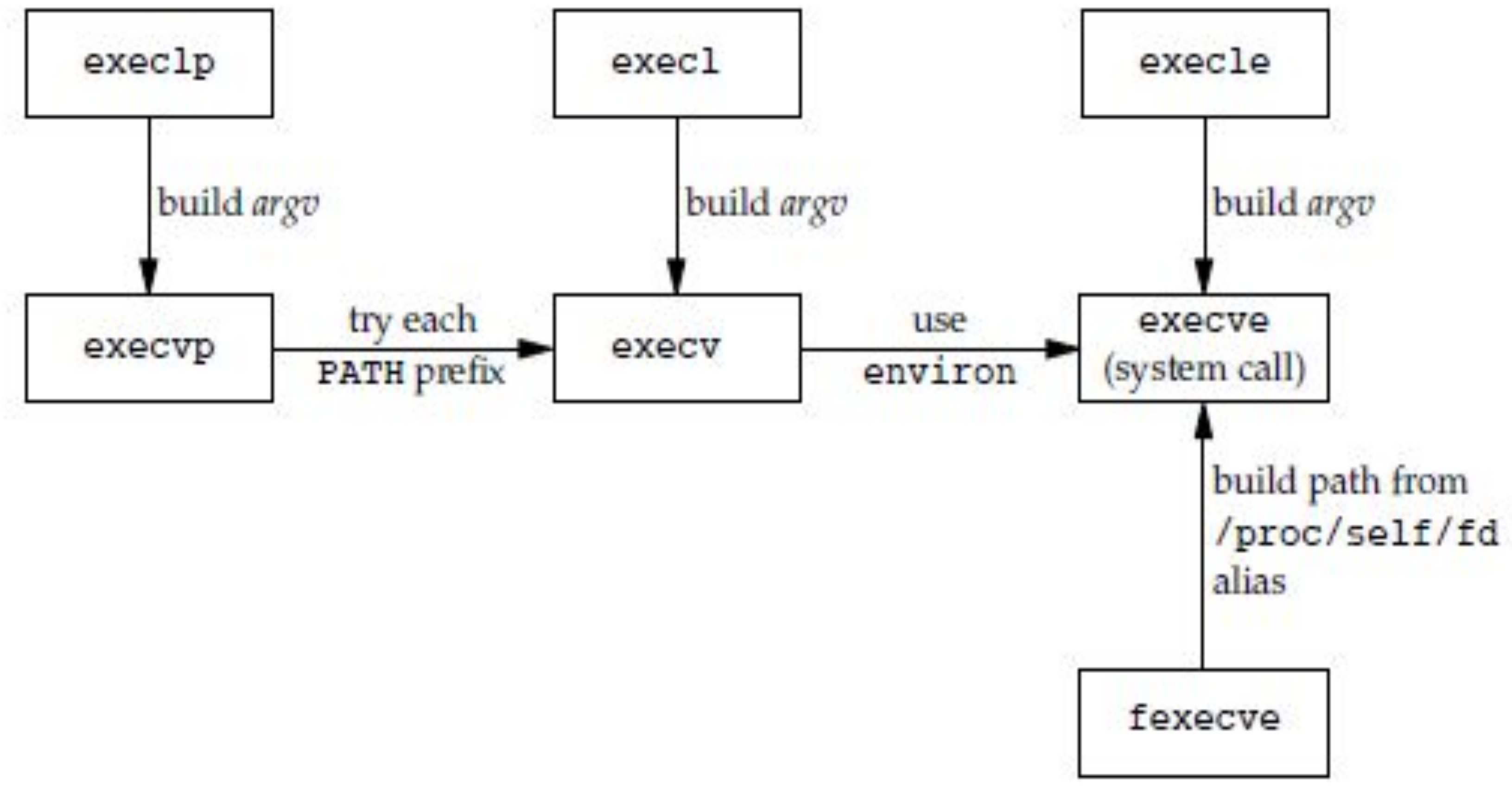
Note: (1) Every open descriptor in a process has a close-on-exec flag(FD_CLOEXEC). If this flag is set, the descriptor is closed across an exec.

- Otherwise, the descriptor is left open across the exec.
- The default is to leave the descriptor open across the exec unless we specifically set the close-on-exec flag using fcntl.

Note: (2) The real user ID and the real group ID remain the same across the exec, but the effective IDs can change, depending on the status of the set-user-ID and the set- group-ID bits for the program file that is executed.

- If the set-user-ID bit is set for the new program, the effective user ID becomes the owner ID of the program file.
- Otherwise, the effective user ID is not changed (it's not set to the real user ID).
- The group ID is handled in the same way.

Relationship among all exec()



```
char *env_init[] = { "USER=unknown“,"PATH=/tmp",  NULL };
int main(void)
{
    pid_t pid;
    if ((pid = fork()) < 0) {
      err_sys("fork error");
    } else if (pid == 0) { /* specify pathname, specify environment */
      if (execle("/home/sar/bin/echoall", "echoall", "myarg1“,"MY
    ARG2", (char *)0, env_init) < 0)
      err_sys("execle error");
}
if (waitpid(pid, NULL, 0) < 0)
    err_sys("wait error");
if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) { /* specify filename, inherit environment */
    if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
     err_sys("execlp error");
}
    exit(0);
}
```

echoall program

```
int main(int argc, char *argv[])
{
    int i; char **ptr;
    extern char **environ;
    for (i = 0; i < argc; i++) /* echo all command-line args */
      printf("argv[%d]: %s\n", i, argv[i]);
    for (ptr = environ; *ptr != 0; ptr++) /* and all env strings */
      printf("%s\n", *ptr);
    exit(0);
}
```

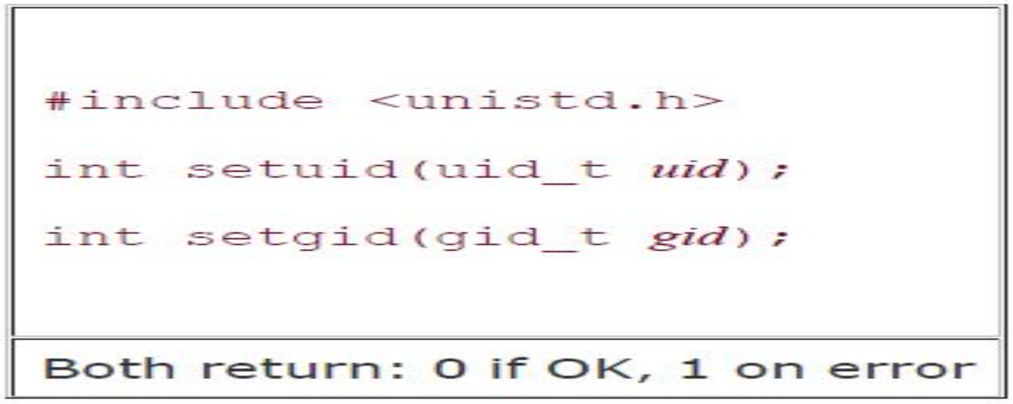
output

```
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
```

47 more lines that aren’t shown

```
HOME=/home/sar
```

Changing User IDs and Group IDs



```
#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.
1. If the process has superuser privileges, the setuid function sets the real user ID, effective user ID, and saved set-user-ID to *uid.*
2. 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.
3. If neither of these two conditions is true, errno is set to EPERM, and 1 is returned.

Ways to change the three user IDs

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

Interpreter files

The interpreter file is the program that runs the rest of the script. The main use of specifying an interpreter file is to make it easier to run the program and to make sure the correct interpreter is used.

Interpreter files are text files that are read in (interpreted) by the program specified on the first line. i.e. text files that begin with a line of the form

#! *pathname [optional-argument]*

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 file, a text file that begins with #!and the interpreter, which is specified by the *pathname on the first line of the interpreter file*

E.g. Shell scripts, perl scripts, awk scripts, and so forth are all examples of interpreter files.

e.g #!/bin/sh

Write a program to create and execute Interpreter file

inter.c

```
 #include<sys/stat.h>
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
int main()
{
    pid_t pid;
    if((pid=fork())<0)
     printf("error");
    else if(pid==0)
if(execl("textinterpreter","test","myarg1",
    "myarg2", "myarg4", (char *)0)<0)
    printf("error1");
if(waitpid(pid,NULL,0)<0)
    printf("wait error");
return 0;
}
```

Interpreter file

text interpreter (textinterpreter)

```
#! /home/guest1/echoarg   my2
```

echoarg.c file

```
int main(int argc,char *argv[])
{   int i;
    for(i=0;i<argc;i++)
          printf("argv[%d]=%s",i,argv[i]);
    return(0);
}
```

How to compile the file

```
cc echoarg.c -o echoarg


chmod 777 textinterpreter


gcc inter.c –o inter
./inter
```

Interpreter files are useful for the following reasons.

1. They hide that certain programs are scripts in some other language. For example, to execute an awk script, we just say

awkexample optional-arguments

instead of needing to know that the program is really an awk script that we would otherwise have to execute as

awk -f awkexample *optional-arguments*

2. Interpreter scripts provide an efficiency gain. There is more overhead in replacing an interpreter script with a shell script.

3. Interpreter scripts let us write shell scripts using shells other than /bin/sh. When it finds an executable file that isn't a machine executable, execlp has to choose a shell to invoke, and it always uses /bin/sh. Using an interpreter script, however, we can simply write

#!/bin/csh

(C shell script follows in the interpreter file)

system()

The C library function

int system(const char *command)

passes the command name or program name specified by **command** to the host environment to be executed by the command processor and returns after the command has been completed.

Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.

1. If either the fork fails or waitpid returns an error other than EINTR (**EINTR** is one of the POSIX errors that you can get from different blocking functions), 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.

Example

```
#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);
}
```

Process Accounting

- When process accounting is 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 on FreeBSD
and Mac OS X, /var/account/pacct on Linux, and /var/adm/pacct on Solaris.
- Accounting is turned off by executing accton without any arguments.

The structure of the accounting records is defined in the header <sys/acct.h> and looks something like

```
typedef  u_short comp_t;   /* 3-bit base 8 exponent; 13-bit fraction */

struct  acct
{
  char   ac_flag;      /* flag (see Figure 8.26) */
  char   ac_stat;      /* termination status (signal & core flag only) */
                       /* (Solaris only) */
  uid_t  ac_uid;       /* real user ID */
  gid_t  ac_gid;       /* real group ID */
  dev_t  ac_tty;       /* controlling terminal */
  time_t ac_btime;     /* starting calendar time */
  comp_t ac_utime;     /* user CPU time (clock ticks) */
  comp_t ac_stime;     /* system CPU time (clock ticks) */
  comp_t ac_etime;     /* elapsed time (clock ticks) */
  comp_t ac_mem;       /* average memory usage */
  comp_t ac_io;        /* bytes transferred (by read and write) */
                       /* "blocks" on BSD systems */
   comp_t ac_rw;        /* blocks read or written */
                        /* (not present on BSD systems) */
   char   ac_comm[8];   /* command name: [8] for Solaris, */
                        /* [10] for Mac OS X, [16] for FreeBSD, and */
                        /* [17] for Linux */
 };
```

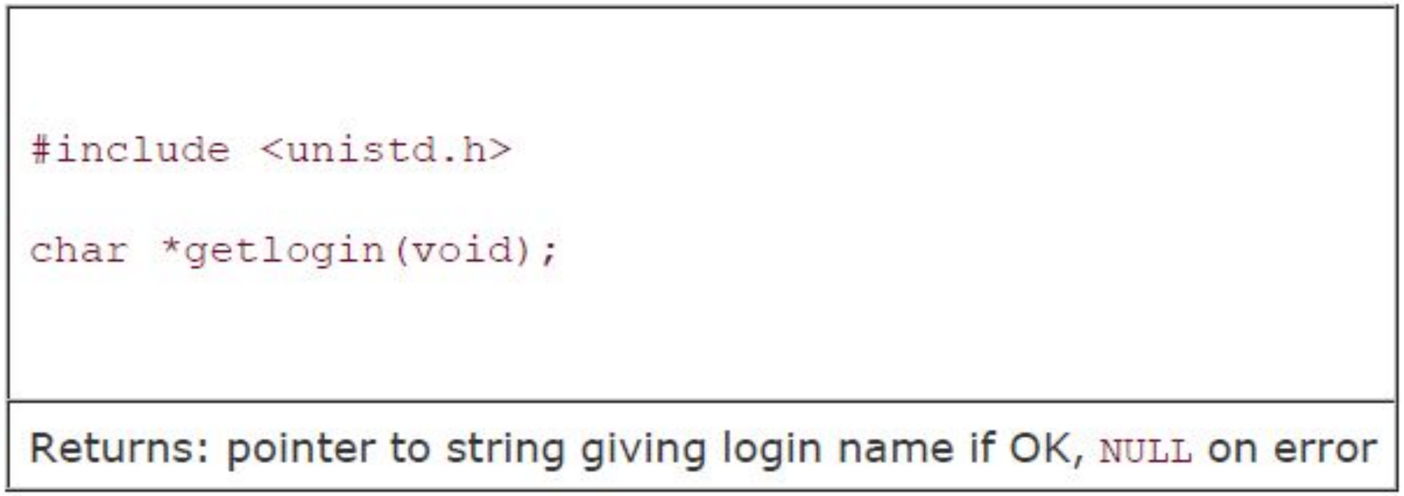
The `ac_flag` member records certain events during the execution of the process. These events are described

Values for `ac_flag` from accounting record

`ac_flag`	Description	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
`AFORK`	process is the result of `fork`, but never called `exec`	•	•	•	•
`ASU`	process used superuser privileges		•	•	•
`ACOMPAT`	process used compatibility mode				
`ACORE`	process dumped core	•	•	•	
`AXSIG`	process was killed by a signal	•	•	•	
`AEXPND`	expanded accounting entry				•

- 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. Although exec doesn't create a new accounting record, the command name changes, and the AFORK flag is cleared.
- This means that if we have a chain of three programs A execs B, then B execs C, and C exits only a single accounting record is written.
- The command name in the record corresponds to program C, but the CPU times, for example, are the sum for programs A, B, and C.

User Identification

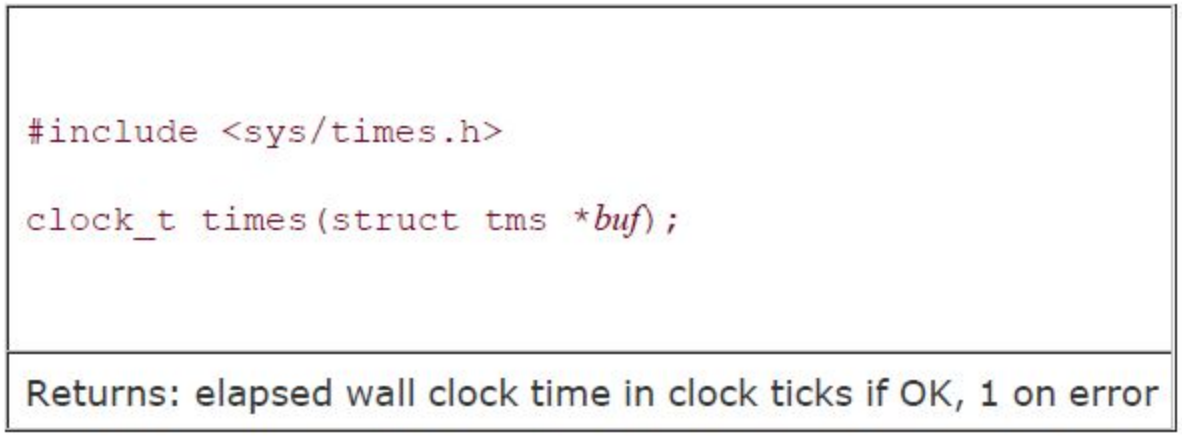


```
#include <unistd.h>

char *getlogin(void);
```

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

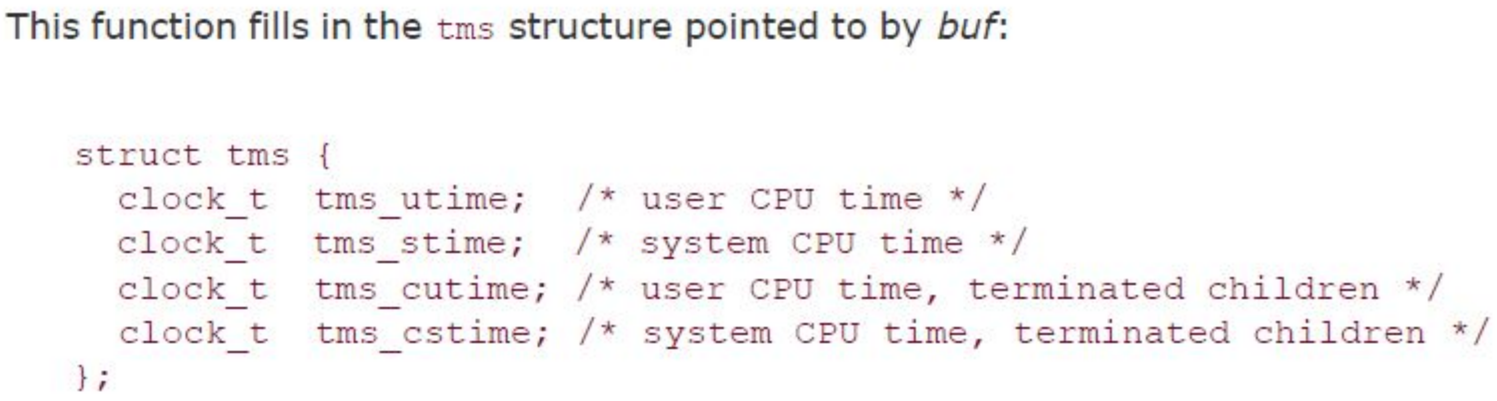
Process Times

We can measure three kinds of times: wall clock time, user CPU time, and system CPU time. A 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 `tms` structure 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 */
};
```

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.
For example, we call times and save the return value. At some later time, we call times again and subtract the earlier return value from the new return value. The difference is the wall clock time.