Chapter 1-process environment

Introduction

A Process is a program under execution in a UNIX or POSIX system

main FUNCTION

A C program starts execution with a function called main. The prototype for the main function is
int main(int argc, char *argv[]);
where argc is the number of command-line arguments, and argv is an array of pointers to the arguments. 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.

PROCESS TERMINATION

There are eight ways for a process to terminate.
Normal termination occurs in five ways:

- Return from main
- Calling exit
- Calling _exit or _Exit
- Return of the last thread from its start routine
- Calling pthread_exitfrom the last thread

Abnormal termination occurs in three ways:

- Calling abort
- Receipt of a signal
- Response of the last thread to a cancellation request
  If the main() function returns, the exit function is called. If the startup routine were added in C the call to main could look like **exit(main(argc,argv));**

EXIT FUNCTIONS

Three functions terminate a program normally: _exit and _Exit, which return to the kernel immediately, and exit, which performs certain cleanup processing and then returns to the kernel.

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

All three exit functions expect a single integer argument, called the exit status. Returning an integer value from the main function is equivalent to calling exit with the same value. Thus **exit(0);** is the same as **return(0);** from the main function.
In the following situations the exit status of the process is undefined.

- any of these functions is called without an exit status.
- main does a return without a return value
- main "falls off the end", i.e if the exit status of the process is undefined.

Example: **$ cc hello.c**
```
$ ./a.out hello, world
 $ echo $? // print the exit status
13
```

atexitFunction

With ISO C, a process can register up to 32 functions that are automatically called by exit. These are called exit handlers and are registered by calling the atexit function.

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

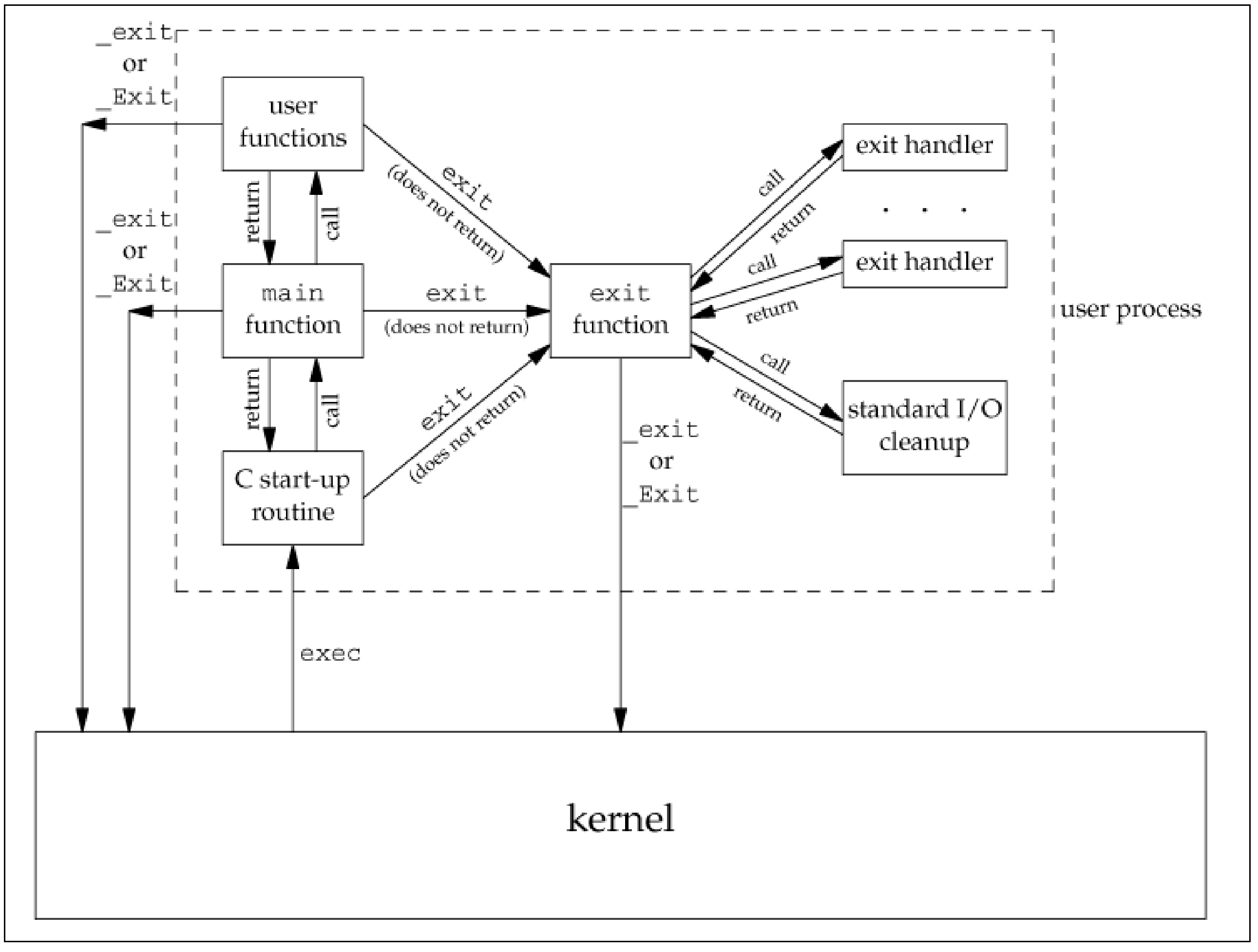
Returns: 0 if OK, nonzero on error This declaration says that we pass the address of a function as the argument to atexit. When this function is called, it is not passed any arguments and is not expected to return a value. The exit function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.
Example of exit handlers

```
#include "apue.h"

static void my_exit1(void);

static void my_exit2(void);

int main(void) {

        if (atexit(my_exit2) != 0)

        err_sys("can't register my_exit2");

        if (atexit(my_exit1) != 0)

        err_sys("can't register my_exit1");

        if (atexit(my_exit1) != 0)

        err_sys("can't register my_exit1");

        printf("main is done\n"); return(0); }

static void my_exit1(void) {

printf("first exit handler\n");}

static void my_exit2(void) {

printf("second exit handler\n");}
```

Output: $ ./a.out

```
main is done
first exit handler
first exit handler
second exit handler
```

The below figure summarizes how a C program is started and the various ways it can terminate.



COMMAND-LINE ARGUMENTS

When a program is executed, the process that does the exec can pass command-line arguments to the new program. Example: Echo all command-line arguments to standard output

```
#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); }
```

Output:

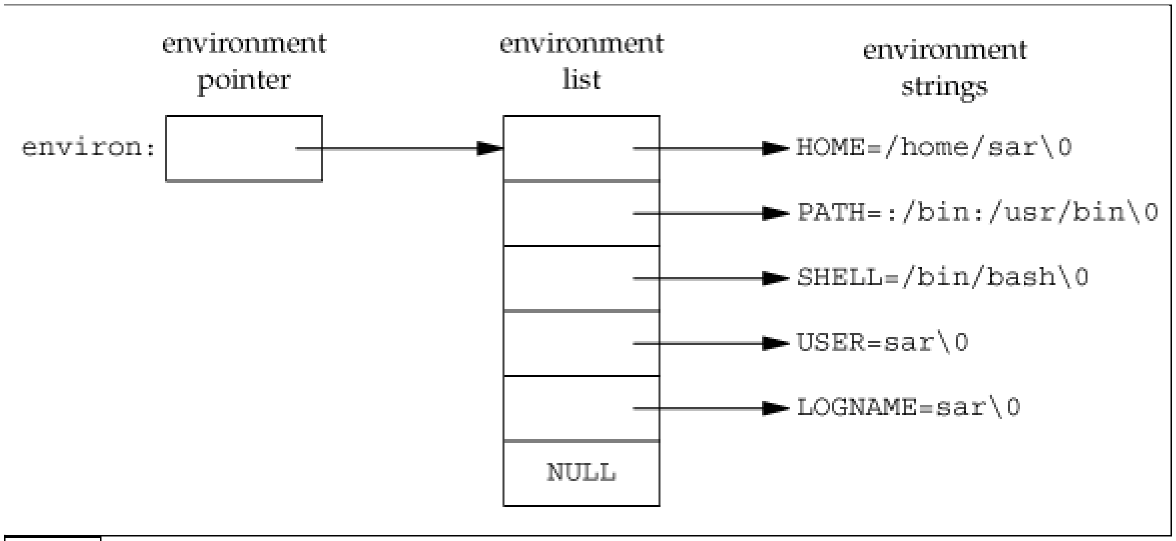
```
$ ./echoarg arg1 TEST foo

argv[0]: ./echoarg

argv[1]: arg1

argv[2]: TEST

argv[3]: foo
```

ENVIRONMENT LIST

Each program is also 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;**

Figure : Environment consisting of five C character strings

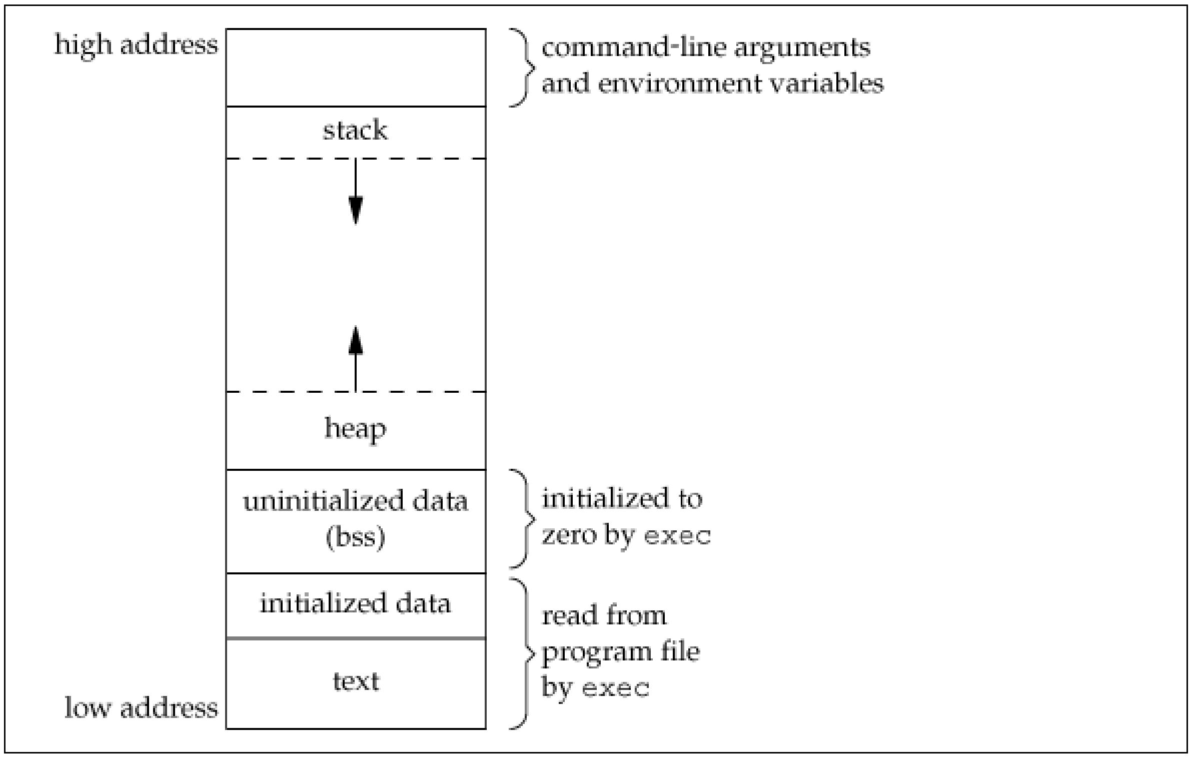


Generally any environmental variable is of the form: ***name=value.***

MEMORY LAYOUT OF A C PROGRAM

Historically, a C program has been composed of the following pieces:

- **Text segment**: the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.
- **Initialized data segment**: usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration **int maxcount = 99;** appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.
- **Uninitialized data segment**: often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration
  long sum[1000]; appearing outside any function causes this variable to be stored in the uninitialized data segment.
- **Stack**: where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.
- **Heap**: where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.


high address
low address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec

SHARED LIBRARIES

Nowadays most UNIX systems support shared libraries. Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference. This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time each shared library function is called. Another advantage of shared libraries is that, library functions can be replaced with new versions without having to re-link, edit every program that uses the library. With cc compiler we can use the option –g to indicate that we are using shared library.

MEMORY ALLOCATION

ISO C specifies three functions for memory allocation:

- malloc, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
- calloc, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
- realloc, which increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_tnobj, size_t size);
void *realloc(void *ptr, size_t newsize);
```

All three return: non-null pointer if OK, NULL on error **void free(void *ptr);**
The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object. Because the three alloc functions return a generic void * pointer, if we #include <stdlib.h>(to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type. The function free causes the space pointed to by ptr to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three alloc functions.
The realloc function lets us increase or decrease the size of a previously allocated area. For example, if we allocate room for 512 elements in an array that we fill in at runtime but find that we need room for more than 512 elements, we can call realloc. If there is room beyond the end of the existing region for the requested space, then realloc doesn't have to move anything; it simply allocates the additional area at the end and returns the same pointer that we passed it. But if there isn't room at the end of the existing region, realloc allocates another area that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area.

The allocation routines are usually implemented with the sbrk(2) system call. Although sbrk can expand or contract the memory of a process, most versions of malloc and free never decrease their memory size. The space that we free is available for a later allocation, but the freed space is not usually returned to the kernel; that space is kept in the malloc pool.

It is important to realize that most implementations allocate a little more space than is requested and use the additional space for record keeping the size of the allocated block, a pointer to the next allocated block, and the like. This means that writing past the end of an allocated area could overwrite this record-keeping information in a later block. These types of errors are often catastrophic, but difficult to find, because the error may not show up until much later. Also, it is possible to overwrite this record keeping by writing before the start of the allocated area.

Because memory allocation errors are difficult to track down, some systems provide versions of these functions that do additional error checking every time one of the three alloc functions or free is called. These versions of the functions are often specified by including a special library for the link editor. There are also publicly available sources that you can compile with special flags to enable additional runtime checking.

Alternate Memory Allocators

Many replacements for malloc and free are available.

- **libmalloc**
  SVR4-based systems, such as Solaris, include the libmalloc library, which provides a set of interfaces matching the ISO C memory allocation functions. The libmalloc library includes mallopt, a function that allows a process to set certain variables that control the operation of the storage allocator. A function called mallinfo is also available to provide statistics on the memory allocator.

- **vmalloc**
  Vo describes a memory allocator that allows processes to allocate memory using different techniques for different regions of memory. In addition to the functions specific to vmalloc, the library also provides emulations of the ISO C memory allocation functions.
- **quick-fit**
  Historically, the standard malloc algorithm used either a best-fit or a first-fit memory allocation strategy. Quick-fit is faster than either, but tends to use more memory. Free implementations of malloc and free based on quick-fit are readily available from several FTP sites.
- **alloca Function**
  The function alloca has the same calling sequence as malloc; however, instead of allocating memory from the heap, the memory is allocated from the stack frame of the current function. The advantage is that we don't have to free the space; it goes away automatically when the function returns. The alloca function increases the size of the stack frame. The disadvantage is that some systems can't support alloca, if it's impossible to increase the size of the stack frame after the function has been called.

ENVIRONMENT VARIABLES

The environment strings are usually of the form: ***name=value.*** The UNIX kernel never looks at these strings; their interpretation is up to the various applications. The shells, for example, use numerous environment variables. Some, such as HOME and USER, are set automatically at login, and others are for us to set. We normally set environment variables in a shell start-up file to control the shell's actions. The functions that we can use to set and fetch values from the variables are setenv, putenv, and getenv functions.

The prototype of these functions are

```
#include <stdlib.h>
char *getenv(const char *name);
```

Returns: pointer to value associated with name, NULL if not found.

Note that this function returns a pointer to the value of a **name=value** string. We should always use getenv to fetch a specific value from the environment, instead of accessing environ directly. In addition to fetching the value of an environment variable, sometimes we may want to set an environment variable. We may want to change the value of an existing variable or add a new variable to the environment. The prototypes of these functions are

```
#include <stdlib.h>
int putenv(char *str);
int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
```

All return: 0 if OK, nonzero on error.

- The **putenv** function takes a string of the form **name=value** and places it in the environment list. If name already exists, its old definition is first removed.
- The **setenv** function sets name to value. If name already exists in the environment, then
  (a) if rewrite is nonzero, the existing definition for name is first removed;
  (b) if rewrite is 0, an existing definition for name is not removed, name is not set to the new value, and no error occurs.
- The **unsetenv** function removes any definition of name. It is not an error if such a definition does not exist.

Note the difference between **putenv** and **setenv**. Whereas **setenv** must allocate memory to create the **name=value** string from its arguments, **putenv** is free to place the string passed to it directly into the environment.

Environment variables defined in the Single UNIX Specification

VARIABLES	**DESCRIPTION**
COLUMNS	terminal width
DATEMSK	getdate(3) template file pathname
HOME	home directory
LANG	name of locale
LC_ALL	name of locale
LC_COLLATE	name of locale for collation
LC_CTYPE	name of locale for character classification
LC_MESSAGES	name of locale for messages
LC_MONETARY	name of locale for monetary editing
LC_NUMERIC	name of locale for numeric editing
LC_TIME	name of locale for date/time formatting
LINES	terminal height
LOGNAME	login name
MSGVERB	fmtmsg(3) message components to process
NLSPATH	sequence of templates for message catalogs
PATH	list of path prefixes to search for executable file
PWD	absolute pathname of current working directory
SHELL	name of user's preferred shell
TERM	terminal type
TMPDIR	pathname of directory for creating temporary files
TZ	time zone information

NOTE:

If we're modifying an existing name:

- If the size of the new value is less than or equal to the size of the existing value, we can just copy the new string over the old string.
- If the size of the new value is larger than the old one, however, we must malloc to obtain room for the new string, copy the new string to this area, and then replace the old pointer in the environment list for name with the pointer to this allocated area.
- If we're adding a new name, it's more complicated. First, we have to call malloc to allocate room for the name=value string and copy the string to this area.
- Then, if it's the first time we've added a new name, we have to call malloc to obtain room for a new list of pointers. We copy the old environment list to this new area and store a pointer to the name=value string at the end of this list of pointers. We also store a null pointer at the end of this list, of course. Finally, we set environ to point to this new list of pointers.
- If this isn't the first time we've added new strings to the environment list, then we know that we've already allocated room for the list on the heap, so we just call realloc to allocate room for one more pointer. The pointer to the new name=value string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

setjmp AND longjmp FUNCTIONS

In C, we can't go to a label that's in another function. Instead, we must use the setjmp and longjmp functions to perform this type of branching. As we'll see, these two functions are useful for handling error conditions that occur in a deeply nested function call.

```
#include <setjmp.h>
int setjmp(jmp_bufenv);
```

Returns: 0 if called directly, nonzero if returning from a call to longjmp

```
void longjmp(jmp_bufenv, int val);
```

The setjmp function records or marks a location in a program code so that later when the longjmp function is called from some other function, the execution continues from the location onwards. The envvariable(the first argument) records the necessary information needed to continue execution. The env is of the jmp_buf defined in <setjmp.h> file, it contains the task.

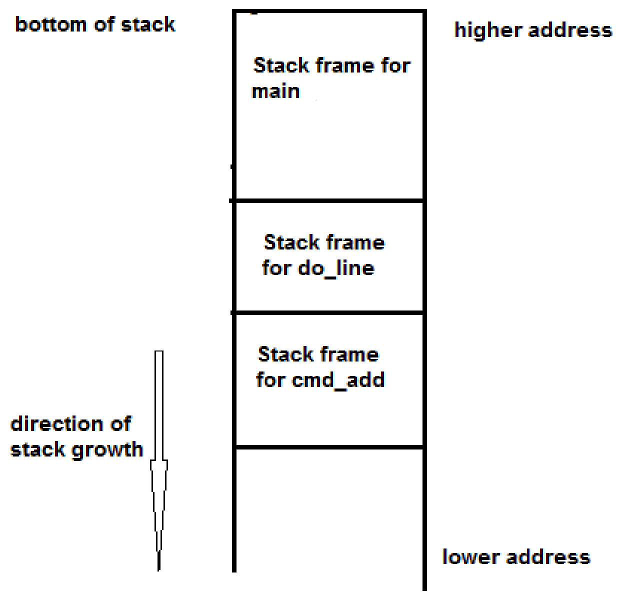
Example of setjmp and longjmp

```
#include "apue.h"

#include <setjmp.h>

#define TOK_ADD 5

jmp_bufjmpbuffer;
```

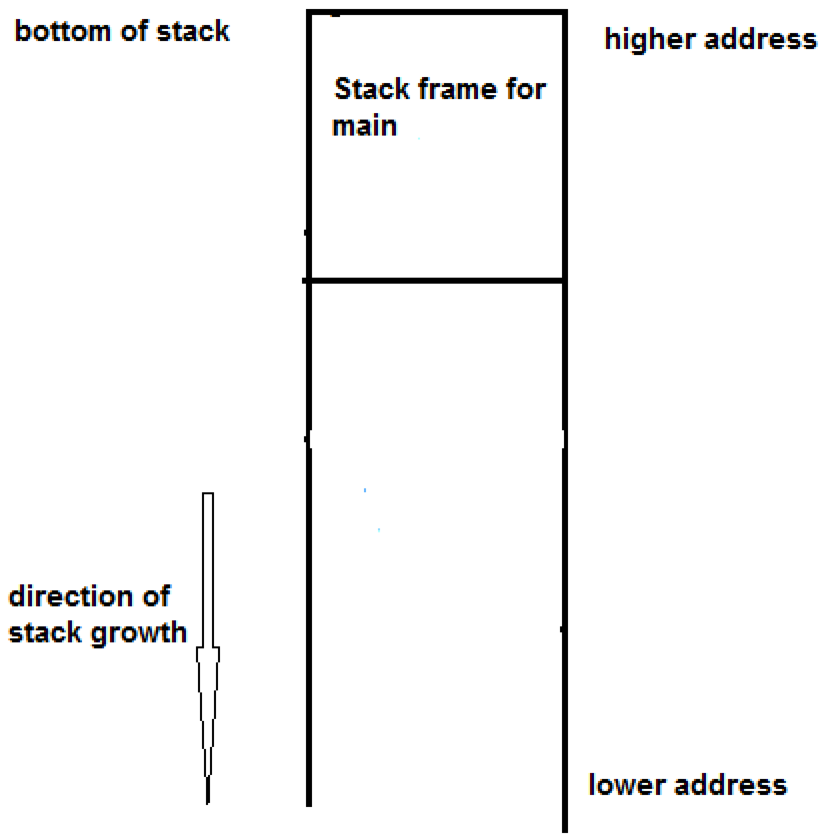
```
int main(void) {

char line[MAXLINE];

if (setjmp(jmpbuffer) != 0)

printf("error");

while (fgets(line, MAXLINE, stdin) != NULL)

do_line(line); exit(0); }

voidcmd_add(void) {

int token;

token = get_token();

if (token < 0)                         /* an error has occurred */

longjmp(jmpbuffer, 1);                 /* rest of processing for this command */

}
```

- The setjmp function always returns ‘0’ on its success when it is called directly in a process (for the first time).
- The longjmp function is called to transfer a program flow to a location that was stored in the env argument.
- The program code marked by the env must be in a function that is among the callers of the current function.
- When the process is jumping to the target function, all the stack space used in the current function and its callers, upto the target function are discarded by the longjmp function.
- The process resumes execution by re-executing the setjmp statement in the target function that is marked by env. The return value of setjmp function is the value(val), as specified in the longjmp function call.
- The ‘val’ should be nonzero, so that it can be used to indicate where and why the longjmp function was invoked and process can do error handling accordingly.

Note: The values of ***automatic*** and ***register*** variables are indeterminate when the longjmp is called but static and global variable are unaltered. The variables that we don’t want to roll back after longjmp are declared with keyword ‘volatile’.



Stack frames after cmd_add has been called



Stack frame after longjmp has been called

getrlimit AND setrlimit FUNCTIONS

Every process has a set of resource limits, some of which can be queried and changed by the getrlimit and setrlimit functions.

```
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, conststruct rlimit *rlptr);
```

Both return: 0 if OK, nonzero on error Each call to these two functions specifies a single resource and a pointer to the following structure:

```
struct rlimit {
rlim_trlim_cur; /* soft limit: current limit */
rlim_trlim_max; /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

- A process can change its soft limit to a value less than or equal to its hard limit.
- A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
- Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant RLIM_INFINITY.

RLIMIT_AS	The maximum size in bytes of a process's total available memory.
RLIMIT_CORE	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
RLIMIT_CPU	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process.
RLIMIT_DATA	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap.
RLIMIT_FSIZE	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the SIGXFSZ signal.
RLIMIT_LOCKS	The maximum number of file locks a process can hold.
RLIMIT_MEMLOCK	The maximum amount of memory in bytes that a process can lock into memory using mlock(2).
RLIMIT_NOFILE	The maximum number of open files per process. Changing this limit affects the value returned by the sysconf function for its _SC_OPEN_MAX argument
RLIMIT_NPROC	The maximum number of child processes per real user ID. Changing this limit affects the value returned for _SC_CHILD_MAX by the sysconf function
RLIMIT_RSS	Maximum resident set size (RSS) in bytes. If

	available physical memory is low, the kernel takes memory from processes that exceed their RSS.
RLIMIT_SBSIZE	The maximum size in bytes of socket buffers that a user can consume at any given time.
RLIMIT_STACK	The maximum size in bytes of the stack.
RLIMIT_VMEM	This is a synonym for RLIMIT_AS.

The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits needs to be built into the shells to affect all our future processes.

Example: Print the current resource limits

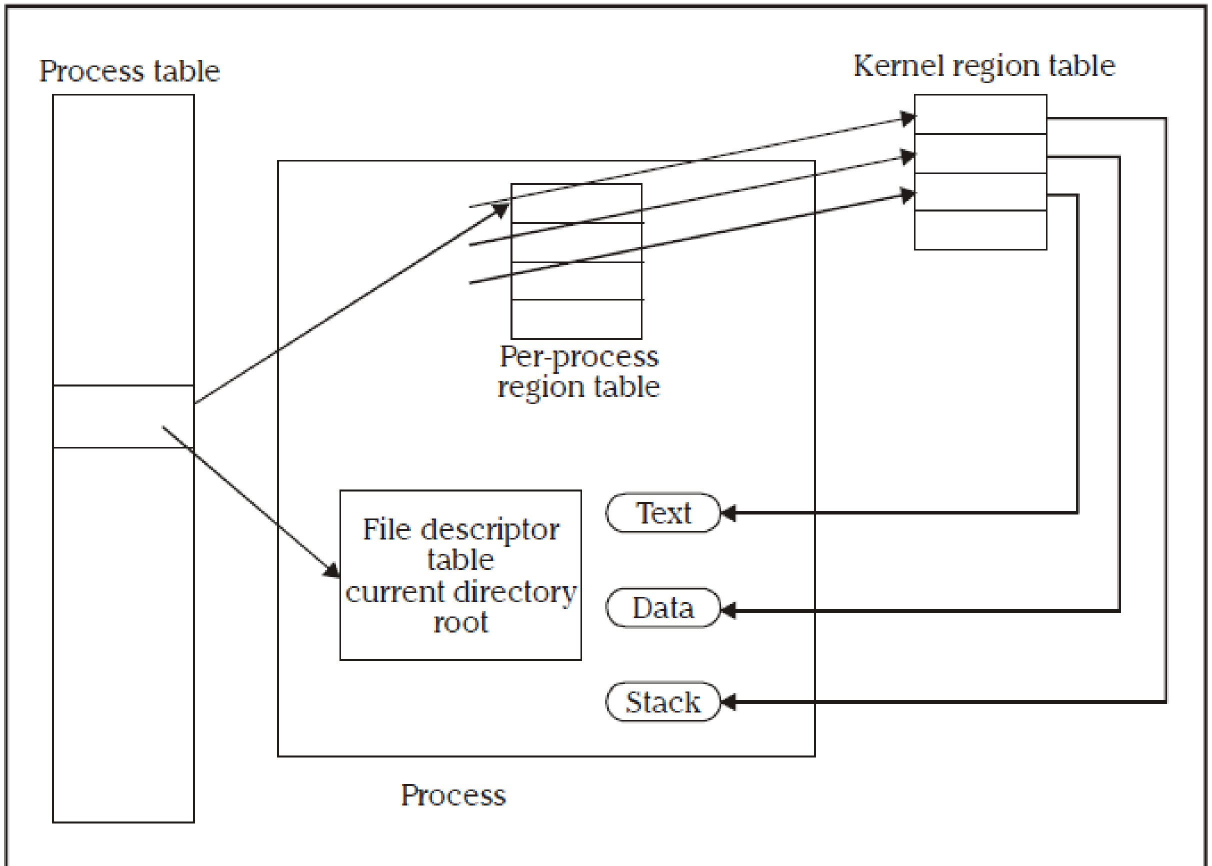
```
#include "apue.h"

#if defined(BSD) || defined(MACOS)

#include <sys/time.h>

#define FMT "%10lld "

#else #define FMT "%10ld "

#endif

#include <sys/resource.h>

#define doit(name) pr_limits(#name, name)

static void pr_limits(char *, int);

int main(void) {

 #ifdef RLIMIT_AS doit(RLIMIT_AS);

#endifdoit(RLIMIT_CORE);

doit(RLIMIT_CPU);

doit(RLIMIT_DATA);

doit(RLIMIT_FSIZE);

#ifdef RLIMIT_LOCKS

doit(RLIMIT_LOCKS);

#endif

 #ifdef RLIMIT_MEMLOCK

doit(RLIMIT_MEMLOCK);

#endifdoit(RLIMIT_NOFILE);
#ifdef RLIMIT_NPROC
doit(RLIMIT_NPROC);
 #endif
#ifdef RLIMIT_RSS
doit(RLIMIT_RSS);
#endif
#ifdef RLIMIT_SBSIZE
doit(RLIMIT_SBSIZE);
#endif
doit(RLIMIT_STACK);
 #ifdef RLIMIT_VMEM
doit(RLIMIT_VMEM);
#endif
exit(0); }
static void pr_limits(char *name, int resource) {
        struct rlimit limit;
        if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
        printf("%-14s ", name);
        if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite) ");
        else
        printf(FMT, limit.rlim_cur);
        if (limit.rlim_max == RLIM_INFINITY)
        printf("(infinite)");
        else
        printf(FMT, limit.rlim_max);
        putchar((int)'\n');
}
```

UNIX KERNEL SUPPORT FOR PROCESS

The data structure and execution of processes are dependent on operating system implementation. A UNIX process consists minimally of a text segment, a data segment and a stack segment. A segment is an area of memory that is managed by the system as a unit.

- A text segment consists of the program text in machine executable instruction code format.
- The data segment contains static and global variables and their corresponding data.
- A stack segment contains runtime variables and the return addresses of all active functions for a process.

UNIX kernel has a process table that keeps track of all active process present in the system. Some of these processes belongs to the kernel and are called as “system process”. Every entry in the process table contains pointers to the text, data and the stack segments and also to U-area of a process. U-area of a process is an extension of the process table entry and contains other process specific data such as the file descriptor table, current root and working directory inode numbers and set of system imposed process limits.



A Unix process data structure

All processes in UNIX system expect the process that is created by the system boot code, are created by the fork system call. After the fork system call, once the child process is created, both the parent and child processes resumes execution. When a process is created by fork, it contains duplicated copies of the text, data and stack segments of its parent as shown in the Figure below. Also it has a file descriptor table, which contains reference to the same opened files as the parent, such that they both share the same file pointer to each opened files.

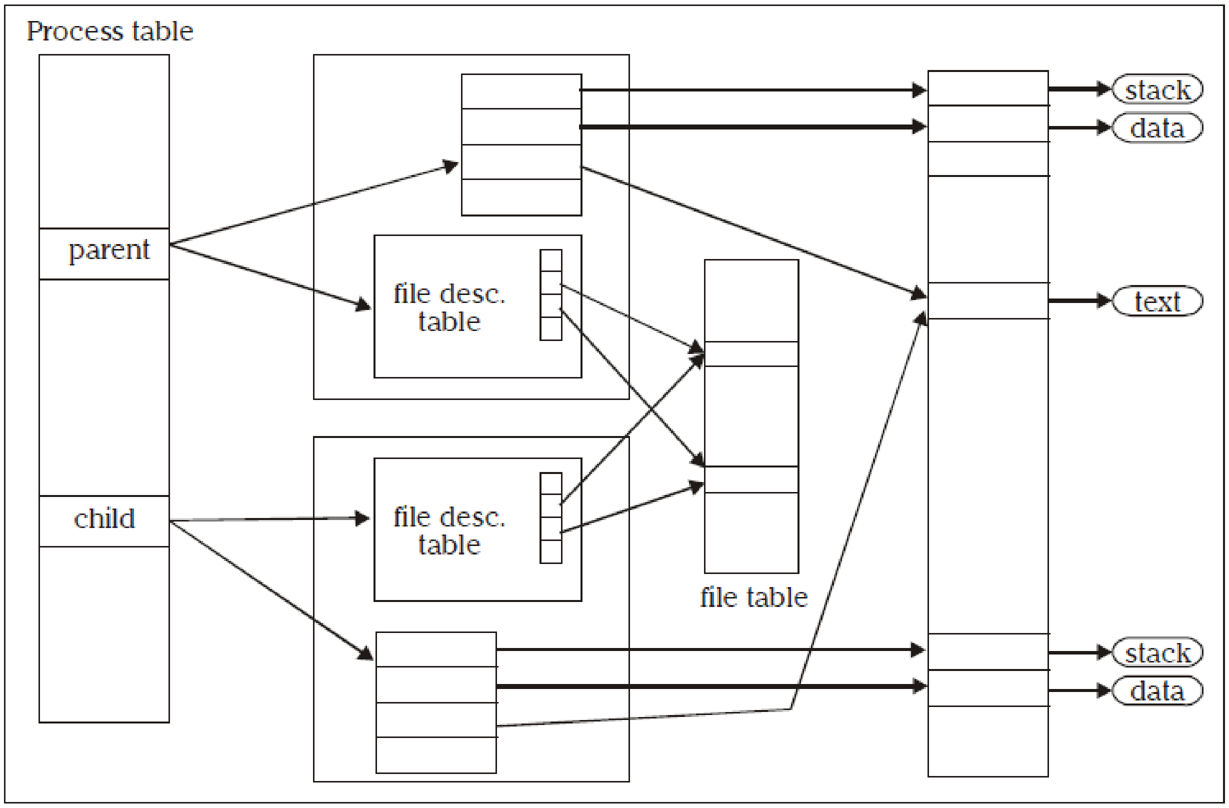


Figure: Parent & child relationship after fork

- ✓ The process will be assigned with attributes, which are either inherited from its parent or will be set by the kernel.
- ✓ **A real user identification number (rUID)**: the user ID of a user who created the parent process.
- ✓ **A real group identification number (rGID)**: the group ID of a user who created that parent process.
- ✓ **An effective user identification number (eUID)**: this allows the process to access and create files with the same privileges as the program file owner.
- ✓ **An effective group identification number (eGID)**: this allows the process to access and create files with the same privileges as the group to which the program file belongs.
- ✓ **Saved set-UID and saved set-GID**: these are the assigned eUID and eGID of the process respectively.
- ✓ **Process group identification number (PGID) and session identification number (SID):** these identify the process group and session of which the process is member.
- ✓ **Supplementary group identification numbers:** this is a set of additional group IDs for a user who created the process.
- ✓ **Current directory:** this is the reference (inode number) to a working directory file.
- ✓ **Root directory**: this is the reference to a root directory.
- ✓ **Signal handling**: the signal handling settings.

- ✓ **Signal mask**: a signal mask that specifies which signals are to be blocked.
- ✓ **Unmask**: a file mode mask that is used in creation of files to specify which accession rights should be taken out.
- ✓ **Nice value**: the process scheduling priority value.
- ✓ **Controlling terminal**: the controlling terminal of the process.
- ✓ In addition to the above attributes, the following attributes are different between the parent and child processes:
- ✓ **Process identification number (PID)**: an integer identification number that is unique per process in an entire operating system.
- ✓ **Parent process identification number (PPID)**: the parent process PID.
- ✓ **Pending signals**: the set of signals that are pending delivery to the parent process.
- ✓ **Alarm clock time**: the process alarm clock time is reset to zero in the child process.
- ✓ **File locks**: the set of file locks owned by the parent process is not inherited by the chid process.

fork and ***exec*** are commonly used together to spawn a sub-process to execute a different program.

The advantages of this method are:

- A process can create multiple processes to execute multiple programs concurrently.
- Because each child process executes in its own virtual address space, the parent process is not affected by the execution status of its child process.

Chapter 2-Process Control

INTRODUCTION

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

PROCESS IDENTIFIERS

Every process has a unique process ID, a non negative integer ,because it is the only well-known identifier of a process that is always unique.
Unique process IDs are reused, when a processes terminates, their IDs become candidates for reuse.

- Process ID 0:The scheduler process often known as swapper. Which is part of the kernel and is known as system process.
- Process ID 1: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(old version), /sbin/init(new version).
    - This init process is responsible for bringing up a UNIX system after the kernel has been bootstrapped.
    - Init usually reads the system dependent initialization files and brings the system to a certain state, such as multiuser.
    - The init process never dies. This is a normal process, not a system process within the kernel and it does run with superuser privileges.
    - Each Unix system implementations has its own set of kernel processes that provide OS services.
- Process ID 2: Known as page daemon, responsible for supporting paging of the virtual memory system.

#include<unistd.h>
pid_tgetpid(void);
Returns: process ID of calling process
pid_tgetppid(void);
Returns: parent process ID of calling process
uid_tgetuid(void);
Returns: real user ID of calling process
uid_tgeteuid(void);
Returns: effective user ID of calling process
gid_tgetgid(void);
Returns: real group ID of calling process
gid_tgetegid(void);
Returns: effective group ID of calling process

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 getppidto 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 programs: **Program 1** /* Program to demonstrate fork function Program name – fork1.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.

Program 2 /* Program name – fork2.c */

```
#include<sys/types.h>
#include<unistd.h>
int main( ) {
printf(“\n 6 sem “);
fork( );
printf(“\n hello USP”); }
```

Output :

```
$ cc fork1.c
$ ./a.out
6 sem
hello USP
hello USP
```

Note: The statement 6 sem is executed only once by the parent because it is called before fork and statement hello USP is executed twice by child and parent.

File Sharing

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 8.2.

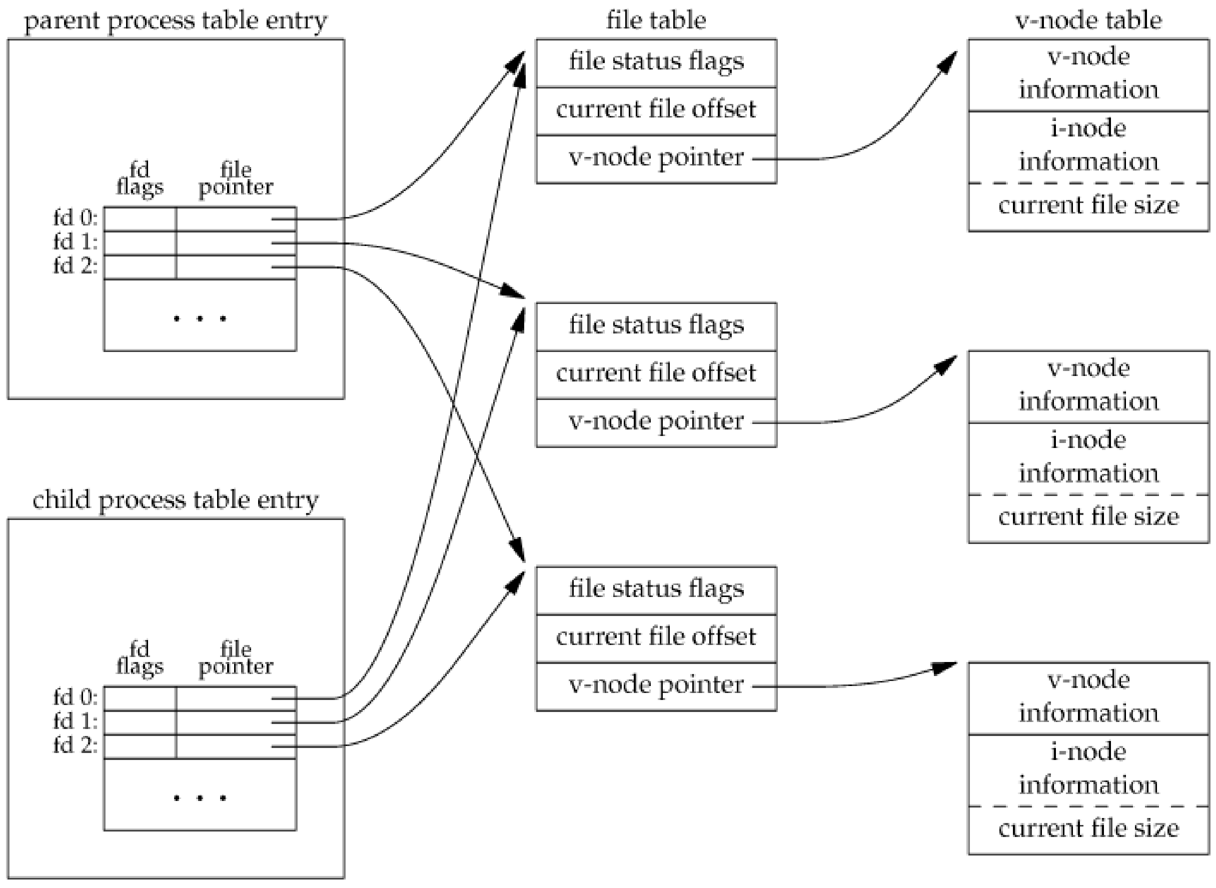


Figure 8.2 Sharing of open files between parent and child after fork

- 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.

- There are two normal cases for 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.

There are numerous other properties of the parent that are inherited by the child:

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

The differences between the parent and child are

- ○ 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_cstimevalues 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

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

vfork FUNCTION

- 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.

Example of vfork function

```
#include "apue.h"
int glob = 6;                           /* external variable in initialized data */
int main(void) {
intvar;                                 /* automatic variable on the stack */
pid_tpid;
var = 88;
printf("before vfork\n");               /* we don't flush stdio */
if ((pid = vfork()) < 0) {
err_sys("vfork error");
}
else if (pid == 0) {                    /* child */
glob++;                                 /* modify parent's variables */
var++;
_exit(0);                                       /* child terminates */
}                                               /* * Parent continues here. */
printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
exit(0);
}
```

Output:

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

exit FUNCTIONS

A process can terminate normally in five ways:

1. **Executing a return from the main function** is equivalent to calling exit.
2. **Calling the exit function:** This function is defined by the ISO C and includes the calling of all exit handlers that have been registered by calling atexit function and closing all std I/O streams. Because ISO C does not deal with file descriptors, multiprocesses and job control.
3. **Calling the _exit or _Exit function**: ISO C defines _Exit to provide a way for a process to terminate without running exit handlers or signal handlers. and do not perform any clean up operations.

In most UNIX system implementations, exit(3) is a function in the standard C library, whereas _exit(2) is a system call.

4. **Executing a return from the start routine of the last thread in the process**: The return value of thread is not used as the return value of the process. When the last thread returns from its start routine, the process exits with a termination status of 0.
5. **Calling the pthread_exit function from the last thread in the process:** The exit status of the process is always '0', regardless of the argument passed to the pthread_exit.

The three forms of abnormal termination are as follows:

1. **Calling abort**. This is a special case of the next item, as it generates the SIGABRT signal.
2. **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.
3. **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.

wait AND waitpid FUNCTIONS

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 waitpidcan:

- ✓ 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_twaitpid(pid_tpid, 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.

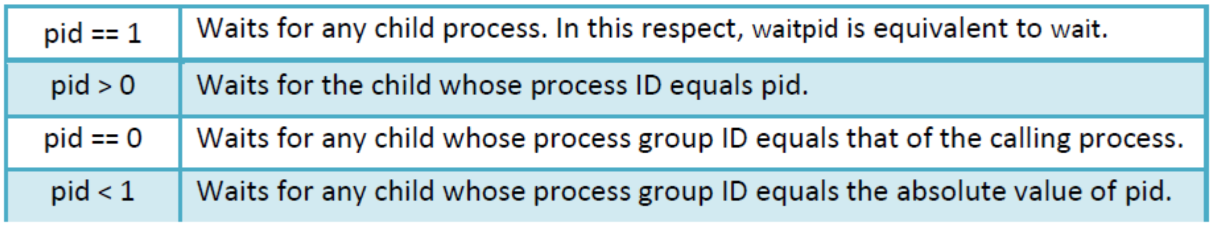
If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates. For both functions, the argument statloc is a pointer to an 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. Print a description of the exit status

```
#include "apue.h"

#include <sys/wait.h>

voidpr_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));

}
```

Program to Demonstrate various exit statuses

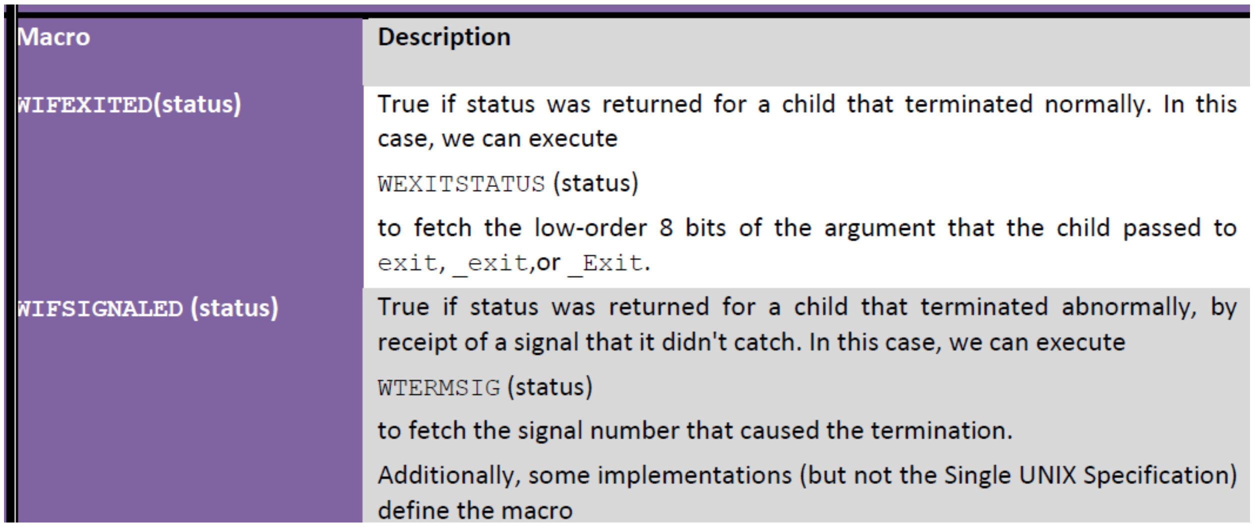
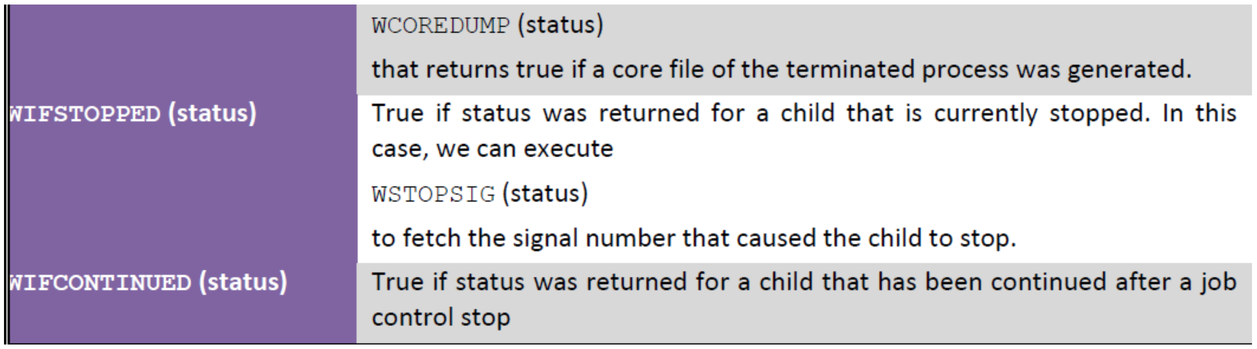
```
#include "apue.h"

#include <sys/wait.h>

int main(void) {

pid_tpid;

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);

}
```

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

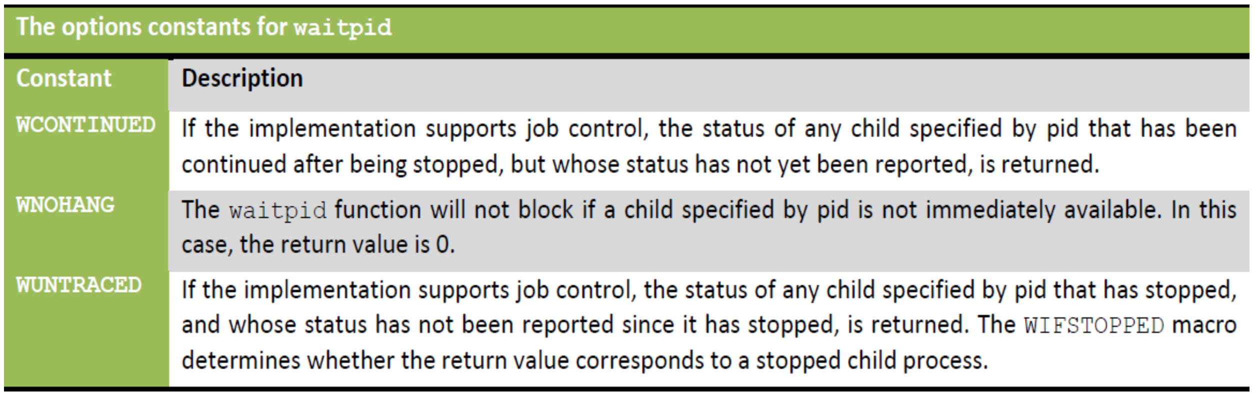


pid == 1	Waits for any child process. In this respect, waitpid is equivalent to wait.
pid > 0	Waits for the child whose process ID equals pid.
pid == 0	Waits for any child whose process group ID equals that of the calling process.
pid < 1	Waits for any child whose process group ID equals the absolute value of pid.

Macros to examine the termination status returned by wait an 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



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.
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.

The waitpid function provides three features that aren't provided by the wait function.

- The waitpid function lets us wait for one particular process, whereas the wait function returns the status of any terminated child. We'll return to this feature when we discuss the popen function.
- The waitpid function provides a non blocking version of wait. There are times when we want to fetch a child's status, but we don't want to block.

- The waitpid function provides support for job control with the WUNTRACED and WCONTINUED options.

Program to Avoid zombie processes by calling fork twice

```
#include "apue.h"
#include <sys/wait.h>
int main(void) {
pid_tpid;
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); /* 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
```

waitidFUNCTION

The waitidfunction is similar to waitpid, but provides extra flexibility.

#include <sys/wait.h>

int waited(idtype_tidtype, id_t id, siginfo_t *infop, int options);

Returns: 0 if OK, -1 on error

The *idtype* constants for waited are as follows:

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.

The options argument is a bitwise OR of the flags as shown below: these flags indicate which state changes the caller is interested in.

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.

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 prototypes of these functions are:

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

#include <sys/wait.h>

#include <sys/time.h>

#include <sys/resource.h>

pid_twait3(int *statloc, int options, struct rusage *rusage);

pid_twait4(pid_tpid, int *statloc, int options, struct rusage *rusage);
```

Both return: process ID if OK,-1 on error The resource information includes such statistics as the amount of user CPU time, the amount of system CPU time, number of page faults, number of signals received etc. the resource information is available only for terminated child process not for the process that were stopped due to job control.

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_tpid;

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
```

Program modification to avoid race condition

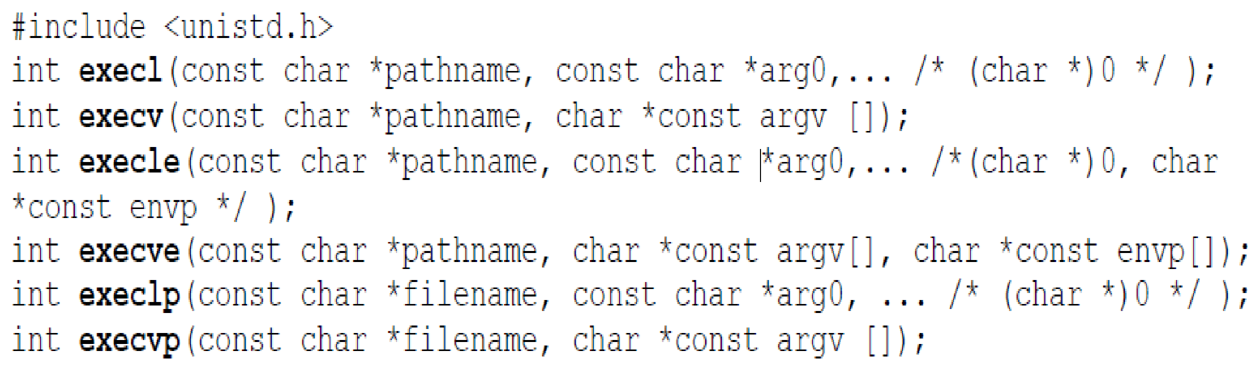
```
#include "apue.h"
static void charatatime(char *);
int main(void) {
```

```
pid_tpid;
TELL_WAIT();
if ((pid = fork()) < 0) {
err_sys("fork error");
}
else if (pid == 0) {
WAIT_PARENT();                                    /* parent goes first */
charatatime("output from child\n");
}
else {
charatatime("output from parent\n");
TELL_CHILD(pid);
}
exit(0); }
static voidcharatatime(char *str)  {
char *ptr;
int c;
setbuf(stdout, NULL);                             /* set unbuffered */
for (ptr = str; (c = *ptr++) != 0; )
putc(c, stdout);
}
```

When we run this program, the output is as we expect; there is no intermixing of output from the two processes.

execFUNCTIONS

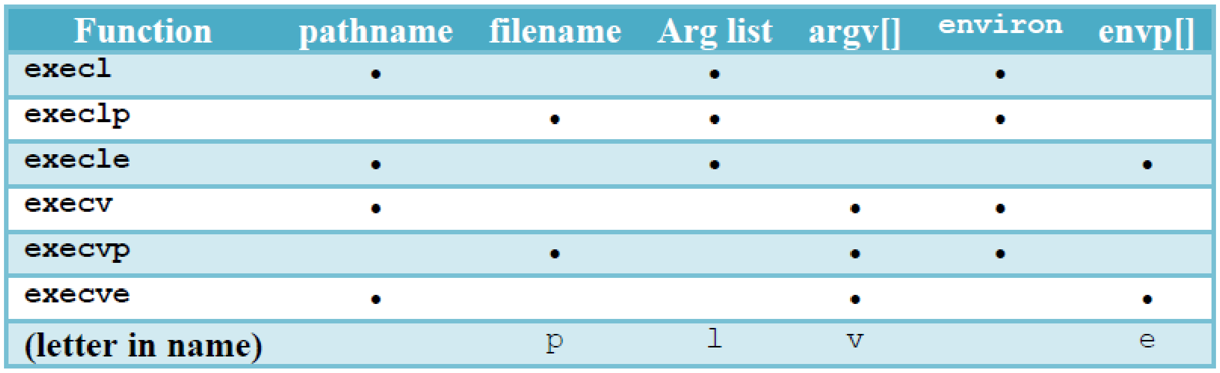
When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process - its text, 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.

- ✓ The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument. When a filename argument is specified
  1. If filename contains a slash, it is taken as a pathname.
  2. Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.
- ✓ The next difference concerns the passing of the argument list (l stands 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.
- ✓ 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.



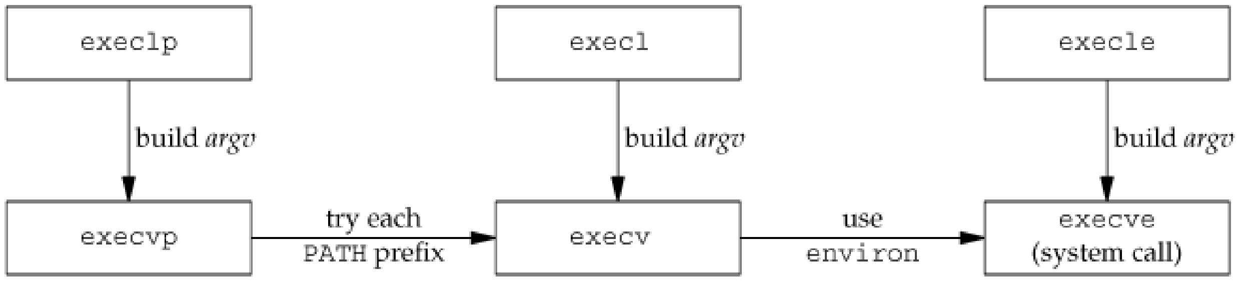
Function	pathname	filename	Arg list	argv[]	environ	envp[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(letter in name)		p	l	v		e

The above table shows the differences among the 6 exec functions.

We've mentioned that the process ID does not change after an exec, but the new program inherits additional properties from the calling process:

- o Process ID and parent process ID
- o Real user ID and real group ID
- o Supplementary group IDs
- o Process group ID
- o 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.



Relationship of the six exec functions

Example of exec functions

```
#include "apue.h"
#include <sys/wait.h>
char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int main(void) {
pid_tpid;
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); }
```

Output:

```
$ ./a.out
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
```

Note that the shell prompt appeared before the printing of argv[0] from the second exec. This is because the parent did not wait for this child process to finish.

Echo all command-line arguments and all environment strings

```
#include "apue.h"
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);
}
```

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>
intsetuid(uid_tuid);
intsetgid(gid_tgid);
```

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.

We can make a few statements about the three 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.

setreuid and setregid Functions

Swapping of the real user ID and the effective user ID with the setreuid function.

```
#include <unistd.h>
int setreuid(uid_truid, uid_teuid);
int setregid(gid_trgid, gid_tegid);
```

Both return : 0 if OK, -1 on error

We can supply a value of 1 for any of the arguments to indicate that the corresponding ID should remain unchanged. The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID. This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user- ID operations.

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_tuid);

int setegid(gid_tgid);                    Both return : 0 if OK, 1 on error
```

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID. For a privileged user, only the effective user ID is set to uid. (This differs from the setuid function, which changes all three user IDs.)

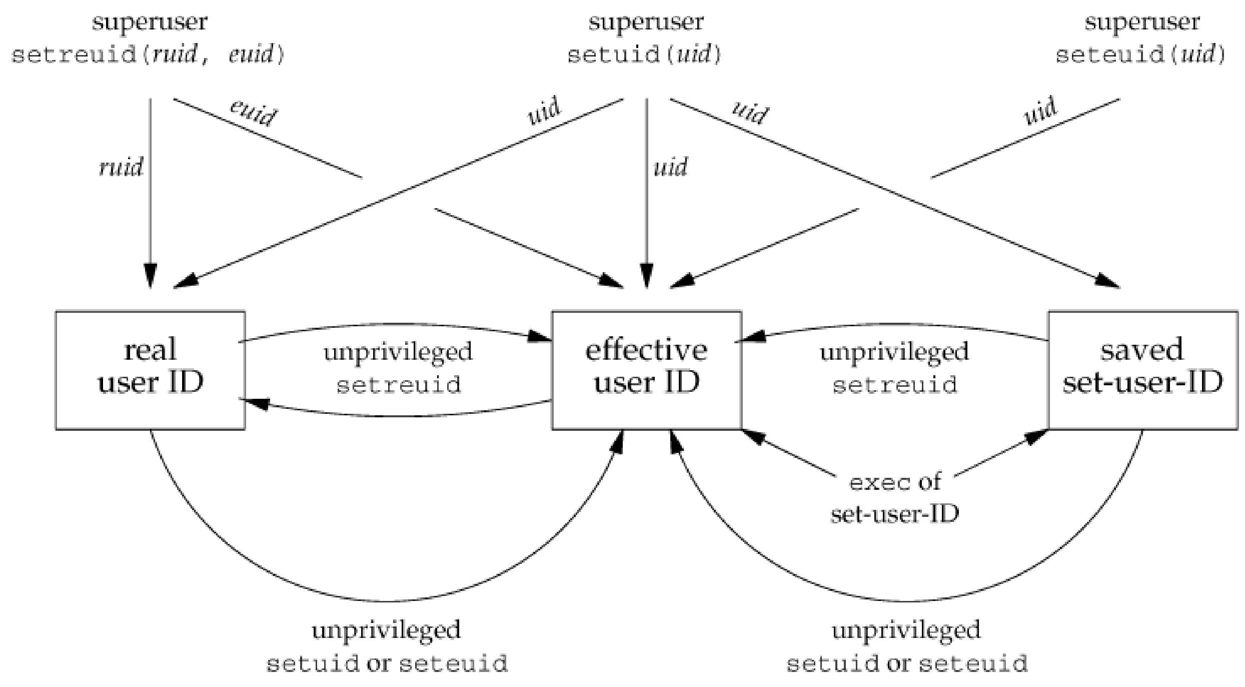


Figure: Summary of all the functions that set the various user Ids

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 file a 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. A program that execs an interpreter file.

```
#include "apue.h"

#include <sys/wait.h>

int main(void) {

pid_tpid;

if ((pid = fork()) < 0) {

err_sys("fork error"); }

else if (pid == 0) {                                              /* child */

if (execl("/home/sar/bin/testinterp","testinterp","myarg1","MY ARG2",(char *)0) < 0)

err_sys("execl error"); }

if (waitpid(pid, NULL, 0) < 0)                                    /* parent */

err_sys("waitpid error");

exit(0);

 }
```

Output:

```
$ cat /home/sar/bin/testinterp
```

```
#!/home/sar/bin/echoarg foo

$ ./a.out

argv[0]: /home/sar/bin/echoarg

argv[1]: foo

argv[2]: /home/sar/bin/testinterp

argv[3]: myarg1

argv[4]: MY ARG2
```

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: The system function, without signal handling

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

#include <errno.h>

#include <unistd.h>

int system(const char *cmdstring)                 /* version without signal handling */

{

pid_tpid;

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);

}
```

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);

 }
```

Program: Execute the command-line argument using system

```
#include "apue.h"

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

int status; if (argc< 2)
err_quit("command-line argument required");

if ((status = system(argv[1])) < 0)

err_sys("system() error");

pr_exit(status);

exit(0);

}
```

Program: Print real and effective user IDs

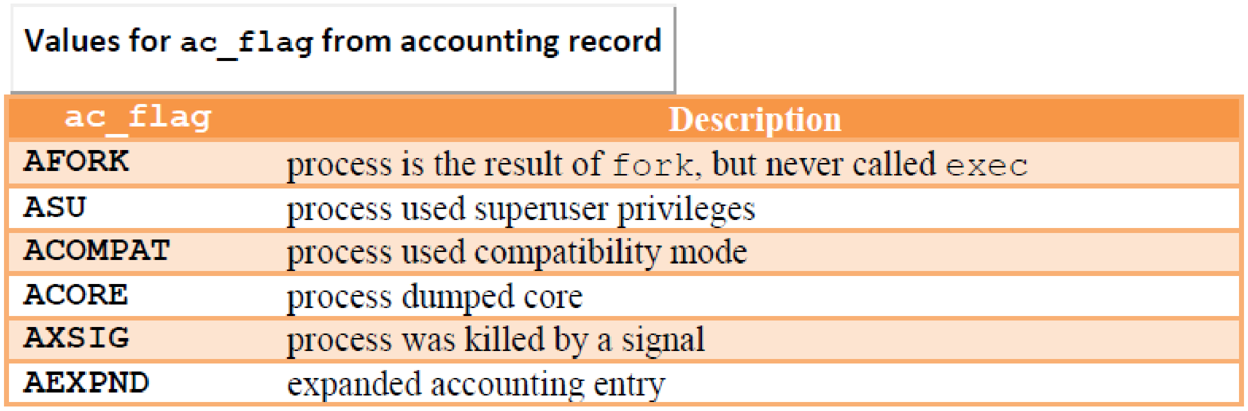
```
#include "apue.h"

int main(void) {

printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());

exit(0);

}
```

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.

The structure of the accounting records is defined in the header <sys/acct.h>and looks something like **typedefu_shortcomp_t; /* 3-bit base 8 exponent; 13-bit fraction */**

```
struct acct {

charac_flag; /* flag */

charac_stat; /* termination status (signal & core flag only) */ /* (Solaris only) */ uid_tac_uid; /* real user ID */
gid_tac_gid; /* real group ID */

dev_tac_tty; /* controlling terminal */

time_tac_btime; /* starting calendar time */

comp_tac_utime; /* user CPU time (clock ticks) */

comp_tac_stime; /* system CPU time (clock ticks) */

comp_tac_etime; /* elapsed time (clock ticks) */

comp_tac_mem; /* average memory usage */

comp_tac_io; /* bytes transferred (by read and write) */ /* "blocks" on BSD systems */ comp_tac_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 */

};
```

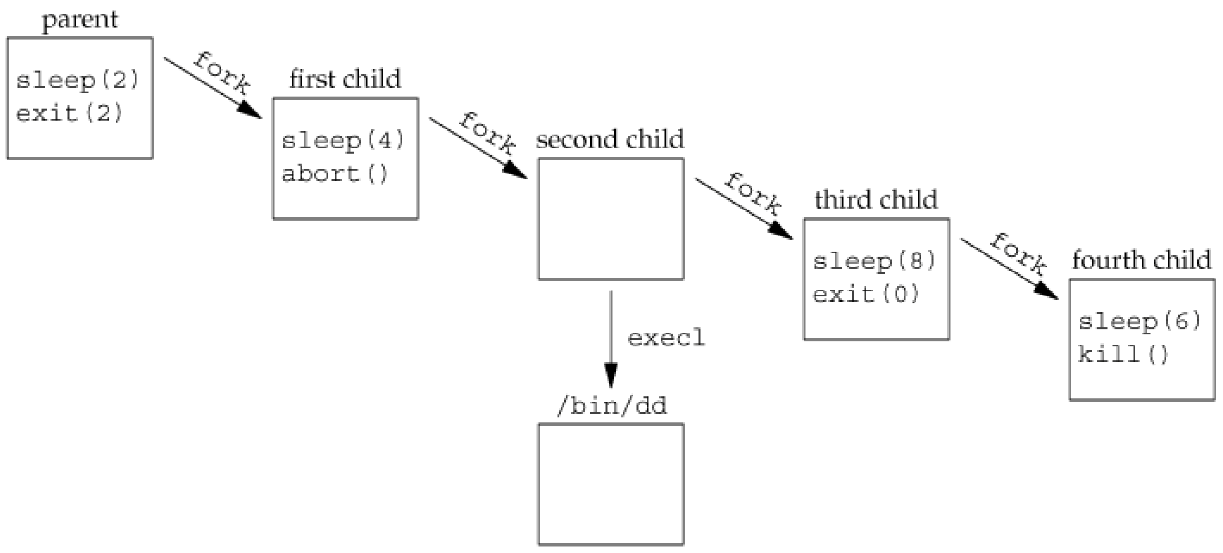


Values for `ac_flag` from accounting record

`ac_flag`	Description
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

Program to generate accounting data

```
#include "apue.h"

int main(void) {

pid_tpid;

if ((pid = fork()) < 0)

err_sys("fork error");

else if (pid != 0) {                                    /* parent */

sleep(2);

exit(2);                                                /* terminate with exit status 2 */
} /* first child */
if ((pid = fork()) < 0)
err_sys("fork error");
else if (pid != 0) {
sleep(4);
abort();                                        /* terminate with core dump */
}                                               /* second child */
if ((pid = fork()) < 0)
err_sys("fork error");
else if (pid != 0) {
execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
exit(7);                                        /* shouldn't get here */
} /* third child */
if ((pid = fork()) < 0)
err_sys("fork error");
else if (pid != 0) {
sleep(8);
exit(0);                                        /* normal exit */
}                                               /* fourth child */
sleep(6);
kill(getpid(), SIGKILL);                        /* terminate w/signal, no core dump */
exit(6);                                        /* shouldn't get here */
}
```



Process structure for accounting example

USER IDENTIFICATION

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 getlogin function 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.

PROCESS TIMES

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 tms structure pointed to by buf:

```
structtms {
clock_ttms_utime;                              /* user CPU time */
clock_ttms_stime;                              /* system CPU time */
clock_ttms_cutime;                             /* user CPU time, terminated children */
clock_ttms_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.

Chapter 3:PROCESS RELATIONSHIPS

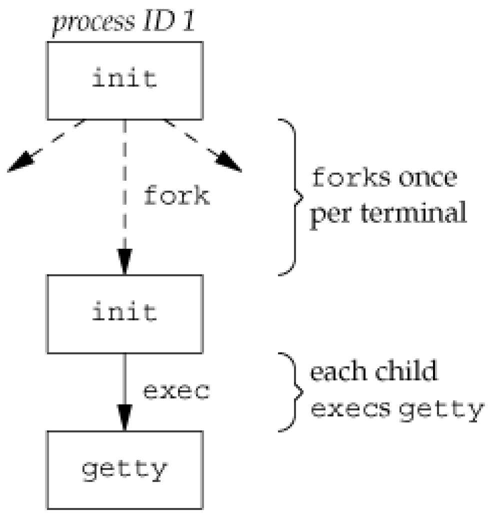
INTRODUCTION

In this chapter, we'll look at process groups in more detail and the concept of sessions that was introduced by POSIX.1. We'll also look at the relationship between the login shell that is invoked for us when we log in and all the processes that we start from our login shell.

TERMINAL LOGINS

The terminals were either local (directly connected) or remote (connected through a modem). In either case, these logins came through a terminal device driver in the kernel.
The system administrator creates a file, usually /etc/ttys, that has one line per terminal device. Each line specifies the name of the device and other parameters that are passed to the getty program. One parameter is the baud rate of the terminal, for example.
When the system is bootstrapped, the kernel creates process ID 1, the init process, and it is init that brings the system up multiuser. The init process reads the file /etc/ttys and, for every terminal device that allows a login, does a fork followed by an exec of the program getty. This gives us the processes shown in Figure 9.1.

Figure 9.1. Processes invoked by init to allow terminal logins



All the processes shown in Figure 9.1 have a real user ID of 0 and an effective user ID of 0 (i.e., they all have super user privileges). The init process also execs the getty program with an empty environment.
It is getty that calls open for the terminal device. The terminal is opened for reading and writing. If the device is a modem, the open may delay inside the device driver until the modem is dialed and the call is answered. Once the device is open, file descriptors 0, 1, and 2 are set to the device. Then getty outputs something like login: and waits for us to enter our user name. When we enter our user name, getty's job is complete, and it then invokes the login program, similar to **execle("/bin/login", "login", "-p", username, (char *)0, envp);**

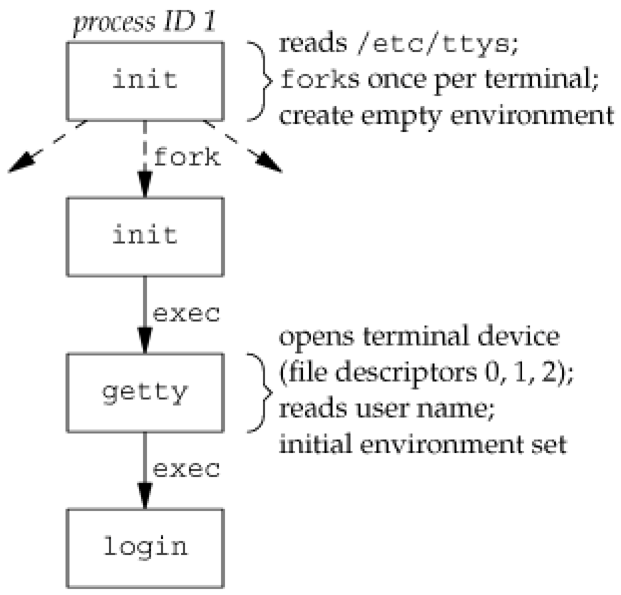


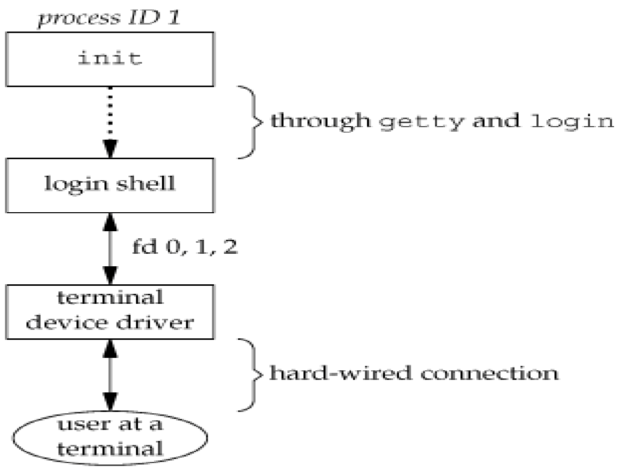
Figure 9.2. State of processes after login has been invoked

All the processes shown in Figure 9.2 have superuser privileges, since the original init process has superuser privileges. If we log in correctly, login will

- Change to our home directory (chdir)
- Change the ownership of our terminal device (chown) so we own it
- Change the access permissions for our terminal device so we have permission to read from and write to it
- Set our group IDs by calling setgid and init groups
- Initialize the environment with all the information that login has: our home directory (HOME), shell (SHELL), user name (USER and LOGNAME), and a default path (PATH)
- Change to our user ID (setuid) and invoke our login shell, as in **execl("/bin/sh", "-sh", (char *)0);**

The minus sign as the first character of argv[0] is a flag to all the shells that they are being invoked as a login shell. The shells can look at this character and modify their start-up accordingly.

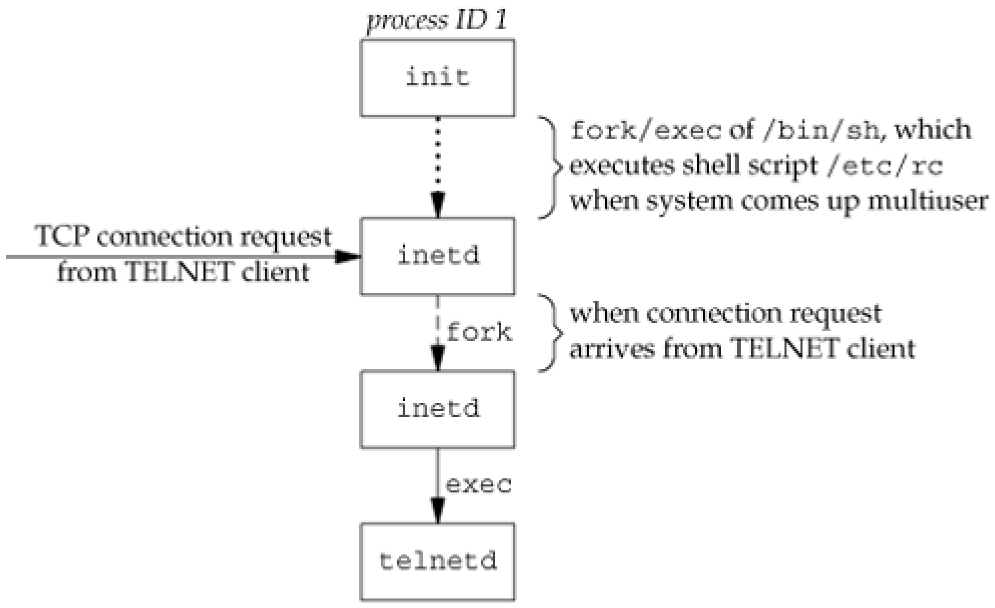
Figure 9.3. Arrangement of processes after everything is set for a terminal login



NETWORK LOGINS

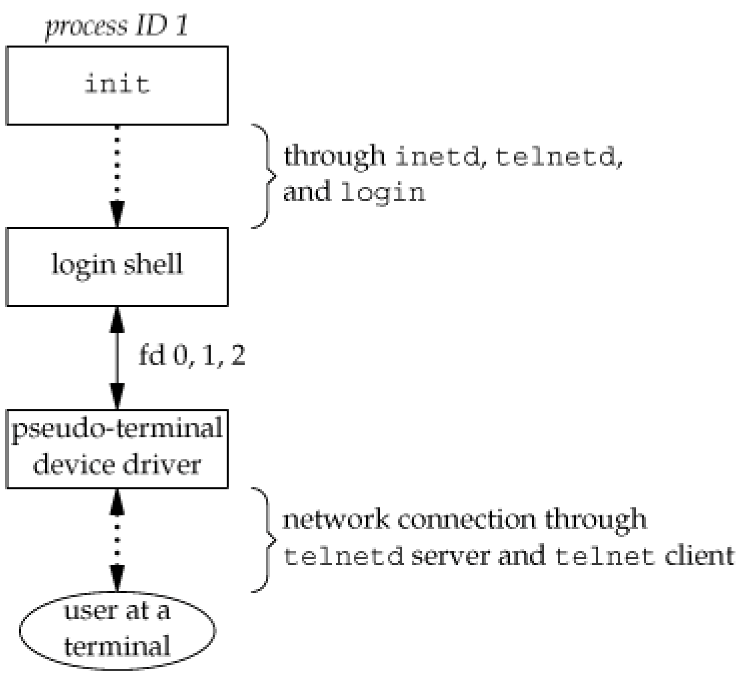
- The main (physical) difference between logging in to a system through a serial terminal and logging in to a system through a network is that the connection between the terminal and the computer isn't point-to-point.
- With the terminal logins that we described in the previous section, init knows which terminal devices are enabled for logins and spawns a getty process for each device.
- In the case of network logins, however, all the logins come through the kernel's network interface drivers (e.g., the Ethernet driver).
- Let's assume that a TCP connection request arrives for the TELNET server. TELNET is a remote login application that uses the TCP protocol.
- A user on another host (that is connected to the server's host through a network of some form) or on the same host initiates the login by starting the TELNET client: **telnet hostname**
- The client opens a TCP connection to hostname, and the program that's started on hostname is called the TELNET server.
- The client and the server then exchange data across the TCP connection using the TELNET application protocol.
- What has happened is that the user who started the client program is now logged in to the server's host. (This assumes, of course, that the user has a valid account on the server's host.) Figure 9.4 shows the sequence of processes involved in executing the TELNET server, called telnetd.

Figure 9.4. Sequence of processes involved in executing TELNET server


process ID 1
init
fork/exec of /bin/sh, which executes shell script /etc/rc when system comes up multiuser
TCP connection request from TELNET client
inetd
fork
when connection request arrives from TELNET client
inetd
exec
telnetd

The telnetd process then opens a pseudo-terminal device and splits into two processes using fork. The parent handles the communication across the network connection, and the child does an exec of the login program. The parent and the child are connected through the pseudo terminal. Before doing the exec, the child sets up file descriptors 0, 1, and 2 to the pseudo terminal. If we log in correctly, login performs the same steps we described in Section 9.2: it changes to our home directory and sets our group IDs, user ID, and our initial environment. Then login replaces itself with our login shell by calling exec.

Figure 9.5 shows the arrangement of the processes at this point.


process ID 1
init
through inetd, telnetd, and login
login shell
fd 0, 1, 2
pseudo-terminal device driver
network connection through telnetd server and telnet client
user at a terminal

PROCESS GROUPS

A process group is a collection of one or more processes, usually associated with the same job, that can receive signals from the same terminal. Each process group has a unique process group ID. Process group IDs are similar to process IDs: they are positive integers and can be stored in a pid_tdata type. The function getpgrp returns the process group ID of the calling process.

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

Returns: process group ID of calling process The Single UNIX Specification defines the getpgid function as an XSI extension that mimics this behavior.

```
#include <unistd.h>
pid_tgetpgid(pid_tpid);
```

Returns: process group ID if OK, 1 on error

- Each process group can have a process group leader.
- The leader is identified by its process group ID being equal to its process ID.
- It is possible for a process group leader to create a process group, create processes in the group, and then terminate.
- The process group still exists, as long as at least one process is in the group, regardless of whether the group leader terminates.
- This is called the process group lifetime-the period of time that begins when the group is created and ends when the last remaining process leaves the group.

- The last remaining process in the process group can either terminate or enter some other process group. A process joins an existing process group or creates a new process group by calling setpgid.

```
#include <unistd.h>
int setpgid(pid_tpid, pid_tpgid);
```

Returns: 0 if OK, 1 on error
This function sets the process group ID to pgid in the process whose process ID equals pid. If the two arguments are equal, the process specified by pid becomes a process group leader. If pid is 0, the process ID of the caller is used.

SESSIONS

A session is a collection of one or more process groups. For example, we could have the arrangement shown in Figure 9.6. Here we have three process groups in a single session.

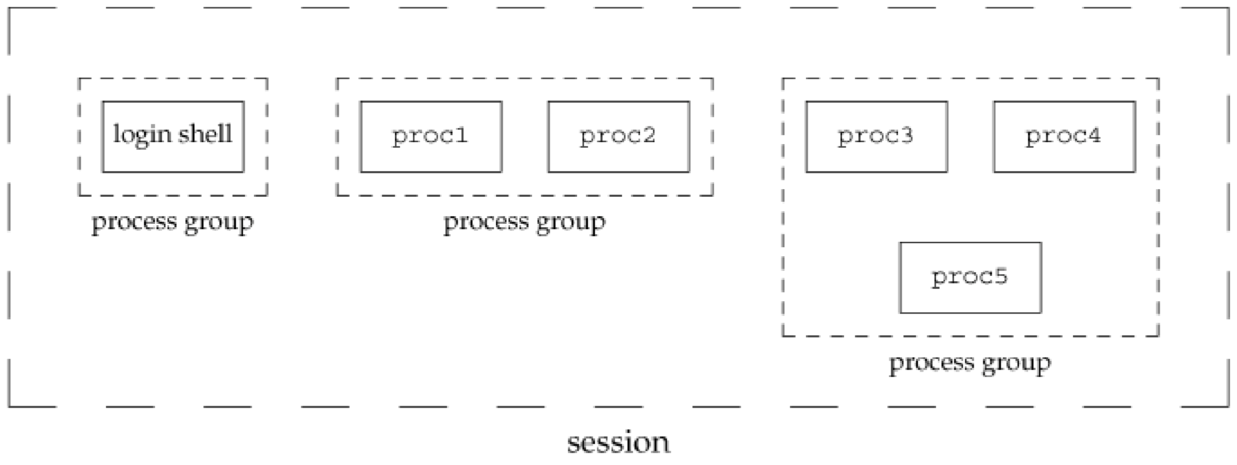


Figure 9.6. Arrangement of processes into process groups and sessions

A process establishes a new session by calling the setsid function.

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

Returns: process group ID if OK, 1 on error

- If the calling process is not a process group leader, this function creates a new session. Three things happen.
- The process becomes the session leader of this new session. (A session leader is the process that creates a session.) The process is the only process in this new session.
- The process becomes the process group leader of a new process group. The new process group ID is the process ID of the calling process.
- The process has no controlling terminal. If the process had a controlling terminal before calling setsid, that association is broken.

- This function returns an error if the caller is already a process group leader. The getsid function returns the process group ID of a process's session leader. The getsid function is included as an XSI extension in the Single UNIX Specification.

```
#include <unistd.h>

pid_tgetsid(pid_tpid);
```

Returns: session leader's process group ID if OK, 1 on error

If pid is 0, getsid returns the process group ID of the calling process's session leader.

CONTROLLING TERMINAL

Sessions and process groups have a few other characteristics.

- A session can have a single controlling terminal. This is usually the terminal device (in the case of a terminal login) or pseudo-terminal device (in the case of a network login) on which we log in.
- The session leader that establishes the connection to the controlling terminal is called the controlling process.
- The process groups within a session can be divided into a single foreground process group and one or more background process groups.
- If a session has a controlling terminal, it has a single foreground process group, and all other process groups in the session are background process groups.
- Whenever we type the terminal's interrupt key (often DELETE or Control-C), this causes the interrupt signal be sent to all processes in the foreground process group.
- Whenever we type the terminal's quit key (often Control-backslash), this causes the quit signal to be sent to all processes in the foreground process group.
- If a modem (or network) disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader).

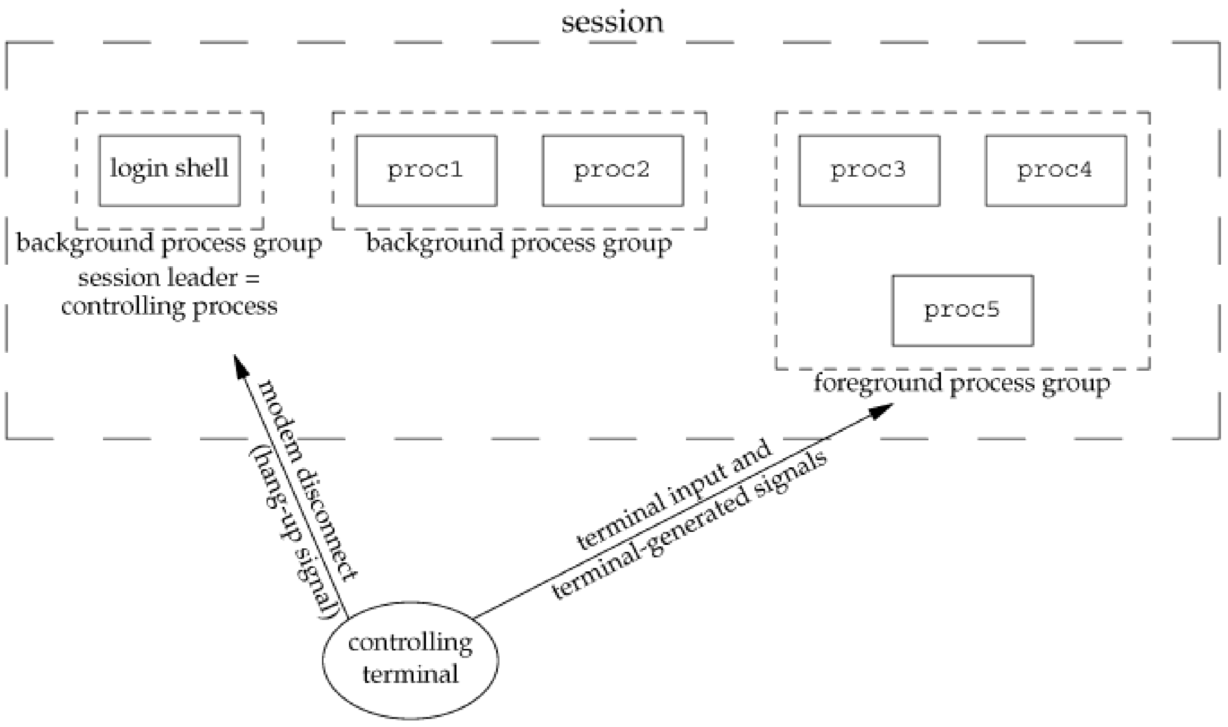


Figure 9.7. Process groups and sessions showing controlling terminal

tcgetpgrp, tcsetpgrp, AND tcgetsid FUNCTIONS

We need a way to tell the kernel which process group is the foreground process group, so that the terminal device driver knows where to send the terminal input and the terminal-generated signals. To retrieve the foreground process group-id and to set the foreground process group-id we can use tcgetprgp and tcsetpgrp function. The prototype of these functions are :

```
#include <unistd.h>
pid_ttcgetpgrp(intfiledes);
```

Returns : process group ID of foreground process group if OK, -1 on error

```
int tcsetpgrp(intfiledes, pid_tpgrpid);
```

Returns: 0 if OK, -1 on error

The function tcgetpgrp returns the process group ID of the foreground process group associated with the terminal open on *file des.* If the process has a controlling terminal, the process can call tcsetpgrp to set the foreground process group ID to pgrpid. The value of pgrpid must be the process group ID of a process group in the same session, and file des must refer to the controlling terminal of the session. The single UNIX specification defines an XSI extension called tcgetsid to allow an application to obtain the process group-ID for the session leader given a file descriptor for the controlling terminal.

```
#include <termios.h>
pid_ttcgetsid(intfiledes);
```

Returns: session leader's process group ID if Ok, -1 on error

JOB CONTROL

This feature allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are to run in the background. Job control requires three forms of support:

- A shell that supports job control
- The terminal driver in the kernel must support job control
- The kernel must support certain job-control signals

The interaction with the terminal driver arises because a special terminal character affects the foreground job: the suspend key (typically Control-Z). Entering this character causes the terminal driver to send the SIGTSTP signal to all processes in the foreground process group. The jobs in any background process groups aren't affected. The terminal driver looks for three special characters, which generate signals to the foreground process group.

- The interrupt character (typically DELETE or Control-C) generates SIGINT.
- The quit character (typically Control-backslash) generates SIGQUIT.
- The suspend character (typically Control-Z) generates SIGTSTP.

This signal normally stops the background job; by using the shell, we are notified of this and can bring the job into the foreground so that it can read from the terminal. The following demonstrates this:

```
$ cat > temp.foo &          start in background, but it'll read from standard input
   [1]     1681
   $                           we press RETURN
   [1] + Stopped (SIGTTIN)     cat > temp.foo &
   $ fg %1                     bring job number 1 into the foreground
   cat > temp.foo              the shell tells us which job is now in the foreground

   hello, world                enter one line

   ^D                          type the end-of-file character
   $ cat temp.foo              check that the one line was put into the file
   hello, world
```

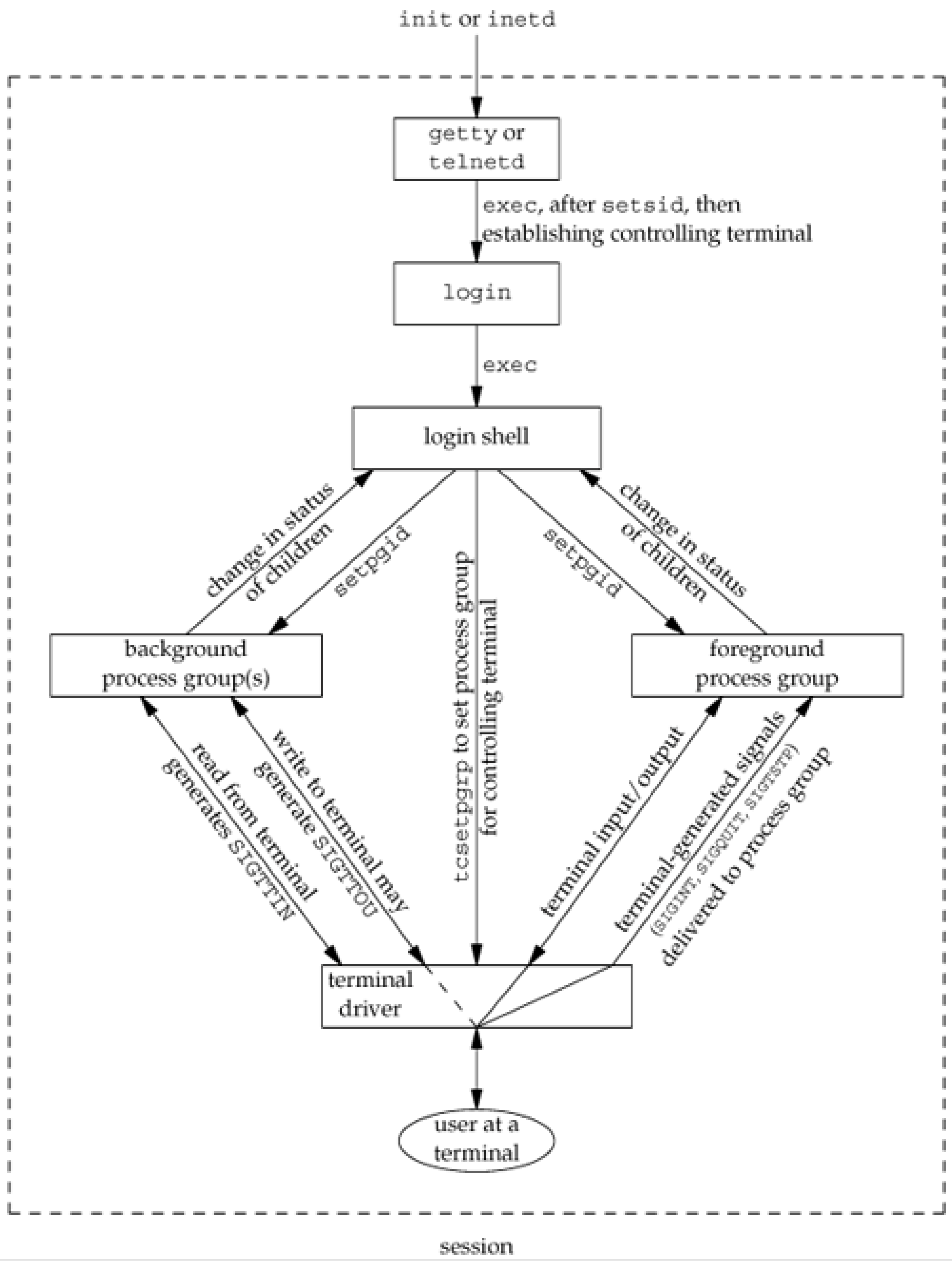
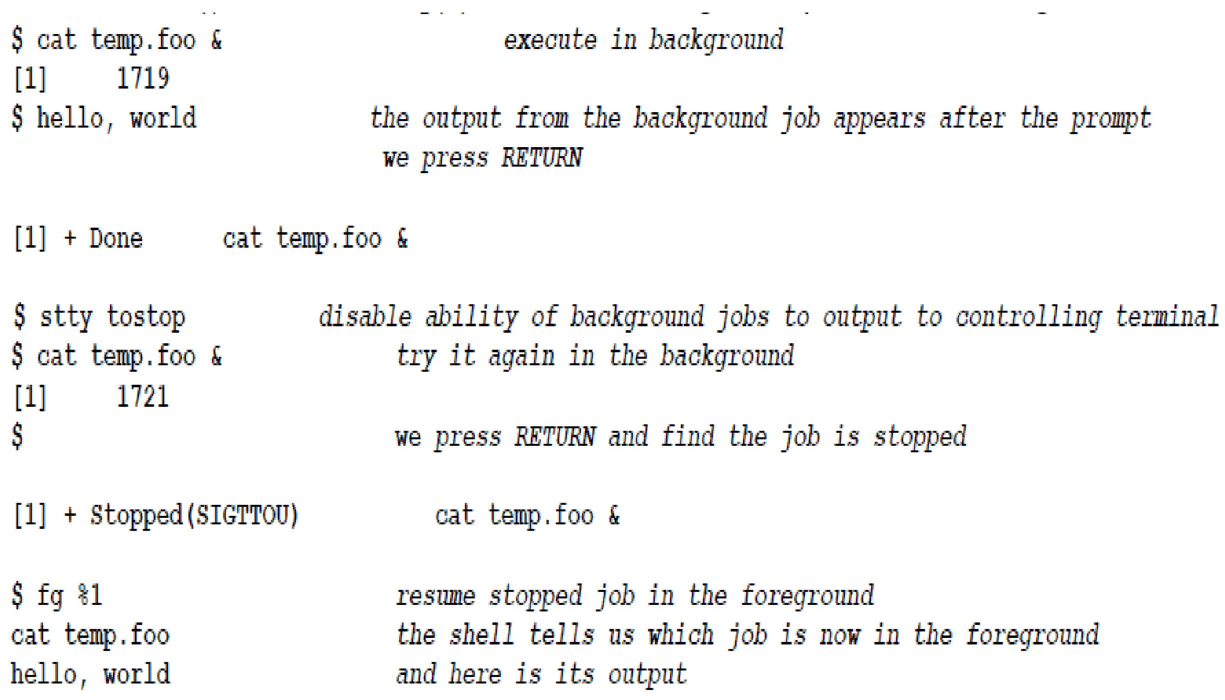

init or inetd
getty or telnetd
exec, after setsid, then establishing controlling terminal
login
exec
login shell
change in status of children
setpgid
tcsetpgrp to set process group for controlling terminal
setpgid
change in status of children
background process group(s)
foreground process group
read from terminal generates SIGTTIN
write to terminal may generate SIGTTOU
terminal input/output
terminal-generated signals (SIGINT, SIGQUIT, SIGTSTP) delivered to process group
terminal driver
user at a terminal
session

Figure 9.8. Summary of job control features with foreground & background jobs & terminal driver

What happens if a background job outputs to the controlling terminal? This is an option that we can allow or disallow. Normally, we use the stty(1) command to change this option. The following shows how this works:

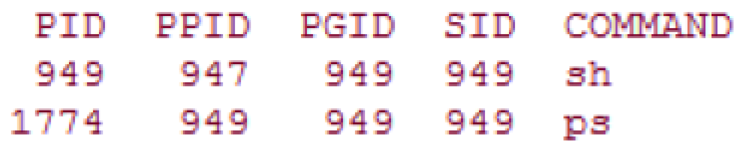


```
$ cat temp.foo &                       execute in background
[1]     1719
$ hello, world              the output from the background job appears after the prompt
                             we press RETURN

[1] + Done      cat temp.foo &

$ stty tostop          disable ability of background jobs to output to controlling terminal
$ cat temp.foo &              try it again in the background
[1]     1721
$                             we press RETURN and find the job is stopped

[1] + Stopped(SIGTTOU)          cat temp.foo &

$ fg %1                       resume stopped job in the foreground
cat temp.foo                  the shell tells us which job is now in the foreground
hello, world                  and here is its output
```

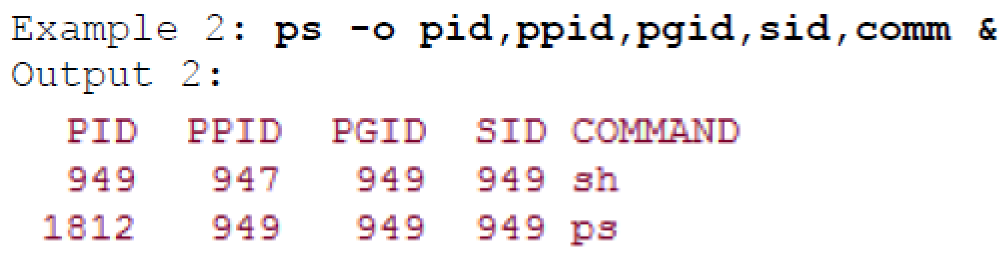
When we disallow background jobs from writing to the controlling terminal, cat will block when it tries to write to its standard output, because the terminal driver identifies the write as coming from a background process and sends the job the SIGTTOU signal.
Figure 9.8 summarizes some of the features of job control that we've been describing. The solid lines through the terminal driver box mean that the terminal I/O and the terminal-generated signals are always connected from the foreground process group to the actual terminal. The dashed line corresponding to the SIGTTOU signal means that whether the output from a process in the background process group appears on the terminal is an option.

SHELL EXECUTION OF PROGRAMS

Example 1: **ps -o pid,ppid,pgid,sid,comm**Output 1:



```
  PID  PPID  PGID  SID  COMMAND
  949   947   949  949  sh
 1774   949   949  949  ps
```



```
Example 2: ps -o pid,ppid,pgid,sid,comm &
Output 2:
  PID  PPID  PGID  SID COMMAND
  949   947   949  949 sh
 1812   949   949  949 ps
```

If we execute the command in the background, the only value that changes is the process ID of the command. Example 3: **ps -o pid,ppid,pgid,sid,comm | cat1**

Output 3:

```
 PID  PPID  PGID  SID COMMAND
 949   947   949  949 sh
1823   949   949  949 cat1
1824  1823   949  949 ps
```

The program `cat1` is just a copy of the standard `cat` program, with a different name. Note that the last process in the pipeline is the child of the shell and that the first process in the pipeline is a child of the last process.

Example 4: **ps -o pid,ppid,pgid,sid,comm | cat1 | cat2**

Output 4:

```
 PID  PPID  PGID  SID COMMAND
 949   947   949  949 sh
1988   949   949  949 cat2
1989  1988   949  949 ps
1990  1988   949  949 cat1
```

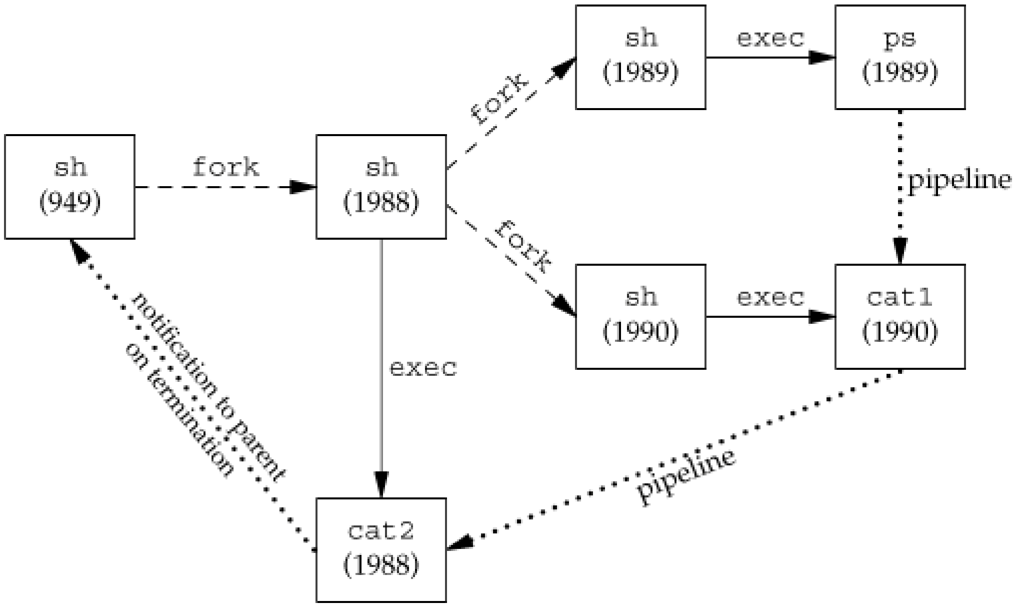
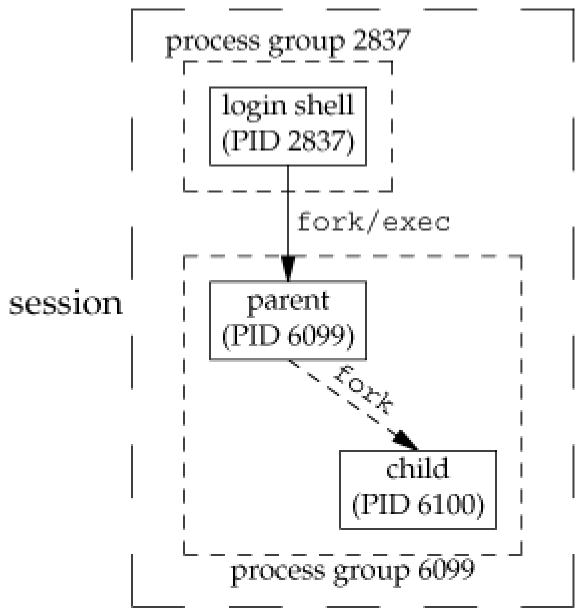


Figure 9.9. Processes in the pipeline ps | cat1 | cat2 when invoked by Bourne shell

ORPHANED PROCESS GROUPS

A process whose parent terminates is called an orphan and is inherited by the init process.

Example of a process group about to be orphaned



Creating an orphaned process group

```
#include "apue.h"
#include <errno.h>
static void sig_hup(intsigno) {
printf("SIGHUP received, pid = %d\n", getpid()); }
static void pr_ids(char *name) {
printf("%s: pid = %d, ppid = %d, pgrp = %d, tpgrp = %d\n", name, getpid(), getppid(),
getpgrp(), tcgetpgrp(STDIN_FILENO));
fflush(stdout);
}
int main(void) {
char c;
pid_tpid;
pr_ids("parent");
if ((pid = fork()) < 0) {
err_sys("fork error"); }
else if (pid> 0) { /* parent */
sleep(5);                               /*sleep to let child stop itself */
exit(0);                        /* then parent exits */
 }
else {                                  /* child */
pr_ids("child");
signal(SIGHUP, sig_hup);                 /* establish signal handler */
kill(getpid(), SIGTSTP);                /* stop ourself */
pr_ids("child");                        /* prints only if we're continued */
if (read(STDIN_FILENO, &c, 1) != 1)
printf("read error from controlling TTY, errno = %d\n", errno);
exit(0);
}
}
```

Output:

```
$ ./a.out
parent: pid = 6099, ppid = 2837, pgrp = 6099, tpgrp = 6099
child: pid = 6100, ppid = 6099, pgrp = 6099, tpgrp = 6099
$ SIGHUP received, pid = 6100 child: pid = 6100, ppid = 1, pgrp = 6099, tpgrp = 2837
read error from controlling TTY, errno = 5
```

- The child inherits the process group of its parent (6099). After the fork,
- The parent sleeps for 5 seconds. This is our (imperfect) way of letting the child execute before the parent terminates.
- The child establishes a signal handler for the hang-up signal (SIGHUP). This is so we can see whether SIGHUP is sent to the child. The child sends itself the stop signal (SIGTSTP) with the kill function. This stops the child, similar to our stopping a foreground job with our terminal's suspend character (Control-Z).
- When the parent terminates, the child is orphaned, so the child's parent process ID becomes 1, the init process ID.
- At this point, the child is now a member of an orphaned process group If the process group is not orphaned, there is a chance that one of those parents in a different process group but in the same session will restart a stopped process in the process group that is not orphaned. Here, the parent of every process in the group belongs to another session.
- Since the process group is orphaned when the parent terminates, POSIX.1 requires that every process in the newly orphaned process group that is stopped (as our child is) be sent the hang-up signal (SIGHUP) followed by the continue signal (SIGCONT).
- This causes the child to be continued, after processing the hang-up signal. The default action for the hang-up signal is to terminate the process, so we have to provide a signal handler to catch the signal. We therefore expect the printf in the sig_hup function to appear before the printf in the pr_ids function.