# Advanced Programming in the UNIX Environment — *Process Control*

Hop Lee
hoplee@bupt.edu.cn

School of Information and Communication Engineering

# Table of Contents I


- Process Identifiers
- FORK Function
- VFORK Function
- EXIT Function
- WAIT and WAITPID Functions
- WAITID Function
- WAIT3 and WAIT4 Functions
- Race Conditions
- EXEC Function Family
- Changing User IDs and Group IDs
  - setreuid and setregid Functions
  - seteuid and setegid Functions
- Interpreter Files
- SYSTEM Function

## Table of Contents II

Set-User-ID Programs

Process Accounting

User Identification

Process Scheduling

Process Times

# Process Identifiers I

- Every UNIX process is guaranteed to have a unique non-negative integer as numeric identifier called **process ID**.
- Example (Figure 1.6, `intro/hello.c`).
- Process ID 0 is usually scheduler process and is often know as the `swapper`. No program on disk corresponds to this process, which is part of the kernel and is known as a system process.
- Process ID 1 is usually the `init`[1] process and is invoked by the kernel at the end of the bootstrap procedure. The program file for this process is `/sbin/init`. The `init` process never dies. It is a normal user process, not a system process within the kernel, although it does run with superuser privileges.

# Process Identifiers II

- Except process ID, there are several other identifications for each process. We can get these identification by such functions:

```c
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

- These functions are always successful.

[1] `launchd` under MacOS 10.10.2

# fork Function I

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

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

- 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 in the parent is the process ID of the new child.

# fork Function II

- 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. 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 do share the text segment, however.
- In modern implementations, fork is implemented using copy-on-write technique, so the only penalty incurred by fork is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.
- Variations of the fork function are provided by some platforms.
- Example (Figure 8.1, proc/fork1.c).

# fork Function III

- All file descriptors that are opened in the parent are duplicated in the child after fork.

# fork Function IV

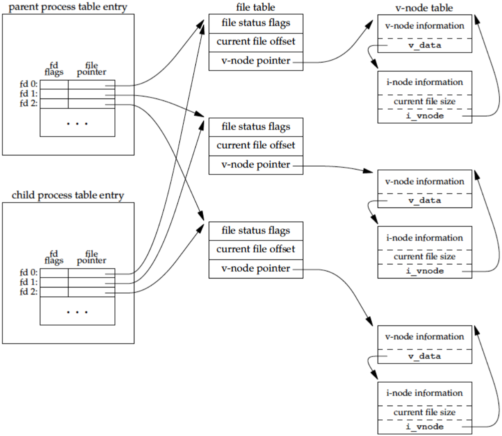


Figure: Sharing of open files between parent and child after fork

# fork Function V

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

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

# fork Function VI

- The differences between the parent and child are
  - The return values from fork.
  - The pids.
  - The ppids.
  - The child's tms_utime, tms_stime, tms_cutime, and tms_cstime values are set to 0 (these times are discussed in Section 8.17).
  - 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 fail are
  1. if too many processes there are already in the system
  2. if the total number of processes for this real user ID exceeds the system's limit.
- fork has two main usage:

# fork Function VII

1. When a process wants to duplicate itself so that the parent and child can each execute different section of code at the same time. This is common for network servers.
2. When a process wants to execute a different program. This is common for shells. In this case the child does an `exec` calling right after it returns from the `fork`.

# vfork Function I

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

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

- vfork creates the new process without copying the address space of the parent into the child. And the child just
    1. calling the exec or exit right after the vfork
    2. running in the address space of the parent until it calls either exec or exit

# vfork Function II

- This optimization is more efficient on some implementations of the UNIX System, but leads to undefined results if the child modifies any data, makes function calls, or returns without calling `exec` or `exit`.
- Another different between the two functions is that `vfotk` guarantees that the child runs first. This feature can lead to deadlock if the child depends on further actions of the parent.
- The `vfork` and the `fork` system calls are identity in Linux.
- Example (Figure 8.3, `proc/vfork1.c`).
- Note in Figure 8.3 that we call `_exit` instead of `exit`.

# exit Function I

- Regardless how a process terminates, the same code in the kernel is eventually executed. This kernel code closes all the open descriptors for the process, release memory that it was using, and so on.
- We want the parent to be notifying how its child terminated. For the three exit functions this is done by passing an **exit status** as the argument to the function. For an abnormal termination, however, the kernel generates a **termination status** to indicate the reason for the abnormal termination.
- If the parent terminates before the child, the `init` process will becomes the parent process of the orphan. Then the terminal status of the child will be collected by the `init` process.

# exit Function II

- If the child terminates before the parent, the kernel will keep a certain amount of information for every terminating process, so that the information is available when the parent calls `wait` or `waitpid`. After this operation, the kernel can discard all the memory used by the process and close its open files.
- In UNIX terminology the process that has terminated, but whose parent has NOT yet waited for it, is called a **zombie**.
- When a process who's parent is `init` terminates, it will definitely NOT become a zombie.

# wait and waitpid Functions I

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

# wait and waitpid Functions II

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

- Both will return the child's pid when success.
- The differences between these two functions are
  - wait can block the caller until a child process terminates, while waitpid has an option that prevents it from blocking,
  - waitpid does'nt wait for the first child to terminate; it has a number of options that control which process it waits for.

# wait and waitpid Functions III

- For both functions the argument *statloc* is a pointer to an integer which will store the termination status of the terminated child process. If we do not care the termination status, we just pass a null pointer as this argument.
- There are four mutually exclusive macros that tell us how the process terminated. Based on which of these three macros is true, other macros are used to obtain the exit status, signal number, and the like.
- Example (Figure 8.5, `lib/prexit.c`).

# wait and waitpid Functions IV

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

| | |
|---|---|
| $pid == -1$ | waits for any child process |
| $pid > 0$ | 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* |

- Example (Figure 8.6, proc/wait1.c).
- The *options* argument lets us further control the operation of waitpid. This argument either is 0 or is constructed from the bitwise OR of the constants in Figure 8.7.

# wait and waitpid Functions V

- The waitpid function provides three features that aren't provided by the wait function.
  1. The waitpid function lets us wait for one particular process, whereas the wait function returns the status of any terminated child.
  2. The waitpid function provides a nonblocking version of wait.
  3. The waitpid function provides support for job control with the WUNTRACED and WCONTINUED options.
- Example (Figure 8.8, proc/fork2.c).

# waitid Function I

- The SUS includes an additional function to retrieve the exit status of a process. The waitid function is similar to waitpid, but provides extra flexibility.

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id,
           siginfo_t *infop, int options);
```

- waitid allows a process to specify which children to wait for. Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used. The *id* parameter is interpreted based on the value of *idtype*. The types supported (P_PID, P_PGID, and P_ALL) are summarized in Figure 8.9.

# waitid Function II

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

# wait3 and wait4 Functions I

- Most UNIX system implementations provide two additional functions, `wait3` and `wait4`. These two functions allow the kernel to return a summary of the resources used by the terminated process and all its child processes.

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

# wait3 and wait4 Functions II

- The resource information includes information such as the amount of user CPU time, amount of system CPU time, number of page faults, number of signals received, and the like. Refer to the getrusage(2) manual page for additional details.

# 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.
- To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes.
- Example (Figure 8.12, `proc/tellwait1.c`), program with a race condition.
- Example (Figure 8.13, `proc/tellwait2.c`), modification of Figure 8.12 to avoid race condition.

# exec Function Family I

- 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.
- exec replaces the current process (its text, data, heap, and stack segments) with a brand new program from disk.

```c
#include <unistd.h>
int execl(const char *pathname, const char *arg0, ...);
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, ...);
int execvp(const char *filename, char *const argv[]);
int fexecve(int fd, char *const argv[],
            char *const envp[]);
```

# exec Function Family II

- When a *filename* argument is specified
  - if *filename* contains a slash, it is taken as a pathname
  - otherwise, the executable file is searched for in the directories specified by the PATH environment variable

  and if the found file is not a machine executable that was generated by the link utility, it assumes the file is a shell script and tries to invoke /bin/sh with the *filename* as input to the shell.
- With fexecve, we avoid the issue of finding the correct executable file altogether and rely on the caller to do this. By using a file descriptor, the caller can verify the file is in fact the intended file and execute it without a race.

# exec Function Family III

- The function execl, execle and execlp require each of the command-line arguments to the new program to be specified as separate arguments with the end as a null pointer. For the other four functions, 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 three functions who's name ended in an e 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.
- Example (Figure 8.16, proc/exec1.c, proc/echoall.c).

# exec Function Family IV

- Only one of these seven functions, `execve`, is a system call within the kernel. The other six are just library functions that eventually invoke this system call.

# exec Function Family V

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

- pid and ppid
- ruid and rgid
- Supplementary gids
- Process group ID
- Session ID
- Root directory
- File locks
- Resource limits
- Current working directory
- Controlling terminal
- Time left until alarm clock
- File mode creation mask
- Process signal mask
- Pending signals
- Nice value
- Values for tms_utime, tms_stime, tms_cutime, and tms_cstime

# exec Function Family VI

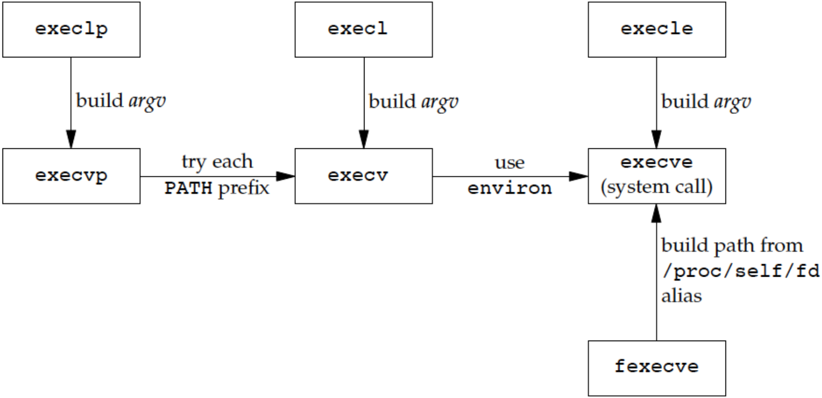


Figure: Relationship of the seven exec functions

# Changing User IDs and Group IDs I

- We can set the real user ID and effective user ID with the `setuid` function. Similarly, the `setgid` function.

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

- There are rules for who can change the IDs:
  - If the process has superuser privileges, the `setuid` functions 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 of the process, `setuid` sets only the effective user ID to *uid*.

# Changing User IDs and Group IDs II

  - If neither of these two conditions is true, `errno` is set to `EPERM` and an error is returned.
- There are several statements about the three user IDs:
  - Only a superuser process can change the real user ID.
  - The effective user ID is set by the `exec` functions, only if the set-user-ID bit is set for the program file.
  - 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.
  - The saved set-user-ID is copied from the effective user ID by `exec` after the `exec` stores the effective user ID from the file's user ID.
  - We can not obtain the current value of the saved set-user-ID.

# setreuid and setregid Functions

- 4.3+BSD supports the swapping of the real user ID and the effective user ID with the following functions.

```
#include <sys/types.h>
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

# seteuid and setegid Functions I

- A proposed change to POSIX.1 includes the two functions which only change the effective user ID or effective group ID:

```
#include <sys/types.h>
#include <unistd.h>
int seteuid(uid_t euid);
int setegid(gid_t egid);
```

- Example, p259.

# seteuid and setegid Functions II

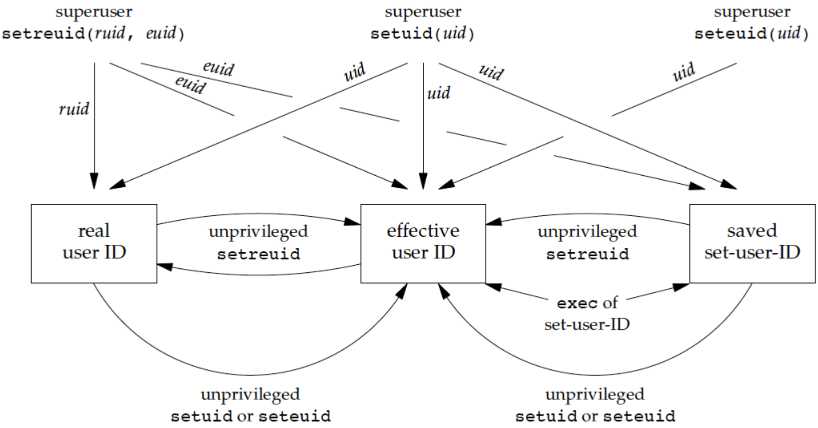


Figure: Summary of all the functions that set the various user IDs

# Interpreter Files I

- The another name of ‘interpreter files’ is **script**.
- Both SVR4 and 4.3+BSD support interpreter files. These files are text files that begin with a line of the form

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

- The *pathname* is normally an absolute pathname since the environment variable `PATH` is not used.
- The actual file that gets `exec`ed by the kernel is not the script file, but the file specified by the *pathname* on the first line of the interpreter file, interpreter.
- Example (Figure 8.20, `proc/exec2.c`, `proc/testinterp`, `environ/echoarg.c`).

# Interpreter Files II

- The corresponding output is:

```
[hop@Hanoi proc]$ cat testinterp
#!/home/hop/bin/echoarg foo
[hop@Hanoi proc]$ ./exec2
argv[0]: /home/hop/bin/echoarg
argv[1]: foo
argv[2]: /home/hop/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

# system Function I

- ISO C defines the system function, but its operation is strongly system dependent.

```
#include <stdlib.h>
int system(const char *string);
```

- system executes a command specified in *string* by calling /bin/sh -c string, and returns after the command has been completed. During execution of the command, SIGCHLD will be blocked, and SIGINT and SIGQUIT will be ignored.

# system Function II

- The value returned is -1 (fork or waitpid fails) or 127 (exec fails) on error, and the return status of the command otherwise. This latter return status is in the format specified in waitpid. Thus, the exit code of the command will be WEXITSTATUS(status).
- In case /bin/sh could not be executed, the exit status will be that of a command that does exit(127).
- If the value of *string* is NULL, system returns nonzero if the shell is available, and zero if not.
- system does not affect the wait status of any other children.
- Figure 8.22 is an implementation of the system function. And Figure 8.23 is a test program.

# system Function III

- Example (Figure 8.22, 8.23, proc/system1.c, proc/systest1.c).
- The advantage in using system, instead of using fork and exec directly, is that system does all the required error handling and all the required signal handling.

# Set-User-ID Programs

- Calling `system` from a set-user-ID program is a security hole and should never be done.
- The superuser permissions that we gave to the set-user-ID program are retained across the `fork` and `exec` that are done by `system`.
- Example (Figure 8.24, 8.25, `proc/systest3.c`, `proc/pruids.c`).

# Process Accounting I

- Most UNIX systems provide an option to do process accounting.
- Process accounting is NOT specified by any of the standards.
- There is a command `accton` under SVR4 and 4.3+BSD. A superuser can executes it with a pathname to the log file as the argument.
- The structure of the accounting records is defined in the header `sys/acct.h`.
- The `ac_flag` member records certain events during the execution of the process. There are described in Table 1:

# Process Accounting II

Table: 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 (not int FreeBSD) |
| ACORE | process dumped core (not in Solaris10) |
| AXSIG | process was killed by a signal (not in Solaris10) |
| AEXPND | expanded accounting entry (Solaris10 only) |
| ANVER | new record format (FreeBSD only) |

- The data required for the accounting record are all kept by the kernel in the process table and initialized whenever a new process is created.

# Process Accounting III

- Each accounting record is written when the process terminates.
- The accounting records correspond to processes (pid), not programs.
- Example,(Figure 8.28 `proc/test1.c`, Figure 8.29 `proc/pracct.c`).

# Process Accounting IV

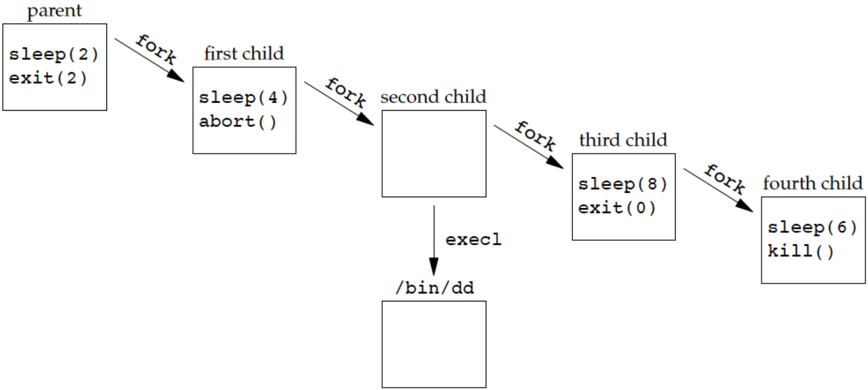


Figure: Process structure for accounting example

# User Identification

- Sometime we want to find out the login name of the user who's running the program. Calling `getpwuid(getuid())` is a approach, but what if a single user has multiple login names?
- The system normally keeps track of the name we log in under, and the `getlogin` function provides a way to fetch that login name.

```
#include <unistd.h>
char *getlogin(void);
```

- This function fails if the process is not attached to a terminal that a user logged into.

# Process Scheduling I

- Historically, the UNIX System provided processes with only coarse control over their scheduling priority.
- The real-time extensions in POSIX added interfaces to select among multiple scheduling classes and fine-tune their behavior.
- In the Single UNIX Specification, nice values range from 0 to (2*NZERO)−1. Lower nice values have higher scheduling priority.
- A process can retrieve and change its own nice value with the nice function.

```
#include <unistd.h>
int nice(int incr);
```

# Process Scheduling II

- The *incr* argument is added to the nice value of the calling process. If *incr* is too large, the system silently reduces it to the maximum legal value. Similarly, if *incr* is too small, the system silently increases it to the minimum legal value.
- It returns `new nice value - NZERO` if OK, −1 on error.
- Because −1 is a legal successful return value, we need to clear `errno` before calling `nice` and check its value if `nice` returns −1.
- Example (Figure 8.30, `proc/nice.c`).
- The `getpriority` function can be used to get the nice value for a process, just like the `nice` function. However, `getpriority` can also get the nice value for a group of related processes.

# Process Scheduling III

```
#include <sys/resource.h>
int getpriority(int which, id_t who);
```

- It returns nice value between -NZERO and NZERO−1 if OK, −1 on error.
- The *which* argument can take on one of three values: PRIO_PROCESS to indicate a process, PRIO_PGRP to indicate a process group, and PRIO_USER to indicate a user ID.
- The *which* argument controls how the *who* argument is interpreted. If the *who* argument is 0, then it indicates the calling process, process group, or user (depending on the value of the which argument).

# Process Scheduling IV

- When the *which* argument applies to more than one process, the highest priority (lowest value) of all the applicable processes is returned.
- The `setpriority` function can be used to set the priority of a process, a process group, or all the processes belonging to a particular user ID.

```
#include <sys/resource.h>
int setpriority(int which, id_t who, int value);
```

- The *which* and *who* arguments are the same as in the `getpriority` function. The *value* is added to `NZERO` and this becomes the new nice value.
- The SUS leaves it up to the implementation whether the nice value is inherited by a child process after a `fork`.

# Process Times I

- We knew that the `time` command can give a report of CPU usage of a give command.
- A process can call the `times` function to obtain these values for itself and any terminated children:

```
#include <sys/times.h>
clock_t times(struct tms *buf);
```

- The function fill in the `tms` structure pointed to by *buf*:

```
struct tms
{
  clock_t tms_utime;  /* User CPU time.  */
  clock_t tms_stime;  /* System CPU time.  */
  clock_t tms_cutime; /* User CPU time of dead children. */
  clock_t tms_cstime; /* System CPU time of dead children.  */
};
```

# Process Times II

- 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.
- Example (Figure 8.31, `proc/times1.c`).

# The End

The End of Chapter 8.