System Programming

Week 6: Process Control

Seongjin Lee

Updated: 2016-10-12
05-process

insight@hanyang.ac.kr
http://esos.hanyang.ac.kr
Esos Lab. Hanyang University

Table of contents

introduction

This chapter covers following items

- Creation of new processes, program execution, and process termination
- Properties of Process
- Interpreter files and the `system` function
- Miscellaneous

Process Identifiers

Every process has a unique process ID (a non-negative integer)

- Although they are unique, the IDs are reused after termination
- Process ID 0 is usually the scheduler known as the *swapper*
- Process ID 1 is usually the `init` process (invoked at the end of the boostrap procedure, located in `/etc/init` or `/sbin/init`)
- Process ID 2 is *pagedaemon* responsible for supporting the paging of the virtual memory system

Functions to find the identifiers

```
#include <unistd.h>
pid_t getpid(void);
// Returns: process ID of calling process

pid_t getppid(void);
// Returns: parent process ID of calling process

uid_t getuid(void);
// Returns: real user ID of calling process

uid_t geteuid(void);
// Returns: effective user ID of calling process

gid_t getgid(void);
// Returns: real group ID of calling process

gid_t getegid(void);
// Returns: effective group ID of calling process
```

Process creation, execution, and termination

Process creation, execution, and termination : Process Creation

`fork(2)` Function

The new process created by `fork` is called the *child process*

```
#include <unistd.h>
pid_t fork(void);
// Returns: 0 in child, process ID of child in parent, -1 on error
```

`fork(2)` Function cont'd

`fork` is called once but returns twice

- The return value in the child is 0
- The return value in the parent is the process ID of the new child
- The child is copy of the parent
- Both the child and the parent continue executing with the instruction that follows the call to `fork`

Must understand that

- The order of execution of the child and the parent depends on the scheduling algorithm
- The parent and the child share the same file offset
- The child has its own copy of the parent's descriptors

fork(2) Example

```c
int gvar = 6; // external variable in initialized data
char buf[] = "a write to stdout\n";

int
main(int argc, char **argv) {
   int var; // automatic variable on the stack
   pid_t pid;

   var = 88;
   if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1) {
      fprintf(stderr, "%s: write error: %s\n", argv[0], strerror(errno));
      exit(1);
   }
   printf("before fork\n"); // we don't flush stdout

   if ((pid = fork()) < 0) {
      fprintf(stderr, "%s: fork error: %s\n", argv[0], strerror(errno));
```

`fork(2)` Example

```
make fork
./fork
./fork | cat
```

When we write to standard output

- we subtract 1 from the size of `buf` to avoid writing the terminating null byte
- `strlen` calculate the length of a string not including the terminating null byte
  - `strlen` requires function call
- `sizeof` calculate the length of string including the terminating null byte
  - `sizeof` calculates the buffer length at compile time

If buffer is not flushed before `fork`, the buffer is copied to child

Sharing of Open Files

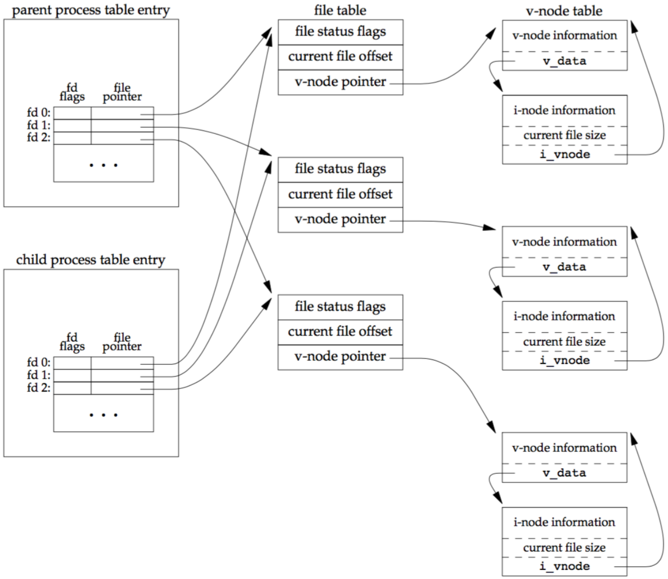

Figure: Sharing of open files between parent and child after `fork`

Two normal cases for handling the descriptors after a `fork`

the parent waits for the child to complete

- 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

- after the `fork`, the parent and the child closes the descriptors that it doesn’t need

Properties that child inherits

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

Why do `fork` fail

- if too many processes are already in the system
- if the total number of processes for this real user ID exceeds the system's limit (`CHILD_MAX`)

Two uses for `fork`

- When a process wants to duplicate itself so that the parent and the child can each execute different sections of code at the same time.
  - This is common for network servers.
- 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.

Process creation, execution, and termination : Process Termination

`exit` Functions

Process can terminate normally in five ways

- Executing a return from the main function.
- Calling the exit function.
- Calling the `_exit` or `_Exit` function.
- Executing a return from the start routine of the last thread in the process.
- Calling the `pthread_exit` function from the last thread in the process.

Regardless of how a process terminates, the same code in the kernel is eventually executed

- the code closes all the open descriptors for the process
- releases the memory that it was using

`wait(2)` `waitpid(2)` Function

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

- This signal is asynchronous notification from the kernel to the parent
- parent can choose to ignore this signal
- parent can provide a function (signal handler) that is called when the signal occurs

A process that calls `wait` or `waitpid` can

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

`wait(2)` `waitpid(2)` Function

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
// Both return: process ID if OK, 0 (see later), or -1 on error
```

- The `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 number of options that control which process it waits for

`statloc` is a pointer to an integer

- if this is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument

`wait(2)` `waitpid(2)` Function

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

we need is a function that waits for a specific process: `waitpid` The interpretation of the pid argument for waitpid depends on its value:

`pid == -1`	waits for any child process, `waitpid` is equivalent to `wait`
`pid>0`	waits for the child whose process ID equals *pid*
`pid == 0`	waits for any child whose process group ID equals that of the calling process
`pid < -1`	waits for any child whose process group ID equals the absolute value of pid

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

Macros to examine the termination status

If we get the termination status in `status`, we can determine how a child died

`WIFEXITED(status)` true if the child terminated normally

- we can execute `WEXITSTATUS(status)` to fetch the status

`WIFSIGNALED(status)` true if status was child terminated abnormally that it didn't catch

- `WTERMSIG(status)` to fetch signal number that caused the termination
- `WCOREDUMP(status)` that returns true if a core file or the terminated process was generated

`WIFSTOPPED(status)` true if status was returned for a child that is currently stopped

- `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

`wait(2)` `waitpid(2)` Function

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

`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

three features of `waitpid` functions

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

Termination status example

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

Termination status example

```
    if (wait(&status) != pid) /* wait for child */
        fprintf(stderr, "wait error");
    pr_exit(status); /* and print its status */

    if ((pid = fork()) < 0)
        fprintf(stderr, "fork error");
    else if (pid == 0) /* child */
        abort(); /* generates SIGABRT */

    if (wait(&status) != pid) /* wait for child */
        fprintf(stderr, "wait error");
    pr_exit(status); /* and print its status */

    if ((pid = fork()) < 0)
        fprintf(stderr, "fork error");
    else if (pid == 0) /* child */
        status /= 0; /* divide by 0 generates SIGFPE
              */

    if (wait(&status) != pid) /* wait for child */
        fprintf(stderr, "wait error");
    pr_exit(status); /* and print its status */
```

To make and run

```
make exit_stat;
./exit_stat
```

Process creation, execution, and termination : Process Creation II

exec Function

When a process calls one of the exec functions, that process is completely replaced by the new program

- The new program starts executing at its `main` function
- Process ID does not change across an `exec`
- `exec` merely replaces the current process (text, data, heap, and stack segment)

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ... /* (char *)0, char *const
     envp[] */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
// All seven return: -1 on error, no return on success
```

Difference among the exec functions

Function	pathname	filename	fd	arg list	arvg[]	environ	envp[]
`execl`	●			●		●	
`execlp`		●			●	●	
`execle`	●			●			●
`execv`	●				●	●	
`execvp`		●			●	●	
`execve`	●				●		●
`fexecve`			●		●		●
letter		p	f	l	v		e

Table: Difference among the seven exec functions

- v in its name, argv's are a vector: `const * char argv[]`
- l in its name, argv's are a list: `const * char arg0, ...`
- e in its name, takes environment variables: `char * const envp[]`
- p in its name, `PATH` environment variable to search for the file

Relationship of the seven exec functions

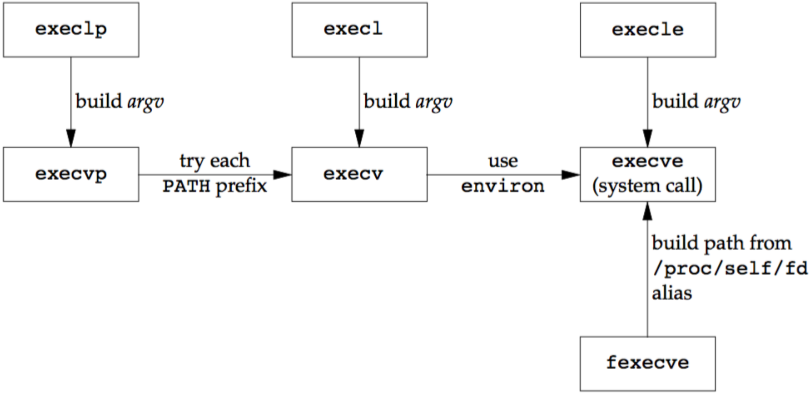

Figure: Relationship of the seven exec functions

Example of exec Family

```
make echoall; ./echoall
```

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

Example of exec Family cont'd

```
make exec; ./exec
```

```
    if ((pid = fork()) < 0) {
        fprintf(stderr, "fork error\n");
    } else if (pid == 0) { /* specify pathname, specify environment */
        if (execle("/Users/James/_projects/class_systemProgramming_2016/lecture_sysprog
               /05_process/codes/echoall", "echoall", "myarg1",
                    "MY ARG2", (char *)0, env_init) < 0){
            fprintf(stderr, "execle error\n");
            printf("#### First fork ended\n");
        }
    }

    if (waitpid(pid, NULL, 0) < 0)
        fprintf(stderr, "wait error\n");

    if ((pid = fork()) < 0) {
        fprintf(stderr, "fork error\n");
    } else if (pid == 0) { /* specify filename, inherit environment */
        if (execlp("./echoall", "echoall", "only 1 arg", (char *)0) < 0)
            fprintf(stderr, "execlp error\n");
    }
```

Properties of Process

Changing User IDs and Group IDs

In the Unix system, privileges are based on user and group IDs

- change the system’s current date
- read or write a particular file

If a program can’t access some resources, they need to change their user or group ID to gain appropriate privilege or access

- In general, we try to use the *least-privilege* model
- This reduces the risk that security might be compromised

APIs to Change User IDs and Group IDs

```
#include <unistd.h>
int setuid(uid_t uid); // set the real and effective user ID
int setgid(gid_t gid); // set the real and effective group ID
// Both return: 0 if OK, -1 on error
```

The rules for who can change the IDs (same applies to user ID and Group ID)

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

Statements about the three user IDs

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

Summary of setting the user IDs

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

Figure: Ways to change the three user IDs

`setreuid` and `setregid` Functions

BSD supported the swapping of the real user ID and the effective user ID with the `setreuid` function.

- We can supply a value of -1 for any of the arguments to indicate that the corresponding ID should remain unchanged

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
// Both return: 0 if OK, -1 on error
```

The rule

- an unprivileged user can always swap between the real user ID and the effective user ID

seteuid and setegid Functions

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

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
// Both return: 0 if OK, -1 on error
```

Summary of all functions

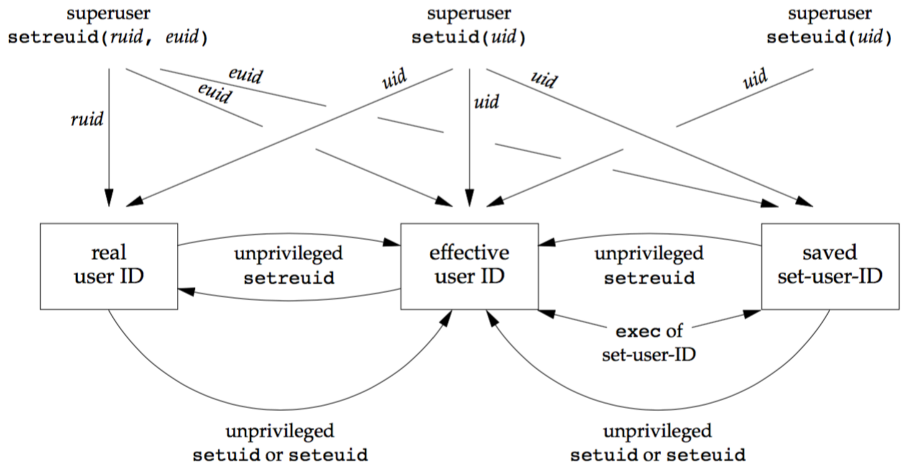

Figure: Summary of all the functions that set the various user IDs

INTERPRETER FILES AND THE `system` FUNCTION

Interpreter Files

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

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

The most common of these interpreter files begin with the line

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

Interpreter Files Example

Example *sharpbang* file

```
#!/Users/James/_projects/class_SystemProgramming_2016/lecture_sysprog/05_process/
    codes/echoarg foo
```

You have to change the PATHNAME on the interpreter line

Example echoarg program

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

Interpreter Files Example

Example interpreter files

```
    pid_t pid;
    if ((pid = fork()) < 0) {
       fprintf(stderr, "fork error\n");
    } else if (pid == 0) { /* child */
       if (execl("/Users/James/_projects/class_systemProgramming_2016/lecture_sysprog
            /05_process/codes/testinterp",
       //if (execl("echoarg",
          "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
       fprintf(stderr, "execl error (%d)\n", errno);
    }
    if (waitpid(pid, NULL, 0) < 0) /* parent */
       fprintf(stderr, "waitpid error\n");
```

You have to change the PWD in the code

Note that on FreeBSD 8.0, *pathname* limit is 4,097 bytes. On Linux 3.2.0, the limit is 128 bytes. Mac OS X 10.6.8 supports a limit of 513 bytes, whereas Solaris 10 places the limit at 1,024 bytes.

system Function

It is convenient to execute a command string from within a program

`system(‘‘date > file’’);` is much simpler than to call `time` to get the current calendar time, then call `localtime` to convert it to a broken-down time, then call `strftime` to format the result

```
#include <stdlib.h>
int system(const char *cmdstring);
// Returns: ---
```

1. If either the `fork` fails or `waitpid` returns an error other than `EINTR`, system returns **-1** with `errno` set to indicate the error.
2. If the `exec` fails, implying that the shell can't be executed, the return value is as if the shell had executed `exit(127)`.
3. Otherwise, all three functions—`fork`, `exec`, and `waitpid`—succeed, and the return value from system is the termination status of the shell, in the format specified for `waitpid`.

The system Function Example

read codes/system.c

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

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

The `system` Function Example

Run it with `make system; ./system`

```
main(int argc, char **argv)
{
   int status;
   if ((status = system("date")) < 0)
      fprintf(stderr, "system() error");

   pr_exit(status);

   if ((status = system("nosuchcommand")) < 0)
      fprintf(stderr, "system() error");

   pr_exit(status);

   if ((status = system("who; exit 44")) < 0)
      fprintf(stderr, "system() error");

   pr_exit(status);

   exit(0);
}
```

Security hole in using system

```
make tsys; make pruid
```

```
./tsys ./pruid
```

```
sudo chown root tsys ; chmod u+s tsys; ls -l tsys
```

```
./tsys ./pruid
```

```
main(int argc, char *argv[])
{
int status;
   if (argc < 2)
      fprintf(stderr, "command-line argument required");
   if ((status = system(argv[1])) < 0)
      fprintf(stderr, "system() error");
   pr_exit(status);
exit(0);
}
```

```
   printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
   exit(0);
```

Some implementations have closed this security hole by changing /bin/sh to reset the effective user ID to the real user ID when they don't match.

Miscellaneous

User Identification

Any process can find out its real and effective user ID and group ID

- `getpwuid(getuid())` to find out the login name of the user who's running the program
- If a single user has multiple login names, each with the same user ID, then we used `char *getlogin(void)` to find the login name
  - A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry

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

Process Scheduling

A process can retrieve and change *its* nice value with `nice` function

```
#include <unistd.h>
int nice(int incr);
// Returns: new nice value - NZERO if OK, -1 on error

#include <sys/resource.h>
int getpriority(int which, id_t who);
// Returns: nice value between -NZERO and NZERO-1 if OK, -1 on error
int setpriority(int which, id_t who, int value);
// Returns: 0 if OK, -1 on error
```

`getpriority` function gets the `nice` value for a process

- *which*: `PRIO_PROCESS` to indicate a process, `PRIO_PGRP` to indicate process group, and `PRIO_USER` to indicate user ID
- *who*: `0` indicates the calling process, process group, or user
    - For example, when which is set to `PRIO_USER` and *who* is `0`, the real user ID of the calling process is used

nice Example

read codes/nice.c

```
    if (argc == 2)
       adj = strtol(argv[1], NULL, 10);
    gettimeofday(&end, NULL);
    end.tv_sec += 10; /* run for 10 seconds */

    if ((pid = fork()) < 0) {
       fprintf(stderr, "fork failed");
    } else if (pid == 0) { /* child */
       s = "child";
       printf("current nice value in child is %d, adjusting by %d\n",
         nice(0)+nzero, adj);
       errno = 0;
       if ((ret = nice(adj)) == -1 && errno != 0)
          fprintf(stderr, "child set scheduling priority");
       printf("now child nice value is %d\n", ret+nzero);
    } else {  /* parent */
       s = "parent";
       printf("current nice value in parent is %d\n", nice(0)+nzero);
    }
    for(;;) {
       if (++count == 0)
          printf("%s counter wrap", s);
```

nice Example

```
make nice

James@maker:codes$ ./nice
NZERO = 20
current nice value in parent is 20
current nice value in child is 20, adjusting by 0
now child nice value is 20
child count = 218686919
parent count = 218744258
James@maker:codes$ ./nice 20
NZERO = 20
current nice value in parent is 20
current nice value in child is 20, adjusting by 20
now child nice value is 39
parent count = 219145730
child count = 216745892
```

Process Times

There are three times that we can measure

1. wall clock time
2. user CPU time
3. system CPU time

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

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

Process Times Example I

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/times.h>

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);
void  pr_exit(int status);

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

   setbuf(stdout, NULL);
   for (i = 1; i < argc; i++)
      do_cmd(argv[i]); /* once for each command-line arg */
   exit(0);
}

static void
do_cmd(char *cmd)  /* execute and time the "cmd" */
```

Process Times Example II

```
{
    struct tms tmsstart, tmsend;
    clock_t  start, end;
    int   status;

    printf("\ncommand: %s\n", cmd);

    if ((start = times(&tmsstart)) == -1) /* starting values */
        fprintf(stderr, "times error");

    if ((status = system(cmd)) < 0)   /* execute command */
        fprintf(stderr, "system() error");

    if ((end = times(&tmsend)) == -1)  /* ending values */
        fprintf(stderr, "times error");

    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
```

Process Times Example III

```
   static long  clktck = 0;

   if (clktck == 0) /* fetch clock ticks per second first time */
      if ((clktck = sysconf(_SC_CLK_TCK)) < 0)
         fprintf(stderr, "sysconf error");

   printf(" real: %7.2f\n", real / (double) clktck);
   printf(" user: %7.2f\n",
     (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
   printf(" sys: %7.2f\n",
     (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
   printf(" child user: %7.2f\n",
     (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
   printf(" child sys: %7.2f\n",
     (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}


void
pr_exit(int status)
{
    if (WIFEXITED(status))
       printf("normal termination, exit status = %d\n", WEXITSTATUS(status));
```

Process Times Example IV

```
    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, singal number = %d\n", WSTOPSIG(status));
}
```

Process Times Example

```
make times
```

```
./times "sleep 5" "date" "man bash > /dev/null"
```

```
command: sleep 5
  real: 5.01
  user: 0.00
  sys: 0.00
  child user: 0.00
  child sys: 0.00
normal termination, exit status = 0

command: date
Mon Oct 10 14:26:00 KST 2016
  real: 0.01
  user: 0.00
  sys: 0.00
  child user: 0.00
  child sys: 0.00
normal termination, exit status = 0

command: man bash > /dev/null
  real: 0.29
  user: 0.00
  sys: 0.00
  child user: 0.29
  child sys: 0.08
normal termination, exit status = 0
```

Last Words

Last Words

- Read chapter 9
- Read and run `./codes/master_fork.c`
- There are total of 18 examples.
- Describe and explain the result for each `forks`
- Draw a diagram where possible, it helps you understanding the behavior
- Due date Oct-26