Processes

Benjamin Brewster

The Process

- Process Management is a necessary component of a multiprogrammable operating system

- Process:

  An instance of an executing program, with a collection of execution resources associated with it

UNIX Process Components

- A unique identity (process id *aka* pid) :: `pid_t pid = getpid();`
- A virtual address space (from 0 to memory limit)
- Program code and data (variables) in memory
- User/group identity (controls what you can access), umask value
- An execution environment all to itself
  - Environment variables
  - Current working directory
  - List of open files
  - A description of actions to take on receiving signals
- Resource limits, scheduling priority
- and more… see the exec() man page

More on this later

Programs vs Processes

- A program is the executable code:

- A process is a running instance of a program:

- More than one process can be concurrently executing the same program code, with separate process resources:

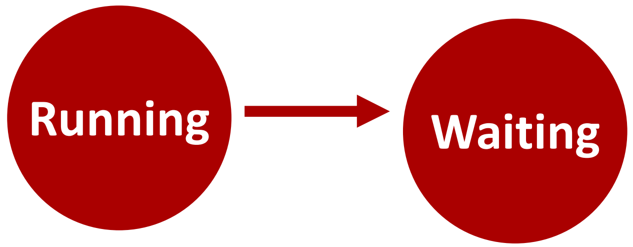
Important Process States in UNIX

On the CPU!

Running

Important Process States in UNIX

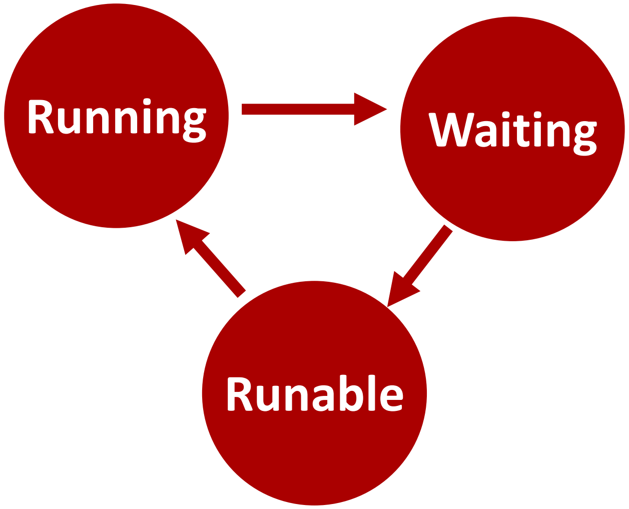
On the CPU!



Waiting for I/O, timer alarm, or signal - also known as "blocked"

Important Process States in UNIX

On the CPU!



Running

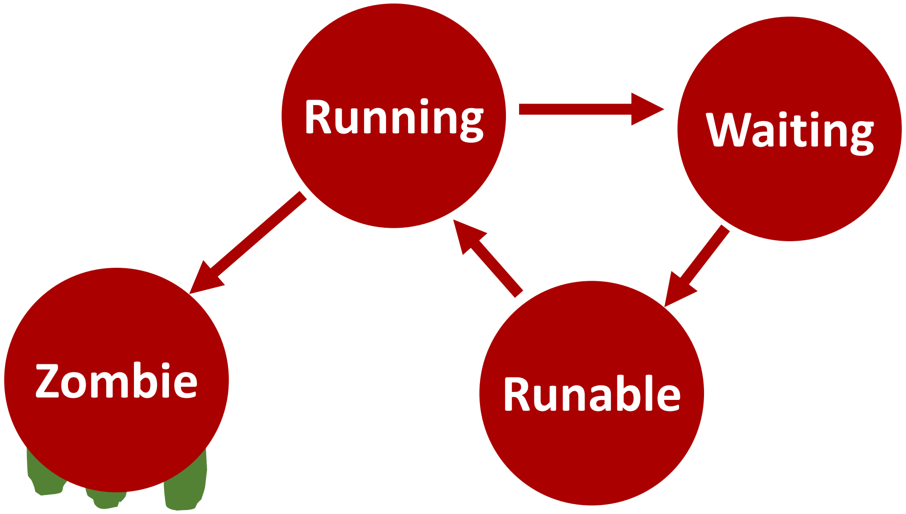
Waiting

Runable

Waiting for I/O, timer alarm, or signal - also known as "blocked"

Waiting for CPU…

Important Process States in UNIX



On the CPU!

Running

Waiting

Waiting for I/O, timer alarm, or signal - also known as "blocked"

Zombie

Exited, waiting for parent to clean it up

Runable

Waiting for CPU…

How Do You Create a Process?

- Let the shell do it for you!
  - When you execute a program, the shell creates the process for you

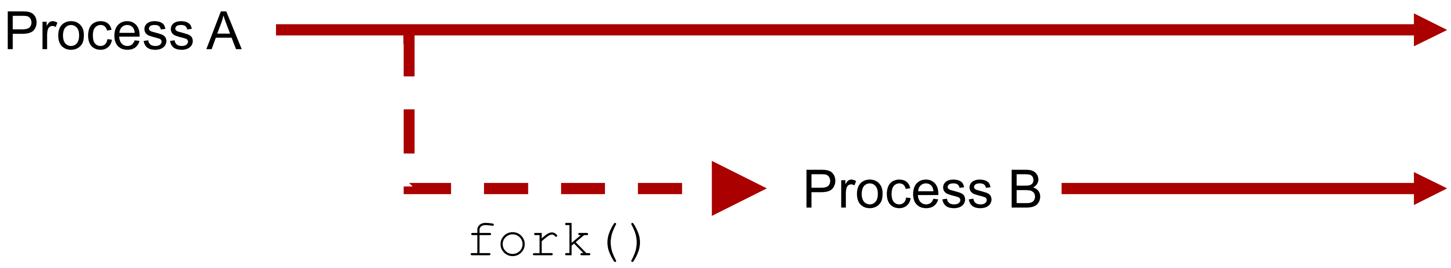
- In some cases, you’ll want to do it yourself
  - Our shell-writing assignment

- Unix provides a C API for creating and managing processes explicitly, as the following material shows

Managing Processes

- Functions we’ll be covering:
  - `fork()`
  - The exec() family:
    - `execl(),execlp(),execv(),execvp()`
  - `exit()`
  - `wait(),waitpid()`
  - `getpid()`
  - `getenv(),putenv()`

How to Start a New Process



- Processes A and B are nearly identical copies, both running the same code, and continuing on from where the fork() call occurred

Process A == Process B ??

- The two processes have different pids
- Each process returns a different value from `fork()`
- Process B gets copies of all the open file descriptors of Process A
- Process B has all of the same variables set to the same values as Process A, but they are now separately managed!
- More to come in a bit

fork()

- A sample program using fork()

```
$ cat forktest.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
    pid_t spawnpid = -5;
    int ten = 10;

    spawnpid = fork();
    switch (spawnpid)
    {
        case -1:
            perror("Hull Breach!");
            exit(1);
            break;
        case 0:
            ten = ten + 1;
            printf("I am the child! ten = %d\n", ten);
            break;
        default:
            ten = ten - 1;
            printf("I am the parent! ten = %d\n", ten);
            break;
    }
    printf("This will be executed by both of us!\n");
}
```

If something went wrong, fork() returns -1 to the parent process and sets the global variable errno; no child process was created

In the child process, fork() returns 0

In the parent process, fork() returns the process id of the child process that was just created

Results

```
$ forktest
I am the child! ten = 11
This will be executed by both of us!
I am the parent! ten = 9
This will be executed by both of us!
```

The order of whether the parent or child reports its text first is up to the OS and its scheduler

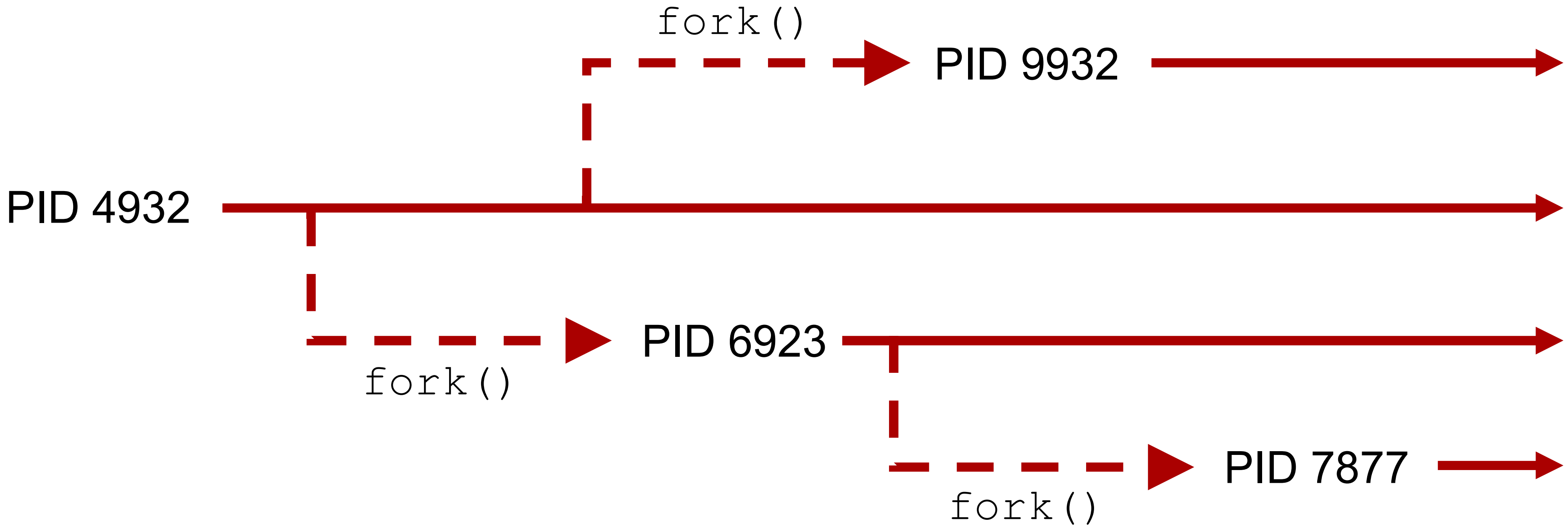
Key Items Inherited

- Inherited by the child from the parent:
  - Program code
  - Process credentials (real/effective/saved UIDs and GIDs)
  - Virtual memory contents, including stack and heap
  - Open file descriptors
  - Close-on-exec flags
  - Signal handling settings
  - process group ID
  - current working directory (CWD)
  - controlling terminal
  - …

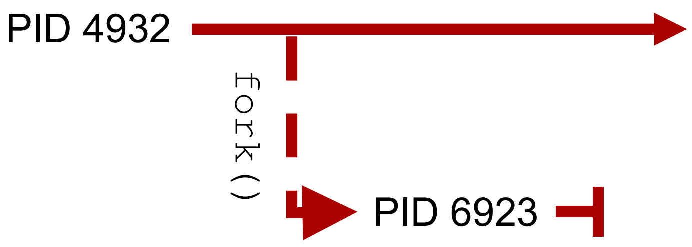
Key Items Unique to the Child Process

- Unique to the child:
  - Process ID
  - Parent process ID is different (it’s the parent that just spawned it)
  - Own copy of file descriptors
  - Process, text, data and other memory locks are NOT inherited
  - Pending signals initialized to the empty set
  - …

`fork()` Forms a Family Tree



Child Process Termination



- A child process can exit for two reasons
    - It completes execution and exits normally
        - Case 1: The child process completed what it was supposed to do and exited with a successful exit status (ie 0)
        - Case 2: The child process encountered an error condition, recognized it, and exited with a non-successful exit status (ie non-zero)
    - It was killed by a signal
        - The child process was sent a signal that by default terminates a process, and the child process *did not catch it*

- How do parents check to see if child processes have terminated?

Checking the Exit Status

- Both of these commands check for child process termination:
  - `wait()`
  - `waitpid()`

- For both functions, you pass in a pointer to which the OS writes an int, which identifies how the child exited
  - We examine this int with various macros to learn what happened

`wait` vs `waitpid`

- `wait()` will block - until *any* one child process terminates; returns the process id of the terminated child

- `waitpid()` will block - until the child process with the *specified* process ID terminates (or has already terminated); returns the process id of the terminated child
  - If you pass it a special flag, it will check if the specified child process has terminated, then immediately return even if the specified child process hasn't terminated yet

`wait()` and `waitpid()` Syntax

```
pid_t wait(int *childExitMethod);

pid_t waitpid(pid_t pid,
              int *childExitMethod,
              int options);
```

- Block this parent until any child process terminates:

  ```
  childPID = wait(&childExitMethod);
  ```

- Block this parent until the specified child process terminates:

  ```
  childPID_actual = waitpid(childPID_intent, &childExitMethod, 0);
  ```

- Check if any process has completed, return immediately with 0 if none have:

  ```
  childPID = waitpid(-1, &childExitMethod, WNOHANG);
  ```

- Check if the process specified has completed, return immediately with 0 if it hasn't:

  ```
  childPID_actual = waitpid(childPID_intent, &childExitMethod, WNOHANG);
  ```

You can use the same variable here, if you like

Proper `waitpid()` Placement

```
$ cat forkwaittest.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
        pid_t spawnPid = -5;
        int childExitMethod = -5;

        spawnPid = fork();
        if (spawnPid == -1) //
        {
                perror("Hull Breach!\n");
                exit(1);
        }
        else if (spawnPid == 0) // Terminate the child process immediately
        {
                printf("CHILD: PID: %d, exiting!\n", spawnPid);
                exit(0);
        }

        printf("PARENT: PID: %d, waiting...\n", spawnPid);
        waitpid(spawnPid, &childExitMethod, 0);
        printf("PARENT: Child process terminated, exiting!\n");
        exit(0);
}
```

```
$ forkwaittest
PARENT: PID: 3311, waiting...
CHILD: PID: 0, exiting!
PARENT: Child process terminated, exiting!
```

Blocks the parent until the child process with specified PID terminates

Checking the Exit Status - Normal Termination

- `wait(&childExitMethod)` and `waitpid(…, &childExitMethod, …)` can identify two ways a process can terminate:

- If the process terminates normally, then the WIFEXITED macro returns non-zero:

```
if (WIFEXITED(childExitMethod) != 0)
    printf("The process exited normally\n");
```

- We can get the actual exit status with the WEXITSTATUS macro:

```
int exitStatus = WEXITSTATUS(childExitMethod);
```

Checking the Exit Status - Signal Termination

- `wait(&childExitMethod)` and `waitpid(…, &childExitMethod, …)` can identify two ways a process can terminate:

- If the process was terminated by a signal, then the WIFSIGNALED macro returns non-zero:

```
if (WIFSIGNALED(childExitMethod) != 0)
    printf("The process was terminated by a signal\n");
```

- We can get the terminating signal with the WTERMSIG macro:

```
int termSignal = WTERMSIG(childExitMethod);
```

Checking the Exit Status - Exclusivity

- Barring the use of the non-standard `WCONTINUED` and `WUNTRACED` flags in `waitpid()`, only one of the `WIFEXITED()` and `WIFSIGNALED()` macros will be non-zero!
- Thus, if you want to know how a child process died, you need to use both WIFEXITED and WIFSIGNALED!
- If the child process has terminated normally, do not run `WTERMSIG()` on it, as there is *no signal number* that killed it!
- If the child process was terminated by a signal, do not run `WEXITSTATUS()` on it, as it has *no exit status* (i.e., no `exit()` or `return()` functions were executed)!

Checking the Exit Status

```
int childExitMethod;
pid_t childPID = wait(&childExitMethod);

if (childPID == -1)
{
   perror("wait failed");
   exit(1);
}

if (WIFEXITED(childExitMethod))
{
   printf("The process exited normally\n");
   int exitStatus = WEXITSTATUS(childExitMethod);
   printf("exit status was %d\n", exitStatus);
}
else
   printf("Child terminated by a signal\n");
```

Non-zero evaluates to true in C

This statement is true, but it never hurts to examine `WIFSIGNALED()`, also, to make sure!

How to Run a Completely Different Program

- `fork()` always makes a copy of your *current* program
- What if you want to start a process that is running a completely different program?
- For this we use the `exec…()` family

`exec…()` - Execute

- `exec…()` replaces the currently running program with a *new* program that you specify
- The `exec…()` functions do not return - they destroy the currently running program
  - No line after a successful `exec…()` call will run
- You can specify arguments to `exec…()`: these become the command line arguments that show up as `argc`/`argv` in C, and as the `$1`, `$2`, etc positional parameters in a bash shell

A møøse once bit my sister

Two Types of Execution

```
int execl(char *path, char *arg1, …, char *argn);
```

- Executes the program specified by *path*, and gives it the command line arguments specified by strings *arg1* through *argn*

```
int execv(char *path, char *argv[]);
```

- Executes the program specified by *path*, and gives it the command line arguments indicated by the pointers in *argv*

Current Working Directory

- `exexl()` and `execv()` do not examine the PATH variable - they only look in the current working directory (but see the next slide)

- If you don't specify a fully qualified path name, then your programs will not be executed, even if they are in a directory listed in PATH, and `execl()` and `execv()` will return with an error

- To move around the directory structure in C, use the following:
  - `getcwd()` :: Gets the current working directory
  - `chdir()` :: Sets the current working directory

Exec…() and the PATH variable

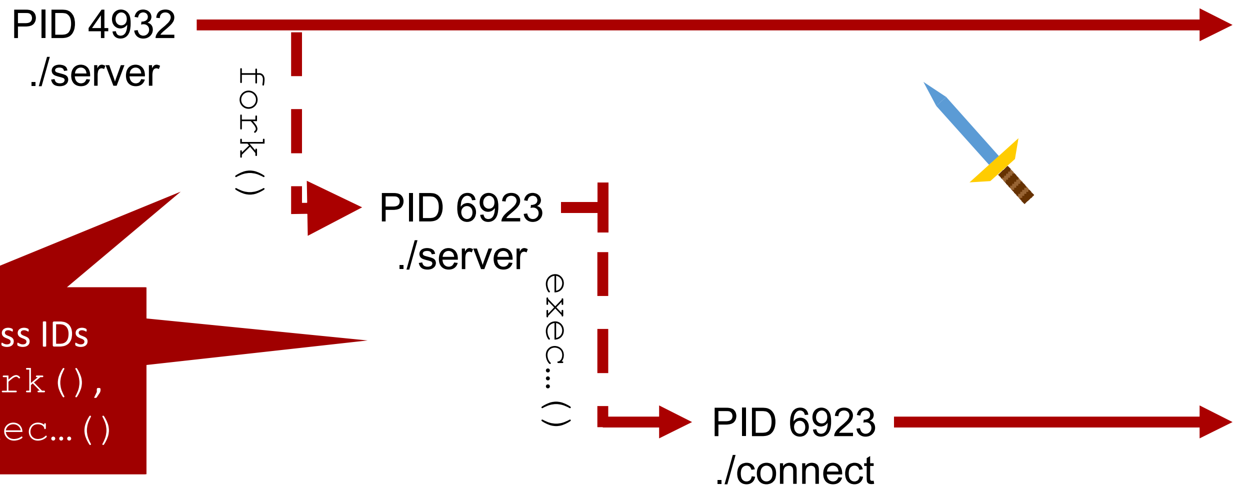
```
int execl(char *path, char *arg1, …, char *argn);
int execlp(char *path, char *arg1, …, char *argn);

int execv(char *path, char *argv[]);
int execvp(char *path, char *argv[]);
```

- The versions ending with *p* will search your PATH environment variable for the executable given in *path*
- In general, you’ll want to use the versions with *p* - `execlp()` or `execvp()` - as they are much more convenient

Execute a New Process

- `exec…()` *replaces* the program it is called from - it does not create a new process!
- Using `fork()` and `exec…()`, we can keep our original program going, and spawn a brand-new process!



PID 4932
./server

fork()

PID 6923
./server

exec…()

PID 6923
./connect

Note that the Process IDs change with the `fork()`, but not with the `exec…()`

Passing parameters to `execlp()`

- `int execlp(char *path, char *arg1, …, char *argn);`
- First parameter to `execlp()` is the pathname of the new program
- Remaining parameters are "command line arguments"
- First argument should be the same as the first parameter (the command itself)
- Last argument must always be NULL, which indicates that there are no more parameters
- Do not pass any shell-specific operators into any member of the `exec…()` family, like `<`, `>`, `|`, `&`, or `!`, because the shell is not being invoked - only the OS is!
- Example:

```
    execlp("ls", "ls", "-a", NULL);
```

fork() + execlp() Example

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
void main() {
        pid_t spawnPid = -5;
        int childExitStatus = -5;

        spawnPid = fork();
        switch (spawnPid) {
                case -1: { perror("Hull Breach!\n"); exit(1); break; }
                case 0: {
                        printf("CHILD(%d): Sleeping for 1 second\n", getpid());
                        sleep(1);
                        printf("CHILD(%d): Converting into \'ls -a\'\n", getpid());
                        execlp("ls", "ls", "-a", NULL);
                        perror("CHILD: exec failure!\n");
                        exit(2); break;
                }
                default: {
                        printf("PARENT(%d): Sleeping for 2 seconds\n", getpid());
                        sleep(2);
                        printf("PARENT(%d): Wait()ing for child(%d) to terminate\n", getpid(), spawnPid);
                        pid_t actualPid = waitpid(spawnPid, &childExitStatus, 0);
                        printf("PARENT(%d): Child(%d) terminated, Exiting!\n", getpid(), actualPid);
                        exit(0); break;
                }
        }
}
```

fork() + execlp() Output

```
$ gcc -o forkexec forkexec.c
$ forkexec
PARENT(8201): Sleeping for 2 seconds
CHILD(8204): Sleeping for 1 second
CHILD(8204): Converting into 'ls -a'
.                    cAd                    Ctests          forkyouzombie    leaky2            python-billion
..                   catsAndDogs            dollars         forkyouzombie.c  leaky2.c          python-billion-fast
addsix-bash          c-billion              doubleparen     forloop          leaky3            pythonmath
addsix-c             c-billion.c            error.txt       greptests        leaky3.c          pythonstring
addsix-c.c           cstring-array          exiter          hardlink1        leaky.c           pythontest
array-of-pointers    cstring-array.c        forkexec        havoc            malloctest        readerror
array-of-pointers.c  cstring-array-unint    forkexec.c      hw               malloctest.c      readpipetest
arraytest            cstring-array-unint.c  forktest        hw.c             memerrors         readtest
arraytest.c          cstring-inlinearray    forktest.c      inodetest        paramtest         rowfile
arraytest.c.backup   cstring-inlinearray.c  forkwaittest    killthesis       perlcamel         rowfile2
billion              cstring-segfault.c     forkwaittest.c  leak2.c_backup   permissionstests  sortdata
PARENT(8201): Wait()ing for child(8204) to terminate
PARENT(8201): Child(8204) terminated, Exiting!
```

Passing parameters to `execvp()`

- `int execvp(char *path, char *argv[]);`
- First parameter to `execvp()` is the pathname of the new program
- Second parameter is an array of pointers to strings
- First string should be the same as the first parameter (the command itself)
- Last string must always be NULL, which indicates that there are no more parameters
- Do not pass any shell-specific operators into any member of the `exec…()` family, like `<`, `>`, `|`, `&`, or `!`, because the shell is not being invoked - only the OS is!
- Example:

```
char* args[3] = {"ls", "-a", NULL};
execvp(args[0], args);
```

execvp() Example

```
$ cat execvptest.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void execute(char** argv)
{
        if (execvp(*argv, argv) < 0)
        {
                perror("Exec failure!");
                exit(1);
        }
}
void main()
{
        char* args[3] = {"ls", "-a", NULL};
        printf("Replacing process with: %s %s\n", args[0], args[1]);
        execute(args);
}
$ gcc -o execvptest execvptest.c
$ execvptest
Replacing process with: ls -a
.         ..        execvptest  execvptest.c
```

`exit()`

- `atexit()`
  - Arranges for a function to be called before `exit()`
- `exit()` does the following:
  - Calls all functions registered by `atexit()`
  - Flushes all stdio output streams
  - Removes files created by `tmpfile()`
  - Then calls `_exit()`
- `_exit()` does the following:
  - Closes all files
  - Cleans up everything - see the man page for `wait()` for a complete list of what happens on exit
- `return()` from `main()` does exactly the same thing as `exit()`

Environment Variables

- A set of text variables, often used to pass information between the shell and a C program
- May be useful if:
  - You need to specify a configuration for a program that you call frequently (LESS, MORE)
  - You need to specify a configuration that will affect many different commands that you execute (TERM, PAGER, PRINTER)
- You can view/edit the environment from bash by using the `printenv` and `export` commands, and assignment (=) operator
- The environment can be edited in C with `setenv()` and `getenv()`

printenv

```
$ printenv
MANPATH=/usr/local/man:/usr/man:/usr/share/man
HOSTNAME=eos-class.engr.oregonstate.edu
SELINUX_ROLE_REQUESTED=
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=128.193.54.168 52413 22
MORE=-c
QTDIR=/usr/lib64/qt-3.3
QTINC=/usr/lib64/qt-3.3/include
SSH_TTY=/dev/pts/17
USER=brewsteb
PAGER=less
MAIL=/var/spool/mail/brewsteb
PATH=/bin:/sbin:/usr/local/bin:/usr/bin:/usr/local/apps/bin:/usr/bin/X11:/nfs/stak/faculty/b/brewsteb/bin:.
PWD=/nfs/stak/faculty/b/brewsteb/tempdir
LANG=en_US.UTF-8
MODULEPATH=/usr/share/Modules/modulefiles:/etc/modulefiles
KDEDIRS=/usr
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
HISTCONTROL=ignoredups
SHLVL=1
HOME=/nfs/stak/faculty/b/brewsteb
LESS=QMcde
LOGNAME=brewsteb
SSH_CONNECTION=128.193.54.168 52413 128.193.37.0 22
LESSOPEN=||/usr/bin/lesspipe.sh %s
G_BROKEN_FILENAMES=1
BASH_FUNC_module()=() {  eval `/usr/bin/modulecmd bash $*`
}
_=/usr/bin/printenv
```

Manipulating the Environment

- Bash:

```
MYVAR="Some text string 1234"
export MYVAR
echo $MYVAR
MYVAR="New text"
```

More on `export` in a bit

- C:

```
setenv("MYVAR", "Some text string 1234", 1);
printf("%s\n", getenv("MYVAR"));
```

`1` means overwrite the value, if it exists

Manipulating the Environment… for Just You

```
$ cat bashAndCEnvironment.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
int main(int argc, char* argv[])
{
  char array[1000];
  printf("Variable %s has value: %s\n", argv[1], getenv(argv[1]));
  printf("Doubling it!\n");
  strcpy(array, getenv(argv[1]));
  strcat(array, getenv(argv[1]));
  printf("New value of %s will be: %s\n", argv[1], array);
  setenv(argv[1], array, 1);
  printf("Variable %s has value: %s\n", argv[1], getenv(argv[1]));
}
```

```
$ MYVAR="TEXT."
$ export MYVAR
$ echo $MYVAR
TEXT.
$ gcc -g -o bashAndCEnvironment bashAndCEnvironment.c
$ bashAndCEnvironment MYVAR
Variable MYVAR has value: TEXT.
Doubling it!
New value of MYVAR will be: TEXT.TEXT.
Variable MYVAR has value: TEXT.TEXT.
$ echo MYVAR
TEXT.
```

All that work for nothing! A processes execution environment belongs to only that process, which gets its initial values from the parent shell - but a process cannot edit the environment variables of it’s parent shell!

Modifications, thus, will only be useful for your current process.

Exporting Environment Variables

```
$ MYTESTVAR="testtext"
$ echo $MYTESTVAR
testtext
$ bashAndCEnvironment MYTESTVAR
Variable MYTESTVAR has value: (null)
Doubling it!
Segmentation fault (core dumped)
$ export MYTESTVAR
$ bashAndCEnvironment MYTESTVAR
Variable MYTESTVAR has value: testtext
Doubling it!
New value of MYTESTVAR will be: testtexttesttext
Variable MYTESTVAR has value: testtexttesttext
$ echo $MYTESTVAR
testtext
```

`export` makes the variable available for all child processes of the shell

But again, remember this environment variable change is only valid for this script - it doesn't affect the shell's environment

Fork Bombs - Notes and Avoidance Techniques

- Yes, they’re hilarious
- Under no circumstances should you be running systems development code on any non-OS class server!
- Consider the following warning signs that you might be about to do something dangerous, where if something goes wrong, your program might consume all of the system resources available and lock you and everyone else out:
  - You've written a loop that calls fork()
  - You've written code in which your child process creates another child process (a fork() within a forked process; these are usually not what you want)
  - You've written code in which your child process is starting up a loop

Fork Bombs - Notes and Avoidance Techniques

- Remember that you need to be really extra sure that you have termination methods built-in to your loops
- Consider having a variable set a flag called `forkNow` in your loop. Then, have a separate function call `fork()` because the flag value was set, with this function *also* resetting the flag value at the end
- Consider during testing, for example, adding an extra condition to a loop with a counting variable: if you hit 50 forks, say, then `abort()`