

# Programs and Processes

COMP 321

Dave Johnson



1

## The State of a Running Process

### ***User-visible state***

- The process's address space (it's memory)
- Current CPU register values: PC, SP, R0, R1, R2, R3, R4, R5, R6, R7, R8, ...
  - (Intel CPUs: rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8, r9, r10, r11, ..., r15)

### ***State inside the kernel (traditionally, the Process Control Block, or PCB)***

- One per process, collects together all OS state for that single process
  - The process's process ID
  - Saved CPU register values (when not currently running on the CPU)
  - Table of the process's open file descriptors
  - Lots of other bookkeeping info about the process

2

## Processes and Threads

*Classically, a process has a single thread of execution*

- One point of execution progress, one set of register values

- Example:

```
main( ... )
{
    ...
    return 0
}
```

*But a process may be “multithreaded”*

- Multiple “**threads**” sharing the same address space
- All running concurrently, all “at once,” cooperating
- Threads are also called **lightweight processes**

## Why Multiple Threads Sharing an Address Space?

*Easy cooperation between these threads since they share all data, such as*

- **A windowing GUI system**

- All threads share the same data structures of what’s on the screen
- One thread tracking the mouse on the screen
- One thread for each open window

- **Microsoft Word**

- One thread managing the user’s keyboard
- One thread doing line breaks, one doing paragraph breaks, one page breaks
- One thread doing spell checking, one for grammar checking, etc.

For now, we will limit ourselves to a **single** thread of control: A “classical” process

## The main() Procedure of Any Program

```
int main(int argc, char *argv[])  
int main(int argc, char *argv[], char *envp[])
```

**Every program has a main() procedure, with these specific arguments**

- Suppose the program was run as

./myprog Hello world

- argc = 3
- argv[0] = “./myprog”
- argv[1] = “Hello”
- argv[2] = “world”
- argv[3] = NULL

By convention, argv[0]  
always equal to the name  
the program was run as

- envp argument points to “environment variable” strings, same format as argv

## main() is Just a Regular Procedure

**The name “main” is special (expected), but it works like any other procedure**

- It gets called like a normal procedure (because it is a normal procedure)
- It returns like a normal procedure (because it is a normal procedure)

**But some mysteries**

- Who actually does the procedure call to main(), passing it those arguments (argc and argv) with the right contents?
- When main() executes any “return” statement, where does that return go back to?
- And what happens if the execution of main() just “falls off the bottom” of that procedure?

## Terminating a Process: Calling exit vs. \_exit

```
[[noreturn]] void _exit(int status);  
[[noreturn]] void exit(int status);
```

### ***\_exit(status) is a kernel call***

- Just causes the process to immediately terminate
- Process becomes a “zombie” until reaped (i.e., collected) by its parent process

### ***exit(status) is a library call***

- Executes various, e.g., “cleanup”-type functions, including flushing and closing all open stdio streams
- And finally calls the kernel call `_exit(status)`

### ***The “exit status” of the process***

- The least significant byte of status (i.e., status & 0xff)

## Terminating a Process: Returning from main()

```
return value;    /* from inside main() */  
return;          /* from inside main(), or “falling off the bottom” of main() */
```

### ***Doing “return value” with some explicit value***

- Then “value” is the return value from that function

### ***Falling off the bottom of a function is the same as “return” from that function***

- Whatever value happens to be in the (e.g., on x86\_64) %rax CPU register becomes the function’s return value
- (So there ***always*** is ***some*** return value from ***any*** function)

### ***The “exit status” of the process***

- The least significant byte of the main() return value (i.e., value & 0xff)

## The C Runtime “Wrapper” Code for main()

*Small piece of assembly language code (traditionally called crt0)*

- The real entry point for any program
- Upon loading the program into memory (i.e., the process’s address space), the kernel initializes this process’s “saved” PC register value to be the address of this C runtime code
- Different systems are a bit different, but in general this code does
  - Packages command line arguments in argv[] format & same for envp[]
  - Saves envp value as “extern char \*\*environ;” /\* see “man 3 getenv” \*/
  - Calls “status = main(argc, argv, envp);”
  - Calls “exit(status);” /\* library exit(), which ultimately calls kernel \_exit() \*/
- This is how return value from main() defines the exist status for the process

COMP 321

Copyright © 2025 David B. Johnson

Page 9

9

## Registering a Function to be Called at Process Exit

```
int atexit(void (*function)(void));
```

**Can register more than one such function, called in the order registered**

- Registered functions are not called on \_exit(), only through library exit()

```
void my_cleanup()
{
    printf("Cleaning up at exit.\n");
    /* ... */
}
```

```
int main()
{
    atexit(my_cleanup);
    return 0;
}
```

Doing “return” here  
results in library exit()  
being called

COMP 321

Copyright © 2025 David B. Johnson

Page 10

10

## What Should the Exit Status of a Process Be?

### *Any 8-bit value you want to exit with*

- But there is a very old, long-standing convention
  - exit 0 for **successful** completion
  - exit any nonzero value (e.g., 1) for any **error/failure** completion
- Example: Using the bash shell

```
$ true
$ echo $?
0
```

```
$ if true; then echo yes; else echo no; fi
yes
```

```
$ false
$ echo $?
1
```

```
$ if false; then echo yes; else echo no; fi
no
```

- Recommendation: `#include <stdlib.h>`
- Use `exit(EXIT_SUCCESS);` or `exit(EXIT_FAILURE);`

## The Termination of a Process

### *Regardless of how a process terminates*

- The kernel frees the process's entire address space (all its memory)
- Does a `close()` on all of its open file descriptors
- Frees all other resources held by the process, except ...

### *The process becomes a "zombie" process*

- Meaning that the kernel retains just enough of the process's state to be able to report the process's termination to the process's parent process
- The process in this "zombie" state remains until it is "reaped" (i.e., collected) by the parent process
- Once reaped, the child is then completely gone (and thus not reported again)

## How the Shell Runs a Program

*Consider the shell running a program such as `./myprog`*

- The program myprog has to get loaded into memory and executed
- The shell must still be there, ready for the next command
  - Can't just throw away the shell's process state and address space
  - And can't allow myprog to possibly mess up the shell
- Means myprog must run as a separate process, with its own address space
- Normally, the shell waits for myprog process to finish
- But if the command includes "&" (as in, e.g., "`./myprog &`"), the shell process and the new myprog process actively run concurrently

13

## How the Shell Runs a Program

### The shell process

The shell uses ***fork()*** to create a new process as an exact clone of itself



The shell (still exists) waits for the new child process to finish



The shell reaps child's exit status



### New child process

The new child process replaces in memory the program it is running (the shell) with the new program (myprog)



myprog runs and eventually exits



14

## A Simple Example of the Shell Running “./myprog”

```
#include <unistd.h>
#include <sys/wait.h>
int main()
{
    pid_t pid = fork();
    if (pid == 0) {
        execl("./myprog", "myprog", NULL);
    } else {
        wait(NULL);
    }
    return 0;
}
```

fork() returns **twice**:  
once in the parent and  
once in the child

**The only difference:**  
fork() returns 0 in the  
child and nonzero in  
the parent

This simple example does not  
do any error checking, but  
you should!

## Creating a New Process

```
pid_t fork(void);
```

***Creates a new process as an identical “clone” of the calling process***

- Kernel creates a new PCB for the new process, substantially as a copy of the calling process’s existing PCB
- Kernel assigns new process a new pid, remembered in the kernel in child’s PCB
  - pids assigned in ascending order, wrapping around, skipping those in use
- Child address space is created as a **copy** of the calling process’s address space
  - Child thus **appears** to have called fork(), since the parent did call fork()
  - So fork() returns **twice**
    - Once (as normal) in the **parent**: returns the new child’s pid
    - And once (appearing to be normal) in the **child** process: returns 0



## Running a New Program in the Current Process

```
int execve(const char *pathname, char *const _Nullable argv[],
           char *const _Nullable envp[]);
int execl(const char *pathname, const char *arg, ...
          /*, (char *) NULL */);
int execv(const char *pathname, char *const argv[]);
```

***Replaces entire calling address space with program specified by pathname***

- Many variants of “exec”: execve() is a **kernel** call, others are **library** calls
- argv (or args ...) is a vector of the individual char \* command line arguments
  - argv[0] should be the program name
- On success, does ***not return to caller*** – begins at entry point of new program

## Waiting for a Child Process to Finish

```
pid_t wait(int *_Nullable wstatus);
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);
```

***wait() waits for any child to exit, waitpid() can wait for a specific child***

- The parent “reaps” (i.e., collects) the exit status (and pid) of its child
- Calling wait() is equivalent to calling waitpid() with pid = -1, options = 0
  - can also give other reasons you want child status report (not relevant here)
- Both wait() and waitpid() return the pid of the child
- If wstatus != NULL, points to int into which to store the exit status of that child
- Returns -1 if no remaining children (none still running and none unreported), with errno = ECHILD

## The Tree of Processes

***The parent/child relationship created by fork() makes all processes form a tree***

- The initial process created at “boot time” is called “init” (pid = 1)
  - init forks one child for log in on each hardware terminal
  - init forks one child for each of several “daemons” (services) such as “sshd”
- Each login process eventually exec’s your login shell
  - When you log out, your shell exits
- init loops, calling wait to reap each of its children
  - When init sees a log in process exit, it fork to create a new child for log in
- If any process terminates while some of its children are still running
  - They are inherited by (“reparented to”) init, so will be reaped when needed

## A wait() Example

```
pid_t wpid;
int wstatus, i;
for (i = 0; i < N; i++)
    if (fork() == 0)
        exit(100 + i); /* exit a child process */
for (i = 0; i < N; i++) {
    wpid = wait (&wstatus);
    if (WIFEXITED(wstatus))
        printf("Child %d terminated with exit status %d\n",
               wpid, WEXITSTATUS(wstatus));
    else
        printf("Child %d terminated abnormally\n", wpid);
}
```

WIFEXITED and WEXITSTATUS are defined by #include <sys/wait.h>

Sees processes in the arbitrary order they exit

## A waitpid() Example

```
pid_t pid[N], wpid;
int wstatus, i;
for (i = 0; i < N; i++)
    if ((pid[i] = fork()) == 0)
        exit(100 + i); /* exit a child process */
for (i = 0; i < N; i++) {
    wpid = waitpid(pid[i], &wstatus, 0);
    if (WIFEXITED(wstatus))
        printf("Child %d terminated with exit status %d\n",
               wpid, WEXITSTATUS(wstatus));
    else
        printf("Child %d terminated abnormally\n", wpid);
}
```

WIFEXITED and WEXITSTATUS are defined by #include <sys/wait.h>

Sees processes in the order created, given their order in pid[] array

COMP 321

Copyright © 2025 David B. Johnson

Page 21

21

## Getting Process IDs

```
pid_t getpid(void);
pid_t getppid(void);
```

***fork() tells your parent the process ID of the new child process***

- For the child process (or any process) to get its own process ID, `getpid()`
  - Always succeeds, perhaps the simplest possible kernel call
- For the child process (or any process) to get the process ID of its own parent, `getppid()`
  - This will generally be the process that did the `fork()` to create you
  - But if your parent already terminated earlier, you will have been inherited by (reparented to) `init` = process id 1

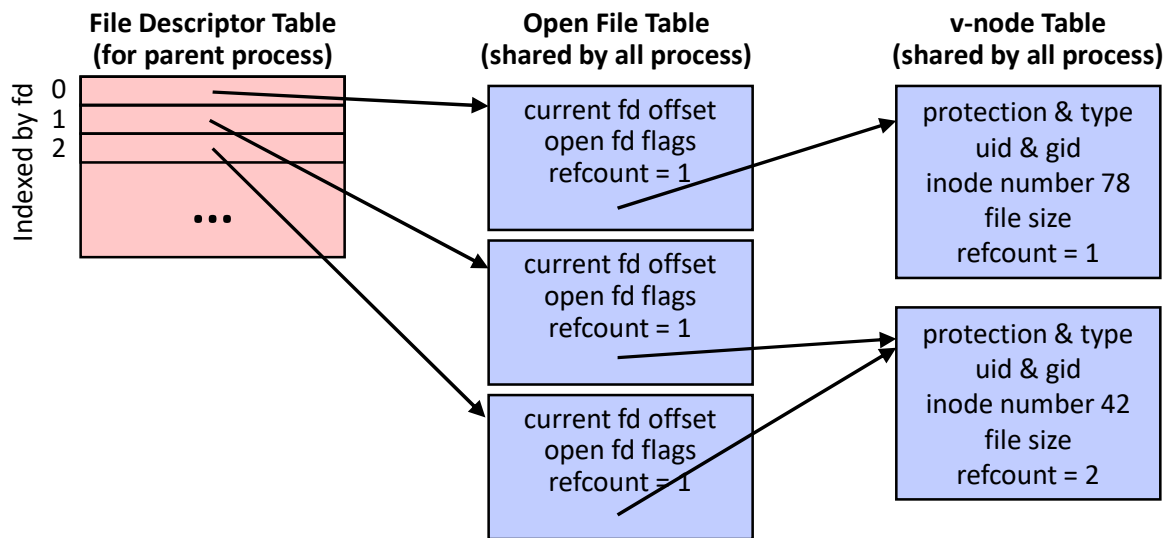
COMP 321

Copyright © 2025 David B. Johnson

Page 22

22

## Kernel File Data Structures Before the Fork



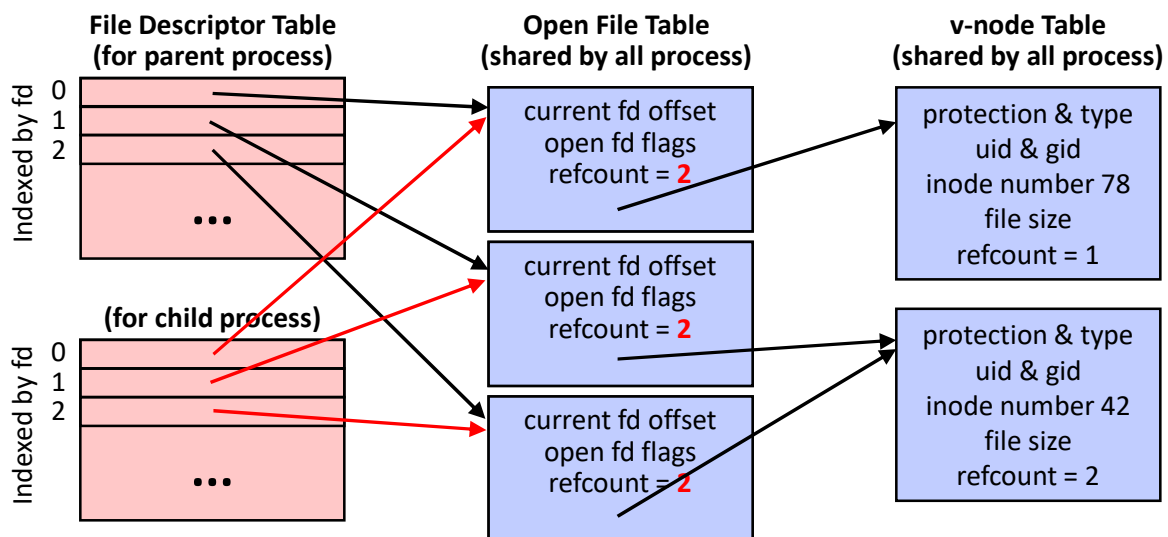
COMP 321

Copyright © 2025 David B. Johnson

Page 23

23

## Kernel File Data Structures After the Fork



COMP 321

Copyright © 2025 David B. Johnson

Page 24

24

## Kernel File Data Structures After the Fork

***The parent and child share all of their open file descriptors***

- Each is open to the same file, for the same O\_RDONLY, O\_WRONLY, O\_RDWR
- And the current offset in each is shared between the processes

### ***Example***

- File “shell\_script” contains:

```
ls  
cat abc
```

- And suppose you run the command  
./shell\_script > outfile

Without sharing the offset in each open fd, the output from “cat abc” would **overwrite** in “outfile” the output from “ls”

## Possible Alternatives to Fork() in Other OSs

***Example: Digital Equipment Corporation VMS Operating System***

```
status = sys$creprc( 12 arguments );
```

***Example: Microsoft Windows Operating System***

```
status = CreateProcess( 10 arguments);
```

In both cases, many arguments are complex structs or arrays of structs

***How can fork() be so simple (0 arguments!) and these are so complex?***

- They try to encode in those arguments everything you might want to do in Unix between the fork() and the exec
  - Redirect file descriptors, change resource limits, etc.
  - In Unix, you can do what you want, since the child shell knows what the parent shell knows and can change anything before the exec