Processses

CSC209H5: Software Tools & Systems Programming

Robert (Rupert) Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

March 6, 2023

Lesson Plan

1. Introduction to Processes
   - `fork`
2. Termination & Status
3. Testing Your Understanding
4. Loading into a Process
   - `exec`
5. Putting it Together

Acknowledgements

Parts of the slides are borrowed from Andi Bergen, Karen Reid, Alan Rosenthal and Arnold Rosenbloom.

Section 1

Processes

Processes: What are they?

A **program** is executable file on disk (either source code or compiled machine code), and a **process** is a running instance of a program.

- Executing multiple instances of the same program launches multiple processes.
  - Example: running multiple instances of Notepad.
- A single instance of a program may launch multiple processes.
  - Example: Firefox/Chrome run one-process-per-tab.
- Each process has its own memory space, including its own stack and heap.
- A process cannot access the variables/memory of another process.
- Process ID (PID): unique, non-negative integer identifier; a handle by which to refer to a process.
- The first process created when the system boots up is `init`, with PID `1`.
  - `init` is typically provided by `systemd` on Linux.
  - Others such as `runit` and `openrc` exist for Linux/BSD/UNIX-like systems.
  - On MacOS, the `init` process is `launchd`.

Processes: What do they do?

Processes can be running, ready or blocked.

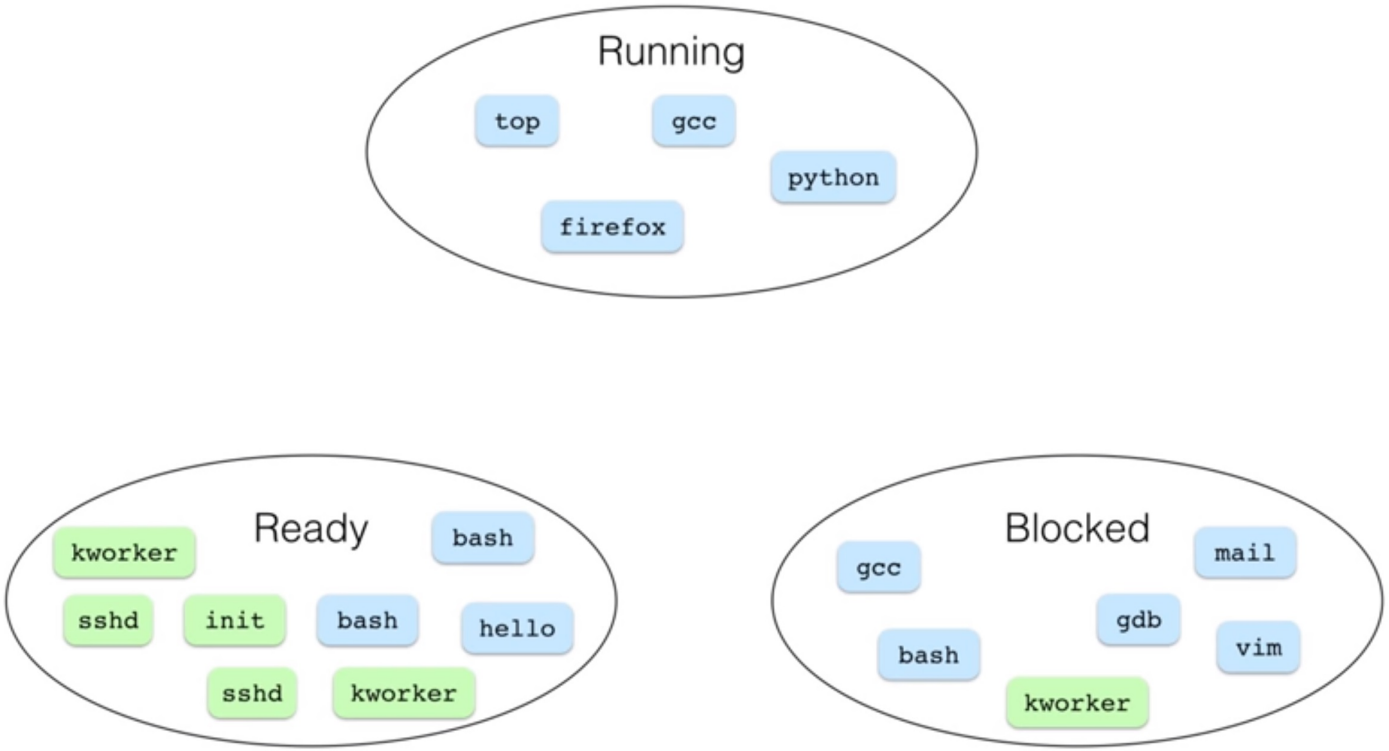


Figure 1: processes grouped by states

Currently-Running Processes

Try `pstree` from `bash` to print the tree of currently-running processes

Processes: What do they do?

Processes can be running, ready or blocked.

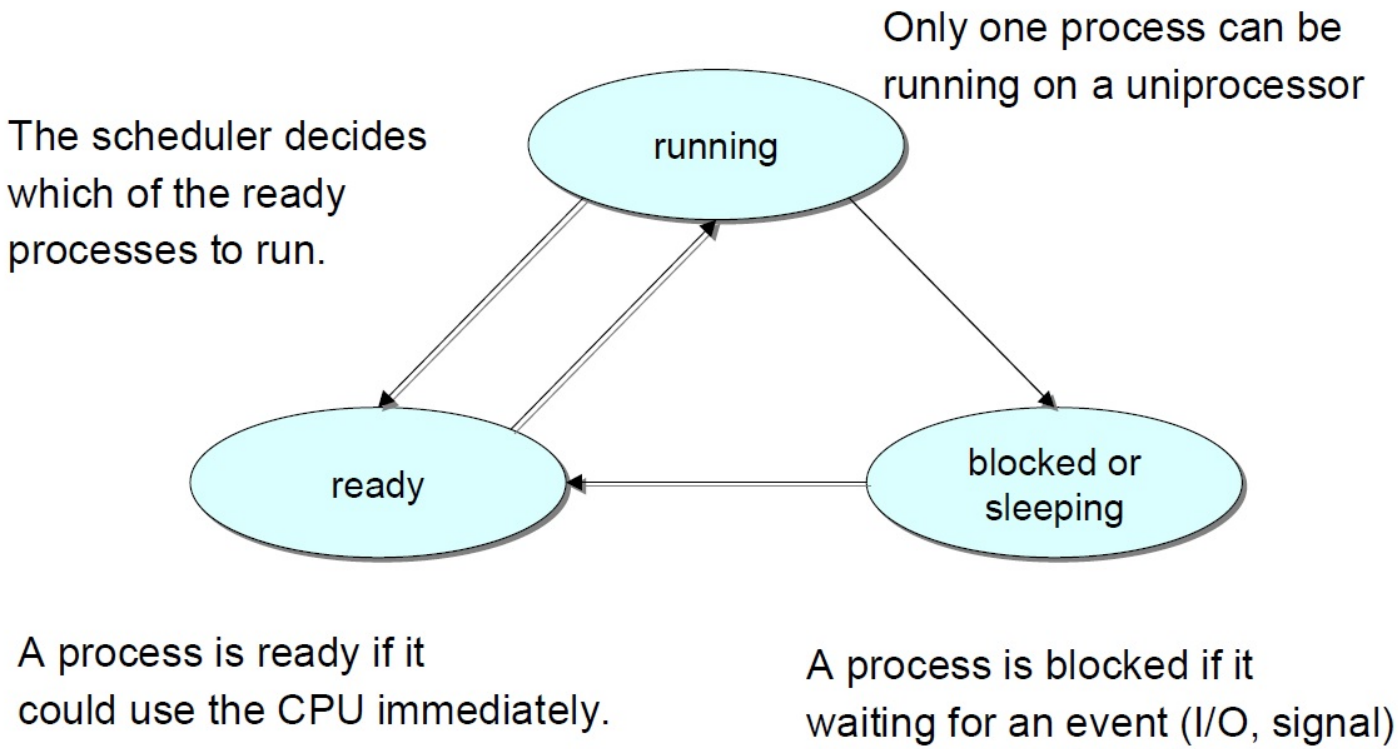


Figure 2: what the states mean

Currently-Running Processes

Try `pstree` from `bash` to print the tree of currently-running processes

Processes: Creation

In UNIX-like systems, processes are created by the `fork()` system call.

- `fork()` allows one process, the parent, to create a new process, the child.
- The new child process is an (almost) exact duplicate of the parent: The two processes execute the same program text and the child obtains copies of the parent's stack, data, and heap. Their PIDs are different though.
- After `fork()` has completed its execution,
  - two processes exist, and, in each process, execution continues from the point where `fork()` returns.
  - each process can modify the variables in its stack, data, and heap segments without affecting the other process;
  - however, we don't know whether the parent process or the child process will execute first.
- The two processes can be distinguished via the value returned from `fork()`:
  - Return value in parent is the PID of the child process.
  - Return value in child is 0 if there was no error.

Processes: pid_t (data type)

The pid_t data type is a signed integer type which is capable of representing a PID. In the GNU C Library, this is an int (but could be long*). Consider the following.

```c
int i = 5;
printf("%d\n", i);
pid_t pid = fork();
if (pid > 0)
    i = 6; /* only parent gets here */
else if (pid == 0) {
    i = 4; /* only child gets here */
    printf(" child: %d\n", pid); // 0, not the actual PID
    printf("parent: %d\n", ???); // parent's PID?
}
printf("%d\n", i);
```

- How to print the actual PID of the child if pid is 0?
- How to print the PID of the parent from the child process?

Processes: pid_t (functions)

How to print the actual PID of the child if pid is 0? How to print the PID of the parent from the child process?

We have access to a few functions that return pid_t identifiers.

```
pid_t getpid (void);  // returns the PID of the current process.
pid_t getppid (void); // returns the parent PID of the current process.
pid_t gettid (void);  // returns the thread ID of the current thread
```

We can then call getpid and getppid in the child process.

```
else if (pid == 0) {
    i = 4; /* only child gets here */
    printf(" child: %d\n", getpid());
    printf("parent: %d\n", getppid());
}
```

Processes: Creation (Example)

What happens when you fork? Conceptually, it looks like this:

```
// original process (parent)
int i = 5;
printf("%d\n", i);
pid_t pid = fork();
if (pid > 0)
    i = 6;
else if (pid == 0) {
    /* doesn't run */


}
printf("%d\n", i);
```

```
// child process


pid_t pid = fork(); // spawns
if (pid > 0)
    /* doesn't run */
else if (pid == 0) {
    i = 4;
    printf("child: %d\n", getpid());
    printf("parent: %d\n", getppid());
}
printf("%d\n", i);
```

Processes: Monitoring

In many applications where a parent creates child processes, it is useful for the parent to be able to monitor the children to find out when and how they terminate. This facility is provided by `wait()` and a number of related system calls.

```
#include "sys/wait.h"
pid_t wait(int *status);
```

The `wait()` system call waits for one of the children of the calling process to terminate and returns the termination `status` of that child in the buffer pointed to by `status`. Returns PID of terminated child, or `-1` on error.

- If no (previously unwaited-for) child of the calling process has not yet terminated, the call blocks until one of the children terminates.
- If a child has already terminated by the time of the call, `wait()` returns immediately, returning the process ID of the child that has terminated. The termination `status` of the child is stored at the memory space `status` is pointing to.
- If the calling process doesn't have any child processes, `wait()` returns immediately with an error.

File Descriptors vs. File Descriptions

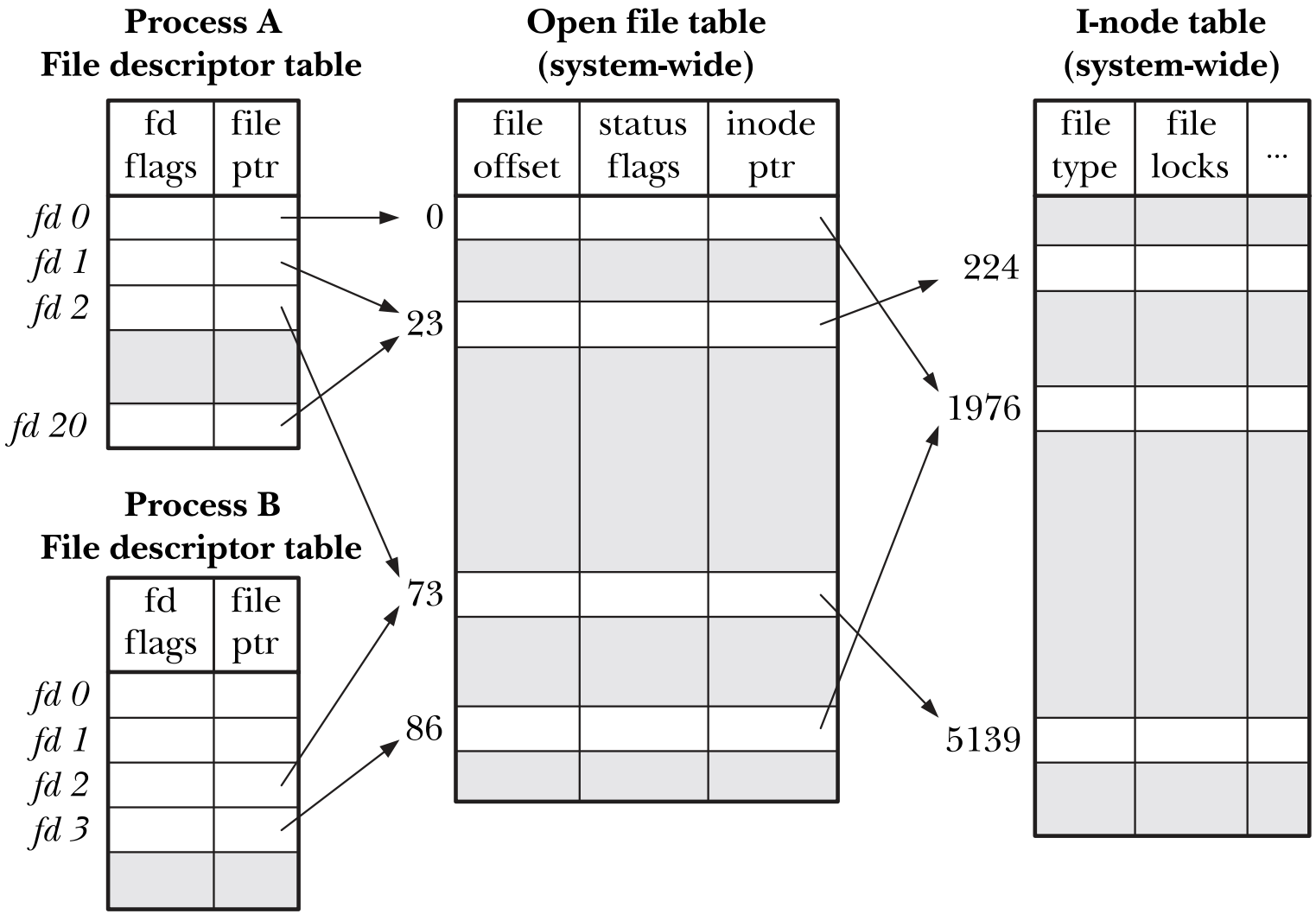


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Figure 3: Mapping between file descriptors and file descriptions

File Descriptors: Sharing the Same Open File Description

A child process gets a **copy** of the parent's file descriptor table. So all open files open in the parent **before** the `fork` are also open in the child.

Immediately after the `fork`, both parent and child's file descriptors refer to the same *open file descriptions* in the system-wide *open file table*.

If the parent process `opens` a file before a `fork`:

- Whichever process (parent or child) calls `fread` first will read the first `N` bytes.
- Whichever process calls `fread` afterwards will read the next `M` bytes.

Since both parent and child processes refer to the same file description, they both share the same *offset* value.

Section 2

Termination & Status

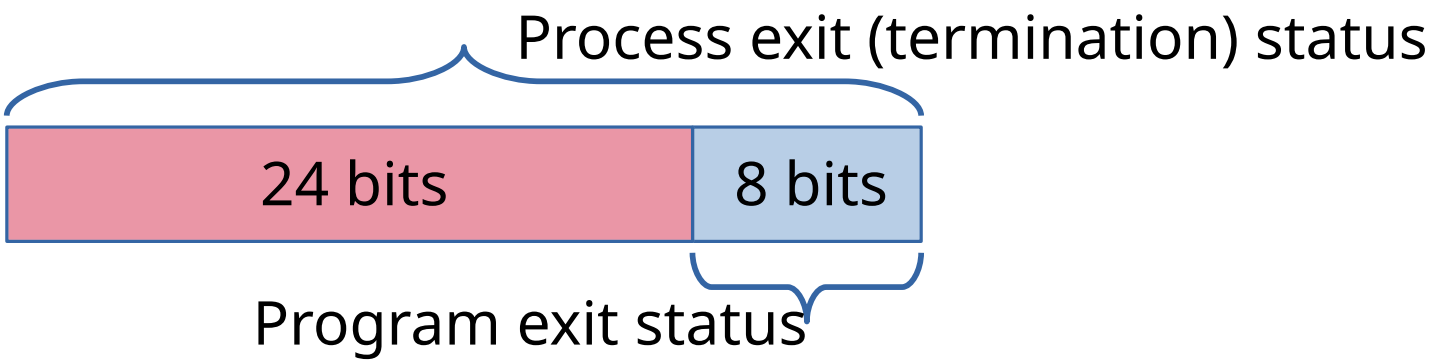
Termination & Status (_exit)

A process may terminate in two general ways:

1. abnormal termination
2. normal termination, using the `_exit()` system call.

```
#include "stdlib.h"
void _exit(int status);
```

A *process exit status* is saved to be reported back to the parent process via `wait` or `waitpid`. If the program exited, this status includes as its low-order 8 bits the *program exit status*.



Source: The GNU C Library

Termination & Status (exit)

Programs generally don't call `_exit()` directly. Instead the `exit()` library function is called. `exit()` performs some cleanup (e.g., flush `stdio` streams) and calls `_exit()`, which sets the process exit status (or termination status), and terminates the process.

```
#include "stdlib.h"
void exit(int status);
```

The `status` argument given to `_exit()` and `exit()` is the program's exit status, which becomes part of the process's exit status.

What's the Difference?

```
#include "stdio.h"
#include "stdlib.h"
#include "unistd.h"

int main() {
    printf("Hi");
    _exit(0);
}
```

```
int main() {
    printf("Hi");
    exit(0);
}
```

Termination: Exit vs. Return

Inside `main()`, `return` and `exit()` are nearly equivalent (save some edge cases):

1. The return value of `main()` is the *program exit status* passed to `exit()`.
2. `exit()` performs some cleanup (e.g., flush `stdio` streams) and calls `_exit()`.
3. `_exit()` sets the *process exit status*, or *termination status*, and terminates the process.

Outside of `main()`, use `exit()` to terminate the process.

Exit Status Conventions

Return `0` on success, any other value on error.

But if you're a real GNU/Linux geek...

> *A general convention reserves status values 128 and up for special purposes.*
>
> *In particular, the value 128 is used to indicate failure to execute another program in a subprocess.*

Termination: Orphans & Zombies

The lifetimes of parent and child processes are usually not the same: either the parent outlives the child or vice versa.

Orphan Processes

- A terminating process may be a parent; in that case all of its children processes become orphans.
- The kernel ensures all of the orphaned processes are adopted by `init`.

Zombie Processes

When a child terminates before its parent, the parent should still be permitted to perform a `wait()` at some later time to determine how the child terminated.

- The kernel deals with this situation by turning the child into a *zombie*, a process that is waiting for its parent to accept its return code.
- When a process become a zombie, most of the resources held by it are released back to the system except an entry in the kernel's process table recording the child's PID, termination status, and resource usage statistics.

Section 3

Processes: Test Your Understanding

Processes: Test Your Understanding

1. How do we obtain the program exit status from the process exit status?
2. What happens if a process terminates before its exit status is obtained by its parent?
3. What happens if a parent process terminates before waiting for all its children?

Processes: Test Your Understanding

Here are some questions:

1. How do we obtain the program exit status from the process exit status?
   - Use macros defined in `wait.h` (see `man 2 wait`), e.g.,
   - `WIFEXITED(status)` to see if process terminated normally or abnormally.
   - `WEXITSTATUS(status)` to obtain program exit status.
2. What happens if a process terminates before its exit status is obtained by its parent?
   - The child process becomes a *zombie process*. Operating system retains minimal information about the process until the parent obtains exit status via `wait()`.
3. What happens if a parent process terminates before waiting for all its children?
   - The child processes become *orphan processes*. Orphan processes are *adopted* by the `init` process.

Processes: Test Your Understanding

Which statements are true when `fork()` is called?

1. True/False? The child process shares the same PID as the parent process.
2. True/False? `fork()` will only fail if the total number of processes under execution by a single user would be exceeded.
3. True/False? After `fork()` has been called successfully, a value of 0 is returned to the child.

Processes: Test Your Understanding

Which statements are true when `fork()` is called?

1. True/False? The child process shares the same PID as the parent process. **False**.
2. True/False? `fork()` will only fail if the total number of processes under execution by a single user would be exceeded. **False**.
3. True/False? After `fork()` has been called successfully, a value of 0 is returned to the child. **True**.

See `man 2 fork`: `linux.die.net/man/2/fork`

Processes: Test Your Understanding

Consider the program below that runs without errors. How many child processes?

```c
int main(void) {
    printf("Mangoes\n");
    int r = fork();
    printf("Apples\n");
    if (r == 0) {
        printf("Oranges\n");
        if (fork() >= 0) printf("Bananas\n");
        return 0;
    }
    printf("Peaches\n");
    for (int i = 0; i < 3; i++) {
        if ((r = fork()) == 0) {
            printf("Pears\n"); exit(0);
            printf("Nectarines\n"); continue;
        }
        printf("Plums\n");
    }
    return 0;
}
```

Processes: Test Your Understanding

1. How many times is each fruit printed?

Fruit Name	Times Printed
`Mangoes`	
`Apples`	
`Oranges`	
`Bananas`	
`Peaches`	
`Pears`	
`Nectarines`	
`Plums`	

2. Several orderings of the fruit names are possible valid output. Some of these orderings even have the unix prompt displaying before the final fruit name (or names). Explain why this happens.
3. Not all of the fruit names could appear after the prompt in a valid output. For example the word Mangoes will never appear after the prompt. List all the fruit names that could occur after the prompt.

Processes: Test Your Understanding

1. How many times is each fruit printed?

Fruit Name	Times Printed
`Nectarines`	0
`Mangoes/Oranges/Peaches`	1
`Apples/Bananas`	2
`Plums/Pears`	3

2. Several orderings of the fruit names are possible valid output. Some of these orderings even have the unix prompt displaying before the final fruit name (or names). Explain why this happens.
   - *The shell is the parent of and waits for the original process before printing the shell prompt. But this original process doesn't wait for its own children, so latters' output may be printed after their parents terminate.*
3. Not all of the fruit names could appear after the prompt in a valid output. For example the word Mangoes will never appear after the prompt. List all the fruit names that could occur after the prompt.
   - `Apples`, `Bananas`, `Oranges`, `Pears`

Section 4

Loading into a Process (exec)

exec: What is it?

The exec functions (provided by `unistd.h`) load a new program into the current process image. The process retains its original PID.

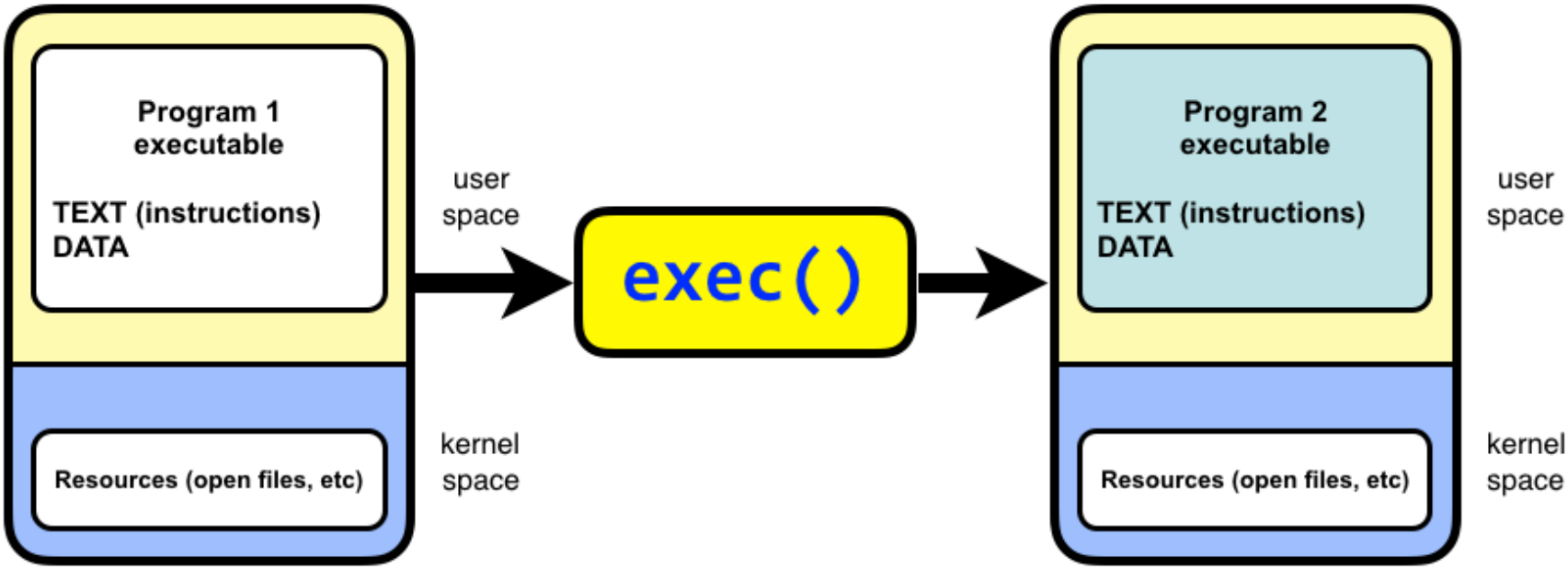


Figure 4: `exec*()`

exec: The Basic execve

The `execve()` system call loads a new program into a process's memory. During this operation, the old program is discarded, and the process's stack, data, and heap are replaced by those of the new program.

```
#include "unistd.h"
extern char **environ; // what's this for?

int execve(const char *filename, char *const argv[], char *const envp[]);
```

- `v`: arguments are passed by array `argv` at run-time.
- `e`: extra arguments can be specified in `envp`.
- The most frequent use of `execve()` is in the child produced by a `fork()`.
- It never returns on success, but returns `-1` on error. Why?
- If it does return, you may want to use `perror` right after.

See: `linux.die.net/man/2/execve`.

Arnold's Examples

`www.cs.toronto.edu/~arnold/209/12s/lectures/process/index.html`

exec: Front-Ends

Various C library functions, all with names beginning with exec, are layered on top of the `execve()` system call. Each of these functions provides a different interface to the same functionality. The front-end functions differ mainly in how they are called.

```c
#include "unistd.h"
extern char **environ;
int execve(const char *filename, char *const argv[], char *const envp[]);
// front-ends
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

See: `linux.die.net/man/3/exec`.

How do they stack?

Try tracing all of these functions in `codebrowser.dev`.

exec: Front-Ends

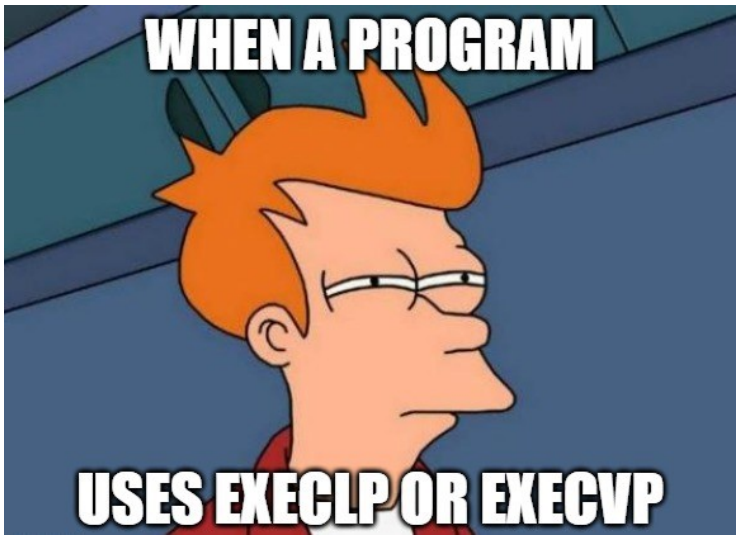
Function	Spec of Program File (-, p)	Spec of Arguments (v,l)	Source of Environment (e,-)
execl()	pathname	list	caller's environ
execle()	pathname	list	envp argument
execlp()	filename + $PATH	list	caller's environ
execv()	pathname	vector (array)	caller's environ
execve()	pathname	vector (array)	envp argument
execvpe()	filename + $PATH	vector (array)	envp argument
execvp()	filename + $PATH	vector (array)	caller's environ

The characters in the function name exec* can be informative.

- l: arguments are statically known at at compile time.
- v: arguments are passed by array argv at run-time.
- e: extra arguments can be specified in envp.
- p: if the program isn't found in the current directory, it'll search in $PATH.

exec: Safety & Security

Because this is C, there are some safety/security implications programmers should consider. Using `execlp` and `execvp` can be very dangerous when used improperly.



Why? What makes these two special?

- Hint: Section 27.2.1 of The Linux Programming Interface by Michael Kerrisk.

Try looking up other vulnerabilities of `exec` functions...

exec: Practice!

1. What's the difference between these?

```
execle("/bin/ls", "ls", NULL, NULL);
```

```
char* args[] = {"ls", NULL};
execve("/bin/ls", args, NULL);
```

2. Write a program similar to `yes` that just prints `no` without the use of loops.
3. Write two programs `tick` and `tock` that `print()` their respective names and call each other after a second has passed (for a specified number of seconds).
   - You can't use loops.
   - You can use `sleep()`, `atoi()`, and `sprintf`.
4. If you're feeling brave, try to write a single binary to mimic `tick` and `tock`.
   - Hint: it's easier if the binary is store somewhere on the `$PATH`.
   - Hint: try using `argv[0]` just like for `no`.

Sample Solution

`github.com/rhubarbwu/csc209/blob/master/lectures/lec08/txck.c`

Section 5

Putting it Together

Putting it Together

A shell can `fork` and `exec` to execute other programs. For example...

1. Shell process `p` waits for keyboard input.
2. You type `ls`.
3. Shell forks child process `c`.
4. Process `c` uses an exec function to run `ls`.
5. Process `p` calls `wait` to wait for `c` to terminate, and then prints new prompt.

The skeleton might look like this:

```
while (1) { // infinite
    print_prompt();
    read_command(command, parameters);
    if (fork()) wait(&status); // parent
    else execve(command, parameters, NULL); // children
}
```

Putting it Together: Arnold's Examples (1)

```c
int main() {
    int x = fork();
    if (x == 0) { /* child */
        execl("/bin/ls", "ls", (char *)NULL);
        perror("/bin/ls");
        return 1;
    }
    /* parent */
    int status;
    wait(&status);
    printf("exit status %d\n", status >> 8);
    return 0;
}
```

Putting it Together: Arnold's Examples (2)

```c
int main() {
    for (int num = 2; num > 0; num--) {
        if (fork() == 0) { /* child */
            execl("/bin/ls", "ls", (char *)NULL);
            perror("/bin/ls");  // should never get here!!
            exit(1);
        }
    }
    /* parent */
    int status;
    while (wait(&status) > 0)
        printf("Parent: exit status %d\n", WEXITSTATUS(status));
    return 0;
}
```

Putting it Together: Arnold's Examples (3)

```c
int main() {
    if (fork() > 0) { /* parent */
        int status, pid = wait(&status);
        printf("pid %d exit status %d\n", pid, status >> 8);
        return 0;
    }
    /* child */
    close(1); // 1==stdout
    // NOTE: the next open takes the first open fd (=1).
    if (open("file", O_WRONLY|O_CREAT|O_TRUNC, 0666) < 0) {
        perror("file"); return 1;
    }
    // where is stdin coming from, stdout going to?
    // remember, command run by exec inherits all open fds!
    execl("/bin/ls", "ls", (char *)NULL);
    perror("/bin/ls"); return 1;
}
```

Finale: Don't Try This At Home

The University of Toronto will not be held liable for any data loss or system damage resulting from the use of the following program.

```c
// C program Sample for FORK BOMB
// It is not recommended to run the program as
// it may make a system non-responsive.
#include "stdio.h"
#include "sys/types.h"
#include "unistd.h"

int main() {
    while(1) fork();
    return 0;
}
```

Read about the fork bomb: `www.geeksforgeeks.org/fork-bomb/`