University of Pittsburgh | School of Computing and Information

Process Management: fork-exec model

CS 0449: Introduction to Systems Software

Shinwoo Kim
Teaching Assistant
shinwookim@pitt.edu
https://www.pitt.edu/~shk148/

Spring 2023, Term 2234
Friday 12 PM Recitation
5502 Sennott Square

Course News!

- Exam II next week
- Quiz 2 graded
- Project 3 due tonight
  - Late submissions until the 27th
- Trying something new
  - www.pollev.com/shinwookim908
  - to improve recitation participation
  - gauge your understanding
  - and provide more practice for the upcoming exam

We use dynamic memory because:

The heap is significantly faster than the stack

Storing data on the stack requires knowing the size of that data at compile time

The stack is prone to corruption from buffer overflows.

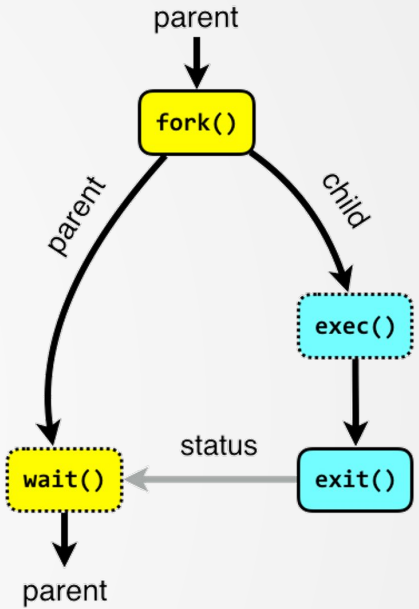
None of the above

Process Management

The Linux Fork-Exec model

Creating new processes & programs

- `fork-exec` model (Linux)
  - `fork()` copies the current process
    - Creating a "child" process that is a duplicate of the memory and state of its parent process
  - `exec*()` replaces the current process's code and address space with the code for a different program
    - Family: `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`
  - `fork()` and `exec()` are system calls
- Other system calls for process management
  - `getpid()` gets process id
  - `exit(int)` ends the current process
    - Argument is known as the **exit code**
    - We can have processes that are no longer running, but not yet deallocated (Zombie processes)
  - `wait()` yields the process and returns only when the child process ends
    - Return value of `wait()` is the process id of the child that exited
    - Specify which child to wait for using `waitpid(pid_t)`

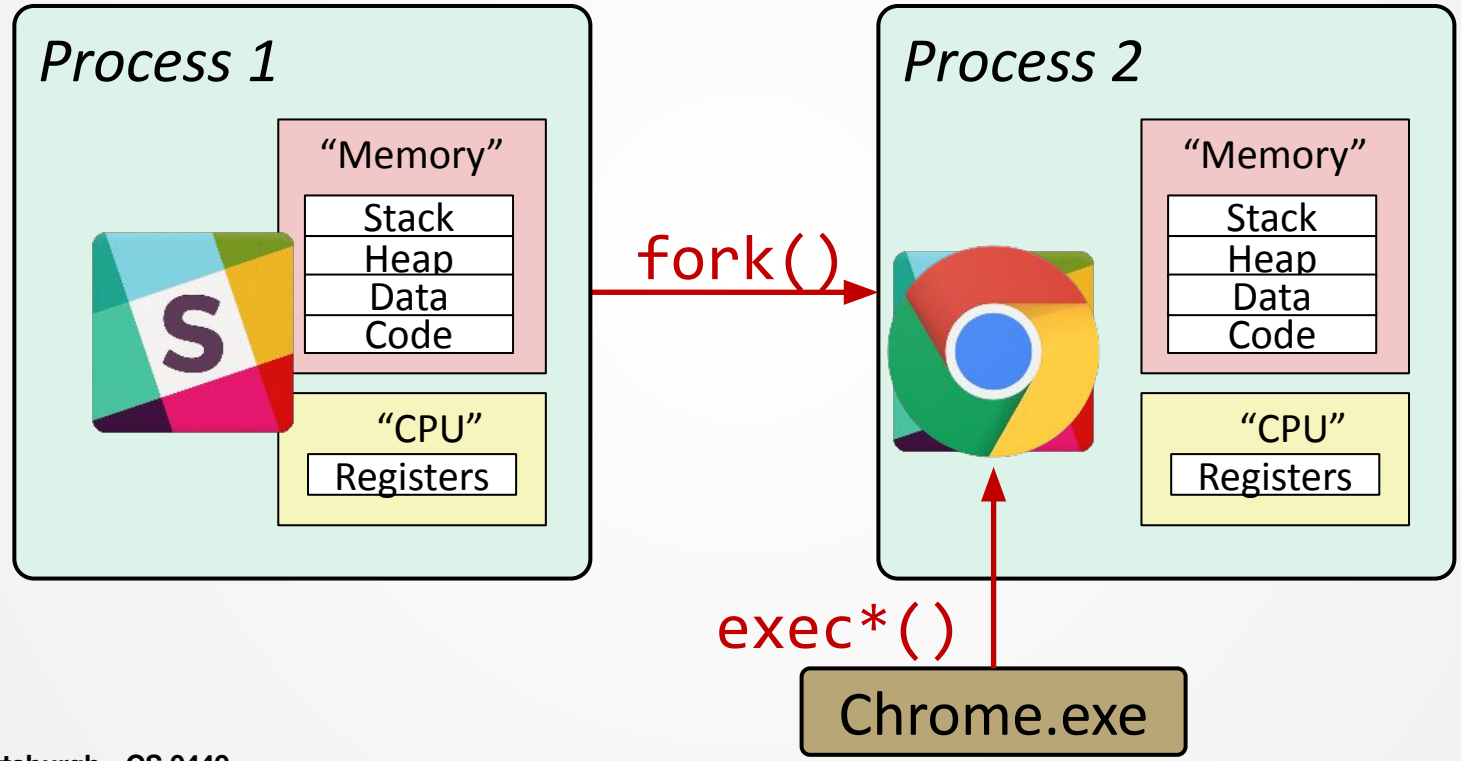


Do parent processes wait for grandchildren processes?

True

False

Creating new processes & programs



`fork()`: creating new processes

- `pid_t fork(void)`
  - **Returns 0 to the child process**
  - **Returns child's process ID (PID) to the parent process**
- Child is almost identical to parent:
  - Child gets an identical (but separate) copy of the parent's address space
  - Child has a different PID than the parent
- fork is unique (and often confusing) because it is called **once** but returns "**twice**"

Understanding fork()

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Understanding fork

Process X

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Process Y (child)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Understanding fork

Process X (parent)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();          pid = Y
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Process Y (child)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();          pid = 0
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
hello from parent
```

```
hello from child
```

Which one appears first?

Modeling `fork()` with process graphs

- A **process graph** is a useful tool for capturing the partial ordering of statements in a concurrent program
  - Each vertex is the execution of a statement
  - a → b means a happens before b
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no in edges

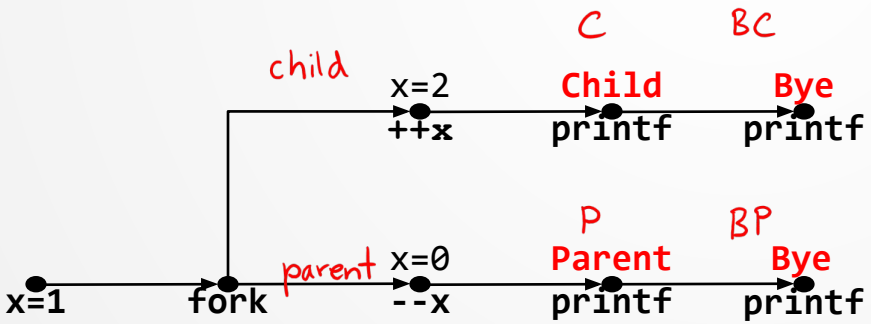
Fork example

```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0)
     printf("Child has x = %d\n", ++x); // child only
    else
     printf("Parent has x = %d\n", --x); // parent only
    printf("Bye from process %d with x = %d\n", getpid(), x); // both
}
```

- Both processes continue/start execution after fork
  - Child starts at instruction after the call to fork (storing into pid)
- Can't predict execution order of parent and child
- Both processes start with x=1
  - Subsequent changes to x are independent

Modeling `fork()` with process graphs

```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0)    printf("Child has x = %d\n", ++x);
    else             printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```



C BC P BP
P BP C BP
C P BC BP
C P BP BC
C BC BP P
P BC C BP

As long as C comes before BC and P comes before BP

} Not possible!

PEV: Is the following sequence of outputs possible?

```
void nestedfork() {
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

```
Seq 1:
L0
L1
Bye
Bye
Bye
L2
```

Are the following sequences of outputs possible?

PEV: Are the following sequences of outputs possible?

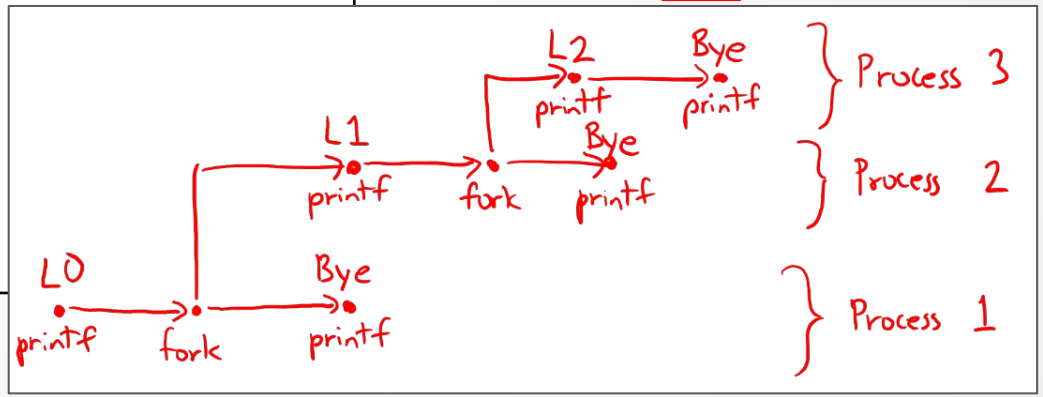
```
void nestedfork() {
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

```
Seq 1:
L0
L1
Bye
Bye
Bye
L2
```

PEV: Are the following sequences of outputs possible?

```
void nestedfork() {
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Seq 1:
L0
L1
Bye
Bye
Bye
L2 NO!



Fork bombs 

- A Fork bomb (AKA rabbit virus, or wabbit)
  - is a denial-of-service attack
  - wherein a process continually replicates itself to deplete available system resources
  - `while(true) { fork(); }`
- `:(){ :|:& };:` ← This is all you need for a fork bomb
  - https://en.wikipedia.org/wiki/Fork_bomb
- Try experimenting on your own machine
  - Preferably on a virtual machine
  - Worst case scenario, you just reboot your machine!
- That being said, if you fork bomb Thoth, your access to it will be revoked
  - *And you will need access for other courses (e.g., 1550)*

Lab 5: Loading & Forking

Executables and Plugins

Now, it's your turn!

- In lab 5, you will practice:
  a. Learn how libraries are loaded dynamically
  b. Learn how processes are created
     - Using `fork()`, `exec()`
     - And `wait()`
- Three parts
  a. Plugging your code!
  b. FORK!
  c. Gradescope Questions

Collaboration on this lab is allowed and encouraged!

Part A: Plugging your code!

- Read the handout on how *function pointers* work
  - A function pointer is a variable that stores the address of a function that can later be called through that function pointer.
- `return_type (*pointer_name)(list,of,argument,types);`
  - `long int (*f_ptr)(int, int);`
  - Really useful for general purpose functions!
    - A sort function that can work on any data type
      - Works as long you pass in a function that can *compare* two values of that type

```
void qsort(void *base, size_t n_elem, size_t elem_size,
int(*compare)(const void *, const void *));
```

Part A: Plugging your code!

- Read the handout on how *function pointers* work
  - A function pointer is a variable that stores the address of a function that can later be called through that function pointer.
- `return_type (*pointer_name)(list,of,argument,types);`
  - `long int (*f_ptr)(int, int);`
  - Really useful for general purpose functions!
    - A sort function that can work on any data type
      - Works as long you pass in a function that can *compare* two values of that type

```
void qsort(void *base, size_t n_elem, size_t elem_size,
int(*compare)(const void *, const void *));
```

If I want to use `qsort()` on a different data type (e.g., Strings), all I need to do is, swap out the comparison function!

```
int compare_ascending(const void *val1, const void *val2) {
        // Compares integers
         return *(const int *)val1-*(const int *)val2;
}
```

Part A: Plugging your code!

- Build a program that accepts a *plugin name* as a parameter and executes that plugin.
  - Plugin file will have the name `plugin-name.so`
  - All Plugin support
    - `int initialize()`
    - `int run()`
    - `int cleanup()`
  - Your program should be run as
    - `$ ./program plugin-name`

```
// Create a shared object
gcc plugin.c -o plugin.so -shared
```

```c
/* Sample Plugin */
int initialize() {
    printf("Initializing plugin\n");
}

int run() {
    printf("Running plugin\n");
}

int cleanup() {
    printf("Cleaning plugin\n");
}
```

To dynamically link libraries you will need to get familiar with `dlfcn.h` *functions (see lecture slides for examples)*

- Create `plugin_manager.c` that
  - reads the first argument - *you may need to format your argument (e.g., "plugin" → ./plugin.so)*
  - Loads the shared object *Dynamic linking requires the* **`-ldl`** *flag when compiling with* **`gcc`**
  - Runs `initialize()`, `run()`, and `cleanup()` in that order(**`gcc plugin_manager.c … -ldl`**)

Part B: FORK!

- Forking allows us to expand our programs to multiple processes
  - But how can processes communicate with one another?
    - That is, how do we **synchronize** processes? ←*This is often known as interprocess communication*
- Signals are primitive standardized that can be sent to processes
  - By other processes, the OS, etc.
- For example, when you kill a program with `ctrl + c`, the shell sends the `SIGINT` signal to that process
  - Which usually terminates the program
- When you get a *segmentation fault*, the OS usually sends the `SIGSEGV` signal to the process
- However, we can *capture* the signals to do something else
  - For example, on when the user tries to kill the process (`SIGINT`) print “No!” and keep running

Part B: FORK!

- Create a program `run_on_demand.c` that:
  - When it receives signal `SIGUSR1`,
    - Fork-execs `ls`
  - When it receives signal `SIGUSR2`,
    - Fork-execs `ls -l a`

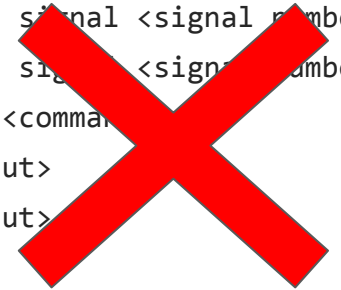
Sample Output

```
Received signal <signal number>.
Running <command>.
<ls output>
Done!
```

- When `CTRL + c` is pressed, the program should print
  - `"Leaving gracefully"`
  - Then exit
- Remember to synchronize the processes
  - Printing order should be respected
  - *The process should "wait" until the* `ls` *is complete*

Sample Output

```
Received signal <signal number>.
Received signal <signal number>.
Running <command>.
<ls output>
<ls output>
Done!
Done!
```



Testing with signals

- How can we test signals?
- Signals are sent by *processes*…so we can create a wrapper program that tests our `run_on_demand` program
  - For example:

This is pseudo-code

```
pid_t pid = fork();
if (pid == 0) // child process
   exec("./run_on_demand");
else // parent process
   kill(pid, SIGUSR1); //send
SIGUSR1 to child
```

- Or we can do so manually:
  - Open up two terminals
  - In terminal 1, run `run_on_demand`
  - In terminal 2, manually send signals
    - `kill(pid, SIGUSR1)`
    - How do we know `pid`?
    - `$ ps ux` gets you the pid of all process (that you are running)

Part C: Gradescope Questions

- Fork tracing questions + extra
- Good exam practice!

Collaboration on this lab is allowed and encouraged!

⇒ You must submit:

1. `plugin_manager.c`
2. `run_on_demand.c`
3. Answer questions on Gradescope