Operating Systems
ECE344

Unix Processes

Ashvin Goel
ECE
University of Toronto
Unix Processes

- A process in traditional Unix consisted of an address space and one thread

- Processes are arranged hierarchically
  - sh process has two children, each process has one parent
## Process-Related System Calls

<table>
<thead>
<tr>
<th>System call</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpid(), getppid()</td>
<td>Get unique id for process, for parent process</td>
</tr>
<tr>
<td>fork(), execve()</td>
<td>Create new process, run new program</td>
</tr>
<tr>
<td>exit(), wait()</td>
<td>Terminate process, and synchronize with terminating process</td>
</tr>
<tr>
<td>kill(), sigaction()</td>
<td>Communicate with another process via signals</td>
</tr>
</tbody>
</table>
Fork System Call

- When a process calls `fork()`, a new process is created
  - Original process is parent, new process is child
- Child process is **identical** to the (one) parent process!
  - Child initially has same address space, thread state as parent
  - After the child or the parent start running, their state diverges

```
sh
ret = fork()
sh (parent)  sh (child)
ret != 0, child’s PID  ret = 0
```

- How can we distinguish the parent from the child?
Fork Example

Understanding fork:
- What does this code print?
- Is the output always the same?
- How can parent and child modify data independently?

```c
int n = 5;
int pid = fork();
if (pid == 0) {
    // run child code
    n = n + 1;
} else { // pid value > 0
    // run parent code
    n = n - 1;
}
printf("n = %d\n", n);
```
How to Run a New Program?

- Recall that fork creates a new, identical process.
- But then how can we run new programs?
  - Rather than running process replicas.
When a process calls `execve(“program”)`, the process starts running a new program

- Appears as if current program exited, new program is running

```
ret = fork()
```

```
sh (pid1)
sh (pid1)  sh (pid2)
execve(“ls”)  ls (pid2)
```
Execve Example

Understanding execve:

- Why does the code after execve not execute under normal circumstances?

```c
char *cmd = "/bin/ls";
char *args[3];
args[0] = "ls";
args[1] = "-l";
args[2] = NULL;
execv(cmd, args);
// code normally
// doesn’t execute
```
Exit System Call

- A process terminates itself by calling `exit(ret_val)`
  - Typically, process calls `exit(0)` when it returns successfully

OS makes `ret_val` available to the parent process

How does parent know when child exits?

- OS makes `ret_val` available to the parent process
A parent process calls `wait(child_pid)` to wait for a child process to exit

- When child calls `exit(ret_val)`, `wait()` returns `ret_val`

```plaintext
sh (pid1)
ret = fork()
sh (pid1)
wait(pid2)
wait returns 0
wait returns 0
```
What happens if child exits before parent issues wait?
  - Child maintains ret_val until wait is issued by parent

**wait before exit**

**wait after exit**

\[ W: \text{wait starts} \quad E: \text{exit starts} \quad C: \text{wait continues} \quad D: \text{exit done} \]
Kill and Sigaction System Calls

- A Unix process can send a message to another process using **signals** that are similar to interrupts
  - Need to use the kill() and sigaction() system calls

- A process uses the `kill(pid, signal_number)` system call to send a **signal** to another process (or itself)
  - Signal is delivered to recipient process, similar to interrupts
    - Normal execution is interrupted
    - A signal handler function is run with the `signal_number` argument
    - When handler function finishes, normal execution continues

- A process uses the `sigaction()` system call to register a **signal handler** function
  - When no handler is setup, a process receiving a signal is forced to exit (hence signals sent via ‘kill’ system call)
Think Time: Code Examples

- Read the man pages of the Unix system calls
  - On google: man fork

- Some code examples are available on the web site
  - shell.c
    - A simple shell program
    - Shows how fork, execve, exit and wait are used
  - unix-signal.c
    - A program that shows how signals are used
  - seg-fault.c
    - Get control when your program has a seg fault!
Unix OS provides various system calls for managing processes
  - fork creates a new clone process
  - execve loads a new program in an existing process
  - exit ends a process
  - wait allows a parent process to wait for a child’s exit
  - kill and sigaction are used to send signals to processes, similar to how external devices interrupt the CPU

Next lecture: Threads in Unix
Think Time: Unix System Calls

- What happens when the various Unix system calls we have discussed fail (due to some error)?
- Unix provides the getppid() call to get the parent’s process id. Why is this call required?
- Why does Unix not provide a getcpid() call to get the child’s process id?
Think Time Answer: Unix System Calls

- What happens when the various Unix system calls we have discussed fail (due to some error)?
  - When Unix system calls fail, their return value is -1, and they set a special variable called errno that indicates the cause of the error. You should always check the return value of a system call to ensure that it has not failed.

- Unix provides the getppid() call to get the parent’s process id. Why is this call required?
  - Recall that fork returns 0 in the child process. the getppid call allows the child to know the parent’s process id, if it needs to communicate with the parent.
Why does Unix not provide a `getcpid()` call to get the child’s process id?

- While a process has one parent (and so a `getppid` call makes sense), a process can have multiple children, and so `getcpid` would be ambiguous. In any case, the parent finds out each child’s PID on a fork.
Think Time: Fork and Execve

- Why does fork have the weird semantics of creating a replica process?
- Why do we need a separate fork and execve – aren’t they always needed together to run a new program?
- The code example in the lecture used “execv” not “execve”. Why?
Why does fork have the weird semantics of creating a replica process?

- Recall that after a fork, the parent and child do not share information because they have their own address space. `fork` allows sharing information implicitly between parent and child. The parent can setup any state that needs to be shared with the child before the call to fork. For example, the parent can setup a variable to a specific value. After fork, the child would see the same value initially. After that, the parent and the child can modify this value independently.
Think Time Answers: Fork and Execve

- Why do we need a separate fork and execve – aren’t they always needed together to run a new program?
  - To run a new program in a separate thread, we do need both fork and exec. However, a program (executable) file can consist of multiple processes. In this case, we can use fork to create the different processes, without requiring an exec to run another program. Also, some of those forked processes could run another program using exec. So the separate fork and exec calls provides significant flexibility.

- The code example in the lecture used “execv” not “execve”. Why?
  - execve has several library variants, including execv, execl, execlp, execle, execv, execvp, execvpe, see man page of execve and execv.
A Unix program typically returns 0 when it is successful, and a non-zero value when it fails for some reason. Why does this choice make sense?

Say a program leaks memory. How can the OS get this memory back?

When a program exits, when is its thread state destroyed?
A Unix program typically returns 0 when it is successful, and a non-zero value when it fails for some reason. Why does this choice make sense?

- Normally the program returns one value (0) on success. However, there can be many reasons for failure. Unix programs return different positive values, 1, 2, 3, ..., for each reason for failure.

Say a program leaks memory. How can the OS get this memory back?

- OS can get this memory back when the program exits. At this point, the program address space is destroyed and any memory that the program had allocated (e.g., stack, heap memory) is freed and returned back to the OS.
When a program exits, when is its thread state destroyed?

- Recall that when exit is called, the OS is still executing in the context of the current thread, and so the thread state of the current thread cannot be destroyed immediately. Instead, it is destroyed when the next thread starts running. However, note that the process’s return value is kept for even longer, until the parent issues wait and lets the child know that it has retrieved the child’s exit value.
A parent calls `wait(child_pid)` to wait for a child process to exit. How does parent know `child_pid`?

Can a Unix process call `wait` on a process that is not its child?

In Lab 3, we allow any thread to invoke `wait` on any other thread. Could this approach cause a problem?

We discussed how the parent can issue `wait` either before or after the child’s exit. What happens if the parent never issues `wait`?
Think Time Answers: Wait

- A parent calls `wait(child_pid)` to wait for a child process to exit. How does parent know `child_pid`?
  - From the fork call.

- Can a Unix process call `wait` on a process that is not its child?
  - Normally no. Some unix variants allow waiting for descendant children, but never on processes that are not a descendant of the process.

- In Lab 3, we allow any thread to invoke `wait` on any other thread. Could this approach cause a problem?
  - A thread A can wait on a thread B, and thread B can wait on thread A, leading to a deadlock.
Think Time Answers: Wait

- We discussed how the parent can issue wait either before or after the child’s exit. What happens if the parent never issues wait?
  - Note that if the parent is still running when a child exits, the parent may still do a wait, so the child needs to assume that a parent may issue a wait. So, after the child exits, it becomes a zombie, and waits for the parent to issue the wait() call.
  - When a parent process exits without calling wait() on the child (regardless of whether the parents exits before or after the child), the kernel makes the init (first) process the parent of the orphaned child. The init process (which never exits) eventually performs wait on this child.
Think Time: Signals

- What are the steps involved in sending and receiving signals?
- When a process sends a signal to a recipient process that has not registered a signal handler, the recipient process is killed. Is the recipient process terminated immediately?
Think Time Answers: Signals

- What are the steps involved in sending and receiving signals?
  - Sender uses kill system call to send signal to another process, receiver registers a signal handler using sigaction call. When signal is received, the signal handler code is invoked in the receiver.

- When a process sends a signal to a recipient process that has not registered a signal handler, the recipient process is killed. Is the recipient process terminated immediately?
  - The recipient process is not terminated immediately. Instead, it exits when it runs next. This avoids race conditions, e.g., the recipient process may be running on another CPU and terminating it while it was running would likely cause errors.
Think Time: Crazy Fork

- What output does this code produce?

```c
int n = 5;
int pid = fork();  // fork 1

if (pid == 0) {
    n = n + 1;
    if (fork()) {  // fork 2
        n = n + 1;
    }
} else {
    n = n - 1;
    if (fork()) {  // fork 3
        n = n - 1;
    }
}

printf("n = %d\n", n);
```