Processes

Operating Systems
Kartik Gopalan

References:
• Chapter 2 of the Tanenbaum’s book
• Chapter 4 of OSTEP book
• man pages in any UNIX/Linux system
Process

- A **process** is a program in execution.
  - A **program** is a set of instructions somewhere (like the disk).
  - These instructions are loaded into the process’ memory “in the beginning” by the OS.

- **Von Neumann model** of computing: Once created, a process continuously does the following
  1. **Fetches** an instruction from memory.
  2. **Decodes** it.
     - i.e., figures out which instruction this is.
  3. **Executes** it.
     - i.e., it does the thing that it is supposed to do, like add two numbers together, access memory, check a condition, jump to a function, and so forth.
Process versus Program

• Program
  • Program is a passive entity stored in the disk
  • Static code and static data

• Process
  • Actively executing code and the associated static and dynamic data.

• Program is just one component of a process.

• There can be multiple process instances of the same program
  • Example: many users can run “ls” at the same time
So what constitutes a process?

- Memory space (static, dynamic)
- Procedure call stack
- Registers and counters:
  - Program counter, Stack pointer, General purpose registers
- Open files, connections
- And more.
Memory Layout of a typical process

- Stack and heap grow towards each other

- Stack:
  - Function Call Arguments, Return Address, Return Values

- Gap

- Heap:
  - Dynamically allocated memory (e.g. malloc())

- Data:
  - Global variables, constants etc

- Text:
  - Program Code
System calls to control process lifetime

- `fork()`
  - Create a process

- `exec()`
  - Run a new program
  - More accurately: Replace the current process with a new program image

- `wait()` or `waitpid()`
  - Wait for a child process to terminate

- `exit()`
  - Terminate the calling process
Process Creation

• Always using the `fork()` system call.

• When?
  • User runs a program at command line

• OS creates a process to provide a service
  • Check the directory `/etc/init.d/` on Linux for scripts that start off different services at boot time.

• One process starts another process
  • For example in servers
Example: fork() and waitpid()

https://oscourse.github.io/examples/fork_ex.c

```c
pid = fork();

if (pid < 0) {
    perror("fork failed:");
    exit(1);
}

if (pid == 0) { // Child executes this block
    printf("This is the child\n");
    exit(0);
}

if (pid > 0) { // Parent executes this block

    printf("This is parent. The child is %d\n", pid);

    ret = waitpid(pid, &status, 0);
    if (ret < 0) {
        perror("waitpid failed:");
        exit(2);
    }

    printf("Child exited with status %d\n", status);
    exit(0);
}
```
The strange behavior of fork()

• fork() is called once …

• But it returns twice!!
  • Once in the parent and
  • Once in the child

• The parent and the child are two different processes.

• Child is an exact “copy” of the parent.

• So how to make the child process do something different?
  • Return value of fork in child = 0
  • Return value of fork in parent = [process ID of the child]
  • By examining fork’s return value, the parent and the child can take different code paths.
Running a new program

- Consider how a shell executes a command

```
$ pwd
/home/user
```

- How does that work?
  - Shell forked a child process
  - The child process executed `/bin/pwd` using the `exec()` system call

- Exec replaces the process’ memory with a new program image.
exec() system call

Parent \[\rightarrow\] Fork \[\rightarrow\] Child

Child \[\downarrow\] Exec

New program image in execution
```c
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork failed\n");
    exit(1);
}

if (pid == 0) {
    if( execlp("echo",
               "echo",
               "Hello from the child",
               (char *) NULL) == -1)
        fprintf(stderr, "execl failed\n");

    exit(2);
}

printf("parent carries on\n");
```
The strange behavior of exec()

- Replaces current process image with new program image.
  - In the last example, parents’ image was replaced by the “echo” program image.

- All I/O descriptors open before exec stay open after exec.
  - I/O descriptors = file descriptors, socket descriptors, pipe descriptors etc.
  - This property is very useful for implementing filters.
Different Types of exec()

- int `exec1(char * pathname, char * arg0, ... , (char *)0);`
  - Full pathname + long listing of arguments

- int `execv(char * pathname, char * argv[]);`
  - Full pathname + arguments in an array

- int `execle(char * pathname, char * arg0, ... , (char *)0, char envp[]);`
  - Full pathname + long listing of arguments + environment variables

- int `execve(char * pathname, char * argv[], char envp[]);`
  - Full pathname + arguments in an array + environment variables

- int `execlp(char * filename, char * arg0, ... , (char *)0);`
  - Short pathname + long listing of arguments

- int `execvp(char * filename, char * argv[]);`
  - Short pathname + arguments in an array

More info: check “man 3 exec”
Terminating a process

• Return from the first function
  • Usually main()

• exit(status)
  • Exit the program.
  • Status is retrieved by the parent using wait().
  • 0 for normal status, non-zero for error
Process Hierarchy Tree

- A created two child processes, B and C
- B created three child processes, D, E, and F
CPU scheduler

• Time-shares many processes on one CPU
Process Lifecycle

- **Ready**
  - Process is ready to execute, but not yet executing
  - It's waiting in the scheduling queue for the CPU scheduler to pick it up.
- **Running**
  - Process is executing on the CPU
- **Blocked**
  - Process is waiting (sleeping) for some event to occur.
  - Once the event occurs, process will be woken up, and placed on the scheduling queue.

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
How do multiple processes share CPU?

If we were to map these states to a graph, we would arrive at the diagram in Figure 4.2. As you can see in the diagram, a process can be moved between the ready and running states at the discretion of the OS. Being moved from ready to running means the process has been scheduled; being moved from running to ready means the process has been descheduled. Once a process has become blocked (e.g., by initiating an I/O operation), the OS will keep it as such until some event occurs (e.g., I/O completion); at that point, the process moves to the ready state again (and potentially immediately to running again, if the OS so decides).

Let’s look at an example of how two processes might transition through some of these states. First, imagine two processes running, each of which only use the CPU (they do no I/O). In this case, a trace of the state of each process might look like this (Figure 4.3).

<table>
<thead>
<tr>
<th>Time</th>
<th>Process$_0$</th>
<th>Process$_1$</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Running</td>
<td>Ready</td>
<td>Process$_0$ now done</td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>–</td>
<td>Running</td>
<td>Process$_1$ now done</td>
</tr>
</tbody>
</table>

Figure 4.3: Tracing Process State: CPU Only
How do multiple processes share CPU?

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

---

**Figure 4.4: Tracing Process State: CPU and I/O**

<table>
<thead>
<tr>
<th>Time</th>
<th>Process$_0$</th>
<th>Process$_1$</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Running</td>
<td>Ready</td>
<td>Process$_0$ initiates I/O</td>
</tr>
<tr>
<td>4</td>
<td>Blocked</td>
<td>Running</td>
<td>Process$_0$ is blocked, so Process$_1$ runs</td>
</tr>
<tr>
<td>5</td>
<td>Blocked</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Blocked</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Ready</td>
<td>Running</td>
<td>I/O done</td>
</tr>
<tr>
<td>8</td>
<td>Ready</td>
<td>Running</td>
<td>Process$_1$ now done</td>
</tr>
<tr>
<td>9</td>
<td>Running</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Running</td>
<td>–</td>
<td>Process$_0$ now done</td>
</tr>
</tbody>
</table>
# Typical Kernel-level data structure for each process

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- See task_struct in Linux source code
Examining Processes in Unix/Linux

- **ps command**
  - Standard process attributes

- **/proc directory**
  - More interesting information if you are the root.

- **top command**
  - Examining CPU and memory usage statistics.
Orphan process

• When a parent process dies, child process becomes an orphan process.

• The init process (pid = 1) becomes the parent of the orphan processes.

• Here’s an example:
  • [https://oscourse.github.io/examples/orphan.c](https://oscourse.github.io/examples/orphan.c)

• Do a ‘ps –l’ after running the above program and check parent’s PID of the orphan process.

• After you are done remember to kill the orphan process ‘kill –9 <pid>’
Zombie Process

• When a child dies, a SIGCHLD signal is sent to the parent.

• If parent doesn’t wait() on the child, and child exit()s, it becomes a zombie (status “Z” seen with ps).

• Zombies hang around till parent calls wait() or waitpid().

• But they don’t take up any system resources.
  • Just an integer status is kept in the OS.
  • All other resources are freed up.
References

- Chapter 2 of the Tanenbaum’s book
- Chapter 4 of OSTEP book
- Man pages for different system calls
  - Try “man 2 <syscall_name>”
    - E.g. man 2 exec
  - Syscalls are normally listed in section 2 of the man page