System Programming: Process Management

CS 2204

* Notes adapted by Doug Bowman from previous work by other members of the CS faculty at Virginia Tech
Processes in Unix

- **Process** – basic unit of execution
  - Executing instance of a program
  - Has a process ID
  - Occurs in a hierarchy of processes (parents and children)
    - Root of the hierarchy is the *init* process
  - Each process has its own state/memory

- **Shell commands dealing with processes:** `ps`, `top`, `kill`, `nice`, ...
Process Management

- System calls dealing with:
  - Process creation
  - Setting the program a process executes
  - Waiting for a process to terminate
  - Terminating a process
  - Sending signals to a process
  - Communicating between processes
Process creation

- pid = fork()
- Creates a new child process that is an exact copy of the current process
  - Same program running at same location
  - Same variable values
  - Same open files
- Only difference: child has new PID
- Returns 0 in the child process
- Returns child’s PID in the parent process
Setting the program a process executes

- **exec family of functions**
- **e.g.** `execlp(executable_name, arg0, arg1, ...);`
- Replaces the current process with a new process image running the executable specified with the arguments listed
- Process retains old PID and any open files
- Other functions: `execl`, `execle`, `execv`, `execvp`, `execve`
Waiting for a process to terminate

- \( \text{pid} = \text{wait}(&\text{status}) \)
  - Suspends execution of the calling process until any child process terminates
  - Returns PID of terminating child
  - Puts exit status of child in status
- \( \text{pid} = \text{waitpid}(\text{pid}, &\text{status}, \text{options}) \)
  - Suspends execution of the calling process until a specific child terminates
Using fork/exec/wait together

```c
pid = fork();
if(pid == 0)
   execl("./program", "program", arg1, NULL);
else
   pid = wait(&status);
//continue execution
```
Process termination

- `exit(status)`
- Terminates the calling process
- Closes all open file descriptors
- Returns status to the parent process
A signal is an interrupt sent to a process.

An operating system supports a small number of signals, identified by signal names.

<table>
<thead>
<tr>
<th>Signal Names</th>
<th>Typical Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>Interrupt (^C)</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>Non-catchable termination</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>Alarm</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>Terminate</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>Quit</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>Abort</td>
</tr>
<tr>
<td>SIGUSR1, SIGUSR2</td>
<td>User-defined</td>
</tr>
</tbody>
</table>

`retval = kill(pid, signal)`
Inter-Process Communication (IPC)

- Information passing between processes
- Two basic paradigms:
  - *Message passing*: processes send information back and forth in message/packets
  - *Shared memory*: processes share a chunk of physical memory and read/write data there to share that information
Pipes are an example of message passing

```c
int fds[2];
retval = pipe(fds);
```

- Creates two file descriptors (a pipe is a file), the first for reading, and the second for writing
- How does another process connect to this pipe?
Basic pipe example

```c
int fds[2]; char s[100];
retval = pipe(fds);
pid = fork();
if(pid == 0){
    read(fds[0], s, 100);
    printf("Read %s\n", s);
}
else
    write(fds[1], "hello", 6);
```

NOTES:
- data is written/read in order (FIFO)
- reads block until there’s something to read
- writes block if the pipe is full
- closing the writing fd causes EOF to be read on the other end