CMPSC 311- Spring 2016
Prof. Bhuvan Urgaonkar
Scribe Notes: Lecture 15- February 17, 2016

Andrew Ren, Shaurya Rastogi, Jesse Rong, Yahya Saad

Previously we learned some ways in which the Operating System creates and manages processes. We have learned that, using C, we have the power to instruct the Operating System to create and end processes. The tools and concepts discussed below serve to further our ability to interface, intimately, with the Operating System

Overview

- Process API (Exit(), Wait()) and potential O.S. generated interrupts (Seg Fault)
- Process Classes - Orphan, Zombie, Daemon
- Signal Generation
- Signal Handling

Normal Process Termination

Normal Process Termination occurs in typical settings in programs, i.e. when a Main function "falls of the end" of its process, or when a function returns a value. Whenever Normal Process Termination occurs, the `exit()` is called, which is defined as:

```
    void exit(int status);
```

The status parameter is set to 0 if `exit` is called successfully. In addition, `exit` can have user-defined handlers, whereas `_Exit` and `_exit` cannot.

When `main()` returns, it returns to some C runtime functions, and leads to some system call such as `exit()`. This is an example of Normal Process Termination because the function successfully terminates itself and removes itself from memory. There are also different ways that a function does not remove itself from memory, which are called *Abnormal Process Terminations*.
In the example below, the `atexit()` function registers the termination function. When the program is terminated normally (from `exit()` or a return from main), the functions registered are called in reverse order from how they were registered.

```
#include <stdio.h>
#include <stdlib.h>

void functionA ()
{
   printf("This is functionA\n");
}

void functionB ()
{
   printf("This is functionB\n");
}

int main ()
{
   /* register the termination function */
   atexit(functionA);

   atexit(functionB);
   // executed in reverse order, defined by the Unix system
   // functionB is executed first, but registered second
   printf("Starting  main program...\n");

   printf("Exiting main program...\n");

   //_Exit(0);
   //exit(0);
   return(0);
}
```

Abnormal Process Terminations

Abnormal Process Termination occurs when the normal process termination is unable to be executed. Inputs like a Ctrl-C or a Segmentation Fault** will cause an Abnormal Process Termination because the process will then be unable to run through the typical rundown sequence. This has a negative effect on the memory because this termination will allow processes to continue running even after a program is terminated because the OS does not call exit safety handlers. This will eventually cause a core-dump, in which the current memory is saved for future debugging.

** Segmentation Faults are caused by a program trying to read or write an illegal memory location.

Where does main() return?

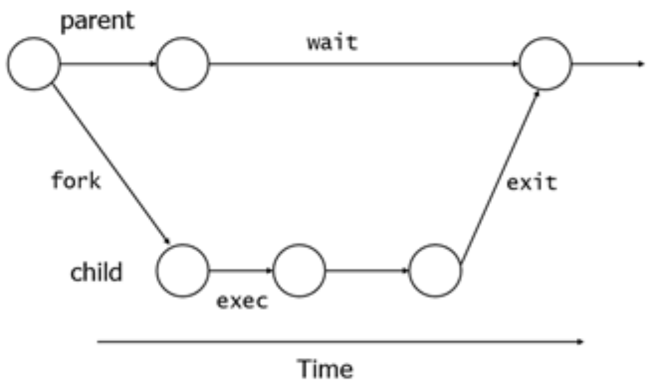
Main returns a value (integer) to indicate successful/unsuccessful program termination to a system call such as `exit`.

Waiting for a Child: wait

The `wait` function causes the caller to suspend execution until the child has exited. For example, when "`ls -l`" is executed on shell, the shell is the parent and "`ls`" is the child. While the child is still executing, the Shell waits until the child exits before accepting new commands. `wait` is defined as:

```
pid_t wait (int *status);
```

This function returns the process id of the child. `status` refers to the child's exit status. If it's 0, the child had successfully exited. If it's another value the child did not.



In this diagram, you can see that after the child is created, the parent waits until the child has exited before continuing.

Waiting for a Specific Child: waitpid

A process may have multiple children, but wait only waits for the quickest child to return and none of the others. So if a parent wants to wait for a specific child, use:

```
     pid_t waitpid(pit_t pid, int *status, int opt);
```

Here, `pid` is the process id of specific child and `status` is the exit status of child. The `opt` parameter gives the option of either waiting or not waiting for the specified child. If the parameter `WNOHANG` is passed in, `waitpid()` will not wait for the child to finish, and if `0` is passed in, `waitpid()` will wait for the child to finish running.

Example Usage:

```
cpid = fork(); // forks a new child process
case (cpid) // determines the status of the cpid
            // case structure will confirm child is forked
case 0: // run some program here

// There is an option here to wait for the child
// if still needs to wait for child
// this will block the program until the child finishes
waitpid(cpid, status, 0);

// does not need to wait for child
// this will not block the program and disregards the child
waitpid(cpid, status, WNOHANG);
```

check example 3.15 in book

```
pid_t r_wait(int *stat_loc)
// restarts wait(0 if interrupted by a signal
```

`fork()` creates a child process
if success, childpid will be returned

```
while(r_wait(NULL) > 0) // this waits for all children
// r_wait() returns the number of children still running
```

Uses for Waiting

Using `wait` will allow synchronization of multiple dependent processes. It allows any parent process to synchronize its children's executions, and has applications in interactive programs. For instance, the shell or terminal running on an operating system utilizes `wait` to wait for user inputs. This is a good example of waiting because the shell has no purpose when there are not inputs from the user, so there is no point in running anything until the user inputs a command. Then, when a user inputs a command, such as `ls -l`, the shell then waits for that child process to finish before accepting the next command. Without synchronization, it is possible for one thread to modify a shared object while another thread is in the process of using or updating that object's value. This often leads to significant errors.

Another important consequence of waiting is "reaping". The OS will only remove a process from memory when a parent waits for it, and if the OS does not remove the process completely, the child may become a zombie process. Although a zombie process is a terminated process, it is undesirable because it is using up memory space in the stack that will not be able to be accessed or used until the memory is dumped. When the parent process waits for its child process, it will successfully remove the child process from memory, which will avoid the possibility of creating a zombie processes.

Process Groups

The default process hierarchy is where a process has one parent and one child. However, sometimes it may be necessary for a process to be the parent of a group of processes. This implementation can also utilize waiting concepts to avoid memory waste. This can be realized in code as such:

```
    waitpid(-group_pid, NULL, 0);
```

Here, the negative sign in front of `group_pid` denotes that negative process ID's indicate group process ID's. In a group, the group process ID is that of the leader of the group.

Orphans and Zombies

Previously, zombie processes were mentioned when a parent did not wait for its child process to finish. There are actually different varieties of wasted memory in processes:

- *Orphan Processes*: Occur when the parent of a process dies before the process can finish
- *Zombie Processes*: Occur when a process finishes when the parent did not wait for it

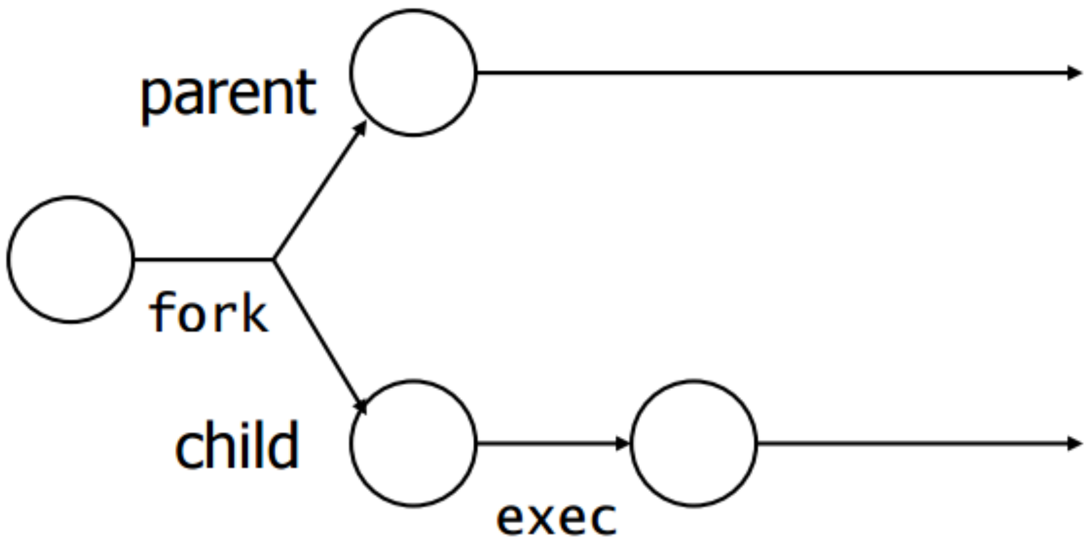
These processes should be avoided because even though they may be done executing, they are still stored in the main memory and are thus wasted memory.

In order to avoid orphan and zombie processes, some steps can be taken. For zombie processes, the parent should always wait for the child process to finish before it executes again, because the system does not have any way of removing zombie processes other than when its parents wait.

A similar approach can be used to limit orphan processes; however, the system also has a means of removing orphan processes. Orphan processes are adopted by the `init` process, which waits from time to time. When `init` waits, it gives any orphan process the opportunity to be reaped from the memory, thus cleaning their space in the main memory.

Background Processes

When the shell creates a background process, it does not wait for it and can accept other commands. A line ending with `&` should be executed as a background process (E.G. "`mozilla &`")



Once the child is forked from the parent, it is executed independent of the parent.

Daemons

A daemon is a background process that runs forever. It's similar to shell in that it gets some input and it can do useful tasks. It's different in that each implements a specific service and may not be interactive like a typical process is. Examples include: web server (httpd) and print server (lpd).

Signal Introduction

A signal is a software interrupt generated by the OS or a process. For example it can happen when a program divides by zero, or when it leaks memory, etc. It can also be generated by a user defined thread.

Signal Names and Types

Signal names start with SIG, like: `SIGKILL` and `SIGALRM`. The names generally correspond to the type of event occurring.

- `SIGCHILD`: child terminated - when you `fork()` a child, `wait()` function catches this signal
- `SIGSEGV`: segmentation fault
- `SIGKILL`: kill signal
- `SIGPIPE`: writing to a pipe with no readers
- `SIGINT`: `^C` (The interrupt key input)

Signal Generation and Delivery

A signal is delivered when a process takes action. The process must be running to get a signal delivered. And the signal is considered "pending" between its generation and delivery. Also, it is important to note that only one instance of a signal can be pending. When there are multiple undelivered signals, they are combined.

Generating Signals: `kill`

The `kill` function is called in a program to send a signal to a process. The sender needs appropriate permissions to send the signal.

```
int kill(pid_t pid, int sig);
```
kill command: `kill -s signal_name pid`

For parameters, it takes process id and signal number. If the process id is:

- greater than 0, the signal is sent to the process with that id
- equal to 0, the signal is sent to the members of the caller's process group
- equal to -1, the signal is sent to all processes for which it has permission to send
- another negative value, the signal is sent to the process group with group ID equal to pid

Handling Signals

A process can take multiple actions once it a signal is sent to it.

One option is to simply ignore the signal that is sent to it. This works for some signals, but not `SIGKILL` and `SIGSTOP`.

Another option is to block the signal. This is a temporary delay to the signal, and the signals get sent to a queue that get executed after blocking stops. Similar to ignore, the `SIGKILL` and `SIGSTOP` signals cannot be blocked.

The third option is to catch the signal. A process can set up a function to be called when the signal is delivered, which then processes that signal.

Default Action

The default action of how a signal is handled depends on signal type. Most signals result in process termination. While some, like SIGCHLD are ignored by default.

Blocking Signals

If you are in the middle of a critical section, you may not want an unexpected signal to interrupt the process. Signals can be blocked so they are not delivered and remain in pending. Once they are unblocked, they will be delivered to the process.

A signal mask is a set of signals that are currently blocked by process and it's manipulated using signal set (type `sigset_t`).

Operations include:

- `sigaddset` add a signal to set
- `sigdelset` remove a signal from set
- `sigismember` check if a signal is in set
- `sigemptyset` clear all signals in set
- `sigfillset` set all signals in set

Signal masks can be manipulated with: `int sigprocmask(int how, sigset_t *set, sigset_t *old_set);`
The parameters include:

1. `how`: how the signal mask would be modified
   - `SIG_BLOCK` Add set of signals to block
   - `SIG_UNBLOCK` Unblock a set of signals
   - `SIG_SETMASK` Set current signal mask to signal set
2. `set`: Signal set to be used for manipulation
3. `old_set`: Old signal mask before modification

Signal Blocking Example

This example uses `SIGINT` to block Ctrl-C Signal and it delays ^C signal until it is unblocked

```
sigset_t newsigset, oldsigset;

/* Add SIGINT to the new signal set */
sigemptyset(&newsigset);
sigaddset(&newsigset, SIGINT);

/* Add SIGINT to the set of blocked signals */
sigprocmask(SIG_BLOCK, &newsigset, &oldsigset);

/* Check if SIGQUIT is in the old signal mask */
if (sigismember(&oldsigset, SIGQUIT))
```

```
  printf(“SIGQUIT already blocked\n”);
```

Sources used:
http://pubs.opengroup.org/onlinepubs/009695399/functions/atexit.html