Section 2

Monday, July 1, 2019 2:05 PM

Section 3: Process Wrap-up and Syscalls

July 1, 2019

Contents

1 Reference **2**
- 1.1 Process Vocabulary (continuation from last week's worksheet) 2
- 1.2 Syscall Vocabulary 2
- 1.3 Signals 3

2 Process Problems **3**
- 2.1 Simple Wait 3
- 2.2 Exec 4
- 2.3 Exec + Fork 5
- 2.4 Implementing fork() efficiently (Design) 5

3 Signal Problems **5**
- 3.1 Using Your Keyboard 5
- 3.2 Signal in Action 6
- 3.3 Sigaction in Action 6
- 3.4 More Sigaction 6

1 Reference

fork() ~ spawns new process

1.1 Process Vocabulary (continuation from last week's worksheet)

- **wait** - A class of C functions that call syscalls, which are used to wait for state changes in a child of the calling process and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

exit(code)

- **exit code** - The exit status or return code of a process is a 1 byte number passed from a child process (or callee) to a parent process (or caller) when it has finished executing a specific procedure or delegated task
- **exec** - The exec() family of functions replaces the current process image with a new process image. The initial argument for these functions is the name of a file that is to be executed.

1.2 Syscall Vocabulary

- **system call** - In computing, a system call is how a program requests a service from an operating system's kernel. This may include hardware-related services, creation and execution of new processes, and communication with integral kernel services such as process scheduling.
- **Signals** - A signal is a software interrupt, a way to communicate information to a process about the state of other processes, the operating system, and the hardware. A signal is an interrupt in the sense that it can change the flow of the program when a signal is delivered to a process, the process will stop what its doing, either handle or ignore the signal, or in some cases terminate, depending on the signal.

signal(SIGINT, handleInt);

- **int signal(int signum, void (*handler)(int))** - signal() is a system call for signal handling, which given a signal and function, will execute the function whenever the signal is delivered. This function is called the signal handler because it handles the signal. Signal is deprecated and sigaction should be used instead; however, signal is useful tool in understanding how these syscalls work.
- **int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)** - sigaction is a system call used to change the action taken by a process on receipt of a specific signal. If act is non-NULL, the new action for signal signum is installed from act. If oldact is non-NULL, the previous action is saved in oldact. Prior to making the sigaction call, the user must create a sigaction struct and populate its fields appropriately.

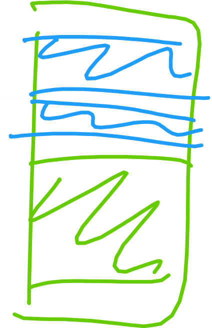
```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
}
```

- **SIG_IGN, SIG_DFL** Usually the sa_handler takes a user defined handler for the signal. However, if you'd like your process to drop the signal you can use SIG_IGN. If you'd like your process to do the default behavior for the signal use SIG_DFL.

1.3 Signals

The following is a list of standard Linux signals:

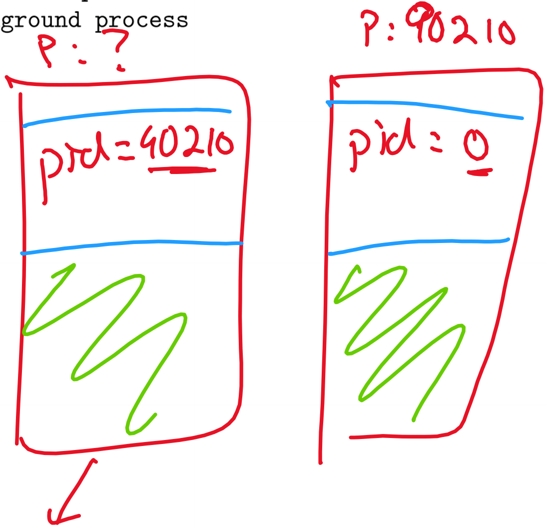
Signal	Value	Action	Comment
SIGHUP	1	Terminate	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Terminate	Interrupt from keyboard (Ctrl - c)
SIGQUIT	3	Core Dump	Quit from keyboard (Ctrl - \)
SIGILL	4	Core Dump	Illegal Instruction
SIGABRT	6	Core Dump	Abort signal from abort(3)
SIGFPE	8	Core Dump	Floating point exception
SIGKILL	9	Terminate	Kill signal
SIGSEGV	11	Core Dump	Invalid memory reference
SIGPIPE	13	Terminate	Broken pipe: write to pipe with no readers
SIGALRM	14	Terminate	Timer signal from alarm(2)
SIGTERM	15	Terminate	Termination signal
SIGUSR1	30,10,16	Terminate	User-defined signal 1
SIGUSR2	31,12,17	Terminate	User-defined signal 2
SIGCHLD	20,17,18	Ignore	Child stopped or terminated
SIGCONT	19,18,25	Continue	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process



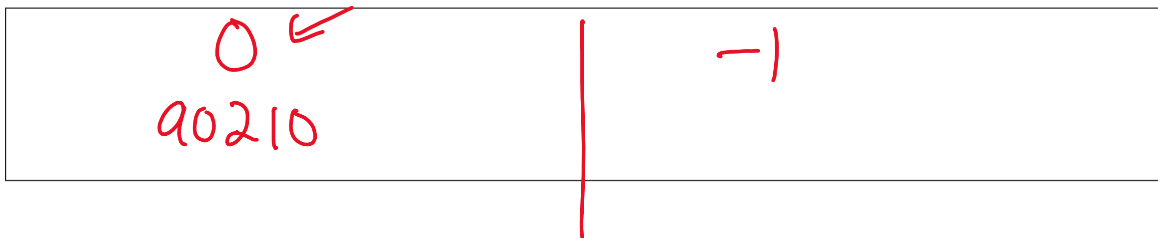
2 Process Problems

2.1 Simple Wait

What can C print? Assume the child PID is 90210.




```
int main(void)
{
pid_t pid = fork();
int exit;
if (pid != 0) {
wait(&exit);
}
printf("Hello World\n: %d\n", pid);
}
```



What is the equivalent program using the `waitpid` function instead of `wait`?

only 90210

wait(&exit) waitpid(90210, &exit, 0)

waitpid(-1, ..)

any child

wait(...) == waitpid(-1, ..)

2.2 Exec

What will C print?

```
int main(void)
{
  char** argv = (char**) malloc(3*sizeof(char*));
  argv[0] = "/bin/ls";
  argv[1] = ".";
  argv[2] = NULL;
  for (int i = 0; i < 10; i++) {
  printf("%d\n", i);
  if (i == 3)
  execv("/bin/ls", argv);}
}
```

for ( ) {
printf ( );
if (i==3){
} execv (...);

3

0
1
2
3
<whatever ls prints out>

2.3 Exec + Fork

How would I modify the above program using fork so it both prints the output of `ls` and all the numbers from 0 to 9 (order does not matter)? You may not remove or reorder lines from the original program; only add statements (and use fork!).

fork
wait

```
if (i==3) {
  pid_t pid = fork();
  if (pid==0) {
    execv( ... );
  }
}
```

0
1
2
3
<ls>
4
5
6
7
:

2.4 Implementing fork() efficiently (Design)

Remember fork() makes the child process's address space exactly the same as its parent's. If you were designing an OS, list some steps you would take to make this address space copy more efficient ?

Copy-on-write

3 Signal Problems

3.1 Using Your Keyboard

kill -2 pid

How do we stop the following program?

```
int main(){
  signal(SIGINT, SIG_IGN);
  while(1);
}
```

SIGQUIT (Ctrl - \ )

3.2 Signal in Action

Fill in the blanks for the following function using syscalls such that when we type Ctrl-C, the user is prompted with a message: "Do you really want to quit [y/n]? ", and if "y" is typed, the program quits. Otherwise, it continues along.

```
void sigint_handler(int sig)
{
    char c;
    printf(Ouch, you just hit Ctrl-C?. Do you really want to quit [y/n]?);
    c = getchar();
    if (c == 'y' || c == 'Y')
        ______________________________;
}

int main() {
    ______________________________;
    ...

}
```

3.3 Sigaction in Action

How would you change the main function to use sigaction instead of signal?

```
int main() {
______________________________;
______________________________;
______________________________;
______________________________;
...

}
```

3.4 More Sigaction

Lets say you wanted to move the signal handler from SIGINT to SIGQUIT. How would you do that without manually constructing another sigaction struct?

```
int main() {
______________________________;
______________________________;
______________________________;
______________________________;
...

}
```