Shell - Lab

A shell is an interactive command-line interpreter that runs programs on behalf of the user. A shell repeatedly prints a prompt, waits for a command line on stdin, and then carries out some action, as directed by the contents of the command line.

The command line is a sequence of ASCII text words delimited by whitespace. The first word in the command line is either the name of a built-in command or the pathname of an executable file. The remaining words are command-line arguments. If the first word is a built-in command, the shell immediately executes the command in the current process. Otherwise, the word is assumed to be the pathname of an executable program. In this case, the shell forks a child process, then loads and runs the program in the context of the child. The child processes created as a result of interpreting a single command line are known collectively as a job. In general, a job can consist of multiple child processes connected by Unix pipes.

If the command line ends with an ampersand ”&”, then the job runs in the background, which means that the shell does not wait for the job to terminate before printing the prompt and awaiting the next command line. Otherwise, the job runs in the foreground, which means that the shell waits for the job to terminate before awaiting the next command line. Thus, at any point in time, at most one job can be running in the foreground. However, an arbitrary number of jobs can run in the background.

Your tsh shell should have the following features:

• The prompt should be the string “tsh> ”.

• The command line typed by the user should consist of a name and zero or more arguments, all separated by one or more spaces. If name is a built-in command, then tsh should handle it immediately and wait for the next command line. Otherwise, tsh should assume that name is the path of an executable file, which it loads and runs in the context of an initial child process (In this context, the term job refers to this initial child process).
• tsh need not support pipes (|) or I/O redirection (< and >).
• Typing ctrl-c (ctrl-z) should cause a SIGINT (SIGTSTP) signal to be sent to the current foreground job, as well as any descendents of that job (e.g., any child processes that it forked). If there is no foreground job, then the signal should have no effect.
• If the command line ends with an ampersand &, then tsh should run the job in the background. Otherwise, it should run the job in the foreground.
• Each job can be identified by either a process ID (PID) or a job ID (JID), which is a positive integer assigned by tsh. Job ID’s are used because some scripts need to manipulate certain jobs, and the process ID’s change across runs. JIDs should be denoted on the command line by the prefix ’%’. For example, “%5” denotes JID 5, and “5” denotes PID 5. (We have provided you with all of the routines you need for manipulating the job list.)

• tsh should support the following built-in commands:
– The quit command terminates the shell.
– The jobs command lists all background jobs.
– The bg command restarts by sending it a SIGCONT signal, and then runs it in the background. The argument can be either a PID or a JID.
– The fg command restarts by sending it a SIGCONT signal, and then runs it in the foreground. The argument can be either a PID or a JID.
• tsh should reap all of its zombie children. If any job terminates because it receives a signal that it didn't catch, then tsh should recognize this event and print a message with the job's PID and a description of the offending signal.

Chapter 8 notes:

8.4

Getpid function returns the PID of the calling process
Getppid returns the PID of its parent

A process is in **3 states**:

Running

Stopped- A process stops as a result of receiving a SIGSTOP, SIGTSTP,SIGTTIN, OR SIGTTOU signals and it remains stopped until it receives a SIGCONT signal.

Terminated: a process becomes terminated because= 1) receiving a signal whose default action is to terminate the process 2) returning from the main routine 3)calling the exit function(void exit (int status);)

A parent process creates a new running child process by calling the fork function

The fork function is called once but it returns twice. In the parent, fork returns the PID of the child. In the child, fork returns a value of 0. Since the PID of the child is always nonzero, the return value provides an unambiguous way to tell whether the program is executing in the parent or the child.

The parent and the child are separate processes that run concurrently.

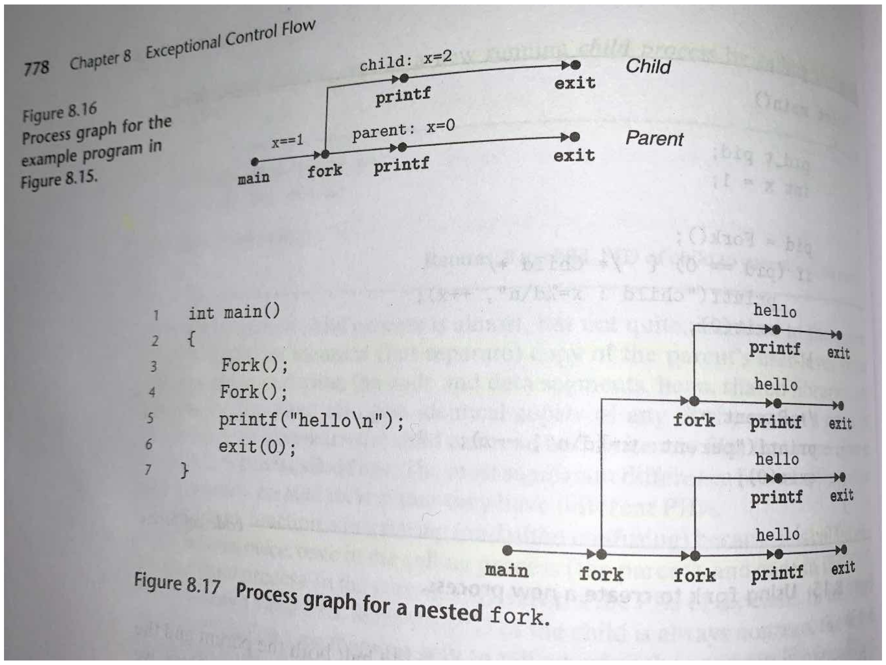
778 Chapter 8 Exceptional Control Flow

Figure 8.16 Process graph for the example program in Figure 8.15.

```
int main()
{
    Fork();
    Fork();
    printf("hello\n");
    exit(0);
}
```

Figure 8.17 Process graph for a nested fork.

^-A process graph captures the partial ordering of program statements. Each vertex A corresponds to the execution of a program statements. A directed edge a->b denotes that statement a "happens before" statement b. Edges can be labeled with info such as the current value of a variable. Vertices corresponding to printf statements can be labeled with the output of the printf. Each graph begins with a vertex that corresponds to the parent process calling main. This vertex has no inedges and exactly one outedge. The sequence of vertices for each process ends with a vertex corresponding to a call to exit.(the printf statements in the parent and child can occur in either order because each of the orderings corresponds to some topological sort of the graph vertices.

-When a process terminates for any reason, the kernel does not remove it from the system immediately. The process is kept around in a terminated state until it is reaped by its parent. When the parent reaps the terminated child, the kernel passes the child's exit status to the parent and then discards the terminated process, at which point it ceases to exist. A process that has not yet been reaped is called a **Zombie**

When a parent process terminates , the kernel arranges for the init process to become the adopted parent of any orphaned children.(Init process has a PID of 1 and it's the ancestor of every process). If parent terminates and doesn't reap their children then the init process reap them. Even tho zombies are not running, they still consume system memory resources and ao shell or servers should always reap their zombie children.

A process waits for its children to terminate or stop by calling the waitpid function.
When options =0 (look at function code) waitpid suspends execution of the calling process until a child process in its wait set terminates. Waitpid returns the PID of the terminated child.

The members of the wait set are determined by the pid argument:

If pid>0 then the wait set is the singleton child process whose process ID is equal to pid
If pid =-1 then the wait set consists of all of the parent's child processes.

Modifying the default behavior: Various combo of WNOHANG, WUNTRACED, & WCONTINUED constants.(page 780)

Checking the exit status of a reaped child: WIFEXITED,WEXITSTATUS,WIFSIGNALED,WTERMSIG,WIFSTOPPED,WSTOPSIG,WIFCONTINUED.

If the calling process has no children, then waitpid returns -1 and sets errno to ECHILD. If the waitpid function was interrupted by a signal, then it returns -1 and sets errno to EINTR.

The wait function is a simpler version of waitpid.

The sleep function suspends a process for a specified period of time.
The pause function which puts the calling function to sleep until a signal is received by the process.
The execve function loads and runs a new program in the context of the current process.
Execve is called once and never return

Int main(int argc, char *argv[], char *envp[]);
1- argc, which gives the number of non-null pointers in the argv[] array,
2- argv, which points to the first entry in the argv[] array
3- envp, which points to the first entry in the envp[] array.

The getenv function searches the environment array for a string name=value. If found, it returns a pointer to value; otherwise it returns NULL

Command line/shell:
Its first task is to call the parseline function, which parses the space-separated command-line arguments and builds the argv vector that will eventually be passed to execve. If the last argument is an & character, then parseline returns 1, indicating that the program should be executed in the background(the shell does not wait for it to complete). Otherwise, it returns 0, indicating that the program should be in the foreground(the shell waits for it to complete).

Notes 8.5:
Signals
A signals is a small message that notifies a process that an event of some type has occurred in the system.Signals provide a mechanism for exposing the occurrence of such exception to user processes.

If a process divide by zero, then the kernel sends it a SIGFPE
If a process executes an illegal instruction, then the kernel sends it a SIGILL

If a process makes an illegal memory reference, the kernel sends it a SIGSEGV
If you type ctrl+c while a process is running in the foreground, then the kernel sends a SIGINT to each process in the foreground process group.
The process can either ignore the signal,terminate, or catch the signal by executing a user-level function called a signal handler.

Every process belongs to exactly one process group, which is identified by a positive integer process group ID. the getpgrp function returns the process group id of the current process.
A process can change the process group of itself or another process by using the setpgid function
A negative PID causes the signal to be sent to every process in process group PID

If pid is greater than zero, then the kill function sends signal number sig to process pid. If pid is equal to zero, then kill sends signal sig to every process in the process group of the calling process, including the calling process itself. If pid is less than zero, then kill sends signal sig to every process in process group pid.

Sending signals with the alarm function - the alarm func arranges for the kernel to send a SIGALARM signal to the calling process in secs seconds.

The default action for the receipt of a SiGKILL is to terminate the receiving process. The default action for the receipt of a SIGCHILD is to ignore the signal. A process can modify the default action associated with a signal by using the signal func. The only exception are SIGSTOP and SIGKILL,whose default actions cannot be changed.

Blocking and unblocking signals

Implicit blocking mechanism- BY default, the kernel blocks any pending signals of the type currently being processed by a handler.

Explicit blocking mechanism-Applications can explicitly block and unblock selected signals using the sigprocmask function and its helpers.

The sigprocmask func changed the set of currently blocked signals
SIG_BLOCK - Add the signals in set to blocked
SIG_UNBLOCK - remove the signals in set from blocked
SIG_SETMASK blocked=set
If oldset is non-nULL, the previous value of the blocked bit vector is stored in oldset
Sigemptyset initializes set to the empty set
Sigfillset function adds every signal to set.
Sigaddset function adds signum to set, sigdelset deletes signum form set, and sigismember returns 1, if signum is a member of set, and 0 if not.

Safe signal handling guidelines:
G0 Keep handlers as simple as possible
G1 Call only async-signal-safe functions in your handlers
G2 Save and restore errno
G3 Protect accesses to shared global data structures by blocking all signals
G4 Declare global variables with volatile
G5 Declare flags with sig_Atomic_t

C provides an integer data type, sig_atomics_t for which reads and writes are guaranteed to be atomic(uninterruptible) because they can be implemented with a single instruction.

Default Signal Handlers

There are several default signal handler routines. Each signal is associated with one of these default handler routine. The different default handler routines typically have one of the following actions:

- Ign: Ignore the signal; i.e., do nothing, just return
- Term: terminate the process
- Cont: unblock a stopped process
- Stop: block the process

fork() Create a duplicate, a “child”, of the process
execve() Replace the running program
exit() End the running program
waitpid() Wait for a child process to terminate