Programming Assignment #1: Process Management

Due: Check My Courses

Description of the Assignment

Lets start with a tiny shell that has minimal functionality. The tiny shell works like the following. We read a line of input, if the input is valid (longer than a single newline character), we run the command. Otherwise, the tiny shell quits.

```
while (1) {
      line = get_a_line();
      if length(line) > 1
            system(line);
      else
            exit(0);
}
```

Note that in the above pseudo code we use a Linux library function called **`system()`** and functions supplied by you such as **`get_a_line()`** and **`length()`**, which are expected to read a line from the terminal and compute the length of a string, respectively. The **`system()`** function is responsible for creating a process and running the command in that process. You get the tiny shell working with this library function. Once it is working, run some commands that successfully run and run others that do not work. That is, try commands that crash or prematurely terminate as well as commands that complete their execution. What happens to your shell? Does your shell crash when a command crash?

Your tiny shell is an interactive program. It reads from the terminal. Such a program can be run non-interactively by redirecting the required input from an input file. For instance, suppose the tiny shell is named **`tshell`** and we have an input file name **`input.txt`**. The **`input.txt`** file has all the commands you want to run in the tiny shell. Then, issue the following command.

```
tshell < input.txt
```

You should be able to run the shell and it should terminate assuming your tiny shell is detecting the end-of-file condition (input lines with just the new line character) and terminating itself.

You can use **`ltrace`** or **`strace`** to trace a program and detect all the library routines or system calls that are used by the program. For instance, the following command should give the library routines that are called by the tiny shell.

```
ltrace tshell < input.txt
```

Similarly, you can find the system calls used by the tiny shell using **`strace`**.

Now, can you identify how the library routine **`system()`** could be implemented? That is, determine the system calls that are used in implementing the **`system()`** library function.

In the next phase of the assignment, replace the **`system()`** library function with your own implementations. You need to implement **`my_system()`** using **`fork()`**, **`vfork()`**, and **`clone()`** system

calls. That is, implement version F using **fork()**, version V using **vfork()**, and version C using **clone()**.
To create the different implementations of **my_system()**, you need to understand what that function needs to do. **my_system()** is a function that spawns (i.e., creates) a child process and runs the command you passed as the argument to the call, assuming the command you requested to run is a valid one and is present in the machine. For example, if you specify '**/bin/ls**' it is going to execute the directory lister command. You could program so that absolute path names of the commands are not necessary. That is you can just specify '**ls**' and it should do the same run as the previous one. It is important that your implementation makes the tiny shell fault tolerant. That is, if the executable crashes, the process (i.e., the tiny shell) that is calling the **my_system()** function **does not crash** as well.

An implementation that uses **fork()** reflects the semantics of the **fork()** system call or library function (note that fork() *is available as both*). Because you cannot do much to change the behavior of **fork()**, the **my_system()** implementation will exhibit a behavior that is dictated by the **fork()** operation.

```
#include <unistd.h>

pid_t fork(void);
```

In parent: returns process ID of child on success, or −1 on error;
in successfully created child: always returns 0

With **fork()**, after a successful call, we have two processes and both continue the execution from the point where **fork()** returns in the program.

The implementation of **my_system()** function is completely up to you. However, you need to ensure that **my_system()** completes an ongoing execution before it launches the next command. That is, it does not put the command in the background. In order, to ensure that **my_system()** waits for an ongoing execution, you need to use the **wait()** system call. Here is a brief description of the call. You can always get more information by consulting the man page.

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
                                    Returns process ID of child, 0 (see text), or –1 on error
```

The return value and *status* arguments of *waitpid()* are the same as for *wait()*. (See Section 26.1.3 for an explanation of the value returned in *status*.) The *pid* argument enables the selection of the child to be waited for, as follows:

- If *pid* is greater than 0, wait for the child whose *process ID* equals *pid*.
- If *pid* equals 0, wait for any child in the *same process group as the caller* (parent). We describe process groups in Section 34.2.
- If *pid* is less than –1, wait for any child whose *process group* identifier equals the absolute value of *pid*.
- If *pid* equals –1, wait for *any* child. The call *wait(&status)* is equivalent to the call *waitpid(–1, &status, 0)*.

Once you have the **my_system()** implementation, run the tiny shell version with your **my_system()** again and see whether it behaves the same way as the implementation that used the **system()** from Linux. If there are differences, you need to explain those differences and why they are so.

Next, you create another version of **my_system()**. This time, you use only **vfork()** in implementing **my_system()**. The **vfork()** system call was created to make the process creation more efficient. It was realized that copying all the memory pages while creating a clone is a waste of work. So, **vfork()** shares the memory between the parent and child until the child loads a new program via **exec()** or one of its variants. **The problem we could run into is that the changes made in the child before it loads a new program can get reflected back in the parent. For instance, if the child goes ahead and changes a variable, it could get reflected in the parent.**

```
#include <unistd.h>

pid_t vfork(void);
                    In parent: returns process ID of child on success, or –1 on error;
                                in successfully created child: always returns 0
```

Two things are important to note with regarding to **vfork()**:

- It does not copy the memory while cloning like **fork()** does. So the child does not have its own memory until it does **exec()**.
- The parent is stopped until the child does **exec()**.

With the **vfork()** semantics, the **child is guaranteed to execute before the parent** after the **vfork()**. At **vfork()** execution, the child gets its own file descriptors.

Implement the **my_system()** using **vfork()** and measure the execution time for starting certain number of commands. Compare the time taken for executing the same commands with the previous

implementations. ***Report this timing in the report you handover with the code***. At the end of this handout we describe how you could measure the execution time in Linux.

Next, we want to implement **`my_system()`** using the **`clone()`** system call. The **`clone()`** is a Linux specific lower-level system call for creating processes. You pass the appropriate flags to get different behaviors from the **`clone()`** system call.

The syntax of the **`clone()`** system call is shown below.

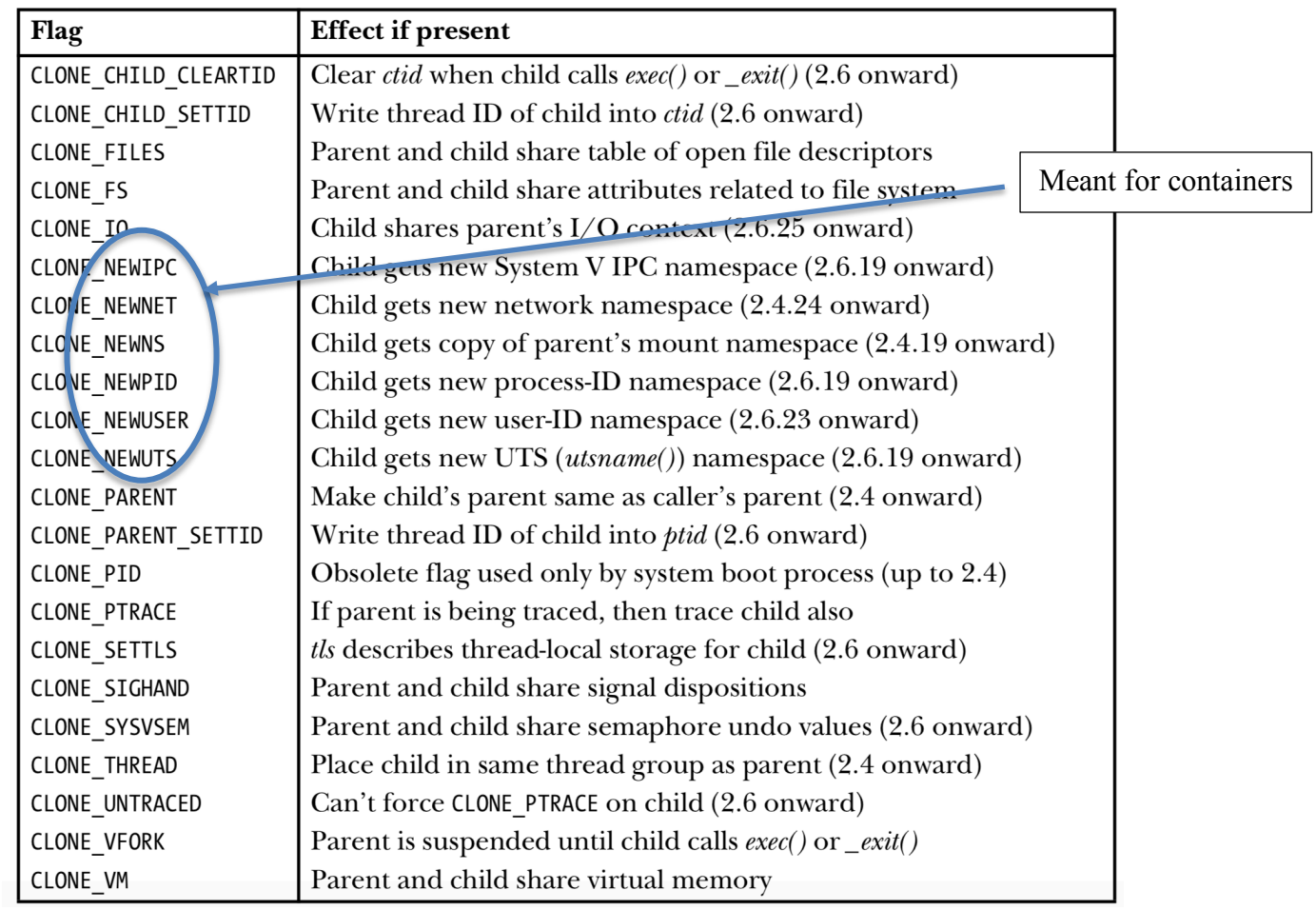
```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*func) (void *), void *child_stack, int flags, void *func_arg, ...
          /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

Returns process ID of child on success, or −1 on error

Like the **`fork()`** call, the clone() makes another replica of the calling process. However, the sharing between the caller (parent) and child can be precisely controlled by the flags. Also, the child runs the func specified in the call and the parameters can be passed to this function. We need to specify a stack for the child as well. The various flags we could use in the **`clone()`** call and their descriptions are shown in the table below.

Flag	Effect if present
CLONE_CHILD_CLEARTID	Clear *ctid* when child calls *exec()* or *_exit()* (2.6 onward)
CLONE_CHILD_SETTID	Write thread ID of child into *ctid* (2.6 onward)
CLONE_FILES	Parent and child share table of open file descriptors
CLONE_FS	Parent and child share attributes related to file system
CLONE_IO	Child shares parent's I/O context (2.6.25 onward)
CLONE_NEWIPC	Child gets new System V IPC namespace (2.6.19 onward)
CLONE_NEWNET	Child gets new network namespace (2.4.24 onward)
CLONE_NEWNS	Child gets copy of parent's mount namespace (2.4.19 onward)
CLONE_NEWPID	Child gets new process-ID namespace (2.6.19 onward)
CLONE_NEWUSER	Child gets new user-ID namespace (2.6.23 onward)
CLONE_NEWUTS	Child gets new UTS (*utsname()*) namespace (2.6.19 onward)
CLONE_PARENT	Make child's parent same as caller's parent (2.4 onward)
CLONE_PARENT_SETTID	Write thread ID of child into *ptid* (2.6 onward)
CLONE_PID	Obsolete flag used only by system boot process (up to 2.4)
CLONE_PTRACE	If parent is being traced, then trace child also
CLONE_SETTLS	*tls* describes thread-local storage for child (2.6 onward)
CLONE_SIGHAND	Parent and child share signal dispositions
CLONE_SYSVSEM	Parent and child share semaphore undo values (2.6 onward)
CLONE_THREAD	Place child in same thread group as parent (2.4 onward)
CLONE_UNTRACED	Can't force CLONE_PTRACE on child (2.6 onward)
CLONE_VFORK	Parent is suspended until child calls *exec()* or *_exit()*
CLONE_VM	Parent and child share virtual memory



Below we explain the operation of some of the important flags in the **`clone()`** system call. You can consult the man page for a complete description.

If CLONE_FILES is set, the calling process and the child process share the same file descriptor table. Any file descriptor created by the calling process or by the child process is also valid in the other process. Similarly, if one of the processes closes a file descriptor, or changes its associated flags the other process is also affected. If CLONE_FILES is not set, the child process inherits a copy of all file descriptors opened in the calling process at the time of **`clone()`**. Subsequent operations that open or close file descriptors, or change file descriptor flags, performed by either the calling process or the child process do not affect the other process.

If CLONE_FS is set, the caller and the child process share the same filesystem information. This includes the root of the filesystem, the current working directory, and the umask. If CLONE_FS is not set, the child process works on a copy of the filesystem information of the calling process at the time of the **`clone()`** call.

If CLONE_VM is set, the calling process and the child process run in the same memory space. In particular, memory writes performed by the calling process or by the child process are also visible in the other process. If CLONE_VM is not set, the child process runs in a separate copy of the memory space of the calling process at the time of **`clone()`**.

If CLONE_VFORK is set, the execution of the calling process is suspended until the child releases its virtual memory resources via a call to **execve(2)** or **_exit(2)** (as with **vfork()**). If CLONE_VFORK is not set, then both the calling process and the child are schedulable after the call, and an application should not rely on execution occurring in any particular order.

You are strongly encouraged to read the man page for a full description of all the flags. One interesting aspect is that **clone()** is much more general than fork() or **vfork()**. It can be used to create containers – pseudo virtual machines. See the *NEW* flags in the above list.

Here are some code snippets for using the **clone()** system call. First you need to allocate a stack.

```
stack = malloc(STACK_SIZE);
if (stack == NULL)
    errExit("malloc");
stackTop = stack + STACK_SIZE;      /* Assume stack grows downward */
```

Notice the need to get stackTop because the stack grows downwards.

```
static int                      /* Startup function for cloned child */
childFunc(void *arg)
{
    if (close(*((int *) arg)) == -1)
        errExit("close");

    return 0;                           /* Child terminates now */
}
```

The clone command uses the childFunc as the starting function. The child terminates when the childFunc terminates.

```
if (clone(childFunc, stackTop, flags | CHILD_SIG, (void *) &fd) == -1)
    errExit("clone");
```

Before the above **clone()** call is executed, we need to setup the flags. The **fd** parameter is part of the example. In this example, a file is opened by the parent and the child (i.e., the childFunc) is closing it.

The above code snippet is just an example that illustrates how the **clone()** could be used.

You can have the childFunc doing something totally different like running the **exec()** and launching another program. The key is the fine-grained control the **clone()** system call offers in determining what is shared between the parent and child.

Next, we need to wait for the child process to terminate. Here, cloned children are treated slightly differently. Check the **wait()** man page for more details.

To wait for children produced by *clone()*, the following additional (Linux-specific) values can be included in the *options* bit-mask argument for *waitpid()*, *wait3()*, and *wait4()*:

__WCLONE

If set, then wait for *clone* children only. If not set, then wait for *nonclone* children only. In this context, a *clone* child is one that delivers a signal other than SIGCHLD to its parent on termination. This bit is ignored if __WALL is also specified.

Implement the **my_system()** using **clone()** and measure the execution time for starting certain number of commands. Compare the time taken for executing the same commands with the previous implementations. ***Report this timing in the report you handover with the code***.

While designing the **my_system()** using **clone()** you don't need to completely emulate **fork()** or **vfork()**. You can set the flags for maximum performance. What would that be? However, it is still necessary to obtain isolation like parent proceeding while child crashing.

How would you set the flags if you want to execute a command like cd (change directory) in the child and parent be affected by it? If you want subsequent commands executed by the parent to run in a specific directory that was changed into by a previous cd command, what should the cloning do? That is, what kind of sharing should be performed while cloning? Your design needs to work with this use case. However, the files opened by the child or parent after cloning should not be available in the other.

The last part of the assignment is working with FIFOs. FIFOs are named pipes. We have already seen anonymous pipes in the lectures. They are one way to perform inter-process communication. FIFOs create the same but the name of the pipe is left in the filesystem. This allow two arbitrary processes to communicate using a FIFO. Like the pipe, a FIFO has a reading end and writing end. You create a FIFO using a command like the following.

We can create a FIFO from the shell using the *mkfifo* command:

```
$ mkfifo [ -m mode ] pathname
```

After running this command, you should see a FIFO with the given name in the file system (i.e., in the current working directory).

You need to modify the tiny shell so that a FIFO can be used as the argument in the shell. Use the version F (my_system() implemented using fork()) in this part of the assignment. If the command running in the tiny shell is outputting data to the terminal, it should go into the FIFO. Similarly, if the command is reading data from the terminal, it should come from the FIFO. Here we are considering commands that are not even written by us. For example, commands such as **ls** (directory listing) output the list of files to the terminal. When you run the command **ls** the tiny shell is responsible for redirecting the output to the FIFO.

Similarly, in another tiny shell instance we could run **`wc`** to count the number of characters, words, or lines. By connecting the second tiny shell instance to the reading end of the FIFO we can make the **`wc`** running there to count the output from **`ls`**.

This command piping could be achieved in many different ways. We want to do that piping using FIFOs.

Modify the tiny shell to carry out the command piping. It should work for any two commands. That is, the **`ls`** and **`wc`** are just illustration purposes.

Helpful Functions for Timing Your Program

You can use the following clock_gettime() library function to get a high resolution time difference. The clockid should be set to CLOCK_REALTIME to measure wall clock times.

The *clock_gettime()* system call returns the time according to the clock specified in *clockid*.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int clock_gettime(clockid_t clockid, struct timespec *tp);
int clock_getres(clockid_t clockid, struct timespec *res);
```

Both return 0 on success, or −1 on error

The time value is returned in the *timespec* structure pointed to by *tp*. Although the *timespec* structure affords nanosecond precision, the granularity of the time value returned by *clock_gettime()* may be coarser than this. The *clock_getres()* system call returns a pointer to a *timespec* structure containing the resolution of the clock specified in *clockid*.

```
struct timespec {
    time_t tv_sec;         /* Seconds */
    long   tv_nsec;        /* Nanoseconds */
};
```

Clock ID	Description
CLOCK_REALTIME	Settable system-wide real-time clock
CLOCK_MONOTONIC	Nonsettable monotonic clock
CLOCK_PROCESS_CPUTIME_ID	Per-process CPU-time clock (since Linux 2.6.12)
CLOCK_THREAD_CPUTIME_ID	Per-thread CPU-time clock (since Linux 2.6.12)

Turn-in and Marking Scheme

The programming assignment should be submitted via My Courses. Other submissions (including email) are not acceptable. **Your programming assignment should compile and run in Linux. Otherwise, the TAs can refuse to grade them**. Here is the mark distribution for the different components of the assignment.

What Should be Turned In?

1. Tiny shell based on Linux system().
2. My_system() implementations using fork(), vfork(), and clone(); a tiny shell version based on each version.
3. Documentation of testing and timing results.
4. Extension of tiny shell to use FIFO.
5. Testing results that show the tiny shell working with FIFO.

A marking scheme will be posted very soon.