Operating Systems (234123) - Winter 2015-2016
Home Assignment 4 - Wet

Due Date: Tuesday, 19.1.2016, 12:30p.m.
Teaching assistant in charge: Matthias Bonne

Postponements can be authorized only by Arthur, the TA in charge. If you need a postponement, please contact him directly.

Important: The Q&A for the exercise will take place at a public forum Piazza only. Critical updates about the HW will be published in pinned notes in the Piazza forum. These notes are mandatory and it is your responsibility to be updated. A number of guidelines to use the forum:

- Read previous Q&A carefully before asking the question; repeated questions will probably go without answers.
- Be polite, remember that course staff does this as a service for the students.
- You are not allowed to post any kind of solution and/or source code in the forum as a hint for other students; in case you feel that you have to discuss such a matter, please come to the reception hour.
- When posting questions regarding hw4, put them in the hw4 folder.

Introduction

In this assignment you need to implement a simplified version of a pseudo-terminal. A pseudo-terminal is a pair of character devices, one called the master, and the other one called the slave. Anything that is written to the slave device can be read from the master device. Conversely, anything that is written to the master device can be read from the slave device, except for a few special characters which are interpreted specially, as described below. Pseudo-terminals are often used by programs such as xterm, ssh, telnet and expect.

A good introduction to terminals and pseudo-terminals can be found here:

http://www.linusakesson.net/programming/tty/
Sessions and Process Groups

Every process in the system belongs to a *process group*. A process group is a collection of processes which should be manipulated simultaneously by the shell. Typically, all processes in a single shell pipeline form a process group. A process group has a *process group ID* (PGID) and may have a *process group leader*. New process groups can be created by calling `setpgid()`.

A *session* is a collection of process groups. Typically, all process groups managed by the same shell belong to the same session. The creator of the session, which is typically the shell itself, is called the *session leader*. Each session has a *foreground process group*, which runs in the foreground and may communicate with the user. Other process groups in the same session may be either suspended, or run in the background. A session has a *session ID*, which is the same as the process ID of the session leader. A session may be attached to a terminal device, known as the *controlling terminal* of the session. A new session can be created by calling `setsid()`.

Technical Information

Your task is to write a kernel module named “simplepty”. Your module will support two character devices, both with major number 62. The master device has minor number 0 and the slave device has minor number 1. As mentioned above, anything that is written to the slave device can be read from the master device, and vice versa, with a few exceptions described below. Essentially, you need to maintain two output buffers, one for the master and one for the slave. You should limit the size of those buffers to 64KB each.

The device may be associated with a session, and it may have a foreground process group. Processes can read and modify the current foreground process group and the current session associated with the device through the use of `ioctl()` commands.

Both devices should support the following operations.

**open()**

Opening the master device should initialize the underlying communication channel between the slave and the master. The master device can only be open once at any given time; attempting to open the master device when it is already open should return `-EIO`. When the master device is opened, the device should be reset so that it is not associated with any session nor does it have a foreground process group. If the master device has previously been open, then closed, and then later re-opened, the old session and foreground process group (if it had any) should be forgotten.

The slave device can only be opened when the master is open. Attempting to open the slave device when the master is not currently open should return `-EIO`. If a session leader opens the slave device, and the device is not currently associated with any session, it should be associated with the current task’s session, and its foreground process group ID should be set to the current task’s process group ID.

Note that the slave device can be opened more than once; in this case, all files that refer to the slave device operate on the same structures and use the same buffers for communicating with the master device. For example, after executing the following commands:

```c
1 setsid() fails if the calling process is a process group leader. In order to be sure that setsid() will succeed, call fork() and have the child call setsid(). See the manpage for setsid(2) for more details.
```
master_fd = open(MASTER, O_RDWR);
slave_fd1 = open(SLAVE, O_RDWR);
slave_fd2 = open(SLAVE, O_RDWR);

any operation (read(), write(), or ioctl()) done on slave_fd1 has the exact same effect as doing the same operation on slave_fd2.

write()

Writing to the slave device is only allowed if the current process is part of the foreground process group of the device (i.e. its process group ID must be equal to the current foreground process group ID of the device). An attempt to write to the slave device by a process not in the foreground process group should send a SIGTTOU signal to all processes in that process’ process group, and return -ERESTARTSYS.

Anything written to the slave device should be added to the slave’s output buffer to be read later by the master. Anything written to the master device should be added to the master’s output buffer to be read later by the slave, except for the following special characters:

- When writing a ^C character (ASCII code 0x03) to the master device, it is not added to the master’s output buffer. Instead, a SIGINT signal should be sent to all processes in the foreground process group of the device. If the device does not have a foreground process group, no action is taken.
- When writing a ^Z character (ASCII code 0x1A) to the master device, it is not added to the master’s output buffer. Instead, a SIGTSTP signal should be sent to all processes in the foreground process group of the device. If the device does not have a foreground process group, no action is taken.

In both cases, when writing a buffer to the master device which contains any of these special characters, the other characters in the buffer should be processed normally and added to the master’s output buffer as usual. Only occurrences of these two special characters should be removed from that buffer and handled specially.

A write() operation should only return after the entire buffer has been written, unless an error occurs or a signal is pending for the current process. If there is not enough space in the output buffers, write() should block (i.e. wait) until the other device reads some data and makes more space in the buffer.

If an error occurs before any characters have been processed, an appropriate error code should be returned. If a signal is pending for the current process and no characters have been processed, -ERESTARTSYS should be returned. Otherwise, write() should return the number of characters processed, including any special characters passed in the user’s buffer.

read()

Reading from a slave device is only allowed if the current process is part of the foreground process group of the device (i.e. its process group ID must be equal to the current foreground process group ID of the device). An attempt to read from the slave device by a process not in the foreground process group should send a SIGTTOU signal to all processes in that process’ process group, and return -ERESTARTSYS.

Otherwise, read() should wait until either there is at least one byte in the other device’s output buffer, or a signal is pending for the current process. In the former case, it should copy data from the other device’s output buffer and return the number of bytes that have been read. In the latter case, read() should return -ERESTARTSYS.
The master and slave devices should support the following ioctl(3) commands. Calling ioctl() with any other command should return -ENOTTY. The type of the optional argument follows each command in parenthesis; a type of void means no argument is used.

**TIOCNOTTY (void)**

Unset the session ID and foreground process group ID of the device. This operation is only allowed if the current process is a session leader, and its session is the one currently associated with the device. Otherwise, no action is taken. In both cases 0 is returned.

**TIOCSCTTY (void)**

Set the session ID and foreground process group ID of the device to the current task’s session ID and process group ID. This operation is only allowed if the current process is a session leader and the device is not associated with any other session. If the current process is not a session leader, or the device is already associated with another session, -EPERM should be returned. Otherwise, 0 is returned. If the device is already associated with the current process’ session, no action is taken and 0 is returned.

**TIOCGPGRP (pid_t *pgrp)**

Copy the current foreground process group ID of the device to the user-supplied address. If the device does not have a foreground process group, -ENOTTY should be returned. Otherwise, use put_user() to copy this value to the user’s address; if it fails, you should return -EFAULT. Otherwise, 0 should be returned.

**TIOCSPGRP (pid_t *pgrp)**

Set the foreground process group ID of the device to the one pointed by the argument. Use get_user() to retrieve this value from the user’s address; if it fails, you should return -EFAULT. Normally, several checks must be made to ensure the operation is allowed; in order to simplify your work, you should only check that pgrp is valid (non-negative) and that the session ID currently associated with the device matches the session ID of the current process. If pgrp is negative, -EINVAL should be returned. If the session ID of the device is not equal to the session ID of the current process, -ENOTTY should be returned. Otherwise, the foreground process group ID of the device should be set to pgrp and 0 should be returned.

**TIOCGSID (pid_t *sid)**

Copy the current session ID of the device to the user-supplied address. If the device is not associated with a session, -ENOTTY should be returned. Otherwise, use put_user() to copy this value to the user’s address; if it fails, you should return -EFAULT. On success, 0 should be returned.
close()

Closing the master device should release all resources associated with the device. This includes both output buffers of the master and the slave, as well as any other resources allocated to support their communication.

If the master device is closed while the slave is open, subsequent calls to `read()` from the slave device should return 0, and subsequent calls to `write()` or `ioctl()` should return -EIO. For example, after the following sequence of calls:

```c
master_fd = open (MASTER, O_RDWR);
slave_fd = open (SLAVE, O_RDWR);
close (master_fd);
```

any calls to `read()` from `slave_fd` should return 0, and any calls to `write()` or `ioctl()` on `slave_fd` should return -EIO. This condition persists until `slave_fd` is closed, even if the master is subsequently opened again. For example, if, after executing the above code, the following calls are made:

```c
master_fd = open (MASTER, O_RDWR);
new_slave_fd = open (SLAVE, O_RDWR);
```

then `master_fd` and `new_slave_fd` can communicate through the device as usual, while `slave_fd` would still return 0 on calls to `read()` and -EIO on calls to `write()` and `ioctl()`.

Closing the slave device has no special side-effects.

Useful functions and structure fields

- If `p` is a pointer to a `task_struct` structure, then:
  - `p->session` is the session ID of `p`.
  - `p->pgrp` is the process group ID of `p`.
  - `p->leader` is non-zero if (and only if) `p` is a session leader.

- In order to send a signal to all processes in a process group, call `kill_pg(pgrp,sig,1)`, where `pgrp` is the process group ID and `sig` is the signal number you wish to send.

- Use `wait_event_interruptible(wq,condition)` to wait for an event (such as data written to or read from a buffer), and `wake_up_interruptible(wq)` to wake up processes waiting for an event.

- To check if there are signals pending for the current process, call `signal_pending(current)`. This function returns non-zero if (and only if) there are pending signals for the current process.

Submission

You should submit a zip file which contains the following files:

- The source files of your module.
A Makefile which can be used to compile your module. The Makefile should create a module object file named "simplypty.o".

A file named `submitters.txt` which includes the ID, name and email of the participating students. The following format should be used:

Linus Torvalds linus@gmail.com 234567890
Steve Jobs jobs@os_is_best.com 345678901

**Important Note:** Make the outlined zip structure exactly. In particular, the zip should contain only the following files (no directories):

```
zipfile
|-- your source file(s)
|--- Makefile
|--- submitters.txt
```

Good Luck!
The Course Staff