Q1 2021 SOA Project

Description of the project

Mr. Baka Baka, as a visionary of the Operating System market, has realized that ZeOS has a lot of potential to compete with other commercial operating systems.

For that, he decides to start implementing features not currently present in ZeOS. One of these features is pipes.

A pipe is an inter-process communication mechanism that, by means of a shared buffer, allows the communication of data between related (parent-child, brother-brother, etc) processes.

To create a pipe, use the system call:

```
int pipe(int *pd);
```

in which pd is declared as *int pd[2]* and it is where the system call will return the file descriptors corresponding to the read channel (*pd[0]*) and write channel (*pd[1]*) of the pipe. Everything that is written in *pd[1]* will be available for reading in *pd[0]*. If everything is ok, the pipe system call returns 0. Otherwise, it returns -1 setting a value describing the error in *errno*.

After a pipe is created, *read* and *write* system calls can be executed using the returned file descriptors. In this case, *read* is a new system call and *write* changes its behavior:

- The *read* system call:

```
int read(int fd, void *buf, size_t count);
```

read *count* bytes from file descriptor *fd*, storing them at memory region *buf*. Reading from an empty pipe blocks the process until:

- there is data available in the pipe, or
- all the write channels of the pipe are closed. In this case, read returns 0.

- Writing to a filled pipe, blocks the process until:

- there is room in the pipe for writing more data, or
- all the read channels of the pipe are closed. In this case, *write* returns -1, setting *errno* to EPIPE.

Notice that for closing a pipe, all the read and write channels must be closed.

Internals

The *pipe* system call performs the following actions:

- Checks if there are enough resources to create the pipe.

- Allocates an available logical page in the process logical address space to store the data of the pipe. So, the pipe will store a maximum of 4096 bytes. This page can only be accessed by the operating system (privilege level 0)
- Allocates two channels in the Table of Channels of the process. Notice that checks must be performed to ensure that operations on those channels are possible (for example, it cannot be written in the read channel of a pipe).
- Allocates one entry of the Opened File Table of the operating system to store all the management data of the pipe. This data is:
    - A variable that points to the next position inside the pipe buffer from which data will be read
    - A variable that points to the next position inside the pipe buffer to which data will be written.
    - The available number of bytes in the buffer.
    - The number of readers of the pipe
    - The number of writers of the pipe
    - *Semaphores* to block the processes when needed

A process can have, at maximum, 10 pipes.

Processes inherit the pipes through the fork system call.

The system calls *pipe*, *close* and *read* are available as service numbers 15, 16 and 17 respectively.

For whatever behavioral detail, please refer to the POSIX standard about these system calls and the respective Linux man pages.

Semaphores

Semaphores is a new feature to help the programmer to synchronize his processes. Semaphores will be used to execute certain code fragments in mutual exclusion or block processes.

The data structures needed for each semaphore, are basically a counter and a queue for the blocked processes.

Since, for this project, semaphores will be only available in kernel mode, you will implement four new functions (*sem_init, sem_wait, sem_signal* and *sem_destroy*) to manage them.

sem_init **implementation**

The header of the function is:

```
int sem_init (int n_sem, unsigned int value)
```

where:

n_sem: identifier of the semaphore to be initialized

value: initial value of the counter of the semaphore returns: -1 if error, 0 if OK

This function initializes a semaphore identified by the number *n_sem*. It initializes the semaphore's counter to *value*. At the same time, it initializes the blocked processes queue in this semaphore and the data necessary for its correct use.

The return value 0 indicates a correct execution. If *n_sem* is already initialized, the returned value will be -1.

***sem_wait* implementation**

Its header is:

```
      int sem_wait (int n_sem)
```

where:

n_sem: identifier of the semaphore

returns: -1 if error, 0 if OK

If the counter for semaphore *n_sem* is less than or equal to zero, this function will block the process that has called it in this semaphore. If the counter is greater than zero, this function will decrement the value of the semaphore.

A return value 0 indicates a correct execution. If *n_sem* is not a valid identifier for a semaphore, the returned value will be -1.

***sem_signal* implementation**

Its header is:

```
      int sem_signal (int n_sem)
```

where:

n_sem: identifier of the semaphore

returns: -1 if error, 0 if OK

This function increments the counter of the semaphore *n_sem*. If there are one or more blocked processes in *n_sem*, this call will unblock the first process.

The return value 0 indicates a correct execution. If *n_sem* is not a valid identifier for a semaphore, the returned value will be -1.

***sem_destroy* implementation**

Its header is:

```
int sem_destroy (int n_sem)
```

where:

n_sem: identifier of the semaphore to destroy

returns: -1 if error, 0 if OK

This call destroys the semaphore *n_sem*. The return value 0 indicates a correct execution. If there are blocked processes, they will be unblocked and their corresponding *sem_wait* will return -1.

If *n_sem* is not a valid identifier for a semaphore, or the semaphore is not initialized, the returned value will be -1.

Milestones

1. (1 point) Implement kernel *semaphores* in ZeOS.
2. (1 point) Implement the file system structures (table of channels and opened file table).
3. (1 point) Implement the *pipe* system call (wrapper and service routine).
4. (1 point) Implement the *read* system call (wrapper and service routine).
5. (1 point) Modify the *write* system call to work with pipes.
6. (1 point) Modify the *sys_fork* service routine to inherit the pipes of the process.
7. (1 point) Implement the close system call (wrapper and service routine).
8. (1 point) Modify the *sys_exit* service routine to work with pipes.
9. (1 point) Implement workloads to measure the transmission bandwidth of pipes in ZeOS
10. (1 point) Implement an optimization to increase the transmission bandwidth of pipes (comment the optimization you plan to implement with the teacher, beforehand)