COSC4740-01 Operating Systems Design, Fall 2001

Lecture Note: Unnamed Pipe and Shared Memory

Unnamed Pipes

Pipes are a form of Inter-Process Communication (IPC) implemented on Unix and Linux variants. The kernel provides the synchronization between the processes accessing the same pipe. Data stored in the pipe is read on a First-In First-Out (FIFO) basis. The read /write operations are guaranteed to be atomic. The pipe is automatically removed by the OS when all the processes using it terminates. In fact, a pipe is a buffer managed by the kernel. It is a temporary storage of the data to be transferred between participating cooperative processes. The kernel takes care of the process synchronization.

Related Standard Header Files

sys/types.h
unistd.h
limits.h

How to create an unnamed pipe?

call the following function (system call):
int pipe (int fd[2]);

The input parameter is an array of two file descriptors fd[0] and fd[1]. A file descriptor is in fact an integer value. The system call returns a –1 in case of a failure. If the call is successful, it will return two integer values which are file descriptors fd[0]&fd[1]. In half-duplex pipes, fd[0] and fd[1] are used for reading and writing, respectively.

How to write to an unnamed pipe:

call the following system call:
ssize_t write (int fd, const void* buffer, size_t nbyte);

1st parameter: conventionally, fd[1] is used.
2nd parameter: the local data buffer (e.g., array), which contains the output data, of the calling process.
3rd parameter: the size (i.e., number of bytes) of the output data, in the buffer, to write out to the pipe. Note, nbyte should not be larger than PIPE_BUF defined in limits.h. The integrity (i.e., the data currently in the pipe will not be overwritten by another write()) is guaranteed only if nbyte <= PIPE_BUF.

The value of –1 is returned if the call fails. Else, the system call returns nbyte value.

See the manpage of the system call to see the details.

How to read from an unnamed pipe:

call the following system call:
ssize_t read(int fd, const void* buffer, size_t nbyte);

1st parameter: conventionally fd[0] is used.
2nd parameter: the local data buffer (e.g., array) of the calling process.
3rd parameter: the size of the input data, in the pipe, to to write in to the local buffer.

See the manpage of the system call to see the details.

What else?

If a process calls read() when the pipe is empty, the process will be blocked in the waiting queue until the data is sent through the pipe or until fd[1] is closed. The system call read() returns the value of 0 if there is no writer.

If a process calls write() when the pipe is full, the process will be blocked in the waiting queue until the pipe is able to receive more data. When the writing process has finished sending its data, it should close fd[1] (i.e., close(fd[1]);) to let the reading process be aware of it. By closing fd[1], a special character EOF (end-of-file) will be put in the pipe. The EOF informs the other process that there is no more data coming from the writing process, consequently keep it from blocked (i.e., does not call the next read() if the last character of the last input from the pipe is EOF. If the writing process terminates, fd[1] will be automatically closed by OS.

Give Me Some Examples

Example #1

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        int fd[2];
        char buf[30];

        if(pipe(fd) == -1) {
                perror("ERROR creating a pipe\n");
                exit(1);
        }
        if(!fork()) {
                printf("CHILD: writing to the pipe\n");
                write(fd[1], "Hello mother (or father?)", 26);
                printf("CHILD: exiting\n");
                exit(0);
        else {
```

```
            printf("PARENT: reading from pipe\n");
            read(fd[0], buf, 26);
            printf("PARENT: read \"%s\"\n", buf);
            wait(NULL);
            printf("PARENT: exiting\n");
        }
        return(0);
}
```

Example #2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
        int fd[2];

        if(pipe(fd) == -1) {
            perror("ERROR creating a pipe\n");
            exit(1);
        }

        if (!fork()) {
            close(1); // close the standard output file
            dup(fd[1]); // now fd[1] (the writing end of the pipe) is the standard output
            close(fd[0]); // I am the writer. So, I don't need this fd
            execlp("ls", "ls", NULL);
        }
        else {
            close(0); // close the standard input file
            dup(fd[0]);
            close(fd[1]);
            execlp("wc", "wc", "-w", NULL);
        }
        return(0);
}
```

Note, files 0, 1, and 2 are the standard input, standard output, and the standard error output, respectively.

Shared Memory

Shared memory is another method of IPC (inter-process-communication). The kernel does not manage the shared (virtual) memory segment. Moreover, the kernel does not take care of the process synchronization. Therefore, this IPC method is the fastest. However, the process synchronization is the programmer's responsibility.

Related standard header files

sys/types.h // defines the type key_t, which is an integer type
sys/ipc.h // defines flags and permissions associated with IPC mechanisms such as
sys/shm.h // functions and variables for shared memory segment

How to create&use a shared memory segment?

call the following function

int shmget(key_t key, int size, int flag)

1st parameter: Internal shared virtual memory segment ID, which is returned by ftok(). The kernel used this to identify the shared memory segment and it's associated structures. This argument should be IPC_PRIVATE (to create a new segment) or created by ftok() (e.g., key = ftok("/home/shared_seg", "R");). See the manpage of ftok for the details.

2nd parameter: the size of the shared memory segment in bytes. The maximum size and the minimum size are 1024KB and 1byte, respectively. A process can have up to 6 shared memory segments attached to it.

3rd parameter: this is composed of the following values:

IPC_CREAT to create a new segment. If this is not given, shmget() will try to find the segment associated with the given key value (the first parameter). If there is the associated segment and the user, who executed the current process, has the permission, then the segment will be used.

IPC_EXCL used with IPC_CREAT.

Four-digit permission code (0 is followed by three digits): $0P_1P_2P_3$ (e.g., 0666, 0644). P1 is the owner's permission, P2 is the group's permission, and P3 is the world's permission. Each P is the sum of read permission (4 (allowed) or 0 (not-allowed)) and write permission (2 (allowed) or 0 (not-allowed)). So, 0644 means, the owner of the segment can read/write to the segment. But other processes can only read from the segment.

Example 1:

```
int shmid;
shmid = shmget (IPC_PRIVATE, sizeof(int), IPC_CREAT|IPC_EXCL|0666);
```

Example 2:
key_t key;
int shmid;

key = ftok("home/yu3", 'R');
shmid = shmget(key, 1024, IPC_CREAT|0666);

shmget() will return the segmentID (success) or –1 (failure)

The shared memory segment cannot be used before it is attached to a process. To attach the shared memory segment, you can call the following function:

void* shmat(int shmid, void* shmaddr, int flag);

1st parameter: the segmentID returned by shmget().
2nd parameter: where to attach the memory segment. If a value of 0 is given, the system decides on the attachment address.
3rd parameter: access permission: 0 is read/write and SHM_RDONLY is read only.

shmat() returns the pointer to the shared segment (success) or –1 (failure)
Note, shmat() returns void type pointer, and you should treat it as a K type pointer, where K is the type of data you have in the shared memory segment.

Example 1:
char* data;

data = shmat(shmid, (void*) 0, 0);
if(data == (char*) (-1)) perror("shmat() failed\n");

Example 2:
int* shmadd;

```
if((shmadd = shmat(shmid, 0, 0)) == -1) {
        perror("ERROR attaching the segment\n");
        exit(1);
}
```

Now, you have a pointer type variable (in above examples, data and shmadd). You can use this variable to access the shared segment.

Example:
```
*shmadd = 5; //store (assign) an integer value of 5 in (to) the shared segment
strcpy(data, "Hello"); //store (assign) "Hello" in (to) the shared segment
printf("Shared content: %s\n", data);
x = *shmadd;
```

When you are done with the shared memory segment, your program should detach itself from it using the following function:

```
int shmdt(void* shmaddr);
```

Note, shmaddr is the return value of shmat().

shmdt() returns 0 (success) or –1 (failure)

Removing shared memory segment from the system

shmctl() releases the shared memory segment and its associated data structures back to the system resources. Since the shared memory segment is not automatically destroyed, you should remove it when there is no more use for it to release the system resources. You can do this using the following function:

```
int shmctl(int shmid, int cmd, struct shmid_ds* buf);
```

1st parameter: the return value of shmget().
2nd parameter: the operation shmctl() should execute. To remove a shared memory segment, IPC_RMID is used.
3rd parameter: put (struct shmid_ds *) 0.

shmctl() returns 0 (success) or –1 (failure).

Example:
```
if(shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0) == -1) {
        perror("ERROR\n"); exit(1);
}
```

Monitoring and removing the shared segment from the command line:
use ipcs and ipcrm. See the man pages.

Example Program

The example program will be given in the "Semaphore" Lecture Note.