

Inter Process Communication

OS Course
Spring 2009

Introduction

- Usually, the OS does everything to “hide” processes from each other
 - Different processes have different memory spaces
 - The scheduling of other processes is hidden from the process
- But sometimes processes need to **communicate** between them.
- We describe traditional mechanisms for communication between different processes that are running on the same operating system.

IPC mechanisms

- Signals
- Pipes
- FIFOs (named pipes)
- Message queues
- Shared memory segments
- Memory mapped files (not discussed here)
- Sockets (not discussed here) – used also for network communication

- Why so many?
 - Different mechanisms have different properties:
 - Other properties: speed, read order, access method...
 - **Synchronization** and **persistence** – next slides

Synchronization

- **Asynchronous**: The receiver has no control on when the data is delivered (signals, shared memory, files).
- **Synchronous**: The receiver asks for the data explicitly (all the rest).
 - **Blocking**: The receiver waits until data has been sent
 - **Non-blocking**: The receiver does not wait, gets data only if it has already been sent
 - The receiver can usually decide between blocking and non-blocking.
 - If the capacity of the channel is limited, the sender may also be blocked when sending.

Persistence

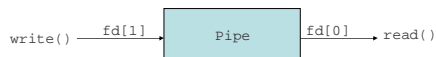
- How long does the “communication channel” last?
 - No persistence (signals)
 - Process (pipes, sockets)
 - Kernel (message queues, shared memory)
 - File System (files, FIFOs)

Signals

- A process can send a signal to another process
 - Also called “raising” a signal
- The signal is sent directly to a process
 - The sender must know the receiver process id
- Asynchronous – the signal is received without the request of the receiving process
- Messages are predefined (no data)
- Process persistence (except for SIGCHLD)

Pipes

- A pipe is a one-way stream between related processes
 - Synchronous
 - Persistence: Process
 - Created using `int pipe(int fd[2])`
 - This function returns two file descriptors
 - Use 'dup' to rename the file descriptors with a different numbers
 - Send: `write()` to `fd[1]`
 - Receive: `read()` from `fd[0]`
 - This is how the shell implements input/output redirection:
 - ~> `ls | wc`
 - To share a pipe, the processes need to be related
 - e.g. created using 'fork' or 'exec' from the same parent process
 - A pipe usually has a limited capacity.
 - By default, both `read()` and `write()` are blocking



Example

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

int main() {
    int pfd[2];
    char buf[30];
    pipe(pfd);
    if (fork()==0) { //create a child process
        printf(" CHILD: writing to the pipe\n");
        close(pfd[0]); //we don't need it
        write(pfd[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
        printf("PARENT: reading from pipe\n");
        close(pfd[1]); //we don't need it
        read(pfd[0], buf, 5); //blocked until child writes
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL); //wait until the child exits
    }
}
```

"ls | wc -l"

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
    int pfd[2];
    pipe(pfd);
    if (!fork()) {
        dup2(pfd[1],1); /* make stdout same as pfd[1] */
        close(pfd[0]); /* we don't need this */
        execlp("ls", "ls", NULL);
    } else {
        dup2(pfd[0],0); /* make stdin same as pfd[0] */
        close(pfd[1]); /* we don't need this */
        execlp("wc", "wc", "-l", NULL);
    }
    return 0;
}
```

FIFO

- Pipes can only be used by related processes
 - There is no way to share a file descriptor between unrelated processes
- A FIFO is sometimes known as a named pipe.
 - The name allows unrelated processes to communicate through it.
- It is actually a special file
 - Persistence: file system.
- Multiple processes can open(), write() and read() from the FIFO.
- open for writing blocks until someone opens for reading
 - And the other way around!
- If the reader closes the FIFO, the writer gets a SIGPIPE (broken pipe) when it tries to write.
- Creating a FIFO:
 - `int mkfifo(const char *path, mode_t mode);`
 - Example: `mkfifo("/tmp/namedpipe" , 0666);`

Producer

```
#include <limits.h>
int main(void)
{
    char buffer[LINE_MAX];
    int num, fd;
    int length;

    mkfifo("/tmp/namedpipe" , 0666);
    printf("waiting for readers...\n");
    fd = open("/tmp/namedpipe", O_WRONLY); //blocked until consumer opens fifo
    printf("got a reader-type some stuff\n");

    while (fgets(buffer, LINE_MAX, stdin) != NULL) {
        length = strlen(buffer)+1;
        num = write(fd, length, sizeof(int));
        if (num < 0) {
            perror("write");
            exit(1);
        }
        num = write(fd, buffer, length);
        if (num < 0) {
            perror("write");
            exit(1);
        }
        printf("producer: wrote %d bytes\n", length*sizeof(int));
    }
    return 0;
}
```

Consumer

```
#include <limits.h>
int main(void)
{
    char buffer[LINE_MAX];
    int num, fd;
    int length;

    mkfifo("/tmp/namedpipe", 0666);
    printf("waiting for writers...\n");
    fd = open("/tmp/namedpipe", O_RDONLY); //releases producer
    printf("got a writer\n");

    do {
        num = read(fd, &length, sizeof(int));
        if (num < 0) {
            perror("read");
            exit(1);
        }
        num = read(fd, buffer, length);
        if (num < 0) {
            perror("read");
            exit(1);
        }
        printf("consumer: read \"%s\"\n", buffer);
    } while (num > 0);

    return 0;
}
```

Message Queues

- A message queue is a list of messages with a unique **key** attached to it
- Any process can send a message to the queue
- Any process can receive a message from the queue
- As long as they know its identifying key...
- Difference from FIFOs:
 - Persistence: kernel
 - No need to open/close the message queue
 - Messages have **sizes** – the receiver gets the whole message sent by the sender
 - Messages have **types** – the receiver may request a specific message type, if so the messages will not be received in order of sending.

Message Queues

- Get or Create:

```
int msgget(key_t key, int msgflg);
```

 - key is a **unique number** identifying the message queue.
 - msgflg defines permissions and whether to create if does not exist.
 - Return value: msgid, used by the following functions

- How to decide on a **shared unique key** for a message queue?

```
key_t ftok(const char *path, int id);
```

- Example:

```
key = ftok("somefile", 'b');
msgid = msgget(key, 0644 | IPC_CREAT);
```

- The process must be allowed to 'stat' the file 'somefile'

Message Queues

- Send:

```
int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg);
```

 - msgp points to the message buffer
 - The first 'long' in msgp must indicate the type of the message
 - msgsz = message size in bytes, not including the type
 - A zero length message is allowed
- Receive:

```
int msgrcv(int msgid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

 - msgsz: limits the size of the received message
 - msgtyp: Allows the receiver to receive only a message with a requested type
 - msgtyp = 0 – get all types of messages
 - msgtyp > 0 – get only messages of type msgtyp
 - msgtyp < 0 – get only message of type <= msgtyp
 - and more...
- Destroy:

```
int msgctl(int msgid, int cmd, struct msgid_ds *buf);
```

 - Example:

```
msgctl(msgid, IPC_RMID, NULL);
```

Example

Shared header "msgbuf.h":

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <limits.h>

struct my_msgbuf {
    long mtype;
    char mtext[LINE_MAX];
};
```

Producer

```
#include "msgbuf.h"
int main(void)
{
    struct my_msgbuf buf;
    int msgid;
    key_t key;

    key = ftok("ipc_example.c", 'B');
    msgid = msgget(key, 0644 | IPC_CREAT);
    printf("Enter lines of text, ^D to quit:\n");

    buf.mtype = 1;
    while(fgets(buf.mtext, LINE_MAX, stdin) != NULL) {
        msgsnd(msgid, &buf, strlen(buf.mtext)+1, 0);
    }
    msgctl(msgid, IPC_RMID, NULL); //deletes the queue immediately
    return 0;
}
```

Consumer

```
#include "msgbuf.h"
int main(void)
{
    struct my_msgbuf buf;
    int msgid;
    key_t key;

    key = ftok("ipc_example.c", 'B');
    msgid = msgget(key, 0644);

    for(;;) {
        if (msgrcv(msgid, &buf, sizeof(buf.mtext), 1, 0) < 0) {
            perror("msgrcv");
            exit(1);
        }
        printf("consumer: \"%s\"\n", buf.mtext);
    }
    return 0;
}
```

Shared Memory Segments

- Processes can ask from the OS a segment of shared memory that will be used for IPC.
- Identification using keys, like message queues.
- This mechanism is **asynchronous**
 - The writer can change the shared memory without the knowledge/control of the reader
 - To synchronize, use semaphores
- Persistence: kernel
- Communication through shared memory is very fast
 - No need for a system call when reading or writing

Shared Memory Segments

- Get or Create:

```
int shmget(key_t key, size_t size, int shmflg);
```

Example:

```
key = ftok("somefile", 'b');  
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```
- Start using:

```
void *shmat(int shmid, void *shmaddr, int shmflg);
```

 - Returns a process-local address for the memory segment
 - Can use it like any other buffer (remember permissions)
- Stop using:

```
int shmdt(void *shmaddr);
```
- Destroy:

```
shmctl(shmid, IPC_RMID, NULL);
```

 - Actual deletion - only after everyone detaches from the memory segment

Summary

- We saw several IPC mechanisms in Unix
 - Signals
 - Pipes
 - FIFOs (named pipes)
 - Message queues
 - Shared memory segments
- There are some others, such as
 - Memory mapped files
 - Sockets – used also for network communication
- The choice of mechanism depends on the requirements.
 - Different mechanisms are most suitable for different uses.