innovate achieve lead

BITS Pilani
Pilani Campus

Operating Systems

Computer Science and Information Systems Department
BITS, Pilani

innovate achieve lead

BITS Pilani
Pilani Campus

Inter Process Communication
Tutorial 9

Today's Agenda

- Inter Process Communication
  - Pipes
  - FIFOs
  - Message Queues
  - Shared Memory

Introduction

- Inter process communication (IPC) is a mechanism which allows processes to communicate each other. This involves synchronizing their actions and managing shared data.
- Communication can be of two types –
  - Between related processes initiating from only one process, such as parent and child processes.
  - Between unrelated processes, or two or more different processes.

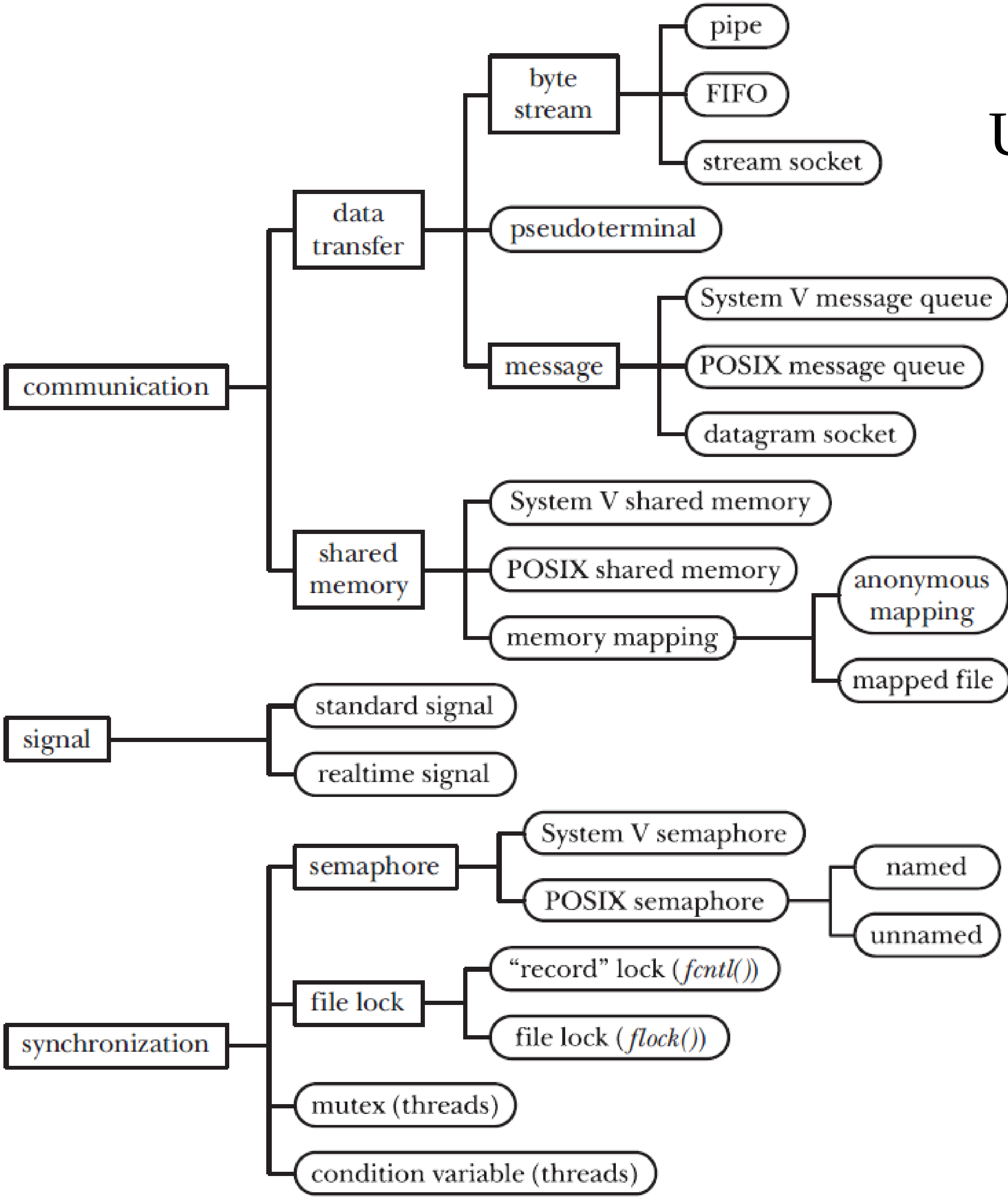
Introduction contd.

- **Pipes**– Communication between two related processes. The mechanism is half duplex meaning the first process communicates with the second process. To achieve a full duplex i.e., for the second process to communicate with the first process another pipe is required.
- **FIFO**– Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.
- **Message Queues**– Communication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message and retrieving it out of the queue.

Introduction contd.

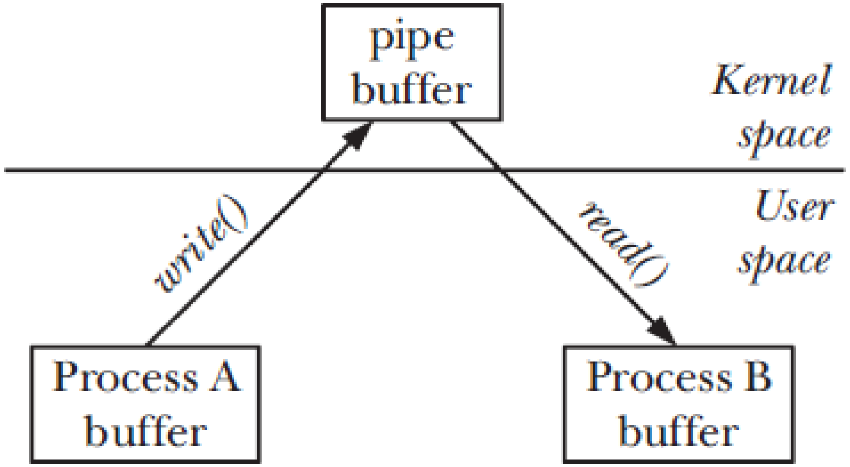
- **Shared Memory**– Communication between two or more processes is achieved through a shared piece of memory among all processes.
- **Semaphores**– Semaphores are meant for synchronizing access to multiple processes. When one process wants to access the memory (for reading or writing), it needs to be locked (or protected) and released when the access is removed.
- **Signals**– Signal is a mechanism to communication between multiple processes by way of signaling. This means a source process will send a signal (recognized by number) and the destination process will handle it accordingly.

A taxonomy of UNIX IPC facilities



Communication

- **Data Transfer Facility**:
  - In order to communicate, one process writes data to the IPC facility, and another process reads the data.
  - These facilities require two data transfers between user memory and kernel memory: one transfer from user memory to kernel memory during writing, and another transfer from kernel memory to user memory during reading.



Communization (Data transfer Categories)

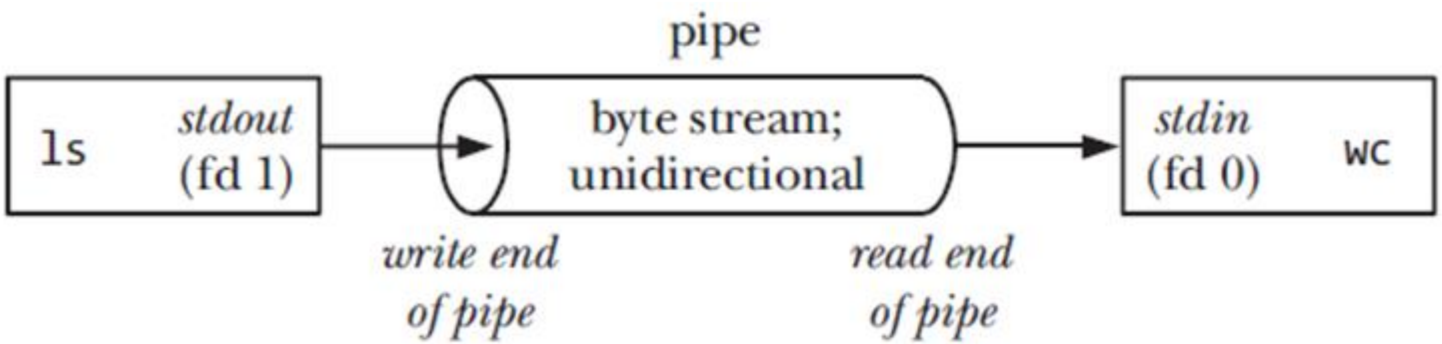
- **Byte stream:**
  - The data exchanged via pipes, FIFOs and datagram sockets is an undelimited byte stream.
  - Each read operation may read an arbitrary number of bytes from the IPC facility, regardless of the size of blocks written by the writer.
- **Message:**
  - The data exchanged message queues
  - These are form of delimited messages.
  - Each read operation reads entire message,
  - It is not possible to read part of a message, leaving the remainder on the IPC facility; nor is it possible to read multiple messages in a single read operation.

Communication

- Shared Memory
  - Shared memory allows processes to exchange information by placing it in a region of memory that is shared between the processes
  - A process can make data available to other processes by placing it in the shared memory region. Because communication doesn't require system calls or data transfer between user memory and kernel memory, shared memory can provide very fast communication.
  - There is a need to synchronize operations on the shared memory.
  - For example, one process should not attempt to access a data structure in the shared memory while another process is updating it.
  - A semaphore is the usual synchronization method used with shared memory.

Pipes (Byte Stream)

- Shell Command
  - **$ ls | wc –l**
- To execute the above command, the shell creates two processes, executing ls and wc, respectively using fork() and exec() system calls.
- Two processes are connected to the pipe so that the writing process (ls) has its standard output (file descriptor 1) joined to the write end of the pipe, while the reading process (wc) has its standard input (file descriptor 0) joined to the read end of the pipe.
- Pipes are unidirectional



Creating and Using Pipes

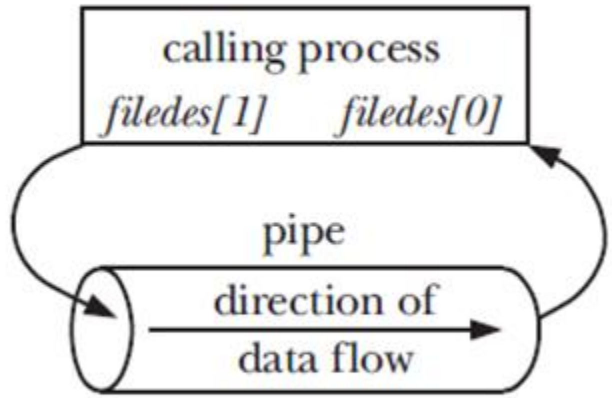
Syntax:

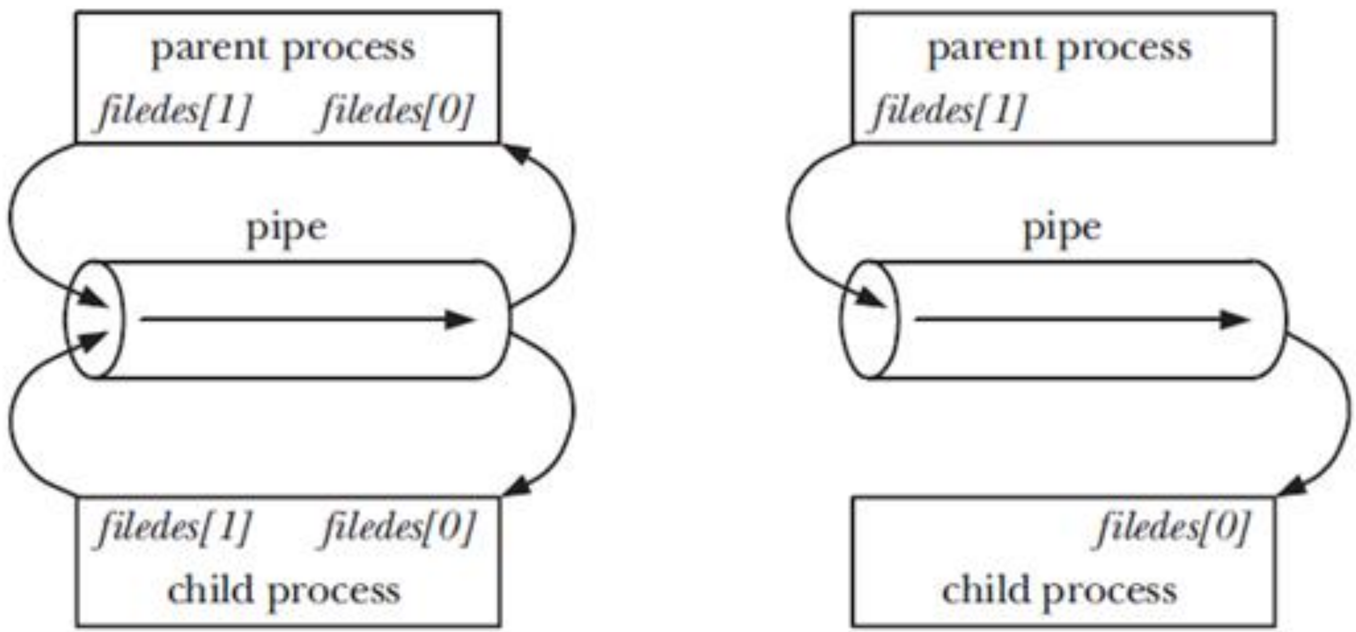
#include <unistd.h>

int pipe(int filedes[2]);

- Returns 0 on success, or –1 on error
- A successful call to pipe() returns two open file descriptors in the array filedes: one for the read end of the pipe (filedes[0]) and one for the write end (filedes[1]).
- we can use the read() and write() system calls to perform I/O on the pipe.

Creating and Using Pipes





a) After *fork()*

b) After closing unused descriptors

PIPE BETWEEN PARENT AND CHILD PROCESS

```
int main()
{
    int filedes[2];
    if (pipe(filedes) == -1)  printf("error creating pipe \n");
    else
        printf("Pipe Created Successfully\nfiledes[0] = %d,filedes[1] = %d\n",filedes[0],filedes[1]);
    switch (fork()) { /* Create a child process */
    case -1:
        printf("fork failed \n");
    case 0: /* Child */
        printf("Child Process.......%d\n",getpid());
        if (close(filedes[1]) == -1)     printf("failed to close \n");
        else
        printf("Closed write end, read file descr = %d\n",filedes[0]);
        break;
    default: /* Parent */
        wait(NULL);
        printf("Parent Process.......%d\n",getpid());
        if (close(filedes[0]) == -1)   printf("failed to close \n");
        else
                printf("Closed read end, write file descr = %d\n",filedes[1]);
        break; }   }
```

```c
int main()
{
    int filedes[2];
    if (pipe(filedes) == -1)  printf("error creating pipe \n");
    else
        printf("Pipe Created Successfully\nfiledes[0] = %d,filedes[1] = %d\n",filedes[0],filedes[1]);
    switch (fork()) { /* Create a child process */
    case -1:
        printf("fork failed \n");
    case 0: /* Child */
        printf("Child Process.......%d\n",getpid());
        if (close(filedes[1]) == -1)     printf("failed to close \n");
        else
        printf("Closed write end, read file descr = %d\n",filedes[0]);
        break;
    default: /* Parent */
        wait(NULL);
        printf("Parent Process.......%d\n",getpid());
        if (close(filedes[0]) == -1)   printf("failed to close \n");
        else
                printf("Closed read end, write file descr = %d\n",filedes[1]);
        break; }   }
```

```
$ ./a.out
Pipe Created Successfully
filedes[0] = 3,filedes[1] = 4
Child Process.......1966
Closed write end, read file
descr = 3
Parent Process.......1965
Closed read end, write file
descr = 4
```

```c
int main(void)
{
    int pfds[2];
    char buf[30];

    pipe(pfds);

    if (!fork()) {
        printf(" CHILD: writing to the pipe\n");
        write(pfds[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    }
    else {
        printf("PARENT: reading from pipe\n");
        read(pfds[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }

    return 0;
}
```

```
PARENT: reading from pipe
 CHILD: writing to the pipe
 CHILD: exiting
PARENT: read "test"
```

Using a pipe to communicate between a parent and child process

```
Int main(int argc, char *argv[])
{
        int pfd[2];
        char buf[BUF_SIZE];
        ssize_t numRead;
        if (argc != 2 || strcmp(argv[1], "--help") == 0)
                printf("%s %s \n", argv[0],argv[1]);
        if (pipe(pfd) == -1)
                printf("failed to create pipe\n");
        else
                printf("Pipe created successfully\nread file des = %d, write file des = %d\n",pfd[0],pfd[1]);
        switch (fork()) {
        case -1:
                printf("failed to fork\n");

        case 0:
                /* Child - reads from pipe */
                printf("Child Created.........%d\n",getpid());
                numRead = read(pfd[0], buf, BUF_SIZE);
                printf("text read from pipe by child : %s , numRead = %d\n",buf,numRead);
                if (numRead == -1)              printf("read error\n");
                if (numRead == 0)               break; /* End-of-file */
                if (write(STDOUT, buf, numRead) != numRead)
                    printf("child - partial/failed write\n");
                write(STDOUT, "\n", 1);
                if (close(pfd[0]) == -1)printf("failed to close");
                    exit(EXIT_SUCCESS);
        default:
                /* Parent - writes to pipe */
                printf("Parent continues %d\n",getpid());
                if (close(pfd[0]) == -1)        /* Read end is unused */
                    printf("failed to close read end in - parent\n");
                if(write(pfd[1],argv[1],strlen(argv[1]))!=strlen(argv[1]))
                    printf("parent - partial/failed write\n");
                else
                    printf("Parent successfully write to pipe (%d): %s\n",pfd[1],argv[1]);
                if (close(pfd[1]) == -1)  printf("failed to close");
                wait(NULL);
                exit(EXIT_SUCCESS);
                }   }
```

```
$ ./a.out BITS
Pipe created successfully
read file des = 3, write file des = 4
Parent successfully write to pipe
(4): BITS
Child Created.........1750
text read from pipe by child : BITS ,
numRead = 4
BITS
```

Usage of Pipes to execute commands like ls | wc

```
int pfd[2];
pipe(pfd);      /* Allocates (say) file descriptors 3 and 4 for pipe */

/* Other steps here, e.g., fork() */

close(STDOUT);            /* Free file descriptor 1 */
dup(pfd[1]);              /* Duplication uses lowest free file  descriptor, i.e.,
                          fd 1 */

dup2(pfd[1], STDOUT);     /* Close descriptor 1, and reopen bound to write
                          end of pipe */
```

Programming Exercises

- Write a C program to implement the unix command “ls -l | sort” using pipes.

innovate achieve lead

Any Queries?