# SCT221-0779/2022 PAUL MAINA NGARUIYA

## **1. Role of system calls in IPC (4 Marks)**

System calls allow processes to **communicate and share resources** by interacting with the kernel. In C, IPC can be achieved using pipes, message queues, and shared memory.

**Pipe Example (parent ↔ child):**

*#include <stdio.h>*

*#include <unistd.h>*

*int main() {*

*int fds[2]; // file descriptors for pipe*

*char buffer[20];*

*pipe(fds);*

*if (fork() == 0) {*

*// Child writes to pipe*

*close(fds[0]);*

*write(fds[1], "Pipe demo", 9);*

*} else {*

*// Parent reads*

*close(fds[1]);*

*read(fds[0], buffer, sizeof(buffer));*

*printf("Parent got: %s\n", buffer);*

*}*

*}*

**Message Queue Example:**

#include <sys/ipc.h>

#include <sys/msg.h>

struct msgbuf { long type; char text[50]; };

int qid = msgget(0x1234, IPC\_CREAT | 0666);

struct msgbuf m = {1, "IPC message"};

msgsnd(qid, &m, sizeof(m.text), 0);

**Shared Memory Example:**

int id = shmget(0x2345, 1024, IPC\_CREAT | 0666);

char \*ptr = (char\*) shmat(id, NULL, 0);

sprintf(ptr, "Shared segment");

shmdt(ptr);

## **2. Client-Server with sockets (3 Marks)**

Client-server communication is achieved using TCP sockets:

**Server code (shortened):**

int serv = socket(AF\_INET, SOCK\_STREAM, 0);

struct sockaddr\_in addr;

addr.sin\_family = AF\_INET;

addr.sin\_addr.s\_addr = INADDR\_ANY;

addr.sin\_port = htons(9090);

bind(serv, (struct sockaddr\*)&addr, sizeof(addr));

listen(serv, 2);

int cli = accept(serv, NULL, NULL);

char buf[50];

read(cli, buf, sizeof(buf));

printf("Server received: %s\n", buf);

**Client code:**

int cli = socket(AF\_INET, SOCK\_STREAM, 0);

struct sockaddr\_in saddr;

saddr.sin\_family = AF\_INET;

saddr.sin\_port = htons(9090);

inet\_pton(AF\_INET, "127.0.0.1", &saddr.sin\_addr);

connect(cli, (struct sockaddr\*)&saddr, sizeof(saddr));

send(cli, "Hello Server", 12, 0);

## **3. Signals in UNIX/Linux (4 Marks)**

Signals are notifications sent to a process. They can terminate, stop, or trigger custom routines.

**Example:**

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

void myhandler(int sig) {

printf("Caught signal: %d\n", sig);

}

int main() {

signal(SIGINT, myhandler); // catch Ctrl+C

printf("Running... Press Ctrl+C\n");

while (1) sleep(1);

}

Here, instead of exiting on Ctrl+C, the handler prints a message.

## **4. Pipes vs FIFO vs Message Queues (3 Marks)**

* **Pipe:** temporary, unnamed, only related processes.
* **FIFO (mkfifo):** exists in filesystem, unrelated processes can use.
* **Message Queue:** kernel-managed, structured messages with priorities.

**FIFO Example:**

mkfifo("mypipe", 0666);

int fd = open("mypipe", O\_WRONLY);

write(fd, "Named Pipe Msg", 14);

## **5. Shared Memory (3 Marks)**

Steps:

1. Create → shmget()
2. Attach → shmat()
3. Access/modify memory
4. Detach → shmdt()
5. Remove → shmctl()

**Example:**

int shmid = shmget(0x5555, 512, IPC\_CREAT | 0666);

char \*data = (char\*) shmat(shmid, NULL, 0);

sprintf(data, "Writing via shared memory");

shmdt(data);

shmctl(shmid, IPC\_RMID, NULL);

Synchronization often requires semaphores to avoid race conditions.

## **6. Semaphores (2 Marks)**

Semaphores control access to a critical section.

**Example:**

sem\_t lock;

sem\_init(&lock, 0, 1);

sem\_wait(&lock); // enter

// critical section

sem\_post(&lock); // leave

This ensures only one process modifies shared data at a time.

## **7.** select() **for multiplexed I/O (2 Marks)**

select() allows one process to watch multiple sockets/files.

**Example:**

fd\_set set;

FD\_ZERO(&set);

FD\_SET(sockA, &set);

FD\_SET(sockB, &set);

select(sockB+1, &set, NULL, NULL, NULL);

if (FD\_ISSET(sockA, &set)) {

recv(sockA, buf, sizeof(buf), 0);

}

## **8. UNIX domain sockets (2 Marks)**

Used for IPC on the same machine.

**Server:**

int s = socket(AF\_UNIX, SOCK\_STREAM, 0);

struct sockaddr\_un addr = {AF\_UNIX, "sockfile"};

bind(s, (struct sockaddr\*)&addr, sizeof(addr));

listen(s, 1);

accept(s, NULL, NULL);

**Client:**

int c = socket(AF\_UNIX, SOCK\_STREAM, 0);

struct sockaddr\_un addr = {AF\_UNIX, "sockfile"};

connect(c, (struct sockaddr\*)&addr, sizeof(addr));

## **9. Makefiles (3 + 2 + 1 Marks)**

**Purpose:** Automates builds, handles dependencies, avoids recompiling everything.

**Sample Makefile:**

CC = gcc

CFLAGS = -Wall

OBJS = main.o server.o ipc.o

app: $(OBJS)

$(CC) -o app $(OBJS)

main.o: main.c ipc.h

server.o: server.c ipc.h

ipc.o: ipc.c ipc.h

clean:

rm -f \*.o app

* make builds project.
* make clean clears binaries.
* Dependencies (ipc.h) ensure rebuilds when headers change.

## **10. Efficiency of Makefiles (1 Mark)**

Makefiles speed up compiling large C projects by rebuilding **only changed files**, instead of everything.  
Example: if only server.c changes, only that .o is rebuilt, then linked. This saves time and reduces errors in system communication projects using sockets, semaphores, etc.