SYSTEM PROGRAMMING -ICS 2305

NAME: PHILOMEN KHAMAL URENDI REG No: SCT221-0799/2022

2025 (Take away)

ANSWER ALL QUESTIONS

1. **Explain the role of system calls in C programming for inter-process communication (IPC).**

Discuss how mechanisms like pipes, message queues, and shared memory are implemented in C, including sample code snippets to illustrate their usage. (4 Marks)

System calls provide the interface between user programs and the operating system, enabling inter-process communication (IPC). IPC mechanisms include:

- **Pipes**: Allow data flow between processes.
- **Message Queues**: Enable message passing between processes.
- **Shared Memory**: Allows multiple processes to access the same memory space.

Pipes

```c
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>


int main() {

    int pipefd[2];

    char buffer[20];

    pipe(pipefd); // Create a pipe


    if (fork() == 0) {

        // Child process

        close(pipefd[1]); // Close write end

        read(pipefd[0], buffer, 20);

        printf("Received: %s\n", buffer);
```

```
        close(pipefd[0]);
    } else {
        // Parent process
        close(pipefd[0]); // Close read end
        write(pipefd[1], "Hello", 6);
        close(pipefd[1]);
    }
    return 0;
}
```

Message Queues

```
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct message {
    long msg_type;
    char msg_text[100];
};

int main() {
    key_t key = ftok("progfile", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);
    struct message msg;

    msg.msg_type = 1;
```

```
    strcpy(msg.msg_text, "Hello");

    msgsnd(msgid, &msg, sizeof(msg), 0);
    printf("Message Sent: %s\n", msg.msg_text);

    // Receiving message
    msgrcv(msgid, &msg, sizeof(msg), 1, 0);
    printf("Message Received: %s\n", msg.msg_text);

    msgctl(msgid, IPC_RMID, NULL); // Destroy the message queue
    return 0;
}
```

Shared Memory

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {
    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT);
    char *str = (char*) shmat(shmid, (void*)0, 0);

    strcpy(str, "Hello, Shared Memory!");
    printf("Data written in memory: %s\n", str);
```

```
    shmdt(str); // Detach from shared memory
    shmctl(shmid, IPC_RMID, NULL); // Destroy shared memory
    return 0;
}
```

2. **Describe the concept of client-server communication in C using socket programming. Write a detailed program for a simple TCP client and server that communicate over a network, explaining each part of the code. (3 Marks)**

In C, **socket programming** enables two processes (often on different machines) to communicate over a network. The **server** waits for incoming connections and provides services, while the **client** initiates the connection to request services.

Server Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT 8080

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
```

```
    address.sin_port = htons(PORT);

    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    listen(server_fd, 3);

    new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen);
    read(new_socket, buffer, 1024);
    printf("%s\n", buffer);
    send(new_socket, "Hello from server", strlen("Hello from server"), 0);

    close(new_socket);
    close(server_fd);
    return 0;
}
```

Client Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT 8080

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char *hello = "Hello from client";
```

```
    char buffer[1024] = {0};

    sock = socket(AF_INET, SOCK_STREAM, 0);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);
    connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    send(sock, hello, strlen(hello), 0);
    read(sock, buffer, 1024);
    printf("%s\n", buffer);

    close(sock);
    return 0;
}
```

3. **Discuss the concept of signals in UNIX/Linux systems. How are signals used for process communication in C? Write a C program to demonstrate signal handling, including signal generation and handling routines. (4 Marks)**

In UNIX/Linux, a **signal** is a limited form of inter-process communication (IPC) used to notify a process that a particular event has occurred.
Signals are **asynchronous** they can be sent to a process at any time, interrupting its normal execution flow.

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

void handle_signal(int sig) {
```

```
    printf("Received signal %d\n", sig);
}

int main() {
    signal(SIGINT, handle_signal); // Register signal handler
    while (1) {
        printf("Running... Press Ctrl+C to send SIGINT\n");
        sleep(1);
    }
    return 0;
}
```

4. **Explain the difference between pipes, FIFO (named pipes), and message queues in the context of process communication in C. Provide example code to demonstrate how each mechanism is used for communication between processes. (3 Marks)**

Pipes: Unidirectional communication.

FIFO (Named Pipes): Allows bidirectional communication and can be accessed via a name in the filesystem.

Message Queues: Allows messages to be queued and retrieved in a prioritized manner.

Pipes & Message Queues as in question 1.

Signal Handling:

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

void handle_signal(int sig) {
    printf("Received signal %d\n", sig);
```

```
}

int main() {
    signal(SIGINT, handle_signal); // Register signal handler
    while (1) {
        printf("Running... Press Ctrl+C to send SIGINT\n");
        sleep(1);
    }
    return 0;
}
```

5. **Describe the steps involved in implementing inter-process communication using shared memory in C. Write a sample program that demonstrates shared memory creation, access, and cleanup, highlighting synchronization issues and their solutions. (3 Marks)**

Steps:

1. Create shared memory segment.
2. Attach to the segment.
3. Read/write data.
4. Detach and remove the segment.

shmget() → shmat() → read/write → shmdt() → shmctl() to remove.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <pthread.h>
```

```
#define SHM_SIZE 1024

void *shared_memory;

int main() {
    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);

    shared_memory = shmat(shmid, NULL, 0);
    strcpy(shared_memory, "Shared Memory Example");

    printf("Data in shared memory: %s\n", (char *)shared_memory);

    // Cleanup
    shmdt(shared_memory);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

5. **Discuss the concept of semaphores in process synchronization within system communication in C. Write a program that demonstrates semaphore usage for controlling access to a critical section in a multi-process environment. (2 Marks)**

Semaphores are used to control access to shared resources in concurrent programming.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

sem_t semaphore;
```

```
void *critical_section(void *arg) {
   sem_wait(&semaphore); // Enter critical section
   printf("In critical section\n");
   sleep(1); // Simulate work
   sem_post(&semaphore); // Exit critical section
   return NULL;
}
int main() {
   pthread_t threads[5];
   sem_init(&semaphore, 0, 1); // Initialize semaphore

   for (int i = 0; i < 5; i++) {
      pthread_create(&threads[i], NULL, critical_section, NULL);
   }
   for (int i = 0; i < 5; i++) {
      pthread_join(threads[i], NULL);
   }
   sem_destroy(&semaphore);
   return 0;
}
```

6. **Explain how select() and poll() system calls facilitate multiplexed I/O in socket programming. Provide a C example that uses select() to handle multiple client connections simultaneously. (2 Marks)**

The select() system call allows a program to monitor multiple file descriptors.

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

```
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/select.h>

#define PORT 8080
#define MAX_CLIENTS 10

int main() {
    int server_fd, client_fd, max_sd, sd;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024];
    fd_set readfds;

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    listen(server_fd, 3);

    int client_sockets[MAX_CLIENTS] = {0};

    while (1) {
        FD_ZERO(&readfds);
        FD_SET(server_fd, &readfds);
        max_sd = server_fd;
```

```
for (int i = 0; i < MAX_CLIENTS; i++) {
    sd = client_sockets[i];
    if (sd > 0) FD_SET(sd, &readfds);
    if (sd > max_sd) max_sd = sd;
}
select(max_sd + 1, &readfds, NULL, NULL, NULL);

if (FD_ISSET(server_fd, &readfds)) {
    client_fd = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen);
    // Add new client socket to array
    for (int i = 0; i < MAX_CLIENTS; i++) {
        if (client_sockets[i] == 0) {
            client_sockets[i] = client_fd;
            break;
        }
    }
}

for (int i = 0; i < MAX_CLIENTS; i++) {
    sd = client_sockets[i];
    if (FD_ISSET(sd, &readfds)) {
        read(sd, buffer, sizeof(buffer));
        printf("Client: %s\n", buffer);
        send(sd, "Message received", strlen("Message received"), 0);
    }
}
```

```
    }
    return 0;
}
```

7. **Describe the process of establishing a communication link between two processes using UNIX domain sockets in C. Include sample code for creating both server and client processes. (2 Marks)**

UNIX domain sockets allow two processes on the **same machine** to communicate via a file system pathname instead of an IP address. They support both stream (SOCK_STREAM) and datagram (SOCK_DGRAM) modes and are faster than network sockets because they bypass the network stack.

Steps:
Server:

- Create socket → socket(AF_UNIX, SOCK_STREAM, 0)
- Bind to a pathname → bind()
- Listen for connections → listen()
- Accept client → accept()
- Exchange data → read()/write() or send()/recv()
- Close socket and remove pathname → close(), unlink()

Client:

- Create socket → socket(AF_UNIX, SOCK_STREAM, 0)
- Connect to server → connect()
- Exchange data → read()/write()
- Close socket → close()

Server Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
```

```
#include <unistd.h>

#define SOCKET_PATH "/tmp/mysocket"

int main() {
    int server_fd, client_fd;
    struct sockaddr_un addr;
    char buffer[100];

    server_fd = socket(AF_UNIX, SOCK_STREAM, 0);
    addr.sun_family = AF_UNIX;
    strcpy(addr.sun_path, SOCKET_PATH);

    bind(server_fd, (struct sockaddr*)&addr, sizeof(addr));
    listen(server_fd, 5);

    client_fd = accept(server_fd, NULL, NULL);
    read(client_fd, buffer, sizeof(buffer));
    printf("Received: %s\n", buffer);

    close(client_fd);
    close(server_fd);
    unlink(SOCKET_PATH);
    return 0;
}
```

Client Code:

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

```
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define SOCKET_PATH "/tmp/mysocket"

int main() {
    int sock;
    struct sockaddr_un addr;
    char *message = "Hello, UNIX Domain Socket";

    sock = socket(AF_UNIX, SOCK_STREAM, 0);
    addr.sun_family = AF_UNIX;
    strcpy(addr.sun_path, SOCKET_PATH);

    connect(sock, (struct sockaddr*)&addr, sizeof(addr));
    send(sock, message, strlen(message), 0);

    close(sock);
    return 0;
}
```

8. **Explain the role of Makefiles in managing complex C projects, especially those involving system communication components such as socket programming, shared memory, and semaphores. Illustrate with an example Makefile for a project that has multiple source files and dependencies. (3 marks)**

Makefiles are essential tools for managing the build process of complex C projects. They define a set of rules and dependencies, allowing developers to automate the compilation and linking of their programs. This is particularly important in projects that involve system communication components such as socket programming, shared memory, and semaphores for the following reasons:

- **Automation**: Makefiles automate the build process, so developers don't have to manually compile each source file every time changes are made.
- **Dependency Management**: They track file dependencies, ensuring that only modified files and those that depend on them are recompiled, which saves time.
- **Modularity**: Large projects often consist of multiple source files. Makefiles help manage these files efficiently, allowing developers to organize code into modules.
- **Consistency**: By defining build rules in a Makefile, teams can ensure that everyone builds the project in the same way, reducing errors and inconsistencies.

```
# Compiler and flags

CC = gcc

CFLAGS = -Wall -g

LDFLAGS = -lrt # Link with real-time library for shared memory


# Source files

SOURCES = main.c socket_comm.c shared_memory.c semaphore_control.c

# Object files

OBJECTS = $(SOURCES:.c=.o)


# Executable name

EXECUTABLE = my_project


# Default target

all: $(EXECUTABLE)


# Link the object files to create the executable

$(EXECUTABLE): $(OBJECTS)
```

$(CC) $(LDFLAGS) -o $@ $^

Compile source files to object files

.c.o:

$(CC) $(CFLAGS) -c $<

Clean target to remove object files and executable

clean:

rm -f $(OBJECTS) $(EXECUTABLE)

Phony targets

.PHONY: all clean

9. **Describe the process of writing a Makefile for a multi-module C project that involves system communication mechanisms like message queues and shared memory. Include rules for compilation, linking, and cleaning, and explain how dependencies are managed. (2marks)**

Creating a Makefile for a multi-module C project that involves system communication mechanisms like message queues and shared memory requires careful planning of dependencies, compilation, and linking. Below is a structured approach to writing such a Makefile, along with explanations of each part.

Example

Compiler and flags

CC = gcc

CFLAGS = -Wall -g

LDFLAGS = -lrt # Link with real-time library for shared memory

Source files

SOURCES = main.c message_queue.c shared_memory.c utils.c

Object files

```
OBJECTS = $(SOURCES:.c=.o)

# Executable name
EXECUTABLE = my_project

# Default target
all: $(EXECUTABLE)

# Link the object files to create the executable
$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) -o $@ $^

# Compile source files to object files
.c.o:
	$(CC) $(CFLAGS) -c $< -o $@

# Clean target to remove object files and executable
clean:
	rm -f $(OBJECTS) $(EXECUTABLE)

# Phony targets
.PHONY: all clean
```

10. **Discuss how Makefiles improve the efficiency of building C programs that involve multiple source files and external libraries for system communication. Provide an example scenario and demonstrate the structure of an appropriate Makefile. (1 mark)**

Makefiles significantly enhance the efficiency of building C programs, especially those involving multiple source files and external libraries for system communication, through several key mechanisms:

1. **Incremental Builds**: Makefiles only recompile files that have changed, along with any files that depend on them. This reduces the time spent recompiling unchanged code.
2. **Dependency Management**: Makefiles automatically track dependencies between source files and headers, ensuring that any changes trigger the necessary recompilation.
3. **Modularity**: Large projects can be broken down into smaller, manageable modules (source files), making it easier to develop and test individual components.

Example Scenario

Consider a project that implements a chat application using sockets (for communication) and utilizes shared memory for message storage. The project consists of the following files:

- main.c: The main application entry point.
- socket_comm.c: Manages socket communication.
- shared_memory.c: Handles shared memory operations.
- utils.c: Contains utility functions.
- Makefile: The build script.

Submission Deadline: Friday 15th August 2025 at 11:59 pm