**BIDWELL WARIO SCT221-0819/2022**

**SYSTEM PROGRAMING**

**Role of System Calls in C Programming for Inter-Process Communication (IPC)**

System calls in C link user apps to the OS core. In the frame of Inter-Process Chat, system calls let tasks talk & sync up. They do this by use of shared goods or bya send of notes.

**1. Pipes**

Pipes allow unidirectional communication between related processes (e.g., parent and child).

**Code Example:**

#include <stdio.h>

#include <unistd.h>

#include <string.h>

int main() {

int fd[2];

char buffer[20];

pipe(fd); // create pipe

if (fork() == 0) {

// Child process

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

read(fd[0], buffer, sizeof(buffer));

printf("Child read: %s\n", buffer);

} else {

// Parent process

close(fd[0]); // Close read end

char message[] = "Hello Child";

write(fd[1], message, strlen(message)+1);

}

return 0;

}

**2. Message Queues**

Message queues allow unrelated processes to send and receive structured messages.

**Code Example:**

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

#include <string.h>

struct msg\_buffer {

long msg\_type;

char msg\_text[100];

};

int main() {

key\_t key = ftok("progfile", 65);

int msgid = msgget(key, 0666 | IPC\_CREAT);

struct msg\_buffer message;

message.msg\_type = 1;

strcpy(message.msg\_text, "Hello via Message Queue");

msgsnd(msgid, &message, sizeof(message), 0);

printf("Message sent: %s\n", message.msg\_text);

return 0;

}

**3. Shared Memory**

Shared memory allows multiple processes to access the same memory segment.

**Code Example:**

#include <stdio.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, "Shared Memory Example");

printf("Data written to shared memory: %s\n", str);

shmdt(str); // detach

return 0;

}

**Client-Server Communication in C Using Socket Programming**

In C, socket code lets two parts talk through a net. They use the client-server style. The server sits for client links. The client starts the link. TCP (Data Send Rule) gives a sure, link-based chat path. It runs 'tween client & server.Bottom of Form

**TCP Server Code (server.c)**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

int main() {

int server\_fd, new\_socket;

struct sockaddr\_in address;

int addrlen = sizeof(address);

char buffer[BUFFER\_SIZE] = {0};

char \*greeting = "Hello from server!";

// 1. Create socket (IPv4, TCP)

server\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

if (server\_fd == 0) {

perror("Socket failed");

exit(EXIT\_FAILURE);

}

// 2. Bind socket to port and IP

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY; // Bind to any local address

address.sin\_port = htons(PORT); // Host-to-network short

if (bind(server\_fd, (struct sockaddr\*)&address, sizeof(address)) < 0) {

perror("Bind failed");

exit(EXIT\_FAILURE);

}

// 3. Listen for incoming connections

listen(server\_fd, 3);

printf("Server listening on port %d...\n", PORT);

// 4. Accept a client connection

new\_socket = accept(server\_fd, (struct sockaddr\*)&address, (socklen\_t\*)&addrlen);

if (new\_socket < 0) {

perror("Accept failed");

exit(EXIT\_FAILURE);

}

// 5. Read data from client

read(new\_socket, buffer, BUFFER\_SIZE);

printf("Message from client: %s\n", buffer);

// 6. Send response

send(new\_socket, greeting, strlen(greeting), 0);

printf("Greeting message sent\n");

// 7. Close socket

close(new\_socket);

close(server\_fd);

return 0;

}

**Client code**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

int main() {

int sock = 0;

struct sockaddr\_in serv\_addr;

char \*message = "Hello from client!";

char buffer[BUFFER\_SIZE] = {0};

// 1. Create socket

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

if (sock < 0) {

perror("Socket creation error");

return -1;

}

// 2. Define server address

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(PORT);

// Convert IPv4 and IPv6 addresses from text to binary

if (inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr) <= 0) {

perror("Invalid address/ Address not supported");

return -1;

}

// 3. Connect to server

if (connect(sock, (struct sockaddr\*)&serv\_addr, sizeof(serv\_addr)) < 0) {

perror("Connection Failed");

return -1;

}

// 4. Send message

send(sock, message, strlen(message), 0);

printf("Message sent to server\n");

// 5. Read response

read(sock, buffer, BUFFER\_SIZE);

printf("Message from server: %s\n", buffer);

// 6. Close socket

close(sock);

return 0;

}

**Concept of Signals in UNIX/Linux Systems**

In UNIX/Linux systems, signals are a form of inter-process communication (IPC) used to notify a process that a particular event has occurred. They are asynchronous — they can be sent at any time and interrupt the normal execution of a process.

**How Signals Are Used in C**

In C, signals are handled using:

* signal(int signum, sighandler\_t handler); – Registers a signal handler.
* kill(pid\_t pid, int sig); – Sends a signal to a process.
* raise(int sig); – Sends a signal to the current process.

**C Program: Signal Handling Example**

This program demonstrates:

* Catching SIGINT (Ctrl+C)
* Using raise() to generate a signal
* Custom signal handler

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

// Signal handler function

void handle\_sigint(int sig) {

printf("\nCaught signal %d (SIGINT). Cleaning up...\n", sig);

exit(0);

}

void handle\_sigusr1(int sig) {

printf("Received SIGUSR1 (signal %d) from self!\n", sig);

}

int main() {

// Register signal handlers

signal(SIGINT, handle\_sigint); // Ctrl+C

signal(SIGUSR1, handle\_sigusr1); // Custom signal

printf("Process ID: %d\n", getpid());

printf("Press Ctrl+C to trigger SIGINT or wait for SIGUSR1...\n");

// Generate a signal to self

sleep(2); // Pause before sending signal

raise(SIGUSR1);

// Infinite loop to keep program alive

while (1) {

printf("Running...\n");

sleep(2);

}

return 0;

}

**Difference Between Pipes, FIFO (Named Pipes), and Message Queues in C**

In C, pipes, FIFOs, & message queues let apps talk. Pipes, FIFOs, & message queues all get info to & from processes. Each has a form, size, & ease of use. They differ. All work in UNIX/Linux. They serve to swap data.

**Example Code:**

#include <stdio.h>

#include <unistd.h>

#include <string.h>

int main() {

int fd[2];

pid\_t pid;

char buffer[100];

pipe(fd); // Create pipe

pid = fork();

if (pid == 0) {

// Child reads from pipe

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

read(fd[0], buffer, sizeof(buffer));

printf("Child received: %s\n", buffer);

} else {

// Parent writes to pipe

close(fd[0]); // Close read end

char msg[] = "Hello from parent";

write(fd[1], msg, strlen(msg)+1);

}

return 0;

}

**FIFO (Named Pipe)**

Used for communication between **unrelated processes** via a **named file**.

**Sender (writer.c):**

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

#include <sys/stat.h>

int main() {

char \*fifo = "/tmp/myfifo";

mkfifo(fifo, 0666); // Create FIFO

int fd = open(fifo, O\_WRONLY);

char msg[] = "Hello from writer!";

write(fd, msg, strlen(msg)+1);

close(fd);

return 0;

}

**Receiver (reader.c):**

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/stat.h>

int main() {

char \*fifo = "/tmp/myfifo";

char buffer[100];

int fd = open(fifo, O\_RDONLY);

read(fd, buffer, sizeof(buffer));

printf("Reader received: %s\n", buffer);

close(fd);

return 0;

}

**3. MESSAGE QUEUES**

Allows structured communication with **message types**.

**Sender (msg\_send.c):**

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>

struct msg\_buffer {

long msg\_type;

char msg\_text[100];

};

int main() {

key\_t key = ftok("progfile", 65); // Generate key

int msgid = msgget(key, 0666 | IPC\_CREAT); // Create queue

struct msg\_buffer msg;

msg.msg\_type = 1;

strcpy(msg.msg\_text, "Hello from sender!");

msgsnd(msgid, &msg, sizeof(msg.msg\_text), 0);

printf("Message sent\n");

return 0;

}

**Receiver (msg\_recv.c):**

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/msg.h>

struct msg\_buffer {

long msg\_type;

char msg\_text[100];

};

int main() {

key\_t key = ftok("progfile", 65); // Same key

int msgid = msgget(key, 0666 | IPC\_CREAT);

struct msg\_buffer msg;

msgrcv(msgid, &msg, sizeof(msg.msg\_text), 1, 0);

printf("Message received: %s\n", msg.msg\_text);

// Delete queue

msgctl(msgid, IPC\_RMID, NULL);

return 0;

}

**Steps Involved in Implementing Shared Memory IPC**

1. **Create/Get shared memory segment**
   * Use shmget() with a unique key, size, and permission flags.
2. **Attach shared memory segment to process's address space**
   * Use shmat() to map the shared memory into the process's address space.
3. **Read/Write data to shared memory**
   * Processes can directly read from/write to the shared memory.
4. **Synchronize access**
   * Shared memory is just a block of memory—no synchronization.
   * Use **semaphores** or other synchronization primitives to avoid race conditions.
5. **Detach shared memory**
   * Use shmdt() when done accessing the memory.
6. **Remove shared memory segment**
   * Use shmctl() with IPC\_RMID to mark the segment for deletion.

**Synchronization Issues**

* Multiple processes can read/write simultaneously leading to **race conditions**.
* Need synchronization tools like **semaphores** or **mutexes** to ensure exclusive access.

.

**shared\_memory.c**

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

#define SHM\_SIZE 1024

typedef struct {

int ready; // Synchronization flag

char message[100];

} shared\_data;

int main() {

key\_t key = ftok("shmfile", 65); // Generate unique key

// 1. Create shared memory segment

int shmid = shmget(key, sizeof(shared\_data), 0666 | IPC\_CREAT);

if (shmid == -1) {

perror("shmget failed");

exit(1);

}

// 2. Attach to shared memory

shared\_data \*shm\_ptr = (shared\_data\*) shmat(shmid, NULL, 0);

if (shm\_ptr == (void\*) -1) {

perror("shmat failed");

exit(1);

}

// 3. Initialize synchronization flag

shm\_ptr->ready = 0;

if (fork() == 0) {

// Child process: waits until parent writes message

while (shm\_ptr->ready == 0) {

sleep(1);

}

printf("Child reads: %s\n", shm\_ptr->message);

// Detach and exit

shmdt(shm\_ptr);

exit(0);

} else {

// Parent process: write message

strcpy(shm\_ptr->message, "Hello from parent!");

shm\_ptr->ready = 1; // Signal child

wait(NULL); // Wait for child

// 4. Detach shared memory

shmdt(shm\_ptr);

// 5. Destroy shared memory

shmctl(shmid, IPC\_RMID, NULL);

}

return 0;

}

**Concept of Semaphores in Process Synchronization**

**What are Semaphores?**

Semaphores are sync tools used to block to shared things in side-by-side code work. They stop race issues. They let tasks wait (block) or go (let in). They guard key zones where shared data is got to.

**Kinds of Semaphores:**

**Binary Semaphore (lock):** Worth is 0 or 1 - used to keep turns.

**Counting Semaphore:** Sets count of free things.

**How semaphores work in C (System V IPC):**

* semget() — Create/get a semaphore set.
* semop() — Perform operations (P / wait / decrement or V / signal / increment).
* semctl() — Control semaphore operations (initialize, remove, etc.).

**Code: semaphore\_demo.c**

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/types.h>

#include <unistd.h>

#include <stdlib.h>

#include <wait.h>

union semun {

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

// Semaphore operations

struct sembuf p = {0, -1, 0}; // wait (P)

struct sembuf v = {0, 1, 0}; // signal (V)

void critical\_section(int process\_num) {

printf("Process %d entering critical section...\n", process\_num);

sleep(2); // Simulate work

printf("Process %d leaving critical section...\n", process\_num);

}

int main() {

key\_t key = ftok("semfile", 65); // Generate unique key

int semid = semget(key, 1, 0666 | IPC\_CREAT); // Create semaphore

// Initialize semaphore to 1 (available)

union semun u;

u.val = 1;

semctl(semid, 0, SETVAL, u);

pid\_t pid = fork();

if (pid < 0) {

perror("fork failed");

exit(1);

}

if (pid == 0) {

// Child process

semop(semid, &p, 1); // Wait (P)

critical\_section(2);

semop(semid, &v, 1); // Signal (V)

exit(0);

} else {

// Parent process

semop(semid, &p, 1); // Wait (P)

critical\_section(1);

semop(semid, &v, 1); // Signal (V)

wait(NULL); // Wait for child to finish

// Cleanup semaphore

semctl(semid, 0, IPC\_RMID);

}

return 0;

}

**Multiplexed I/O in Socket Programming: select() and poll()**

**How select() and poll() Help**

Both select() and poll() allow a program to:

* **Monitor multiple file descriptors (FDs)** at once.
* **Wait/block** until one or more FDs become ready for reading, writing, or have exceptional conditions.
* Avoid **busy-waiting** or continuously polling sockets inefficiently.
* Implement **event-driven** socket handling in a single-threaded process.

**select() System Call**

* **Interface:** Takes sets of FDs divided into **read**, **write**, and **exceptional** sets, plus a timeout.
* **How it works:**
  1. You initialize **fd\_sets** for the FDs you want to monitor.
  2. Call select(), which blocks until one or more FDs are ready or timeout expires.
  3. After select() returns, you check which FDs are ready by testing membership in fd\_sets.
* **Limitations:**
  1. Maximum number of FDs limited by FD\_SETSIZE (often 1024).
  2. Modifies fd\_sets, so you must reset them before each call.
  3. Slightly complex interface due to bitmask fd\_sets.

**poll() System Call**

* **Interface:** Uses an array of struct pollfd where each element specifies:
  + FD to monitor.
  + Events of interest (readable, writable, errors).
* **How it works:**
  + Populate array with FDs and desired events.
  + Call poll(), which blocks until any FD matches event or timeout expires.
  + After return, check revents field of each struct to see which events occurred.
* **Advantages over select():**
  + No fixed limit on number of FDs.
  + More scalable and easier to manage large numbers of descriptors.
  + Cleaner API without bitmask fiddling.

**Code**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <sys/select.h>

#define PORT 8080

#define MAX\_CLIENTS 30

#define BUFFER\_SIZE 1024

int main() {

int master\_socket, addrlen, new\_socket, client\_socket[MAX\_CLIENTS];

int max\_sd, sd, activity, i, valread;

struct sockaddr\_in address;

char buffer[BUFFER\_SIZE];

fd\_set readfds;

// Initialize all client\_socket[] to 0 (not connected)

for (i = 0; i < MAX\_CLIENTS; i++) client\_socket[i] = 0;

// Create master socket

if ((master\_socket = socket(AF\_INET, SOCK\_STREAM, 0)) == 0) {

perror("Socket failed");

exit(EXIT\_FAILURE);

}

// Prepare sockaddr\_in structure

address.sin\_family = AF\_INET;

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

address.sin\_port = htons(PORT);

// Bind

if (bind(master\_socket, (struct sockaddr\*)&address, sizeof(address)) < 0) {

perror("Bind failed");

exit(EXIT\_FAILURE);

}

printf("Listener on port %d \n", PORT);

// Listen

if (listen(master\_socket, 3) < 0) {

perror("Listen");

exit(EXIT\_FAILURE);

}

addrlen = sizeof(address);

puts("Waiting for connections ...");

while (1) {

// Clear fd set

FD\_ZERO(&readfds);

// Add master socket to set

FD\_SET(master\_socket, &readfds);

max\_sd = master\_socket;

// Add child sockets to set

for (i = 0; i < MAX\_CLIENTS; i++) {

sd = client\_socket[i];

if (sd > 0)

FD\_SET(sd, &readfds);

if (sd > max\_sd)

max\_sd = sd;

}

// Wait for activity on sockets (no timeout)

activity = select(max\_sd + 1, &readfds, NULL, NULL, NULL);

if ((activity < 0)) {

perror("select error");

}

// If something happened on master socket: new connection

if (FD\_ISSET(master\_socket, &readfds)) {

if ((new\_socket = accept(master\_socket,

(struct sockaddr\*)&address,

(socklen\_t\*)&addrlen)) < 0) {

perror("accept");

exit(EXIT\_FAILURE);

}

printf("New connection, socket fd is %d, IP: %s, Port: %d\n",

new\_socket,

inet\_ntoa(address.sin\_addr),

ntohs(address.sin\_port));

// Add new socket to array

for (i = 0; i < MAX\_CLIENTS; i++) {

if (client\_socket[i] == 0) {

client\_socket[i] = new\_socket;

printf("Added to list of sockets as %d\n", i);

break;

}

}

}

// Check all clients for incoming data

for (i = 0; i < MAX\_CLIENTS; i++) {

sd = client\_socket[i];

if (FD\_ISSET(sd, &readfds)) {

// Check if it was for closing, and read data

if ((valread = read(sd, buffer, BUFFER\_SIZE)) == 0) {

// Client disconnected

getpeername(sd, (struct sockaddr\*)&address,

(socklen\_t\*)&addrlen);

printf("Host disconnected, IP %s, Port %d\n",

inet\_ntoa(address.sin\_addr),

ntohs(address.sin\_port));

close(sd);

client\_socket[i] = 0;

} else {

// Echo back message

buffer[valread] = '\0';

printf("Received from client %d: %s\n", sd, buffer);

send(sd, buffer, valread, 0);

}

}

}

}

return 0;

}

**UNIX Domain Sockets for Interprocess Communication (IPC) in C**

**What are UNIX Domain Sockets?**

* UNIX Domain Sockets (UDS) allow **fast and efficient communication** between **processes on the same machine**.
* Unlike TCP/IP sockets, UDS use a **file path** as the socket address (e.g., /tmp/mysocket).
* They support both:
  + **Stream-based** (like TCP)
  + **Datagram-based** (like UDP)

**Steps to Establish Communication:**

**Server:**

1. Create a socket with socket(AF\_UNIX, SOCK\_STREAM, 0)
2. Set up a sockaddr\_un structure with a **file path**.
3. Bind the socket to that path using bind().
4. Listen for connections using listen().
5. Accept connections with accept().
6. Read/write data using read() and write().

**Client:**

1. Create a socket.
2. Set up a sockaddr\_un structure with the same path.
3. Connect to the server using connect().
4. Read/write data.

unix\_socket\_server.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <unistd.h>

#include <string.h>

#define SOCKET\_PATH "/tmp/unix\_socket"

int main() {

int server\_fd, client\_fd;

struct sockaddr\_un addr;

char buffer[100];

// 1. Create socket

if ((server\_fd = socket(AF\_UNIX, SOCK\_STREAM, 0)) < 0) {

perror("Socket failed");

exit(EXIT\_FAILURE);

}

// 2. Set up address

memset(&addr, 0, sizeof(addr));

addr.sun\_family = AF\_UNIX;

strncpy(addr.sun\_path, SOCKET\_PATH, sizeof(addr.sun\_path) - 1);

// Remove the socket file if it already exists

unlink(SOCKET\_PATH);

// 3. Bind

if (bind(server\_fd, (struct sockaddr\*)&addr, sizeof(addr)) < 0) {

perror("Bind failed");

exit(EXIT\_FAILURE);

}

// 4. Listen

listen(server\_fd, 5);

printf("Server listening on %s...\n", SOCKET\_PATH);

// 5. Accept

client\_fd = accept(server\_fd, NULL, NULL);

if (client\_fd == -1) {

perror("Accept failed");

exit(EXIT\_FAILURE);

}

// 6. Read and respond

read(client\_fd, buffer, sizeof(buffer));

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

char reply[] = "Hello from server!";

write(client\_fd, reply, strlen(reply) + 1);

// Cleanup

close(client\_fd);

close(server\_fd);

unlink(SOCKET\_PATH);

return 0;

}

unix\_socket\_client.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <unistd.h>

#include <string.h>

#define SOCKET\_PATH "/tmp/unix\_socket"

int main() {

int sock;

struct sockaddr\_un addr;

char buffer[100];

// 1. Create socket

if ((sock = socket(AF\_UNIX, SOCK\_STREAM, 0)) < 0) {

perror("Socket failed");

exit(EXIT\_FAILURE);

}

// 2. Set up address

memset(&addr, 0, sizeof(addr));

addr.sun\_family = AF\_UNIX;

strncpy(addr.sun\_path, SOCKET\_PATH, sizeof(addr.sun\_path) - 1);

// 3. Connect to server

if (connect(sock, (struct sockaddr\*)&addr, sizeof(addr)) < 0) {

perror("Connect failed");

exit(EXIT\_FAILURE);

}

// 4. Send message

char \*msg = "Hello from client!";

write(sock, msg, strlen(msg) + 1);

// 5. Read response

read(sock, buffer, sizeof(buffer));

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

close(sock);

return 0;

}

**Benefits of Makefiles in System Communication Projects:**

| **Feature** | **Benefit** |
| --- | --- |
| **Automation** | Avoid manually running gcc multiple times |
| **Dependency management** | Recompiles only changed files |
| **Separation of concerns** | Organize code into modules (e.g., socket.c, shm.c, sem.c) |
| **Ease of building** | Simple make builds entire project |
| **Custom targets** | e.g., make clean, make debug, etc. |

**Example Project Structure**

my\_ipc\_project/

├── Makefile

├── main.c # Entry point

├── socket\_comm.c # Socket communication code

├── socket\_comm.h

├── shared\_mem.c # Shared memory logic

├── shared\_mem.h

├── semaphore.c # Semaphore logic

├── semaphore.h

**Example Makefile**

# Compiler and flags

CC = gcc

CFLAGS = -Wall -g

# Output executable

TARGET = ipc\_app

# Source files

SRCS = main.c socket\_comm.c shared\_mem.c semaphore.c

# Header files

HDRS = socket\_comm.h shared\_mem.h semaphore.h

# Object files

OBJS = $(SRCS:.c=.o)

# Default target

all: $(TARGET)

# Link object files to create final executable

$(TARGET): $(OBJS)

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

# Compile .c files into .o files

%.o: %.c $(HDRS)

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

# Clean compiled files

clean:

rm -f $(TARGET) \*.o

# Optional: Debug target

debug: CFLAGS += -DDEBUG

debug: clean all

.PHONY: all clean debug

**Writing a Makefile for a Multi-Module C Project with Message Queues & Shared Memory**

In a C project that uses **system communication mechanisms** like **message queues** and **shared memory**, the code is typically split across multiple modules for:

* Modularity (e.g., msgqueue.c, shared\_mem.c)
* Reusability
* Easier testing and debugging

A Makefile automates:

* Compiling each module separately
* Linking object files into one executable
* Rebuilding only what’s changed
* Cleaning up generated files

**Example Project Structure**

ipc\_project/

├── Makefile

├── main.c # Main application

├── msgqueue.c # Message queue implementation

├── msgqueue.h

├── shared\_mem.c # Shared memory implementation

├── shared\_mem.h

**Makefile with Compilation, Linking, and Cleaning Rules**

# Compiler and flags

CC = gcc

CFLAGS = -Wall -g

# Output executable name

TARGET = ipc\_app

# Source and header files

SRCS = main.c msgqueue.c shared\_mem.c

HDRS = msgqueue.h shared\_mem.h

# Object files

OBJS = $(SRCS:.c=.o)

# Default rule: build the executable

all: $(TARGET)

# Link object files to create executable

$(TARGET): $(OBJS)

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

# Rule to compile .c files into .o files

%.o: %.c %.h

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

# Clean rule: remove object files and executable

clean:

rm -f $(TARGET) \*.o

# Dependency rule (optional but useful)

depend:

$(CC) -MM $(SRCS) > .depend

-include .depend

.PHONY: all clean depend

How Makefiles Improve Efficiency in Multi-Source C Projects with System Communication

Makefiles Help By:

Benefit Description

Automatic builds Avoids retyping complex gcc commands

Incremental compilation Only recompiles changed files, saving time

Dependency tracking Ensures updated headers trigger rebuilds

Modular development Allows clean separation of code into components

Consistent linking Handles linking with external libraries (-lrt, -lpthread, etc.)

Example Makefile

# Compiler and flags

CC = gcc

CFLAGS = -Wall -g

# Output binary

TARGET = chat\_app

# External libraries

LIBS = -lrt -lpthread

# Source and object files

SRCS = main.c socket\_comm.c msg\_queue.c shared\_mem.c

OBJS = $(SRCS:.c=.o)

# Header files (for dependency tracking)

HDRS = socket\_comm.h msg\_queue.h shared\_mem.h

# Default target: build everything

all: $(TARGET)

# Link object files into final binary

$(TARGET): $(OBJS)

$(CC) $(CFLAGS) -o $@ $^ $(LIBS)

# Compile each .c into .o, include headers to track changes

%.o: %.c $(HDRS)

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

# Clean up compiled files

clean:

rm -f $(TARGET) \*.o

.PHONY: all clean