Hospital Management System with Distributed Donor Database

Technical Documentation

October 28, 2025

Contents

1 Project Overview

1.1 Introduction

A distributed, multi-process hospital management system implementing real-time patient monitoring (up to 5 concurrent patients), remote donor database management via TCP sockets, and automated critical care response with intelligent donor matching.

1.2 Core Objectives

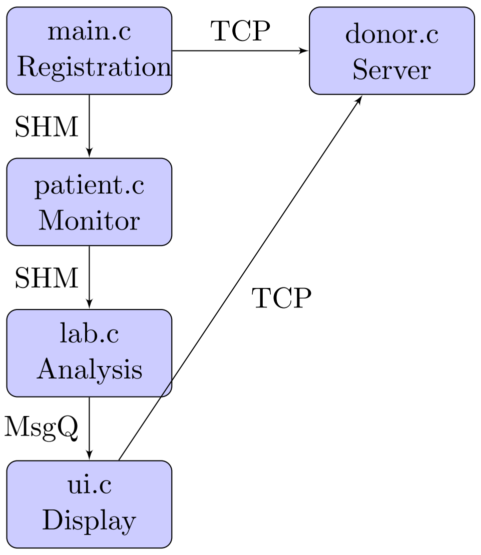
- Automate patient registration and bed management
- Monitor vital signs in real-time using multi-threading
- Detect critical health conditions automatically
- Manage distributed donor database across network
- Provide automated donor matching and discharge workflow
- Offer web-based dashboards for monitoring

1.3 Technologies

- **Language**: C (POSIX-compliant)
- **IPC**: Shared Memory, Message Queues, Semaphores
- **Network**: TCP Sockets (IPv4)
- **Threading**: POSIX pthreads
- **UI**: ncurses (terminal), HTTP (web)
- **OS**: Linux (Ubuntu/Debian/Fedora/Arch)

2 System Architecture

2.1 Component Diagram



2.2 Communication Mechanisms

- **Shared Memory (Key: 123456)**: Patient registration data
- **Shared Memory (Key: 12345678)**: Real-time vital signs
- **Message Queue (Key: 789)**: Status messages (lab $\rightarrow$ ui)
- **Message Queue (Key: 256)**: Donor database (donor server)
- **Semaphore (/semos)**: Patient registration signal
- **TCP Socket (Port: 8080)**: Donor operations
- **HTTP Server (Port: 8081)**: Patient vitals dashboard
- **HTTP Server (Port: 8082)**: Lab results dashboard

3 File-by-File Explanation

3.1 headers.h - Common Definitions

Purpose: Centralized header file containing all data structures, constants, and library includes shared across components.

Key Constants:

```
#define MAX_PATIENTS 5          // Bed capacity
#define MSG_KEY 789             // Status message queue
#define PORT 8080               // Donor server port
#define SERVER_IP "192.168.1.140"  // Donor server IP
```

Critical Data Structures:

```
// Patient registration (SHM: 123456)
typedef struct persons {
    char name[20];
    int age;
    char bg[5];              // Blood group
    int active;              // 1=active, 0=discharged
} person;

// Patient vitals (SHM: 12345678)
typedef struct PatientDetails {
    int age;
    float hr;                // Heart rate (60-140 bpm)
    float o2;                // Oxygen (88-100%)
    char blood_group[5];
    blood_pressure bp;       // Systolic/Diastolic
    char name[20];
    int active;
    int slot_id;             // Bed number (0-4)
} patient;

// Donor socket communication
typedef struct {
    int type;                // 1=Add, 2=Search
    char name[40];
    int age;
    char blood_group[5];
} DonorRequest;

typedef struct {
    int found;               // 1=found, 0=not found
    char name[40];
    int age;
    char blood_group[5];
} DonorResponse;

// Lab to UI message (MsgQ: 789)
typedef struct {
    long mtype;              // Message type (required)
    int slot_id;
    char name[20];
    char blood_group[5];
    int is_critical;         // 1=critical, 0=stable
    char message[100];
} StatusMessage;
```

Included Libraries: stdio, stdlib, string, pthread, semaphore, sys/ipc, sys/shm, sys/msg, sys/socket, netinet/in, arpa/inet, fcntl, unistd, ncurses, wchar, locale.

3.2 donor.c - Remote Donor Server

Purpose: TCP server managing blood donor database using message queue for fast blood-group-based retrieval and binary file for persistence.

Architecture: Runs independently on separate device, listens on port 8080, handles concurrent client connections.

Key Functions:

3.2.1 add_donor(DonorRequest *req)

```c
void add_donor(DonorRequest *req) {
    donor d;
    strcpy(d.name, req->name);
    d.age = req->age;
    strcpy(d.blood_group, req->blood_group);

    // Map blood group to message type (1-8)
    char* arr[8] = {"O+", "O-", "A+", "A-",
                    "B+", "B-", "AB+", "AB-"};
    for(int i = 0; i < 8; i++) {
        if(strcmp(arr[i], d.blood_group) == 0) {
            d.mtype = i + 1;  // O+=1, O-=2, ..., AB-=8
            break;
        }
    }

    // Add to message queue (in-memory, fast search)
    msgsnd(msgid, &d, sizeof(donor) - sizeof(long), 0);

    // Persist to binary file
    lseek(fd, 0, SEEK_END);
    write(fd, &d, sizeof(donor));
}
```

How it works:

1. Receives donor details from client via socket
2. Assigns message type based on blood group (1-8)
3. Stores in message queue for fast retrieval
4. Appends to donor.bin file for persistence
5. Message type enables efficient blood-group filtering

3.2.2 search_donor(char* blood_group)

```c
DonorResponse search_donor(char* blood_group) {
    DonorResponse response;
    response.found = 0;
    donor h;
    int type = 0;

    // Convert blood group to message type
    char* arr[8] = {"O+", "O-", "A+", "A-",
                    "B+", "B-", "AB+", "AB-"};
    for(int i = 0; i < 8; i++) {
        if(strcmp(arr[i], blood_group) == 0) {
            type = i + 1;
            break;
        }
    }

    // Search by message type (blood group filter)
    if(msgrcv(msgid, &h, sizeof(donor) - sizeof(long),
              type, IPC_NOWAIT) != -1) {
        response.found = 1;
        strcpy(response.name, h.name);
        response.age = h.age;
        strcpy(response.blood_group, h.blood_group);
    }

    return response;
}
```

Critical Detail: msgrcv() with specific type automatically removes the message from queue, acting as donor removal after match.

3.2.3 main() - Server Loop

```c
int main() {
    // Initialize message queue (Key: 256)
    msgid = msgget((key_t)KEY_MSG, IPC_CREAT | 0777);

    // Open persistent storage
    fd = open("donor.bin", O_CREAT | O_RDWR | O_APPEND, 0777);

    // Create TCP socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    // Bind to port 8080, all interfaces
    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, 5);

    while(1) {
        client_socket = accept(server_fd, ...);

        DonorRequest req;
        read(client_socket, &req, sizeof(DonorRequest));

        if(req.type == 1) {
            add_donor(&req);
            int success = 1;
            write(client_socket, &success, sizeof(int));
        } else if(req.type == 2) {
            DonorResponse response = search_donor(req.blood_group);
            write(client_socket, &response, sizeof(DonorResponse));
        }

        close(client_socket);
    }
}
```

Flow: Accept connection → Read request → Route to add/search → Send response → Close connection.

3.3 main.c - Registration Interface

Purpose: ncurses-based terminal UI for registering donors and patients with dynamic bed availability tracking.

Key Functions:

3.3.1 connect_to_donor_server()

```c
int connect_to_donor_server() {
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, SERVER_IP, &serv_addr.sin_addr);

    if(connect(sock, (struct sockaddr *)&serv_addr,
               sizeof(serv_addr)) < 0) {
        close(sock);
        return -1;
    }
    return sock;
}
```

Purpose: Establishes TCP connection to donor server for donor registration.

3.3.2 add_donor_to_server()

```c
void add_donor_to_server(char* name, int age, char* blood_group) {
    int sock = connect_to_donor_server();

    DonorRequest req;
    req.type = 1;  // Add operation
    strcpy(req.name, name);
    req.age = age;
    strcpy(req.blood_group, blood_group);

    write(sock, &req, sizeof(DonorRequest));

    int success;
    read(sock, &success, sizeof(int));

    close(sock);
}
```

Flow: Connect → Send DonorRequest → Wait for confirmation → Close.

3.3.3 count_available_beds()

```c
int count_available_beds(person* shared) {
    int available = 0;
    for(int i = 0; i < MAX_PATIENTS; i++) {
        if(shared[i].active == 0) {
            available++;
        }
    }
    return available;
}
```

Purpose: Real-time bed availability calculation by checking shared memory.

3.3.4 main() - Registration Loop

```c
int main() {
    setlocale(LC_ALL,"");  // UTF-8 for emojis

    // Initialize ncurses
    initscr();
    cbreak();
    noecho();
    curs_set(1);

    WINDOW *main_win = newwin(20, 60, starty, startx);

    // Create semaphore
    sem = sem_open("/semos", O_CREAT, 0666, 0);

    // Create shared memory for patient registration
    shmid = shmget((key_t)123456, sizeof(person) * MAX_PATIENTS,
                   IPC_CREAT | 0777);
    shared = (person*)shmat(shmid, NULL, 0);
    memset(shared, 0, sizeof(person) * MAX_PATIENTS);

    while(1) {
        int beds_available = count_available_beds(shared);
        draw_ui(main_win, beds_available);

        int choice;
        wscanw(main_win, "%d", &choice);

        if(choice == 1) {
            // Donor Registration
            // Get donor details via wscanw()
            add_donor_to_server(p.name, p.age, p.bg);
        }
        else if(choice == 2) {
            // Patient Registration
            if(beds_available == 0) {
                mvwprintw(main_win, 4, 2, "BEDS␣ARE␣FULL!");
                continue;
            }

            // Get patient details
            p.active = 1;

            // Find first empty slot
            for(int i = 0; i < MAX_PATIENTS; i++) {
                if(shared[i].active == 0) {
                    memcpy(&shared[i], &p, sizeof(person));
                    sem_post(sem);  // Signal patient.c
                    break;
                }
            }
        }
    }

    // Cleanup
    endwin();
    shmdt(shared);
    sem_close(sem);
}
```

Workflow:

1. Display menu with bed availability

2. Choice 1: Register donor → Send to donor server via TCP
3. Choice 2: Register patient → Store in SHM + post semaphore
4. Enforce bed capacity limit (max 5 patients)

3.4 patient.c - Vital Signs Monitor

Purpose: Multi-threaded application generating real-time patient vitals with embedded HTTP server.

Architecture: 3 threads per patient (HR, BP, O2), web server thread, main monitoring loop.

Thread Functions:

```
// Heart Rate Thread
void* heart_rate(void* arg) {
    *((float*)arg) = 60 + rand() % 80;  // 60-140 bpm
    pthread_exit(NULL);
}

// Blood Pressure Thread
void* bloodPressure(void* arg) {
    blood_pressure* bp = ((blood_pressure*)arg);
    bp->systollic = 90 + rand() % 60;   // 90-150 mmHg
    bp->diastollic = 60 + rand() % 40;  // 60-100 mmHg
    pthread_exit(NULL);
}

// Oxygen Level Thread
void* oxygen_level(void* arg) {
    *((float*)arg) = 88 + rand() % 13;  // 88-100%
    pthread_exit(NULL);
}
```

Purpose: Generate realistic vital signs in parallel, simulating real medical devices.

3.4.1 web_server() - HTTP Dashboard

```c
void* web_server(void* arg) {
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_port = htons(WEB_PORT);  // 8081
    bind(server_fd, (struct sockaddr*)&address, sizeof(address));
    listen(server_fd, 5);

    while(1) {
        client_sock = accept(server_fd, ...);

        // Build HTML response
        char html[4096] = "HTTP/1.1␣200␣OK\r\n"
                          "Content-Type:␣text/html\r\n\r\n"
                          "<meta␣http-equiv='refresh'␣content='3'>"
                          "<table␣border='1'>"
                          "<tr><th>Slot</th><th>Name</th>...";

        // Add patient data rows
        for(int i = 0; i < MAX_PATIENTS; i++) {
            if(s_global[i].active) {
                sprintf(html + strlen(html),
                    "<tr><td>%d</td><td>%s</td>...</tr>",
                    i, s_global[i].name, ...);
            }
        }

        write(client_sock, html, strlen(html));
        close(client_sock);
    }
}
```

Features: Auto-refresh every 3 seconds, displays real-time vitals in HTML table.

3.4.2 main() - Monitoring Loop

```c
int main() {
    // Open semaphore (wait for patient registration)
    sem = sem_open("/semos", O_CREAT, 0666, 0);

    // Shared memory for vitals (Key: 12345678)
    shmid = shmget(12345678, sizeof(patient) * MAX_PATIENTS,
                   IPC_CREAT | 0666);
    s = (patient*)shmat(shmid, NULL, 0);
    s_global = s;  // For web server access

    // Start web server thread
    pthread_create(&web_thread, NULL, web_server, NULL);
    pthread_detach(web_thread);

    // Shared memory for registration (Key: 123456)
    shmid_reg = shmget(123456, sizeof(person) * MAX_PATIENTS,
                       IPC_CREAT | 0666);
    patient_reg = (person*)shmat(shmid_reg, NULL, 0);

    while(1) {
        sem_wait(sem);  // Wait for new patient signal

        // Find newly registered patient
        for(int i = 0; i < MAX_PATIENTS; i++) {
            if(patient_reg[i].active == 1 && s[i].active == 0) {
                // Copy registration data to vitals structure
                strcpy(s[i].name, patient_reg[i].name);
                s[i].age = patient_reg[i].age;
                strcpy(s[i].blood_group, patient_reg[i].bg);
                s[i].active = 1;
                s[i].slot_id = i;
                break;
            }
        }

        // Update vitals for all active patients
        for(int i = 0; i < MAX_PATIENTS; i++) {
            if(s[i].active == 1) {
                pthread_t t1, t2, t3;

                // Spawn 3 threads per patient
                pthread_create(&t1, NULL, heart_rate, &s[i].hr);
                pthread_create(&t2, NULL, bloodPressure, &s[i].bp);
                pthread_create(&t3, NULL, oxygen_level, &s[i].o2);

                // Wait for all threads
                pthread_join(t1, NULL);
                pthread_join(t2, NULL);
                pthread_join(t3, NULL);
            }
        }

        sleep(3);  // Update every 3 seconds
    }
}
```

Workflow:

1. Wait on semaphore for patient registration
2. Copy registration data from SHM:123456 to SHM:12345678
3. For each active patient: Spawn 3 threads for vitals
4. Join threads (ensures all vitals updated)
5. Sleep 3 seconds, repeat

3.5 lab.c - Laboratory Analysis

Purpose: Analyzes patient vitals to detect critical conditions, sends status messages via message queue.

Critical Condition Rules:

- HR >120 AND HB <8
- BP <90/65 AND HB <8
- O2 <90% AND HB <8

3.5.1 main() - Analysis Loop

```
int main() {
    // Shared memory for vitals
    shmid = shmget(12345678, sizeof(patient) * MAX_PATIENTS,
                   IPC_CREAT | 0666);
    patients = (patient*)shmat(shmid, NULL, 0);

    // Message queue for status
    msgid = msgget((key_t)MSG_KEY, IPC_CREAT | 0777);

    // Start web server thread (port 8082)
    pthread_create(&web_thread, NULL, web_server, NULL);
    pthread_detach(web_thread);

    while(1) {
        for(int i = 0; i < MAX_PATIENTS; i++) {
            if(patients[i].active == 1) {
                // Generate random hemoglobin
                int hb = 6 + rand() % 10;  // 6-15 g/dL

                int is_critical = 0;
                char status[100];

                // Critical condition detection
                if(patients[i].hr > 120 && hb < 8) {
                    is_critical = 1;
                    sprintf(status, "CRITICAL: High HR=%.0f, Low HB=%d",
                            patients[i].hr, hb);
                }
                else if(patients[i].bp.systollic < 90 ||
                        patients[i].bp.diastollic < 65) {
                    if(hb < 8) {
                        is_critical = 1;
                        sprintf(status, "CRITICAL: Low BP, Low HB");
                    }
                }
                else if(patients[i].o2 < 90 && hb < 8) {
                    is_critical = 1;
                    sprintf(status, "CRITICAL:␣Low␣O2,␣Low␣HB");
                }
                else {
                    sprintf(status, "STABLE:␣All␣vitals␣normal");
                }

                // Store in global array for web dashboard
                lab_results[i].slot_id = i;
                lab_results[i].hb_level = hb;
                lab_results[i].is_critical = is_critical;

                // Send status message to ui.c
                StatusMessage msg;
                msg.mtype = 1;
                msg.slot_id = i;
                strcpy(msg.name, patients[i].name);
                strcpy(msg.blood_group, patients[i].blood_group);
                msg.is_critical = is_critical;
                strcpy(msg.message, status);

                msgsnd(msgid, &msg, sizeof(StatusMessage) - sizeof(long), 0);
            }
        }

        sleep(3);  // Analyze every 3 seconds
    }
}
```

Flow: Read vitals → Generate HB → Apply rules → Send StatusMessage → Repeat.

3.6 ui.c - Display Interface

Purpose: Three-panel ncurses UI displaying vitals, status, and donor search results with automated workflow.

Architecture: Top panel (vitals), middle panel (status), bottom panel (donor), main event loop.

Key Functions:

3.6.1 search_donor()

```
DonorResponse search_donor(char* blood_group) {
    DonorResponse response;
    response.found = 0;

    int sock = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, SERVER_IP, &serv_addr.sin_addr);

    connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    // Send search request
    DonorRequest req;
    req.type = 2;  // Search operation
    strcpy(req.blood_group, blood_group);

    write(sock, &req, sizeof(DonorRequest));
    read(sock, &response, sizeof(DonorResponse));
    close(sock);
    return response;
}
```

3.6.2 draw_vitals_box(), draw_status_box(), draw_donor_box()

Three separate functions to render each panel using ncurses. Color coding: Red (critical), Green (stable).

3.6.3 main() - Automated Workflow

```
int main() {
    // Shared memory for vitals
    shmid = shmget(12345678, sizeof(patient) * MAX_PATIENTS,
                   IPC_CREAT | 0666);
    patients = (patient*)shmat(shmid, NULL, 0);

    // Message queue for status
    msgid = msgget((key_t)MSG_KEY, IPC_CREAT | 0777);

    // Shared memory for registration (to discharge)
    shmid_reg = shmget(123456, sizeof(person) * MAX_PATIENTS,
                       IPC_CREAT | 0666);
    patient_reg = (person*)shmat(shmid_reg, NULL, 0);

    // Initialize ncurses with colors
    initscr();
    start_color();
    init_pair(1, COLOR_RED, COLOR_BLACK);
    init_pair(2, COLOR_GREEN, COLOR_BLACK);

    // Create 3 windows
    WINDOW *vitals_win = newwin(vitals_height, width, 0, 0);
    WINDOW *status_win = newwin(status_height, width, vitals_height, 0);
    WINDOW *donor_win = newwin(donor_height, width,
                               vitals_height + status_height, 0);

    int critical_patient_slot = -1;

    while(1) {
        // Display vitals
        draw_vitals_box(vitals_win, patients);

        // Check for status messages
        StatusMessage msg;
        if(msgrcv(msgid, &msg, sizeof(StatusMessage) - sizeof(long),
                  0, IPC_NOWAIT) != -1) {

            // Update status array
            patient_statuses[msg.slot_id].has_status = 1;
            patient_statuses[msg.slot_id].is_critical = msg.is_critical;
            strcpy(patient_statuses[msg.slot_id].message, msg.message);

            // AUTOMATED CRITICAL CARE WORKFLOW
            if(msg.is_critical && critical_patient_slot != msg.slot_id) {
                critical_patient_slot = msg.slot_id;

                // Step 1: Show "Searching..."
                searching = 1;
                draw_donor_box(donor_win, &donor_response, has_donor, searching
                    );
                sleep(1);

                // Step 2: Search donor via TCP
                donor_response = search_donor(msg.blood_group);
                searching = 0;
                has_donor = 1;

                // Step 3: If found, discharge patient
                if(donor_response.found) {
                    patients[msg.slot_id].active = 0;
                    patient_reg[msg.slot_id].active = 0;
                    patient_statuses[msg.slot_id].has_status = 0;
                    critical_patient_slot = -1;
                }
            }

            draw_status_box(status_win, patients, patient_statuses);
            draw_donor_box(donor_win, &donor_response, has_donor, searching);
        }

        // Check for 'q' to quit
        int ch = getch();
        if(ch == 'q' || ch == 'Q') break;

        usleep(500000);  // 500ms refresh rate
    }

    // Cleanup
    endwin();
    shmdt(patients);
    shmdt(patient_reg);
}
```

Automated Workflow:

1. Receive critical status message from lab.c
2. Display "Searching for donor..."
3. Connect to donor server via TCP
4. Send DonorRequest (type=2, blood group)
5. Receive DonorResponse
6. If donor found:
   - Set patients[slot].active = 0 (stop monitoring)
   - Set patient_reg[slot].active = 0 (free bed)
   - Display donor details
   - Patient discharged
7. If not found: Display "No donor available"

3.7 Makefile - Build Automation

```
CC = gcc
CFLAGS = -pthread -lrt -lncursesw
TARGETS = main patient lab ui

all: $(TARGETS)

main: main.c
        $(CC) main.c -o main $(CFLAGS)

patient: patient.c
        $(CC) patient.c -o patient -pthread -lrt

lab: lab.c
        $(CC) lab.c -o lab -lrt

ui: ui.c
        $(CC) ui.c -o ui $(CFLAGS)

clean:
        rm -f $(TARGETS)
        ipcrm -a

stop:
        pkill -f "./patient"
        pkill -f "./lab"
        pkill -f "./ui"
        ipcrm -a
```

Commands:

- `make all`: Compile all components
- `make clean`: Remove executables + IPC resources
- `make stop`: Kill all processes + cleanup

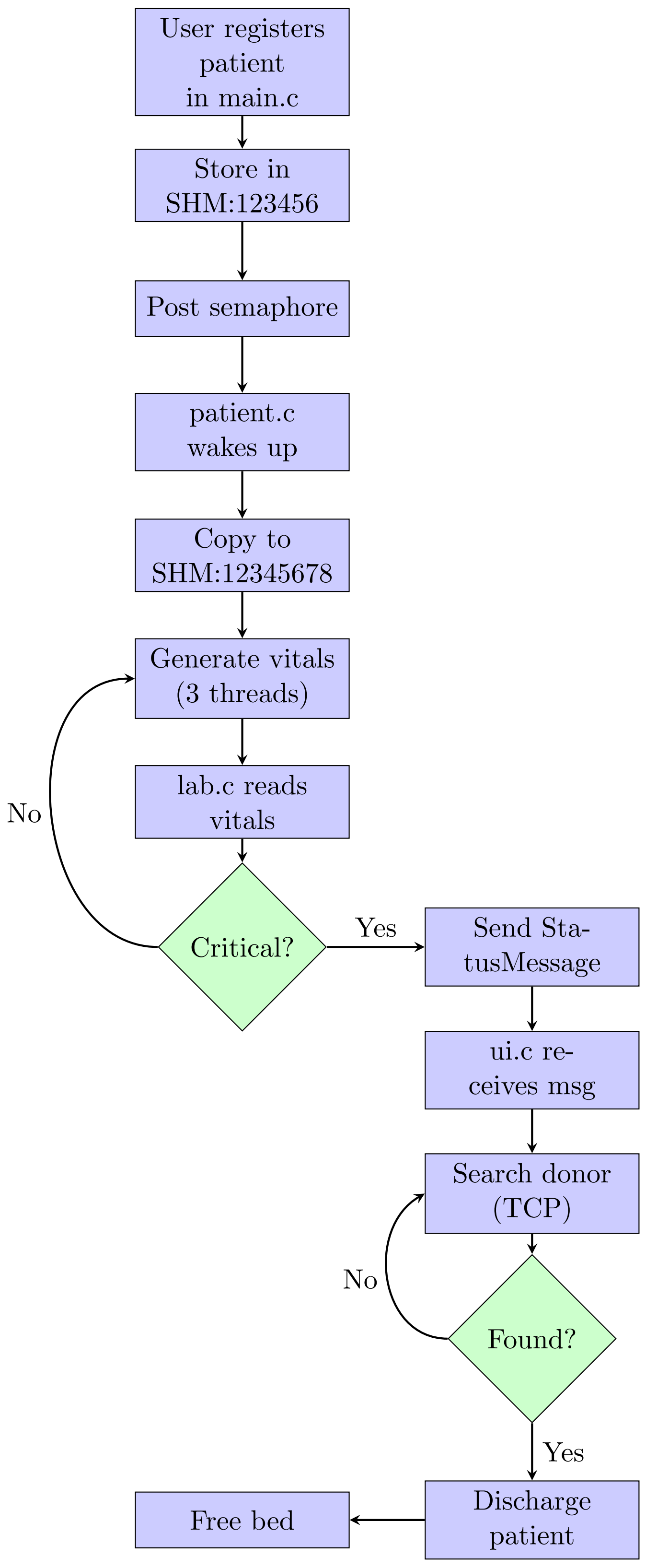
3.8 main.sh - Launch Script

```
#!/bin/bash
echo "Starting␣Hospital␣Management␣System..."

gnome-terminal -- bash -c "./patient;␣exec␣bash" &
sleep 2
gnome-terminal -- bash -c "./lab;␣exec␣bash" &
sleep 2
gnome-terminal -- bash -c "./ui;␣exec␣bash" &
sleep 2

echo "All␣services␣started!"
```

Purpose: Automates launching all components in separate terminals with 2-second delays.

4 Overall System Flow

4.1 Complete Workflow Diagram



4.2 Data Flow Summary

1. **Registration**: main.c → SHM:123456 → Semaphore
2. **Monitoring**: patient.c waits on semaphore → Copies to SHM:12345678 → Generates vitals every 3s
3. **Analysis**: lab.c reads SHM:12345678 → Detects critical → Sends to MsgQ:789
4. **Display**: ui.c reads MsgQ:789 → Displays in ncurses
5. **Critical Response**: ui.c → TCP to donor server → Receives donor → Discharges patient
6. **Bed Release**: ui.c sets active=0 in both SHMs → Bed available in main.c

4.3 Timing Sequence

- **t=0s**: Patient registered
- **t=0-3s**: patient.c generates initial vitals
- **t=3s**: lab.c analyzes (first analysis)
- **t=3s**: ui.c receives status message
- **t=3s**: If critical, donor search initiated (1-2s TCP roundtrip)
- **t=5s**: Donor found, patient discharged
- **t=5s**: Bed becomes available

Total Response Time: Patient critical detection to discharge: ˜2-5 seconds.

5 Key Technical Achievements

- **Distributed Architecture**: Donor server independently deployable across networks
- **Multi-threading**: 15 threads (3 per patient × 5 patients) for parallel vital generation
- **IPC Mastery**: 2 shared memories, 2 message queues, 1 semaphore working in coordination
- **Network Programming**: TCP client-server with custom protocol (DonorRequest/Response)
- **Real-time Processing**: Sub-second critical detection, 3-second refresh rate
- **Resource Management**: Dynamic bed allocation, automatic donor removal, IPC cleanup
- **User Interface**: Professional ncurses UI with UTF-8 emojis, HTTP dashboards
- **Automation**: Zero-touch critical care workflow from detection to discharge

6 Compilation and Execution

6.1 Prerequisites

```
# Ubuntu/Debian
sudo apt-get install build-essential libncurses5-dev libncursesw5-dev

# Compile
make all

# Or individually
gcc donor.c -o donor -pthread
gcc main.c -o main -pthread -lrt -lncursesw
gcc patient.c -o patient -pthread -lrt
gcc lab.c -o lab -lrt
gcc ui.c -o ui -pthread -lrt -lncursesw
```

6.2 Running the System

Device 1 (Donor Server):

```
./donor
```

Device 2 (Main System):

```
# Separate terminals
./patient &
./lab &
./ui &
./main

# Or use automation script
./main.sh
```

Access Dashboards:

- Patient Vitals: `http://localhost:8081`
- Lab Results: `http://localhost:8082`

7 Conclusion

This Hospital Management System demonstrates comprehensive mastery of systems programming concepts including distributed architecture, inter-process communication, multi-threading, network programming, and real-time data processing. The automated workflow from patient registration through critical care response showcases practical application of theoretical concepts in a meaningful healthcare context.

The system successfully achieves its objectives of real-time monitoring, automated donor matching, and efficient resource management while maintaining data consistency across distributed components. The modular design enables easy expansion and enhancement for production deployment.

Project Contributors

- Aditya Andotra
- Harsha Vardhan
- Madhuri V