METU CEng 536

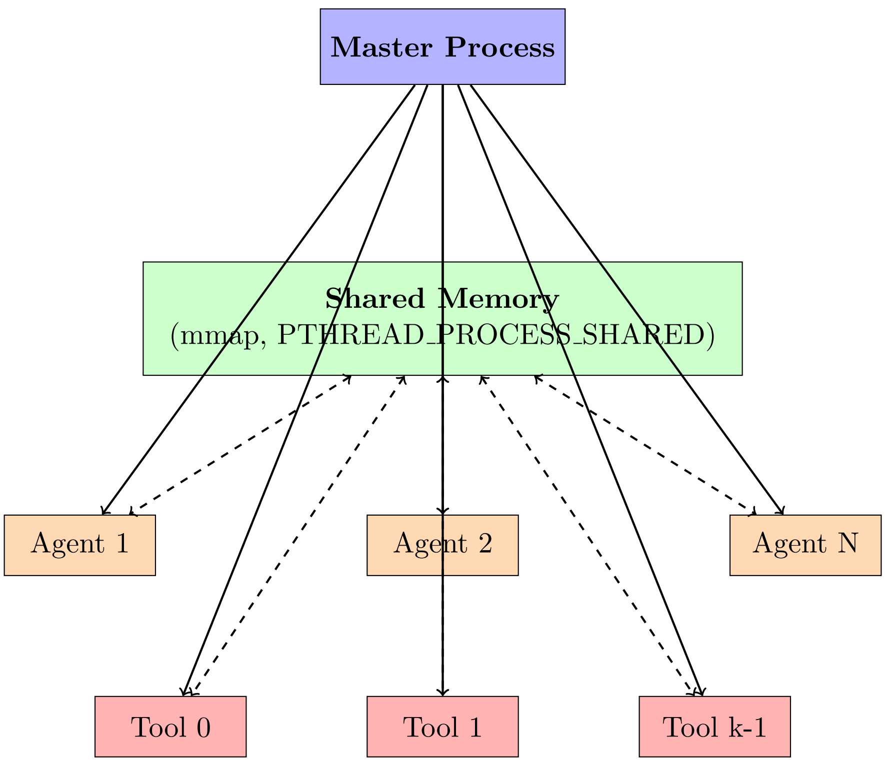
Advanced Unix Programming

Homework 1

Complete Implementation Guide

IPC Socket-Based Tool Assignment System

Multi-Process Server with Fairness Algorithm



Complete Reference Documentation

Every Line of Code Explained

From Zero to Full Implementation

Fall 2025

Version 2.0 - Extended Edition

November 3, 2025

Contents

List of Figures

List of Tables

Listings

Part I

Understanding the Problem

Chapter 1

Assignment Overview

1.1 What Are We Building?

You are building a **multi-process server system** that manages gym equipment (tools) with a sophisticated fairness algorithm. This is NOT a simple program - it's a complex distributed system with:

- **Multiple concurrent processes** (master, agents, tools)
- **Shared memory** for inter-process communication
- **Socket-based networking** (both Unix domain and TCP)
- **Complex synchronization** using mutexes and condition variables
- **Fairness algorithm** based on accumulated usage
- **Preemption mechanism** for resource allocation

1.1.1 Real-World Analogy

Imagine a gym with k treadmills:

1. Customers arrive and want to use treadmills
2. If treadmills are free, customers get assigned immediately
3. If all treadmills are busy, customers wait in a queue
4. The queue is ordered by **fairness**: who has used the treadmill the least total time gets priority
5. There are time limits: after using for q milliseconds, the system checks if someone with less total usage is waiting
6. After Q milliseconds, the customer MUST leave the treadmill (unless nobody is waiting)

1.1.2 Technical Complexity Assessment

1.2 Project Metrics

1.2.1 Code Statistics

1.2.2 Time Estimates

- **Phase 1 (Foundation)**: 8–12 hours
- **Phase 2 (Basic IPC)**: 10–15 hours

Component	Difficulty	Why It's Hard
Multi-process architecture	⋆⋆⋆⋆⋆	Coordinating multiple processes without deadlocks or race conditions
Shared memory management	⋆⋆⋆⋆	Cannot use malloc(), must implement custom allocation with integer indices
Process synchronization	⋆⋆⋆⋆⋆	PTHREAD_PROCESS_SHARED attributes, condition variable usage
Fairness algorithm	⋆⋆⋆⋆	Complex decision tree for tool assignment and preemption
Timing simulation	⋆⋆⋆⋆	Accurate millisecond timing with preemption checks
Socket programming	⋆⋆⋆	Supporting both Unix domain and TCP sockets
Agent threading	⋆⋆⋆⋆	Two threads per agent: one blocking on socket, one waiting for events

Table 1.1: Technical Difficulty Breakdown

File	Lines	Description
hw1.c	2,500–3,000	Main implementation
heap.c	300	Provided min-heap (modified)
Makefile	20	Build system
Total	**3,000**	Core implementation
Plus test clients, documentation, etc.		

Table 1.2: Estimated Line Counts

- **Phase 3 (Tool Assignment)**: 12–18 hours
- **Phase 4 (Timing & Preemption)**: 10–15 hours
- **Phase 5 (Testing & Debug)**: 10–15 hours
- **Phase 6 (Documentation)**: 5–8 hours

Total estimated time: 55–83 hours

1.3 Learning Objectives

By completing this assignment, you will master:

1. **Process Management**: fork(), process hierarchy, parent-child relationships
2. **Inter-Process Communication**: Shared memory (mmap), process-shared synchronization
3. **Socket Programming**: Both Unix domain and IPv4 TCP sockets
4. **Multi-threading**: pthread creation, synchronization within and across processes
5. **Synchronization Primitives**: Mutexes, condition variables, avoiding race conditions
6. **Resource Scheduling**: Implementing a fairness algorithm with preemption
7. **Timing and Real-time Constraints**: Millisecond-precision timing
8. **Memory Management**: Custom allocation without malloc() for shared memory
9. **Data Structures**: Heap-based priority queue for fair scheduling

Chapter 2

System Architecture Deep Dive

2.1 Process Hierarchy

2.1.1 The Four Process Types

Our system consists of four distinct process types, each with specific responsibilities:

2.1.2 Process Creation Flow

2.1.3 Key fork() Concept

`fork()` creates a complete copy of the calling process:

```c
// BEFORE fork():
// One process exists (parent)

pid_t pid = fork();

// AFTER fork():
// TWO processes exist!

if (pid == 0) {
    // CHILD PROCESS
    printf("I am the child, PID=%d\n", getpid());
    printf("My parent PID=%d\n", getppid());

    // Child gets COPY of parent's variables
    // But shared memory is SHARED (not copied)

} else if (pid > 0) {
    // PARENT PROCESS
    printf("I am the parent, PID=%d\n", getpid());
    printf("My child PID=%d\n", pid);

} else {
    // ERROR
    perror("fork");
}
```

Listing 2.1: fork() Behavior

Critical Understanding:

- Normal variables are **copied** (child has separate copy)
- Shared memory (mmap with MAP_SHARED) is **shared** (both see same memory)
- File descriptors are **copied** (both can use socket)
- Both processes continue from the `if` statement

Process Type	**Count**	**Responsibilities**
Master	1	• Create shared memory with mmap() • Set up socket (bind/listen) • Fork k tool processes at startup • Accept client connections in infinite loop • Fork agent process for each client • Never terminates

Agent	N	• One per connected client
		• Allocate customer slot in shared memory
		• Create two threads:
		   – Thread 1: Read from socket (blocking)
		   – Thread 2: Wait for tool events
		• Parse commands (REQUEST/REST/REPORT/QUIT)
		• Update customer state
		• Send responses to client
		• Deallocate customer on disconnect
		• Terminate when client quits

Tool	k	• One per tool (e.g., treadmill) • Infinite loop: simulate tool usage timing • Sleep for appropriate duration • Check q and Q limits • Implement preemption logic • Update customer share values • Assign next customer from waiting queue • Send events to agent processes • Never terminates
Client	N	• External test programs (not part of hw1) • Connect via socket • Send text commands • Receive and display responses • User-controlled

Table 2.1: Process Types and Responsibilities

2.2 Data Flow Architecture

2.2.1 Customer Lifecycle

The complete lifecycle of a customer from connection to termination:

2.2.2 REQUEST Command Detailed Flow

When a customer sends `REQUEST 5000`:

1. **Agent socket thread** receives the command
2. Thread calls `handle_request(customer_idx, 5000)`
3. `handle_request()` locks global_mutex
4. Check customer's current state:
   - If RESTING: Update resting_customers count
   - If already USING: Error (invalid state)
5. Set request_duration = 5000, remaining_duration = 5000
6. **Tool Assignment Logic**:
   - Search for free tool: `find_free_tool()`
   - If found: Immediately assign with `assign_tool_to_customer()`
   - If not found:
     - Check if can preempt: `find_preemption_candidate()`
     - If can preempt: Remove current user, assign to new customer
     - If cannot preempt: Add to waiting queue (heap_insert())
7. Unlock global_mutex
8. If assigned: Agent notify thread wakes up and sends message to client
9. If in queue: Agent waits for tool to become available

2.3 Synchronization Strategy

2.3.1 Why Synchronization is Essential

Without proper synchronization, consider this scenario:

```
// Shared: int counter = 0;

// Process A (Tool 0):          Process B (Tool 1):
tmp = counter;  // Read: 0      tmp = counter;  // Read: 0
tmp = tmp + 1;  // Compute: 1   tmp = tmp + 1;  // Compute: 1
counter = tmp;  // Write: 1     counter = tmp;  // Write: 1

// RESULT: counter = 1 (WRONG! Should be 2)
```

Listing 2.2: Race Condition Example - NO MUTEX

With mutex:

```
// Process A:                       Process B:
pthread_mutex_lock(&mutex);        // BLOCKED - waiting...
tmp = counter;  // Read: 0         // Still waiting...
tmp = tmp + 1;  // Compute: 1      // Still waiting...
counter = tmp;  // Write: 1        // Still waiting...
pthread_mutex_unlock(&mutex);      pthread_mutex_lock(&mutex);
                                   tmp = counter;  // Read: 1
                                   tmp = tmp + 1;  // Compute: 2
                                   counter = tmp;  // Write: 2
                                   pthread_mutex_unlock(&mutex);

// RESULT: counter = 2 (CORRECT!)
```

Listing 2.3: Correct Synchronization - WITH MUTEX

2.3.2 Our Synchronization Primitives

Critical Rule: *Always lock global_mutex before accessing ANY shared memory data!*

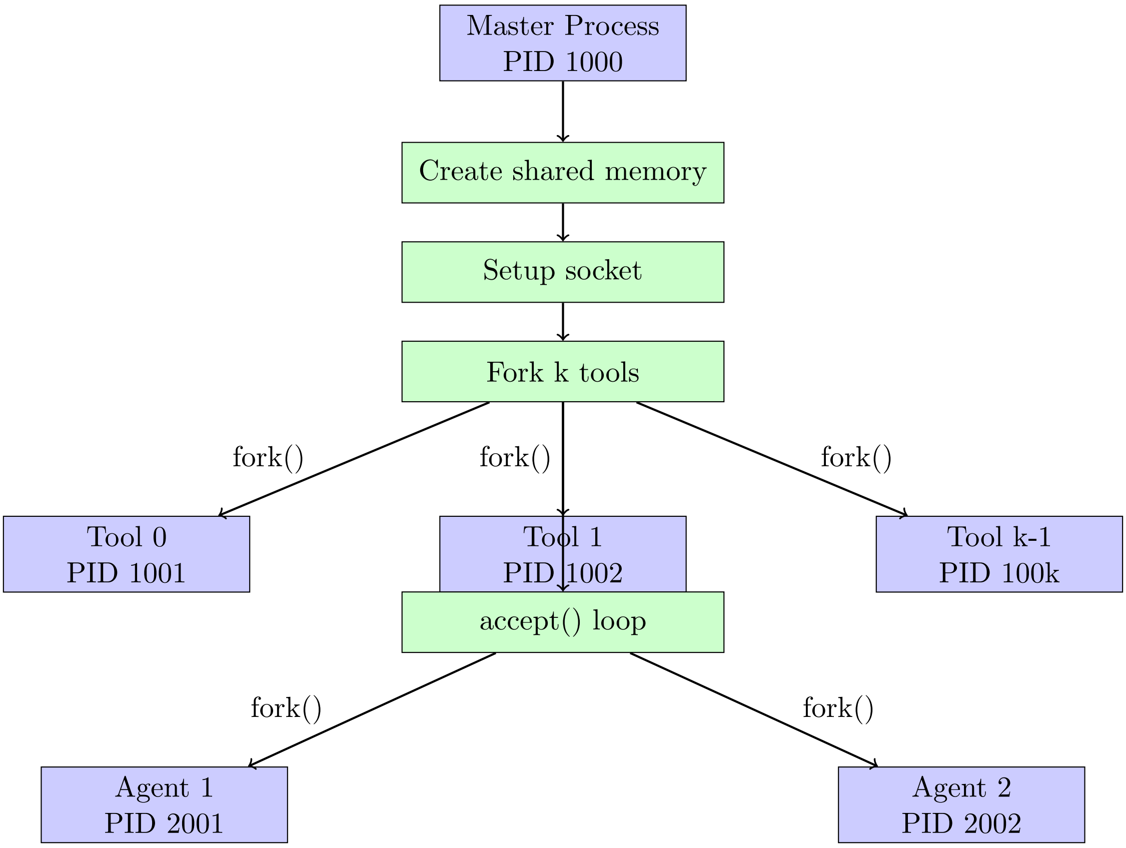


Figure 2.1: Process Creation Sequence

Primitive	**Count**	**Purpose**
global_mutex	1	Protects ALL shared memory access
new_customer_cond	1	Signals tool processes when new customer arrives
agent_cond	N	One per customer, tool notifies agent of events
tool_cond	k	One per tool, agent wakes tool when customer assigned

Table 2.2: Synchronization Primitives

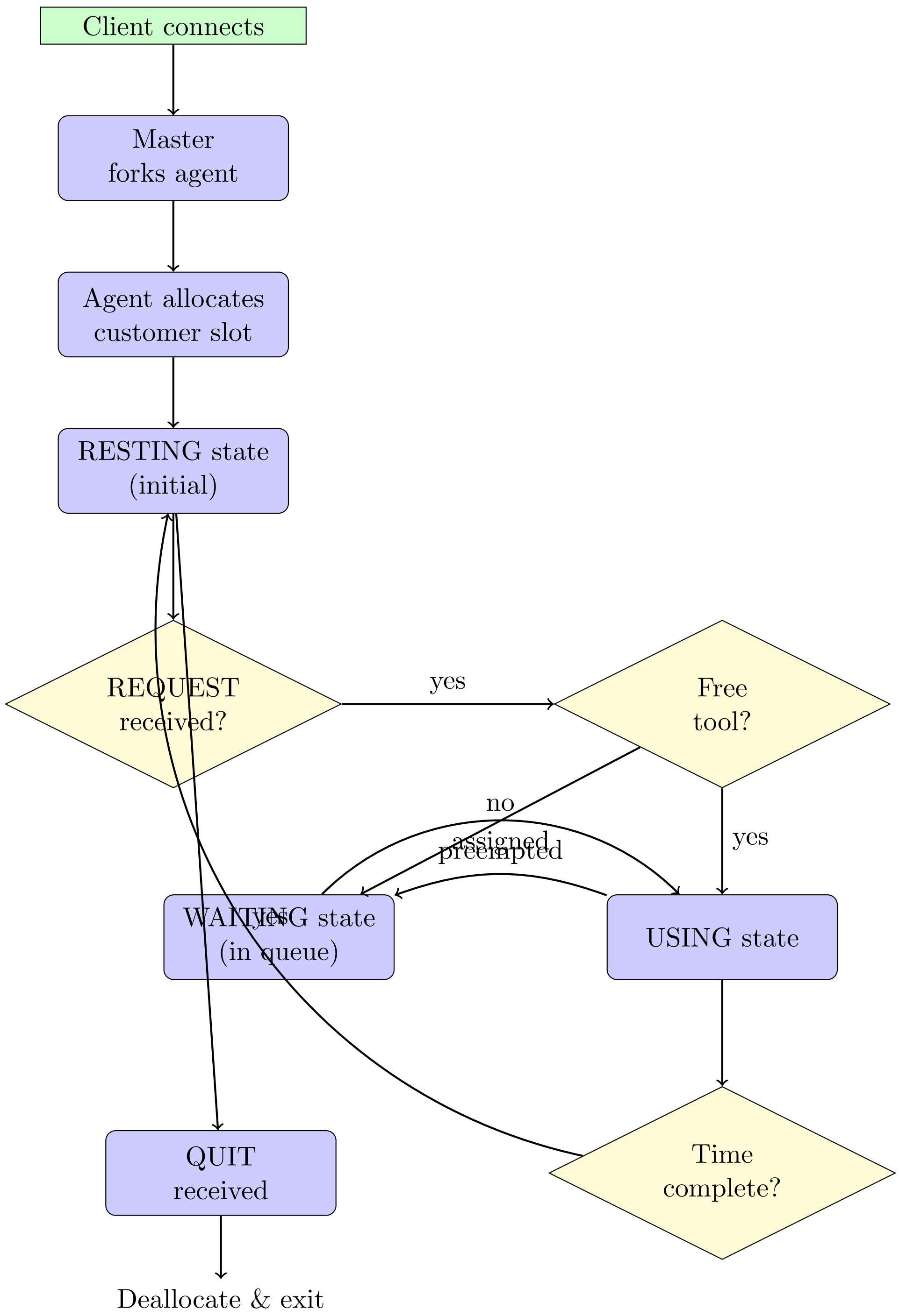


Figure 2.2: Complete Customer State Machine

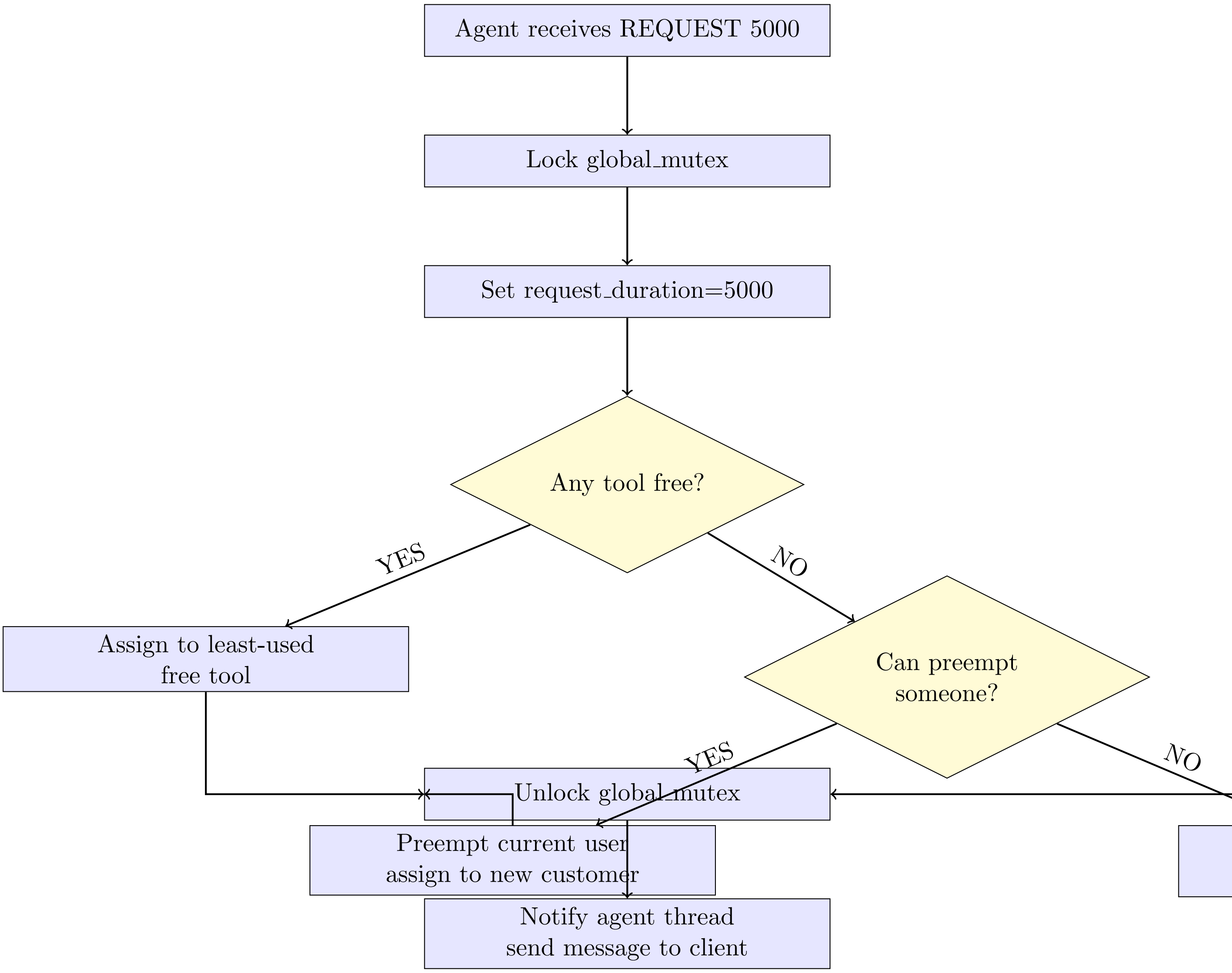


Figure 2.3: REQUEST Command Processing Flow

Part II

Data Structures in Detail

Chapter 3

Core Structures

3.1 Customer Structure - Line by Line

The Customer structure is the heart of the fairness algorithm. Let's examine every field:

```
typedef struct {
    //================================================================
    // IDENTIFICATION
    //================================================================

    int customer_id;           // Agent process PID
                              // Why: Uniquely identify each customer
                              // Used in: printf output, debugging

    int is_allocated;          // 1 = slot in use, 0 = free
                              // Why: Track which slots are occupied
                              // Used in: allocation/deallocation

    //================================================================
    // STATE MANAGEMENT
    //================================================================

    CustomerState state;       // Current state enum
                              // Why: Track customer's current activity
                              // Values: RESTING | WAITING | USING | DELETED
                              // Used in: state machine transitions

    //================================================================
    // FAIRNESS TRACKING (MOST CRITICAL!)
    //================================================================

    double share;              // Total accumulated usage time (milliseconds)
                              // Why: THIS IS THE FAIRNESS METRIC
                              // Lower share = higher priority in queue
                              // Used in: heap sorting, preemption decisions
                              // Example: Customer with share=5000 has used
                              //          tools for 5 seconds total

    //================================================================
    // CURRENT REQUEST
    //================================================================

    int request_duration;      // Duration requested in REQUEST command
                              // Why: Track how much time customer wants
                              // Used in: timing calculations
                              // Example: REQUEST 5000 sets this to 5000

    int remaining_duration;    // How much time left to use
                              // Why: Track progress through request
                              // Used in: session completion check
```

```c
                                // Example: After 2s, this becomes 3000

    //================================================================
    // TOOL USAGE
    //================================================================

    int current_tool;            // Which tool is customer using?
                                // Why: Track tool assignment
                                // Value: 0 to k-1, or -1 if not using
                                // Used in: tool release, reporting

    long long session_start;    // Timestamp when tool usage began (ms)
                                // Why: Calculate elapsed time
                                // Used in: timing, duration calculations
                                // Example: start=1000, now=3000     elapsed=2000ms

    //================================================================
    // WAITING QUEUE
    //================================================================

    long long wait_start;       // Timestamp when entered waiting queue (ms)
                                // Why: Calculate how long customer has waited
                                // Used in: REPORT command duration column

    int heap_index;             // Index in GLB array (from heap.c)
                                // Why: Link customer to heap node
                                // Used in: heap operations
                                // Value: Same as customer array index

    //================================================================
    // AGENT COMMUNICATION
    //================================================================

    pthread_cond_t agent_cond; // Condition variable for events
                                // Why: Tool wakes up agent thread
                                // Used in: pthread_cond_wait/signal

    int event_pending;          // Flag: new event available? (1=yes, 0=no)
                                // Why: Prevent spurious wakeups
                                // Used in: while(!event_pending) wait()

    EventType event_type;       // What kind of event occurred?
                                // Why: Tell agent what happened
                                // Values: ASSIGNED | REMOVED | COMPLETED

    int event_tool_id;          // Which tool is this event about?
                                // Why: Include tool number in message
                                // Used in: printf output

} Customer;
```

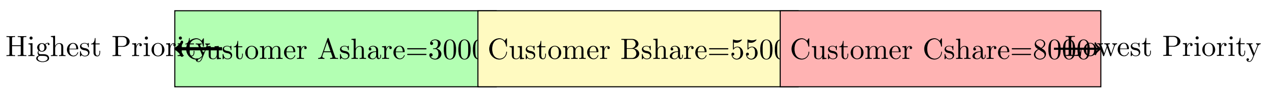
Listing 3.1: Customer Structure - Complete

3.1.1 Share Field Deep Dive

The `share` field is the **most important field** for fairness.

Calculation Examples:

Priority Ordering:



In Waiting Queue: Customer A will be assigned tool first.

3.2 Tool Structure Explained

```c
typedef struct {
    //==============================================================
    // IDENTIFICATION
    //==============================================================

    int tool_id;                 // Tool number (0 to k-1)
                                // Why: Identify this tool
                                // Used in: printf output, reports

    //==============================================================
    // USAGE STATISTICS
    //==============================================================

    long long total_usage;       // Total milliseconds this tool has been used
                                // Why: Balance load across tools
                                // Used in: find_free_tool() selection
                                // Example: Tool 0 used for 50s, Tool 1 for 45s
                                //              Assign to Tool 1 (less used)

    //==============================================================
    // CURRENT SESSION
    //==============================================================

    int current_user;            // Customer index (-1 if free)
                                // Why: Track who is using this tool
                                // Value: Index in customers[] array, or -1

    int current_usage;           // Milliseconds in THIS session
                                // Why: Check q and Q limits
                                // Used in: preemption logic
                                // Example: If current_usage >= Q, must preempt

    long long session_start;     // When did current session start? (ms)
                                // Why: Calculate elapsed time
                                // Used in: timing calculations

    //==============================================================
    // SYNCHRONIZATION
    //==============================================================

    pthread_cond_t tool_cond;    // Condition variable for this tool
                                // Why: Wake up tool thread
                                // Used in: When customer assigned or freed

    int tool_should_exit;        // Flag for cleanup (1=exit, 0=run)
                                // Why: Graceful shutdown
                                // Used in: Signal handling

} ToolInfo;
```

Listing 3.2: Tool Structure - Complete

3.2.1 Tool Selection Algorithm

When multiple tools are free, which one do we assign?

Rule: Select the tool with the **lowest total_usage**. If tied, select **lowest tool_id**.

Example:

Decision Process:

1. Tools 0, 1, 3 are free
2. Tools 1 and 3 have lowest usage (45,000 ms)

3. Tool 1 has lower ID than Tool 3
4. **Result: Assign to Tool 1**

```c
int find_free_tool(void) {
    int best_tool = -1;
    long long min_usage = LLONG_MAX;

    for (int i = 0; i < shm->num_tools; i++) {
        if (shm->tools[i].current_user == -1) {
            // Tool is free
            if (shm->tools[i].total_usage < min_usage) {
                min_usage = shm->tools[i].total_usage;
                best_tool = i;
            } else if (shm->tools[i].total_usage == min_usage) {
                // Tie: select lower ID
                if (i < best_tool || best_tool == -1) {
                    best_tool = i;
                }
            }
        }
    }

    return best_tool;  // -1 if no free tools
}
```

Listing 3.3: Tool Selection Code Snippet

3.3 Shared Memory Structure

```c
typedef struct {
    //================================================================
    // CUSTOMER POOL (Memory Management)
    //================================================================

    Customer customers[MAX_CUSTOMERS];
    // Why: Fixed-size array of all customer slots
    // Size: 1,000,000 slots
    // Note: Cannot use malloc() in shared memory!

    int free_customer_slots[MAX_CUSTOMERS];
    // Why: Free list for available slots
    // Contains: Indices of free slots
    // Example: [0, 1, 2, ..., 999999] initially
    // After allocating slot 5: [0,1,2,3,4,6,7,...]

    int free_customer_count;
    // Why: How many free slots remain?
    // Initial value: 1,000,000
    // After 3 customers: 999,997

    //================================================================
    // WAITING QUEUE
    //================================================================

    int waiting_count;
    // Why: Track number of customers in waiting queue
    // Note: Actual heap is in heap.c (GLB, HEAP_ARRAY)
    // Used in: Reports, queue empty checks

    //================================================================
    // TOOL INFORMATION
    //================================================================
```

```
    ToolInfo tools[MAX_TOOLS];
    // Why: Fixed-size array of tool structures
    // Size: 100 slots (max k=100)

    int num_tools;
    // Why: Actual number of tools (k from command line)
    // Example: ./hw1 ... 3 means num_tools=3

    //================================================================
    // GLOBAL STATISTICS
    //================================================================

    int total_customers;
    // Why: Count of active customers (allocated slots)
    // Used in: Average share calculation, reports

    int resting_customers;
    // Why: Count in RESTING state
    // Used in: Reports

    double total_share;
    // Why: Sum of all customer shares
    // Used in: Calculate average for new customers
    // Example: 3 customers with shares 5000,8000,6000
    //          total_share = 19000
    //          New customer gets 19000/3 = 6333.33

    //================================================================
    // SYNCHRONIZATION
    //================================================================

    pthread_mutex_t global_mutex;
    // Why: THE main mutex protecting all shared data
    // Rule: ALWAYS lock before accessing shared memory!
    // Type: PTHREAD_PROCESS_SHARED

    pthread_cond_t new_customer_cond;
    // Why: Signal all tools when customer enters queue
    // Used in: Wake idle tools

    //================================================================
    // CONFIGURATION
    //================================================================

    int q;
    // Why: Minimum uninterrupted time (milliseconds)
    // From: Command line argument
    // Used in: Check preemption after q ms

    int Q;
    // Why: Maximum uninterrupted time (milliseconds)
    // From: Command line argument
    // Used in: Force preemption after Q ms

    long long start_time;
    // Why: Server start timestamp
    // Used in: Calculate uptime, debugging

} SharedMemory;
```

Listing 3.4: SharedMemory Structure - Complete

3.3.1 Memory Management Strategy

Problem: Cannot use `malloc()` in shared memory!

Solution: Pre-allocate fixed-size arrays and use free list.

Allocation Algorithm:

```c
// Get free slot
int idx = free_customer_slots[--free_customer_count];

// Use slot
customers[idx] = /* initialize */;
```

Deallocation Algorithm:

```c
// Return slot to free list
free_customer_slots[free_customer_count++] = idx;
```

Event	Elapsed (ms)	New Share	Calculation
Customer arrives	—	0	First customer
Uses tool for 3s	3000	3000	$0 + 3000$
Uses tool for 2s more	2000	5000	$3000 + 2000$
Rests for 10s	0	5000	No change (not using tool)
Uses tool for 1s	1000	6000	$5000 + 1000$

Table 3.1: Share Calculation Examples

Tool ID	Total Usage (ms)	Current User	Selection
0	50,000	-1 (FREE)	
1	45,000	-1 (FREE)	← **SELECT THIS!**
2	52,000	Customer 5	Not available
3	45,000	-1 (FREE)	

Table 3.2: Tool Selection Example

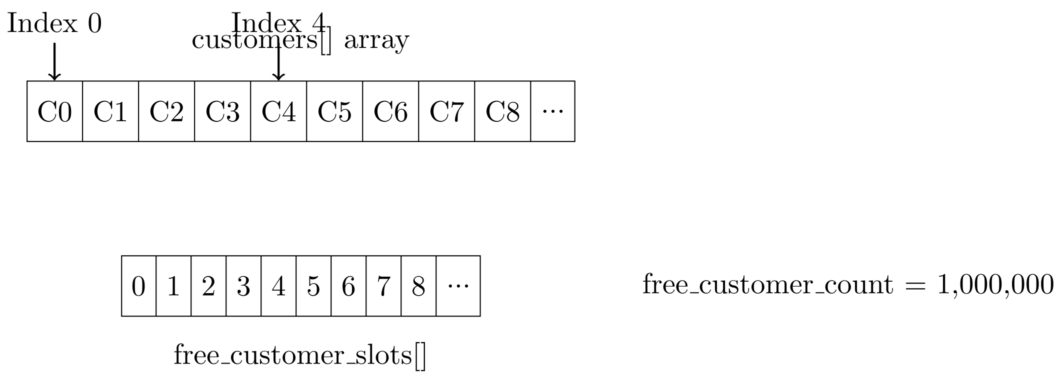


Figure 3.1: Customer Pool Memory Layout

Part III

Complete Implementation - Every Function

Chapter 4

Phase 1: Foundation

4.1 Project Structure Setup

4.1.1 Directory Organization

Create this exact structure:

```
~/ceng536-hw1/
 hw1/                       # ← Submission directory
    hw1.c                 # ← Your main code
    heap.c                # ← Instructor's code (modified)
    Makefile              # ← Build system

 test/                      # ← Testing tools (DON'T submit)
    test_client.py        # ← Python test client
    test_client.c         # ← C test client (optional)
    test_scenarios.sh     # ← Automated tests

 docs/                      # ← Your notes
    notes.md
    algorithm.txt

 original/                  # ← Backup
    heap.c                # ← Original from instructor
    rb.c                  # ← (not using this)
    ceng536-hw1.pdf

 submission/                # ← Final tar.gz goes here
     yourname_id.tar.gz
```

4.1.2 Step-by-Step Setup

Commands to run:

```
# 1. Create directory structure
cd ~
mkdir -p ceng536-hw1/{hw1,test,docs,original,submission}

# 2. Copy instructor's files
cd ~/ceng536-hw1
cp ~/Downloads/heap.c original/
cp ~/Downloads/rb.c original/
cp ~/Downloads/ceng536-hw1.pdf original/

# 3. Copy heap.c to working directory
```

```
cp original/heap.c hw1/

# 4. Verify structure
tree ceng536-hw1
```

4.1.3 Modify heap.c

Open `hw1/heap.c` and make these changes:

Change 1: MAX_NODES

Find line 32:

```
#define MAX_NODES 1024
```

Change to:

```
#define MAX_NODES 1000000
```

Change 2: Remove main()

Find line 509 (the main function):

```
int main() {
    // ... test code ...
    return 0;
}
```

Comment it out:

```
/*
int main() {
    // ... test code ...
    return 0;
}
*/
```

Why remove main()? Because your hw1.c will have its own main()!

4.1.4 Create Makefile

Create `hw1/Makefile`:

```
# Makefile for CEng 536 HW1

CC = gcc
CFLAGS = -Wall -Wextra -Werror -g -pthread -std=c11 \
         -D_POSIX_C_SOURCE=200809L
LDFLAGS = -lpthread -lrt

TARGET = hw1
SRCS = hw1.c heap.c
OBJS = $(SRCS:.c=.o)

# Main target
$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $(TARGET) $(OBJS) $(LDFLAGS)

# Compile .c to .o
%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

# Clean
clean:
	rm -f $(TARGET) $(OBJS)

# Debug build
debug: CFLAGS += -DDEBUG -O0
```

```
debug: clean $(TARGET)

# Run with example parameters
run-test: $(TARGET)
	./$(TARGET) @/tmp/gym.sock 1000 5000 3

# Check for memory leaks
valgrind: $(TARGET)
	valgrind --leak-check=full --show-leak-kinds=all \
		./$(TARGET) @/tmp/gym.sock 1000 5000 2

.PHONY: clean debug run-test valgrind
```

Test the Makefile:

```
cd ~/ceng536-hw1/hw1
make clean
# Should output: rm -f hw1 *.o
```

4.2 hw1.c: Basic Skeleton

Create `hw1/hw1.c` with this initial structure:

```c
/*==================================================================
 * CEng 536 - Advanced Unix Programming
 * Homework 1: IPC Gym Tool Assignment System
 *
 * Multi-process server with fairness algorithm
 *================================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <ctype.h>

// Socket headers
#include <sys/socket.h>
#include <sys/un.h>        // Unix domain sockets
#include <netinet/in.h>    // Internet sockets
#include <arpa/inet.h>

// Process & threading
#include <sys/types.h>
#include <sys/wait.h>
#include <pthread.h>

// Shared memory
#include <sys/mman.h>

// Timing
#include <time.h>
#include <sys/time.h>

// Limits
#include <limits.h>
#include <float.h>

/*==================================================================
 * HEAP INTERFACE (from heap.c)
 *================================================================*/

// Heap functions
```

```c
extern int heap_insert(int nodeindex);
extern int heap_pop(void);
extern int heap_delete(int nodeindex);
extern int heap_size;

// Heap node structure
struct Node {
    double key;
    int heap_index;
};
extern struct Node GLB[];
extern int HEAP_ARRAY[];

/*==================================================================
 * CONSTANTS
 *================================================================*/

#define MAX_CUSTOMERS 1000000
#define MAX_TOOLS 100
#define BUFFER_SIZE 4096
#define NIL -1

/*==================================================================
 * DEBUG LOGGING
 *================================================================*/

#ifdef DEBUG
#define LOG(fmt, ...) \
    do { \
        struct timespec ts; \
        clock_gettime(CLOCK_MONOTONIC, &ts); \
        fprintf(stderr, "[%ld.%03ld][PID:%d] " fmt "\n", \
                ts.tv_sec, ts.tv_nsec/1000000, getpid(), \
                ##__VA_ARGS__); \
        fflush(stderr); \
    } while(0)
#else
#define LOG(fmt, ...) do {} while(0)
#endif

/*==================================================================
 * ENUMERATIONS
 *================================================================*/

typedef enum {
    CUSTOMER_STATE_RESTING,
    CUSTOMER_STATE_WAITING,
    CUSTOMER_STATE_USING,
    CUSTOMER_STATE_DELETED
} CustomerState;

typedef enum {
    EVENT_NONE = 0,
    EVENT_TOOL_ASSIGNED,
    EVENT_TOOL_REMOVED,
    EVENT_TOOL_COMPLETED
} EventType;

/*==================================================================
 * STRUCTURE DECLARATIONS
 *================================================================*/

typedef struct Customer Customer;
typedef struct ToolInfo ToolInfo;
typedef struct SharedMemory SharedMemory;
typedef struct ServerAddress ServerAddress;
```

```c
typedef struct AgentContext AgentContext;

/*=================================================================
 * CUSTOMER STRUCTURE
 *================================================================*/

struct Customer {
    // Identification
    int customer_id;
    int is_allocated;

    // State
    CustomerState state;

    // Fairness
    double share;

    // Request
    int request_duration;
    int remaining_duration;

    // Tool usage
    int current_tool;
    long long session_start;

    // Waiting queue
    long long wait_start;
    int heap_index;

    // Communication
    pthread_cond_t agent_cond;
    int event_pending;
    EventType event_type;
    int event_tool_id;
};

/*=================================================================
 * TOOL STRUCTURE
 *================================================================*/

struct ToolInfo {
    int tool_id;

    long long total_usage;

    int current_user;
    int current_usage;
    long long session_start;

    pthread_cond_t tool_cond;
    int tool_should_exit;
};

/*=================================================================
 * SHARED MEMORY STRUCTURE
 *================================================================*/

struct SharedMemory {
    Customer customers[MAX_CUSTOMERS];
    int free_customer_slots[MAX_CUSTOMERS];
    int free_customer_count;

    int waiting_count;

    ToolInfo tools[MAX_TOOLS];
    int num_tools;
```

```c

    int total_customers;
    int resting_customers;
    double total_share;

    pthread_mutex_t global_mutex;
    pthread_cond_t new_customer_cond;

    int q;
    int Q;
    long long start_time;
};

/*================================================================
 * SERVER ADDRESS
 *================================================================*/

struct ServerAddress {
    int is_unix;
    char path[256];
    char ip[64];
    int port;
};

/*================================================================
 * AGENT CONTEXT
 *================================================================*/

struct AgentContext {
    int socket_fd;
    int customer_idx;
    int should_exit;
};

/*================================================================
 * GLOBAL VARIABLES
 *================================================================*/

static SharedMemory *shm = NULL;

/*================================================================
 * FUNCTION PROTOTYPES
 *================================================================*/

// Utility
long long get_current_time_ms(void);
ServerAddress parse_address(const char *addr_str);

// Shared memory
void setup_shared_memory(int q, int Q, int k);
void setup_heap(void);

// Customer management
int allocate_customer(int customer_id);
void deallocate_customer(int customer_idx);

// Tool assignment
int find_free_tool(void);
int find_preemption_candidate(double new_customer_share);
void assign_tool_to_customer(int customer_idx, int tool_id);
void assign_next_from_queue(int tool_id);

// Request handlers
void handle_request(int customer_idx, int duration);
void handle_rest(int customer_idx);
void handle_report(int socket_fd);
```

```c

// Tool process
void* tool_thread_func(void* arg);
void handle_session_complete(int tool_id);
void handle_Q_limit(int tool_id);
void check_preemption(int tool_id);

// Agent process
void agent_process(int client_socket);
void* agent_socket_thread(void* arg);
void* agent_notify_thread(void* arg);

// Socket
int create_server_socket(ServerAddress addr);

// Main
int main(int argc, char *argv[]);

/*================================================================
 * IMPLEMENTATION BEGINS HERE
 *================================================================*/
```

Listing 4.1: hw1.c - Initial Skeleton (Part 1/10)

Now test compilation:

```bash
cd ~/ceng536-hw1/hw1
make clean
make hw1
# Expected: errors because functions not implemented yet
```

This is normal! We'll implement each function step by step.