OS Lab Assignment - 3

CS 334: Operating Systems Lab

Shared Memory and Mailbox Implementation

Group 11

Prayansh Kumar (230101081)
Deval Singhal (230101035)
Aeyaz Adil (2301010007)
Jayant Kumar (230101050)

Contents

Chapter 1

Task 1: Implement Kernel Primitives for IPC

1.1 Introduction

The default xv6 kernel's IPC capabilities are limited. In this task, we implemented two core primitives to enable more complex process coordination: shared memory for efficient data sharing and mailboxes for reliable messaging between processes. To solve this task, we first define some fixed parameters like the number of maximum semaphores, maximum number of mailboxes, and value limits of those mailboxes. These are defined in `param.h` in the kernel folder

1.2 System Call Changes

For implementing the system calls, we:

- Define kernel helper functions in `defs.h`.
- Add system call numbers in `syscall.h`.
- Add entries in `syscall.c`.
- Implement syscall functions in `sysproc.c`.
- Declare on the user side in `user.h` and `usys.pl`.

1.3 Shared Memory Kernel Logic

We define the file `shm.c` to implement kernel functions for shared memory.

1.3.1 Data Structures

- `struct shm_region {used, key, pa, refcnt}`
  - **used**: active (1) or free (0).
  - **key**: user-visible identifier.

  - **pa**: backing physical page.
  - **refcnt**: number of processes mapped.

- Global table with:
  - lock: protects table
  - reg[NSHM]: slots
  - inited: lazy init flag

1.3.2 Function Flow

1. `shm_init()`, acquire lock.
2. If key exists → return slot index.
3. Else, find free slot, allocate page, memset to 0.
4. Fill slot and return index.

1.3.3 Detach and Cleanup

- Unmap VA, decrement refcnt.
- If refcnt = 0 → free slot and kfree.

1.3.4 On Exit and Exec

We implement cleanup functions to prevent panic in xv6 when a process exits or execs with mappings still active.

1.4 Mailbox Kernel Functions

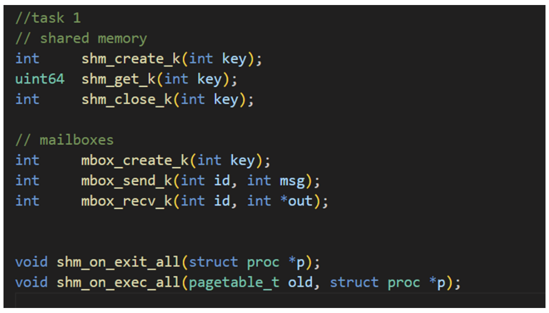
We define `mbox` structure with locks and implement:

- `mb_init()` – lazy init, mark slots free.
- `find_mbox_by_key()`, `find_mbox_free()`.
- `mbox_create_k()` – ensures init, allocates/free slot.
- `mbox_send_k()`, `mbox_recv_k()` – blocking send/receive with ring buffer.

1.5 Test Program

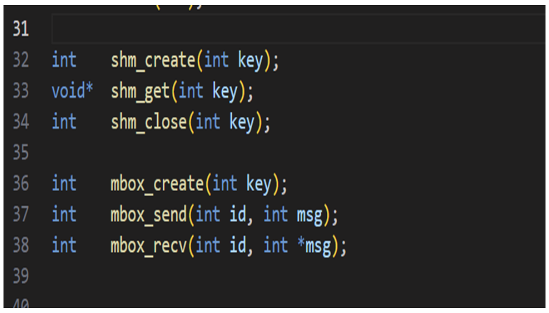
We implement `shmmbox_test.c` to validate shared memory and mailbox. Results confirm:

- Shared memory reflects changes across processes.
- Mailboxes deliver messages FIFO and block correctly.



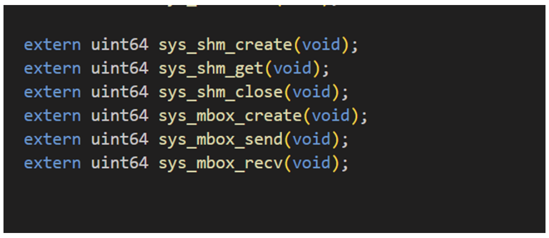
```c
//task 1
// shared memory
int     shm_create_k(int key);
uint64  shm_get_k(int key);
int     shm_close_k(int key);

// mailboxes
int     mbox_create_k(int key);
int     mbox_send_k(int id, int msg);
int     mbox_recv_k(int id, int *out);


void shm_on_exit_all(struct proc *p);
void shm_on_exec_all(pagetable_t old, struct proc *p);
```

(a) defs.h



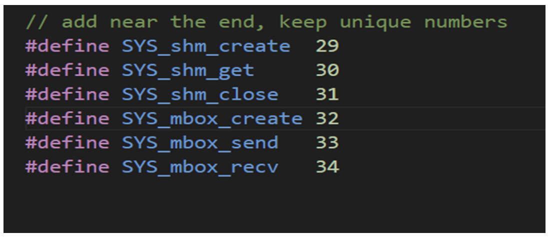
```c
int    shm_create(int key);
void*  shm_get(int key);
int    shm_close(int key);

int    mbox_create(int key);
int    mbox_send(int id, int msg);
int    mbox_recv(int id, int *msg);
```

(b) user.h



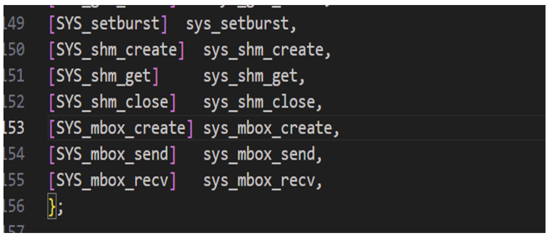
```c
extern uint64 sys_shm_create(void);
extern uint64 sys_shm_get(void);
extern uint64 sys_shm_close(void);
extern uint64 sys_mbox_create(void);
extern uint64 sys_mbox_send(void);
extern uint64 sys_mbox_recv(void);
```

(c) syscall.c



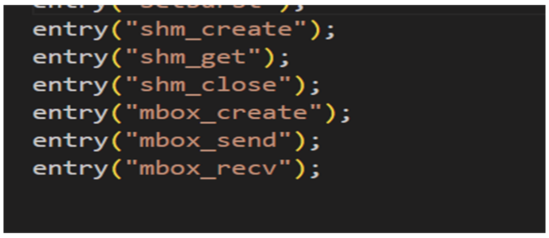
```c
// add near the end, keep unique numbers
#define SYS_shm_create   29
#define SYS_shm_get      30
#define SYS_shm_close    31
#define SYS_mbox_create  32
#define SYS_mbox_send    33
#define SYS_mbox_recv    34
```

(d) syscall.h



```c
[SYS_setburst]  sys_setburst,
[SYS_shm_create] sys_shm_create,
[SYS_shm_get]    sys_shm_get,
[SYS_shm_close]  sys_shm_close,
[SYS_mbox_create] sys_mbox_create,
[SYS_mbox_send]  sys_mbox_send,
[SYS_mbox_recv]  sys_mbox_recv,
};
```

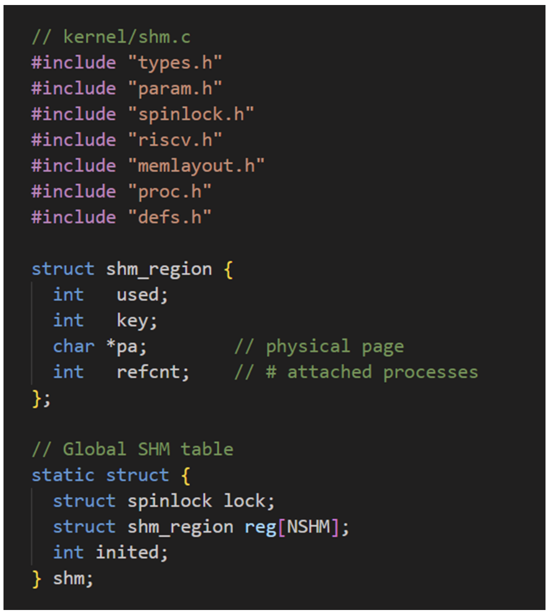
(e) sysproc.c



```perl
entry("shm_create");
entry("shm_get");
entry("shm_close");
entry("mbox_create");
entry("mbox_send");
entry("mbox_recv");
```

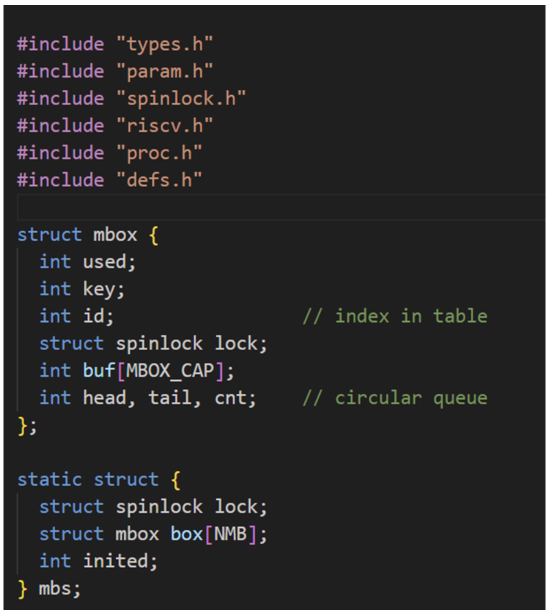
(f) usys.pl

Figure 1.1: Changes across header and syscall-related files



```c
// kernel/shm.c
#include "types.h"
#include "param.h"
#include "spinlock.h"
#include "riscv.h"
#include "memlayout.h"
#include "proc.h"
#include "defs.h"

struct shm_region {
  int   used;
  int   key;
  char *pa;        // physical page
  int   refcnt;    // # attached processes
};

// Global SHM table
static struct {
  struct spinlock lock;
  struct shm_region reg[NSHM];
  int inited;
} shm;
```

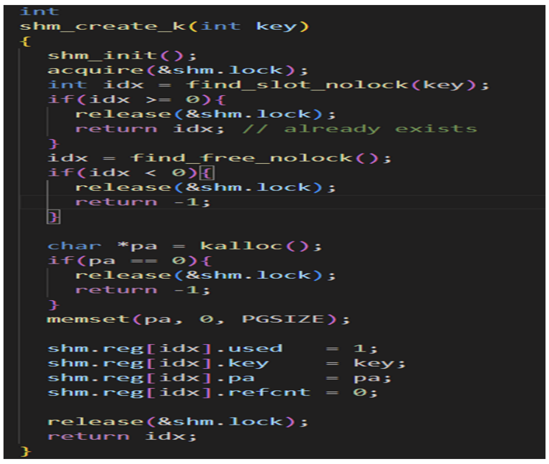
(a) shm.c



```c
#include "types.h"
#include "param.h"
#include "spinlock.h"
#include "riscv.h"
#include "proc.h"
#include "defs.h"

struct mbox {
  int used;
  int key;
  int id;                     // index in table
  struct spinlock lock;
  int buf[MBOX_CAP];
  int head, tail, cnt;        // circular queue
};

static struct {
  struct spinlock lock;
  struct mbox box[NMB];
  int inited;
} mbs;
```

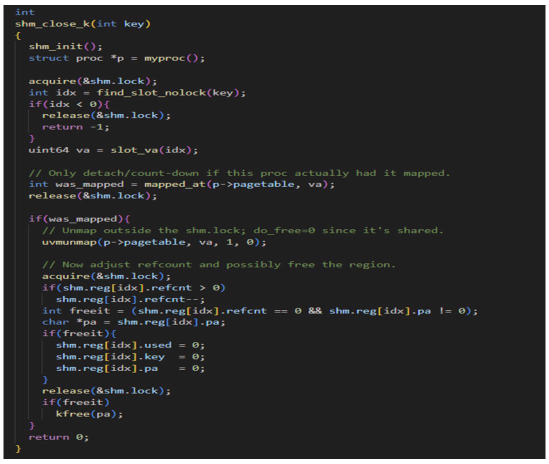
(b) mbox.c

Figure 1.2: Files where we implement kernel functions



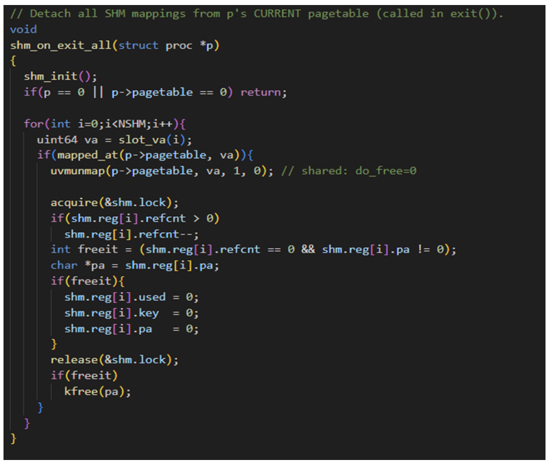
```c
int
shm_create_k(int key)
{
    shm_init();
    acquire(&shm.lock);
    int idx = find_slot_nolock(key);
    if(idx >= 0){
      release(&shm.lock);
      return idx; // already exists
    }
    idx = find_free_nolock();
    if(idx < 0){
      release(&shm.lock);
      return -1;
    }

    char *pa = kalloc();
    if(pa == 0){
      release(&shm.lock);
      return -1;
    }
    memset(pa, 0, PGSIZE);

    shm.reg[idx].used    = 1;
    shm.reg[idx].key     = key;
    shm.reg[idx].pa      = pa;
    shm.reg[idx].refcnt  = 0;

    release(&shm.lock);
    return idx;
}
```

(a) shm_create_k



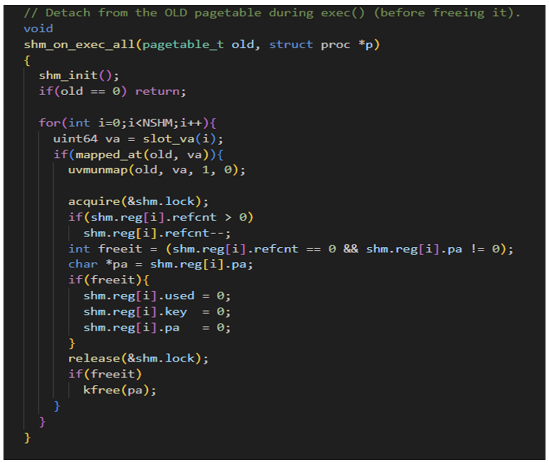
```c
int
shm_close_k(int key)
{
  shm_init();
  struct proc *p = myproc();

  acquire(&shm.lock);
  int idx = find_slot_nolock(key);
  if(idx < 0){
    release(&shm.lock);
    return -1;
  }
  uint64 va = slot_va(idx);

  // Only detach/count-down if this proc actually had it mapped.
  int was_mapped = mapped_at(p->pagetable, va);
  release(&shm.lock);

  if(was_mapped){
    // Unmap outside the shm.lock; do_free=0 since it's shared.
    uvmunmap(p->pagetable, va, 1, 0);

    // Now adjust refcount and possibly free the region.
    acquire(&shm.lock);
    if(shm.reg[idx].refcnt > 0)
      shm.reg[idx].refcnt--;
    int freeit = (shm.reg[idx].refcnt == 0 && shm.reg[idx].pa != 0);
    char *pa = shm.reg[idx].pa;
    if(freeit){
      shm.reg[idx].used = 0;
      shm.reg[idx].key  = 0;
      shm.reg[idx].pa   = 0;
    }
    release(&shm.lock);
    if(freeit)
      kfree(pa);
  }
  return 0;
}
```

(b) shm_close_k



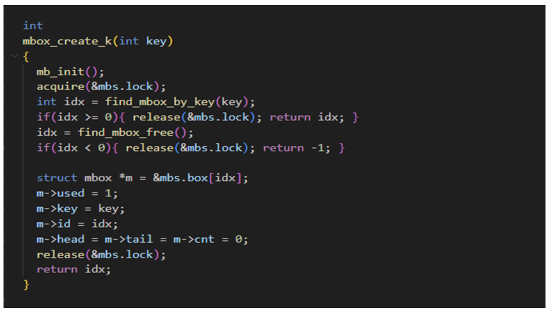
```c
// Detach all SHM mappings from p's CURRENT pagetable (called in exit()).
void
shm_on_exit_all(struct proc *p)
{
  shm_init();
  if(p == 0 || p->pagetable == 0) return;

  for(int i=0;i<NSHM;i++){
    uint64 va = slot_va(i);
    if(mapped_at(p->pagetable, va)){
      uvmunmap(p->pagetable, va, 1, 0); // shared: do_free=0

      acquire(&shm.lock);
      if(shm.reg[i].refcnt > 0)
        shm.reg[i].refcnt--;
      int freeit = (shm.reg[i].refcnt == 0 && shm.reg[i].pa != 0);
      char *pa = shm.reg[i].pa;
      if(freeit){
        shm.reg[i].used = 0;
        shm.reg[i].key  = 0;
        shm.reg[i].pa   = 0;
      }
      release(&shm.lock);
      if(freeit)
        kfree(pa);
    }
  }
}
```

(c) shm_on_exit_all



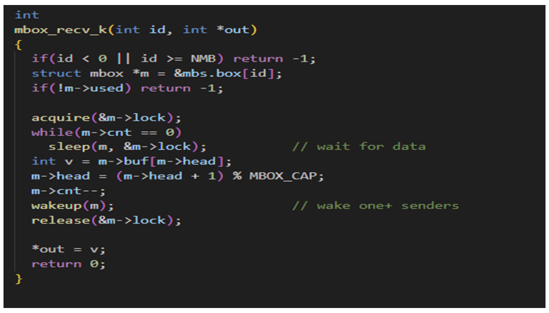
```c
// Detach from the OLD pagetable during exec() (before freeing it).
void
shm_on_exec_all(pagetable_t old, struct proc *p)
{
  shm_init();
  if(old == 0) return;

  for(int i=0;i<NSHM;i++){
    uint64 va = slot_va(i);
    if(mapped_at(old, va)){
      uvmunmap(old, va, 1, 0);

      acquire(&shm.lock);
      if(shm.reg[i].refcnt > 0)
        shm.reg[i].refcnt--;
      int freeit = (shm.reg[i].refcnt == 0 && shm.reg[i].pa != 0);
      char *pa = shm.reg[i].pa;
      if(freeit){
        shm.reg[i].used = 0;
        shm.reg[i].key  = 0;
        shm.reg[i].pa   = 0;
      }
      release(&shm.lock);
      if(freeit)
        kfree(pa);
    }
  }
}
```

(d) shm_on_exec_all



```c
int
mbox_create_k(int key)
{
  mb_init();
  acquire(&mbs.lock);
  int idx = find_mbox_by_key(key);
  if(idx >= 0){ release(&mbs.lock); return idx; }
  idx = find_mbox_free();
  if(idx < 0){ release(&mbs.lock); return -1; }

  struct mbox *m = &mbs.box[idx];
  m->used = 1;
  m->key = key;
  m->id = idx;
  m->head = m->tail = m->cnt = 0;
  release(&mbs.lock);
  return idx;
}
```

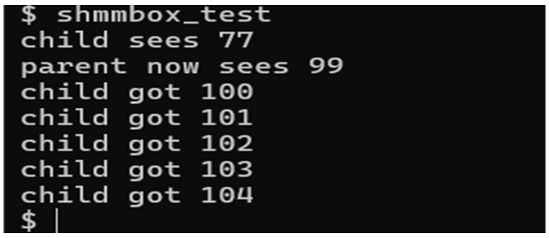
(e) mbox_create_k



```c
int
mbox_recv_k(int id, int *out)
{
  if(id < 0 || id >= NMB) return -1;
  struct mbox *m = &mbs.box[id];
  if(!m->used) return -1;

  acquire(&m->lock);
  while(m->cnt == 0)
    sleep(m, &m->lock);              // wait for data
  int v = m->buf[m->head];
  m->head = (m->head + 1) % MBOX_CAP;
  m->cnt--;
  wakeup(m);                         // wake one+ senders
  release(&m->lock);

  *out = v;
  return 0;
}
```

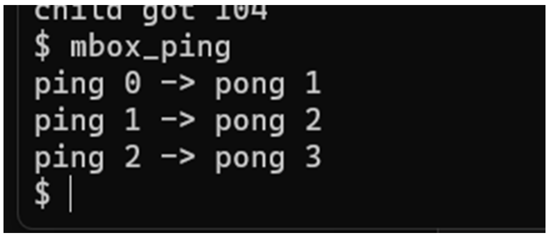
(f) shm_recv_k

Figure 1.3: Functions in shm.c and mbox.c



```
$ shmmbox_test
child sees 77
parent now sees 99
child got 100
child got 101
child got 102
child got 103
child got 104
$ |
```

(a) shm_test



```
child got 104
$ mbox_ping
ping 0 -> pong 1
ping 1 -> pong 2
ping 2 -> pong 3
$ |
```

(b) mbox_test

Figure 1.4: Output of our program

Chapter 2

Task 2: The Intertwined Memory Challenge

2.1 Introduction

In this task, we applied our understanding of shared memory and message passing to solve a synchronization and data-sharing problem. The challenge involves two processes navigating an intertwined path stored in shared memory. Neither process is aware of its own path and must rely on the other process for directions. To address this challenge, we used the primitives created in Task 1 to build a concurrent application, where a team of two processes must communicate and cooperate to successfully navigate the shared data structure.

2.2 Master Program (master.c)

- Create & map one shared page using `shm_create` and `shm_get`.
- Define shared structure: arrays `a2b[]`, `b2a[]`, and `startA`, `startB`.
- Build intertwined paths from Aseq and Bseq, linking moves via `a2b`, `b2a`.
- Create two mailboxes (`A2BKEY`, `B2AKEY`) for communication.
- Spawn two processes with roles (A: send→recv, B: recv→send); exchange END as termination signal.
- Wait for both children, close shared memory, print completion.

2.3 Process Program (process.c)

- Each player (A or B) attaches to shared memory (`a2b`, `b2a`) and mailboxes.
- Protocol: A = send→recv, B = recv→send.
- Each uses teammate's index to compute next move (`b2a` for A, `a2b` for B).
- Prints progress atomically; logger prevents garbled output.
- On END: send END, perform END/ACK handshake for clean termination.

2.4 Output

Execution of our program shows the following output:

- Process B: at 0 → next 4 (peer 0).
- Process A: at 0 → next 3 (peer 0).
- ...
- Process B: reached `END`.
- Process A: reached `END`.
- Master: both processes finished.

2.5 Deadlock Avoidance

2.5.1 Problem

The application involves two cooperating processes, A and B, that must repeatedly exchange their current positions through mailboxes and then use the peer's position to compute their own next step from shared memory. However, if both processes attempt to receive before sending, neither message can be delivered, causing both to block indefinitely. This situation represents a classic circular wait, where process A waits on process B and process B simultaneously waits on process A.

2.5.2 Protocol

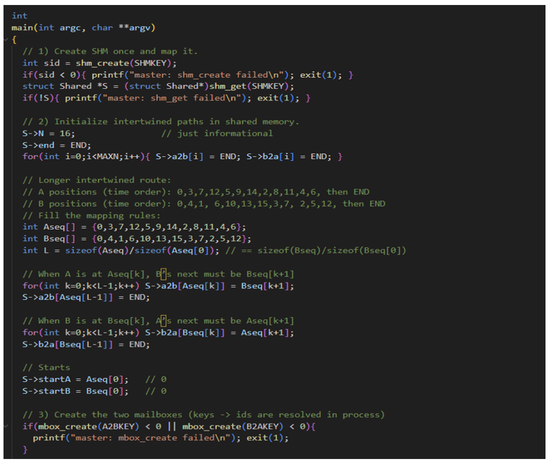
- **Process A (role 0):** In each round, A sends its position to B, then waits to receive B's position. Using B's data, A computes its next step.
- **Process B (role 1):** B first receives A's position), computes its own next step, then replies with its current position .

This creates a lockstep rendezvous: A's send always matches B's receive, and B's send always matches A's receive. Because the roles are asymmetric, both cannot be stuck waiting to receive at the same time.

2.5.3 Termination

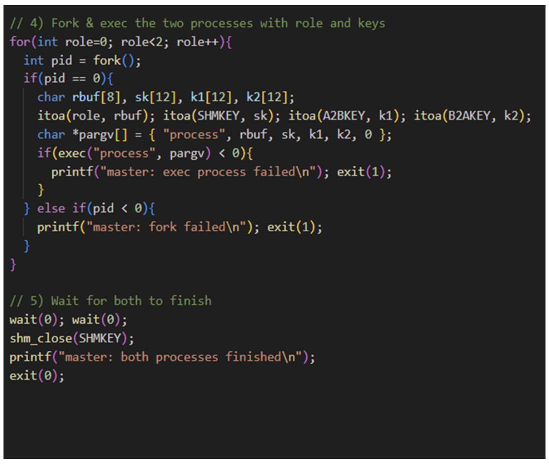
- **If A finishes first:** A sends `END`. B (waiting to receive) immediately gets it, replies with `END`, and exits. A then receives this acknowledgment and exits.
- **If B finishes first:** Since B always starts with receive, it cannot block indefinitely. On receiving `END`, it replies with `END` and exits; otherwise it processes one more step until the final handshake occurs.

The two-way `END` exchange ensures clean termination with no process left waiting for a message that never arrives.



```c
int
main(int argc, char **argv)
{
  // 1) Create SHM once and map it.
  int sid = shm_create(SHMKEY);
  if(sid < 0){ printf("master: shm_create failed\n"); exit(1); }
  struct Shared *S = (struct Shared*)shm_get(SHMKEY);
  if(!S){ printf("master: shm_get failed\n"); exit(1); }

  // 2) Initialize intertwined paths in shared memory.
  S->N = 16;                // just informational
  S->end = END;
  for(int i=0;i<MAXN;i++){ S->a2b[i] = END; S->b2a[i] = END; }

  // Longer intertwined route:
  // A positions (time order): 0,3,7,12,5,9,14,2,8,11,4,6, then END
  // B positions (time order): 0,4,1, 6,10,13,15,3,7, 2,5,12, then END
  // Fill the mapping rules:
  int Aseq[] = {0,3,7,12,5,9,14,2,8,11,4,6};
  int Bseq[] = {0,4,1,6,10,13,15,3,7,2,5,12};
  int L = sizeof(Aseq)/sizeof(Aseq[0]); // == sizeof(Bseq)/sizeof(Bseq[0])

  // When A is at Aseq[k], B’s next must be Bseq[k+1]
  for(int k=0;k<L-1;k++) S->a2b[Aseq[k]] = Bseq[k+1];
  S->a2b[Aseq[L-1]] = END;

  // When B is at Bseq[k], A’s next must be Aseq[k+1]
  for(int k=0;k<L-1;k++) S->b2a[Bseq[k]] = Aseq[k+1];
  S->b2a[Bseq[L-1]] = END;

  // Starts
  S->startA = Aseq[0];   // 0
  S->startB = Bseq[0];   // 0

  // 3) Create the two mailboxes (keys -> ids are resolved in process)
  if(mbox_create(A2BKEY) < 0 || mbox_create(B2AKEY) < 0){
    printf("master: mbox_create failed\n"); exit(1);
  }
```

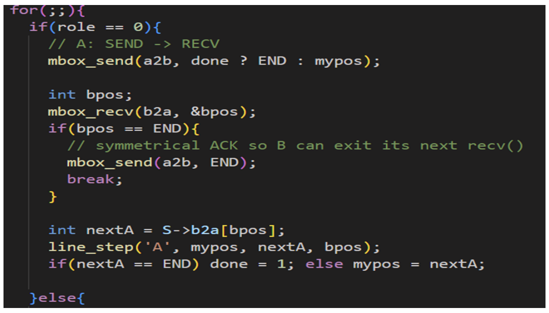
(a) Initialization and mapping of shared memory



```c
// 4) Fork & exec the two processes with role and keys
for(int role=0; role<2; role++){
  int pid = fork();
  if(pid == 0){
    char rbuf[8], sk[12], k1[12], k2[12];
    itoa(role, rbuf); itoa(SHMKEY, sk); itoa(A2BKEY, k1); itoa(B2AKEY, k2);
    char *pargv[] = { "process", rbuf, sk, k1, k2, 0 };
    if(exec("process", pargv) < 0){
      printf("master: exec process failed\n"); exit(1);
    }
  } else if(pid < 0){
    printf("master: fork failed\n"); exit(1);
  }
}

// 5) Wait for both to finish
wait(0); wait(0);
shm_close(SHMKEY);
printf("master: both processes finished\n");
exit(0);
```

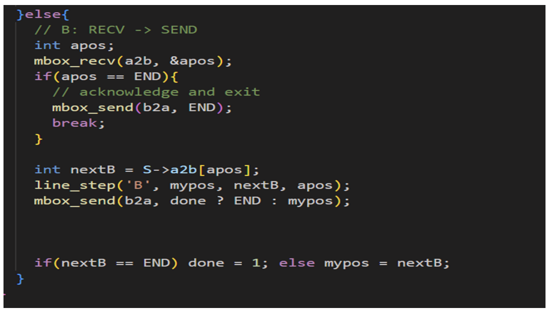
(b) Forking child processes and synchronizing execution

Figure 2.1: Core implementation and changes in master.c



```c
for(;;){
  if(role == 0){
    // A: SEND -> RECV
    mbox_send(a2b, done ? END : mypos);

    int bpos;
    mbox_recv(b2a, &bpos);
    if(bpos == END){
      // symmetrical ACK so B can exit its next recv()
      mbox_send(a2b, END);
      break;
    }

    int nextA = S->b2a[bpos];
    line_step('A', mypos, nextA, bpos);
    if(nextA == END) done = 1; else mypos = nextA;

  }else{
```

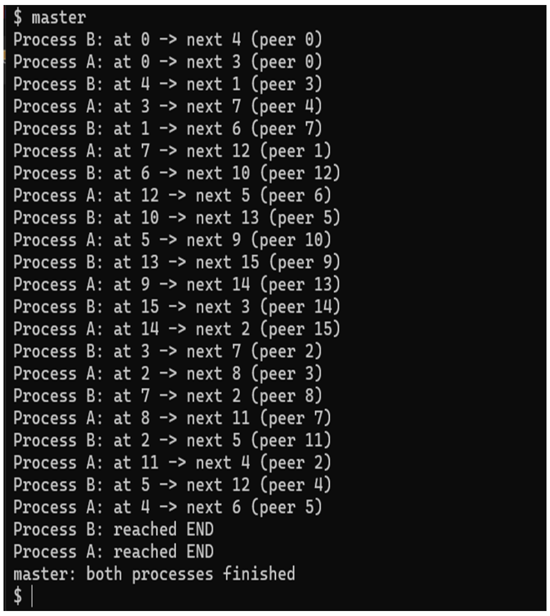
(a) Code for Player 1 to share its position with the team



```c
}else{
  // B: RECV -> SEND
  int apos;
  mbox_recv(a2b, &apos);
  if(apos == END){
    // acknowledge and exit
    mbox_send(b2a, END);
    break;
  }

  int nextB = S->a2b[apos];
  line_step('B', mypos, nextB, apos);
  mbox_send(b2a, done ? END : mypos);


  if(nextB == END) done = 1; else mypos = nextB;
}
```

(b) Code for Player 2 to share its position with the team

Figure 2.2: Core implementation and changes in process.c



```
$ master
Process B: at 0 -> next 4 (peer 0)
Process A: at 0 -> next 3 (peer 0)
Process B: at 4 -> next 1 (peer 3)
Process A: at 3 -> next 7 (peer 4)
Process B: at 1 -> next 6 (peer 7)
Process A: at 7 -> next 12 (peer 1)
Process B: at 6 -> next 10 (peer 12)
Process A: at 12 -> next 5 (peer 6)
Process B: at 10 -> next 13 (peer 5)
Process A: at 5 -> next 9 (peer 10)
Process B: at 13 -> next 15 (peer 9)
Process A: at 9 -> next 14 (peer 13)
Process B: at 15 -> next 3 (peer 14)
Process A: at 14 -> next 2 (peer 15)
Process B: at 3 -> next 7 (peer 2)
Process A: at 2 -> next 8 (peer 3)
Process B: at 7 -> next 2 (peer 8)
Process A: at 8 -> next 11 (peer 7)
Process B: at 2 -> next 5 (peer 11)
Process A: at 11 -> next 4 (peer 2)
Process B: at 5 -> next 12 (peer 4)
Process A: at 4 -> next 6 (peer 5)
Process B: reached END
Process A: reached END
master: both processes finished
$ |
```

Figure 2.3: Output of our program

Bibliography

[1] R. Cox, F. Kaashoek, R. Morris. *xv6: a simple, Unix-like teaching operating system.*

[2] A. Silberschatz, P. Galvin, G. Gagne. *Operating System Concepts.*