CS344 Assignment 3

Group-3, CSE

August 30, 2025

Team Members

- **230101024** (Baswa Dhanush Sai)
- **230101092** (Seella Mythresh Babu)
- **230101114** (Yeredla Koushik Reddy)
- **230101120** (Taes Padhihary)

Please find the Google Drive folder containing all the edited files here:

drive.google.com/drive/folders/1RHZeH6g_jVeL2LWed-yeC2ZoqEYwA2R1?usp=sharing

Contents

---

1. Task 3.1: Kernel Primitives for IPC

Objective: *To implement two core primitives to enable complex process coordination: shared memory for efficient data sharing and mailboxes for reliable messaging between processes.*

1.1. System calls for shared memory

We created kernel/shm.h and kernel/shm.c. We have listed some of the major changes in this report. The list of details can be found in the Google Drive folder above.

Kernel design. The kernel keeps a global table of shared pages in kernel/shm.c and a shared page structure **struct** shm_region is defined in kernel/shm.h:

```
// kernel/shm.c
static struct {
  struct spinlock lock;
  struct shm_region reg[MAX_SHM];
} shm;                          // table of all shared pages

// kernel/shm.h
#define MAX_SHM   64
#define SHM_BASE  ((uint64)0x40000000ULL)
#define SHMVA(slot) (SHM_BASE + ((uint64)(slot) * PGSIZE)) // virtual address slots

struct shm_region {
  int used;     // 1 if allocated
  int key;      // application key
  char *page;     // physical page
  int ref_count;   // mappings of the processes
};

void shminit(void);
int shm_create(int key); // returns the id if successful, −1 if failed
uint64 shm_get(int key); // returns the VA if successful, 0 if failed
int shm_close(int key); // returns 0 if successful, −1 if failed
void shm_cleanup(struct proc *p); // unmap any shared memory for p
```

struct shm_region is used to define a shared page. The table **struct** shm is initialized once. Every slot records whether it is used, the user key, its physical page, and a reference count. **All table operations are serialized by shm.lock (TABLE LOCK).**

We added shminit() to kernel/main.c that'll initialize the shm table of 64 available shared pages.

shm_create(key).

1. Acquire shm.lock; scan for an existing slot with key.
2. If absent, pick a free slot, set used=1,key=key,ref_count=0, and allocate a zeroed page (using kalloc ()+memset).
3. Release the lock; return the slot index (or −1 on failure).

shm_get(key).

1. Acquire shm.lock, find the slot and compute its virtual address using va = SHMVA(slot).
2. If the PTE is not present in the caller, mappages maps the page at va with PTE_R|PTE_W|PTE_U and then do ref_count++. Return va as uint64.

```
uint64 va = SHMVA(s);
pte_t *pte = walk(p−>pagetable, va, 0);
if (pte == 0 || ((*pte) & PTE_V) == 0)
{
   if ( mappages(p−>pagetable, va, PGSIZE, (uint64)shm.reg[s].page,
   PTE_R|PTE_W|PTE_U) < 0 ) {
      release(&shm.lock);
      return 0;
   }
   shm.reg[s].ref_count++;
} // . . .
```

shm_close(key) and exit cleanup.

1. Under shm.lock, if the caller has a valid PTE at SHMVA(slot), uvmunmap it and decrement ref_count.
2. If ref_count==0, kfree(page) and clear the slot (used=0, page=0). The same logic is used in **void** shm_cleanup(proc∗) at process exit to avoid leaks. This is then added to kernel/proc.c under **static void** freeproc(**struct** proc ∗p).

```
// kernel/proc.c
// . . .
static void
freeproc(struct proc *p)
{
  // . . .
  if (p->pagetable) {
    // Task 3.1
    shm_cleanup(p);
    proc_freepagetable(p->pagetable, p->sz);
  }
  // . . .
}
```

This is done because proc_freepagetable(p–>pagetable, p–>sz) unmaps only 0 to PGROUNDUP(sz). But our shared memory pages do not lie in that zone.

User Space. We added the following three system calls in user space (prototypes added to user/user.h, and then to user/usys.pl to generate Assembly code):

```
int   shm_create(int key);   // allocate slot+page for key
void* shm_get(int key);      // map backing page at fixed VA
int   shm_close(int key);    // unmap from caller; free page
```

1.2. System calls for mailboxes

Kernel design. We keep a fixed array of queues; each queue has its own spinlock.

```
// kernel/mbox.c
static struct {
  struct mailbox box[MAX_MBOX];
} mboxes;

// kernel/mbox.h
struct mailbox {
  struct spinlock lock;     // per-queue lock
  int used, key;
  int buf[MBOX_CAP];
  int head, tail, count;
  int closed;
};
```

We initialize with used=key=head=tail=count=closed=0. This initializes each lock.

mbox_create(key) first does an unlocked search: if not found, it acquires each queue's lock and claims the first free slot by setting used=1,key=key and the rest as 0. We do not keep a global table lock and hence duplicate mailboxes may be formed having the same key. **So, for mailboxes we will use different keys every time we run the code.**

Send/receive and close. Both send and receive follow the xv6 sleep/wakeup pattern with the *same* channel and the per-queue lock held:

```
// send: block while full; error if closed
acquire(&b->lock);
while (b->count == MBOX_CAP) sleep(b, &b->lock);
if (b->closed) { release(&b->lock); return −1; }
b->buf[b->tail] = v; b->tail = (b->tail+1)%MBOX_CAP; b->count++;
wakeup(b);
release(&b->lock);

// recv: block while empty and not closed; EOF if empty+closed
acquire(&b->lock);
while (b->count == 0 && b->closed == 0) sleep(b, &b->lock);
if (b->count == 0 && b->closed) { release(&b->lock); return −1; }
*v = b->buf[b->head]; b->head = (b->head+1)%MBOX_CAP; b->count−−;
wakeup(b);
release(&b->lock);

// close: mark closed and wake sleepers
acquire(&b->lock);
b->closed = 1; wakeup(b);
release(&b->lock);
```

We get the expected producers/consumers block. After closing, a blocked receiver returns EOF.

User Space. We added the following four system calls in user space (prototypes added to user/user.h, and then to user/usys.pl to generate Assembly code):

```
int mbox_create(int key);        // return id >= 0, reusable per key
int mbox_send(int id, int v);    // block if full; 0 on success, −1 on closed/error
int mbox_recv(int id, int *v);   // block if empty; 0 on success, −1 on EOF/closed
int mbox_close(int id);          // set closed=1; wake all sleepers; free slot
```

Then we added the test files user/shmtest.c and shm/mboxtest.c to the Makefile UPROGS.

1.3. Handling race conditions and deadlocks

Shared memory. (Details in subsection 1.1) All updates are under shm.lock, so we guarantee one slot per key, consistent refcounts, and "free on last close". There are no sleeps.

Mailboxes. (Details in subsection 1.2)

- All mailboxes are created under mboxes.tbl_lock. So, unique mailboxes are created for each key.
- For a given mailbox id, head/tail/count and the buffer are protected by that queue's lock; blocking uses **while**(cond) sleep(chan,&lock) / wakeup(chan) to avoid lost wakeups.

Deadlock avoidance.

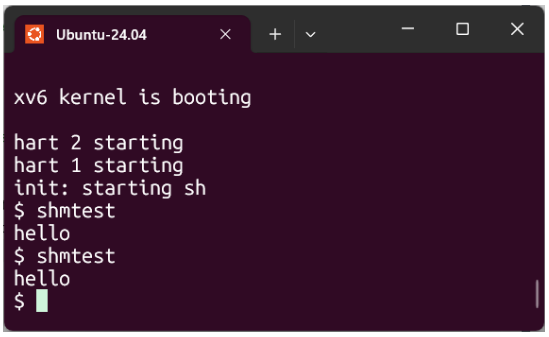
- **SHM:** no sleeps while holding shm.lock; all mapping calls are short and non-blocking.
- **Mailboxes:** senders block only when full; receivers block only when empty; mbox_close wakes sleepers to prevent shutdown hangs.

1.4. Test Programs

1.4.1. Test Program 1: shmtest.c

Sketch. The following code is a sketch of user/shmtest.c and does not include error handling.

```
int key = 1;
int id  = shm_create(key);
char *p = shm_get(key);
if (fork()==0) {
  char *q = shm_get(key);
  strcpy(q, "hello");
  exit(0);
}
wait(0);
printf("%s\n", p); // should print hello
shm_close(key);
```



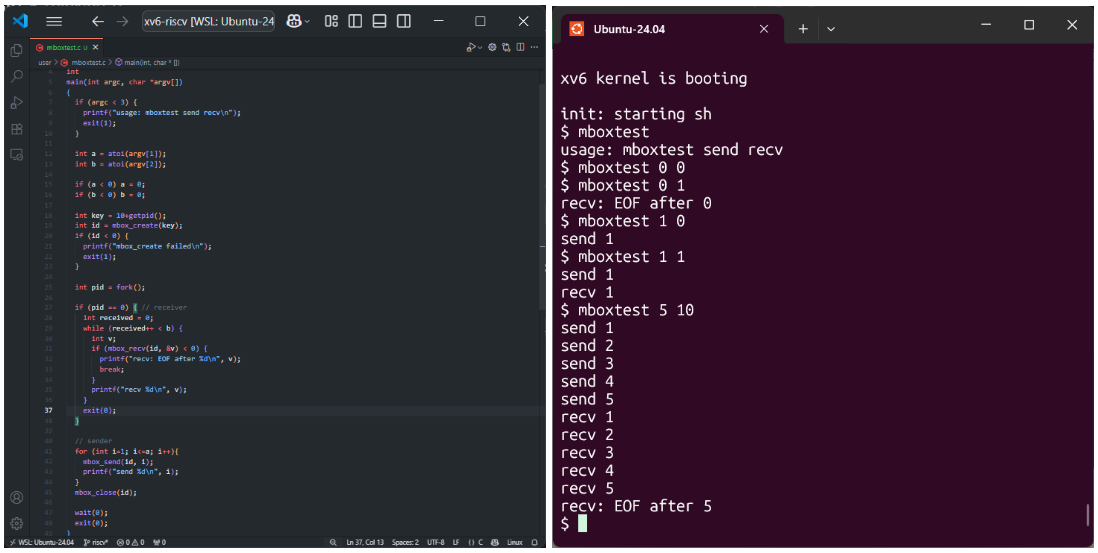
```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ shmtest
hello
$ shmtest
hello
$ 
```

Expected output: a single line hello. Re-running shmtest.c does not require a different key as the slot is reclaimed on last close (see Shared memory). Reference

1.4.2. Test Program 2: mboxtest.c

Sketch. Round-trip integers over a mailbox; confirm blocking behavior and EOF on close. In Figure 1 [left], the usage is mboxtest a b where a is the number of messages to be sent and b is the number of messages to be received. We are able to reuse the command mboxtest because the line: key = 10 + getpid() in user/mboxtest.c changes the key each time and hence a new mailbox is assigned.



```
int
main(int argc, char *argv[])
{
  if (argc < 3) {
    printf("usage: mboxtest send recv\n");
    exit(1);
  }

  int a = atoi(argv[1]);
  int b = atoi(argv[2]);

  if (a < 0) a = 0;
  if (b < 0) b = 0;

  int key = 10+getpid();
  int id = mbox_create(key);
  if (id < 0) {
    printf("mbox_create failed\n");
    exit(1);
  }

  int pid = fork();

  if (pid == 0) { // receiver
    int received = 0;
    while (received++ < b) {
      int v;
      if (mbox_recv(id, &v) < 0) {
        printf("recv: EOF after %d\n", v);
        break;
      }
      printf("recv %d\n", v);
    }
    exit(0);
  }

  // sender
  for (int i=1; i<=a; i++){
    mbox_send(id, i);
    printf("send %d\n", i);
  }
  mbox_close(id);

  wait(0);
  exit(0);
}
```

```
xv6 kernel is booting

init: starting sh
$ mboxtest
usage: mboxtest send recv
$ mboxtest 0 0
$ mboxtest 0 1
recv: EOF after 0
$ mboxtest 1 0
send 1
$ mboxtest 1 1
send 1
recv 1
$ mboxtest 5 10
send 1
send 2
send 3
send 4
send 5
recv 1
recv 2
recv 3
recv 4
recv 5
recv: EOF after 5
$ 
```

Figure 1. [left] user/mboxtest.c (please zoom in); [right] different outputs

Expected output: See Figure 1 [right] (we have compiled using `make CPUS=1 qemu`). Details of the implementation are in System calls for mailboxes.

- mboxtest 0 0 yields nothing since loops do not run.
- mboxtest 0 1 gives EOF after 0 because mailbox is empty and mbox_close sets closed=1.

- mboxtest 1 1 gives "send 1\n receive 1" proving the correctness. The receiver does not send any mbox_close sets closed=1 and wakeups sleepers; empty+closed yields EOF.
- mboxtest 5 10 gives output analogous to mboxtest 0 1 with EOF called after 5.

2. Task 3.2: Intertwined Memory Challenge

Objective: *To apply understanding of shared memory and message-passing to solve a synchronization and data-sharing problem. The challenge involves two processes that navigate an "intertwined" path stored in shared memory, where neither process knows its own path and must rely on the other process for directions.*

Traversal Logic

This occurs on a single shared memory page. This page contains two or more intertwined paths. An intertwined path means that the data at Process A's current location holds the index for Process B's next location, and vice versa. The shared memory page contains the structure of the maze.

- Each process (process A and B) begins at a specific starting address within shared memory.
- The path is "intertwined", i.e. the value stored at process A's current location is the next address for process B and the value at process B's current location is the next address for process A.
- To proceed, A must communicate its current location (using message passing) to process B. Process B then reads the value at that location to discover its own next move. During the process, B must communicate its location with process A, so process A can find its next location.
- This cycle of communication and movement continues until a process's determined next address is a special end-marker value (here, we set it as -1). A team wins only when both processes have reached their respective end-markers.

To implement the above goal, we implemented a shared page (in the code shown by **struct** details):

- The shared page holds:
  - a length $L \leq 255$, an end marker -1, and two **user-provided** starts startA, startB;
  - two arrays next_for_A[256], next_for_B[256];
  - win flags doneA, doneB.

```
struct details {
  int L, end, startA, startB;                    // length, end−marker, start of A, start of B
  int next_for_A[256], next_for_B[256];   // next positions of A and B [hard−coded]
  int doneA, doneB;                              // doneA = 1 when B reaches end and vice versa
};
```

- **Protocol is asymmetric to avoid deadlock**: A does *send→recv* and B does *recv→send*.

2.1. Design of master.c

The master creates the shared page, fills the plan, creates two mailboxes (A→B and B→A), then fork+execs two copies of process. The tests are isolated because each run has unique keys.

```
// fresh keys every run (master.c)
int base   = 10 + getpid();
int SHM_KEY= base;           // shared page
int AB_KEY = base + 100;   // mailbox A−>B
int BA_KEY = base + 101;   // mailbox B−>A
```

On the SHM page we initialize the intertwined plan (length of maze, startA and startB come from the user as inputs) and the two done flags:

Listing 1. maze plan

```
g->L = atoi(argv[1]); g->end = -1; g->startA = atoi(argv[2]); g->startB = atoi(argv[3]);
g->doneA = g->doneB = 0;

// maze plan for the "values" of next addresses for A and B
for (int i=0; i<g->L; i++) {
  g->next_for_A[i] = ((i+1) < g->L ? (i+1) : g->end);
  g->next_for_B[i] = ((i+2) < g->L ? (i+2) : g->end);
}
```

We then create mailboxes and exec the two roles with the four integers passed as argv to process:

```
if (fork() == 0) {
  char role[8], shm[8], ab[8], ba[8];
  itoa10(0, role); itoa10(SHM_KEY, shm); itoa10(mail_ab, ab); itoa10(mail_ba, ba);
  char *args[] = {"process", role, shm, ab, ba, 0};
  exec("process", args);
  exit(1);
}
if (fork() == 0) {
  char role[8], shm[8], ab[8], ba[8];
  itoa10(1, role); itoa10(SHM_KEY, shm); itoa10(mail_ab, ab); itoa10(mail_ba, ba);
  char *args[] = {"process", role, shm, ab, ba, 0};
  exec("process", args);
  exit(1);
}
```

We added the function itoa10 to user space to convert an integer to string in user/ulib.c. After both children exit, master checks the two flags on SHM to decide the result:

```
wait(0); wait(0);
if (g->doneA && g->doneB) printf("WIN: both processes reached end\n");
else if (g->doneA) printf("LOSS: B didn't finish\n");
else if (g->doneB) printf("LOSS: A didn't finish\n");
else printf("LOSS: neither finished\n");
```

2.2. Design of process.c

Each process receives its role and communication parameters through argv from above.

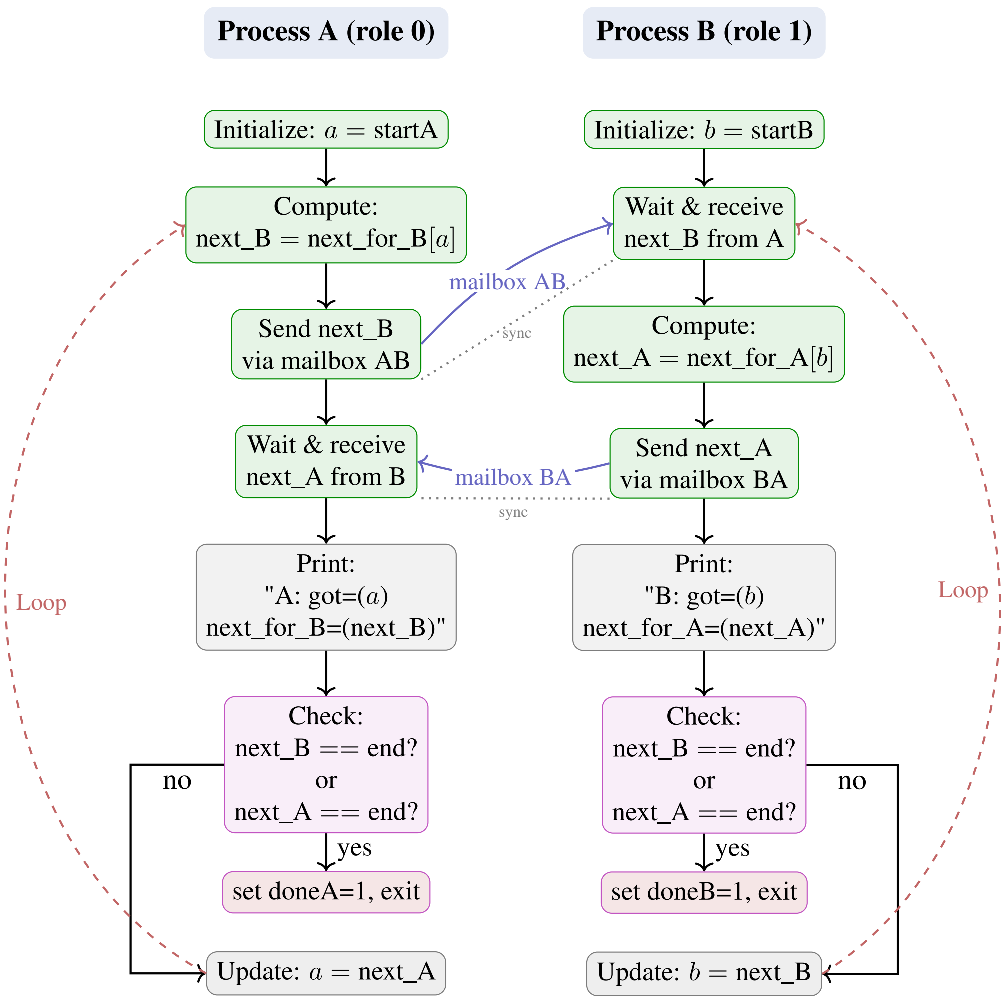
Process A (Role 0) Logic Process A sends first:

```
int a = g->startA;
while(1) {
  int next_B = (a >= 0 && a < g->L) ? g->next_for_B[a] : g->end;
  mbox_send(mail_ab, next_B);
  printf("A:got=%d next_for_B=%d\t", a, next_B);
  if (next_B == g->end) { g->doneA = 1; shm_close(SHM_KEY); exit(0); }
  int next_A = recv_loc(mail_ba);
  if (next_A == g->end) { g->doneA = 1; shm_close(SHM_KEY); exit(0); }
  a = next_A;
}
```

Process B (Role 1) Logic Process B follows by receiving first:

```
int b = g->startB;
while(1) {
  int next_B = recv_loc(mail_ab);
  int next_A = (b >= 0 && b < g->L) ? g->next_for_A[b] : g->end;
  mbox_send(mail_ba, next_A);
  printf("B: next_for_A=%d got=%d\n", next_A, b);
  if (next_B == g->end) { g->doneB = 1; shm_close(SHM_KEY); exit(0); }
  if (next_A == g->end) { g->doneB = 1; shm_close(SHM_KEY); exit(0); }
  b = next_B;
}
```

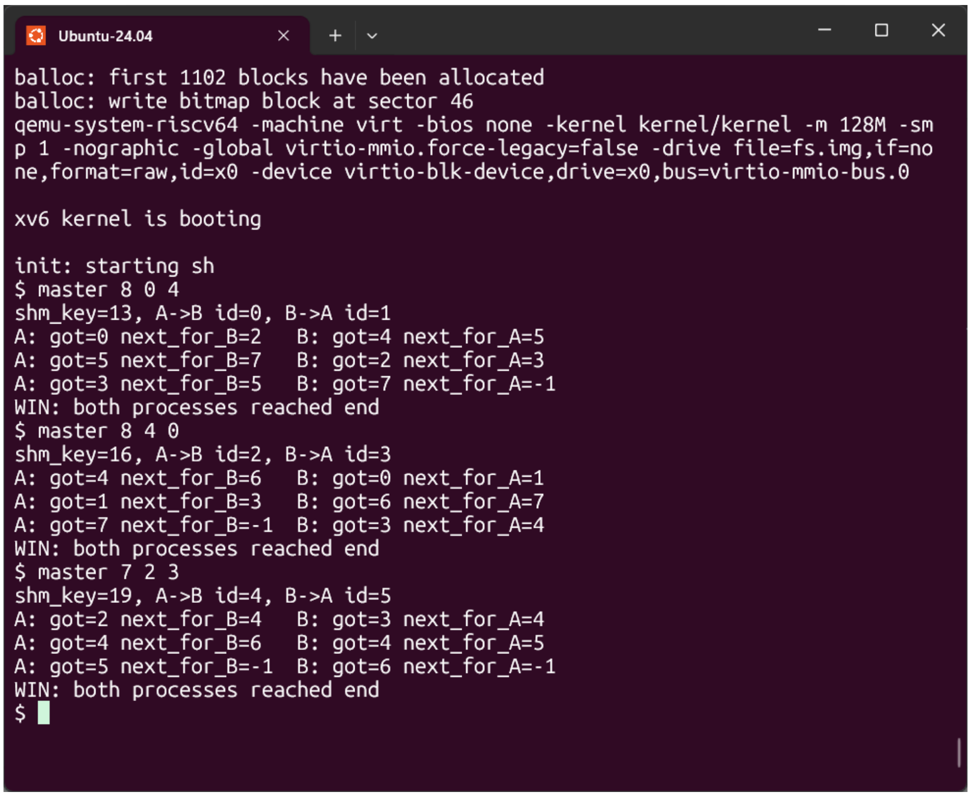
Finally add both the files user/master.c and user/process.c to UPROGS in Makefile.


Process A (role 0)
Process B (role 1)
Initialize: a = startA
Initialize: b = startB
Compute: next_B = next_for_B[a]
Wait & receive next_B from A
mailbox AB
sync
Send next_B via mailbox AB
Compute: next_A = next_for_A[b]
Wait & receive next_A from B
mailbox BA
Send next_A via mailbox BA
sync
Print: "A: got=(a) next_for_B=(next_B)"
Print: "B: got=(b) next_for_A=(next_A)"
Loop
Loop
Check: next_B == end? or next_A == end?
no
yes
set doneA=1, exit
set doneB=1, exit
Update: a = next_A
Update: b = next_B

Expected output

If we use `make CPUS=1 qemu` then the output formatting will be readable. More CPUs may interleave the output prints since printf is not atomic. We present three test cases (outputs in Figure 2:

1. master 8 0 4: A starts at 0 and B starts at 4. Using the maze plan, A goes to 5, B to 2, A to 3, B to 7, A to 8 $\implies$ -1 (end) and B to 5 (will not be printed) which means both A and B exit. Output is verified.



```
balloc: first 1102 blocks have been allocated
balloc: write bitmap block at sector 46
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -sm
p 1 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=no
ne,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ master 8 0 4
shm_key=13, A->B id=0, B->A id=1
A: got=0 next_for_B=2   B: got=4 next_for_A=5
A: got=5 next_for_B=7   B: got=2 next_for_A=3
A: got=3 next_for_B=5   B: got=7 next_for_A=-1
WIN: both processes reached end
$ master 8 4 0
shm_key=16, A->B id=2, B->A id=3
A: got=4 next_for_B=6   B: got=0 next_for_A=1
A: got=1 next_for_B=3   B: got=6 next_for_A=7
A: got=7 next_for_B=-1  B: got=3 next_for_A=4
WIN: both processes reached end
$ master 7 2 3
shm_key=19, A->B id=4, B->A id=5
A: got=2 next_for_B=4   B: got=3 next_for_A=4
A: got=4 next_for_B=6   B: got=4 next_for_A=5
A: got=5 next_for_B=-1  B: got=6 next_for_A=-1
WIN: both processes reached end
$
```

Figure 2. Various outputs under `CPUS=1` in task 3.2

2. master 8 4 0: A starts at 4 and B starts at 0. Using the maze plan, A goes to 1, B to 6, A to 7, B to 3, A to 4 and B to 9 $\implies -1$ (end) which means both A and B exit now. Output is verified.
3. master 7 2 3: A starts at 2 and B starts at 3. Using the maze plan, A goes to 4, B to 4, A to 5, B to 6, A to 7 $\implies -1$ (end) and B to 7 $\implies -1$ (end) which means both A and B exit now. Checked.

2.3. Deadlock avoidance

- **No "both waiting" deadlock:** asymmetric protocol (A: send→recv, B: recv→send). If both started with recv, they'd block forever on empty queues.
- **No "send before receive" deadlock:** The strict ordering prevents scenarios where both processes are trying to send simultaneously with full mailboxes. A sends exactly one message before waiting to receive, and B receives exactly one message before sending.
- **No "infinite loop" deadlock:** Both processes check for the end marker after sending and after receiving. When either process determines it should send end, it sends the marker and exits. The receiving process detects this and also exits.
- **No lost wakeups:** both send/recv use **while**(cond) sleep(b,&b–>lock); with wakeup(b). The sleep atomically releases/reacquires the same lock, so a wakeup cannot be missed.
- **Clean termination:** When a process computes `next == end`, it either:
  - Sends `end` to unblock its peer, sets its `done` flag, and exits
  - Receives `end` from its peer, sets its `done` flag, and exits

  The master process checks `doneA && doneB` to determine if both completed successfully.
- **No race conditions on shared memory:** The `done` flags are written once per process (A only writes `doneA`, B only writes `doneB`). The next_for_A and next_for_B arrays are read-only after initialization and so, no locks are needed for them.