OS-344 Lab - 3 (Revised): Inter-Process Communication

Instructions

- This assignment must be completed in a group. One member should submit the completed assignment before the deadline.
- Each group must submit a single report containing relevant screenshots, data, and findings.
- All submissions will be checked for plagiarism; plagiarized submissions will be heavily penalized.
- A viva will be conducted for this assignment, and attendance for all group members is mandatory. Your final grade will be based on the submitted code, the report, and the viva performance.
- Start early, as no extensions will be granted.
- If any information is missing, you may make reasonable assumptions. All assumptions must be clearly documented in your report.
- Before you start coding, read Chapters 3, 6 and 7 of the xv6 book.

Task 1: Implement Kernel Primitives for IPC

In this task, you'll add fundamental Inter-Process Communication (IPC) mechanisms to the xv6 kernel: a shared memory system for data exchange and a mailbox system for synchronization and messaging.

Objective: The default xv6 kernel's IPC capabilities are limited. You will implement two core primitives to enable more complex process coordination: shared memory for efficient data sharing and mailboxes for reliable messaging between processes.

1. Add System Calls for Shared Memory

- `int shm_create(int key)`: Creates a new shared memory region of one page associated with an integer `key`. If a region with this key already exists, it should not create a new one. It returns a handle or identifier for this region.
- `void* shm_get(int key)`: Allows a process to access (or "attach") to the shared memory region identified by `key`. This call should map the physical page associated with the key into the process's virtual address space and return a pointer to it. All processes that call `shm_get` with the same key should receive a pointer to the same physical page.
- `int shm_close(int key)`: Detaches the shared memory region from the current process's address space. The physical page should only be freed when the last process attached to it calls close.

Implementation Hints:

- Use `kalloc()` to allocate a physical page for a new shared memory region.
- Use `mappages()` to map this page into a process's user space.
- You will need a global table to keep track of active shared memory regions (e.g., an array mapping keys to physical page addresses) and a reference counter for each region.
- Protect this global table with a **spinlock** to prevent race conditions.

2. Add System Calls for Mailboxes

- `int mbox_create(int key)`: Creates a message queue (mailbox) identified by a `key`. This mailbox will be used for passing integer messages.
- `int mbox_send(int mbox_id, int msg)`: Sends an integer message `msg` to the mailbox identified by `mbox_id`. If the mailbox is full, the process should block until space becomes available.
- `int mbox_recv(int mbox_id, int *msg)`: Receives an integer message from the mailbox `mbox_id` and stores it in the location pointed to by `msg`. If the mailbox is empty, the process should block until a message arrives.

Implementation Hints:

- You can implement mailboxes as a global array of simple, fixed-size buffers (e.g., a circular queue).
- Use the `sleep` and `wakeup` mechanisms (Chapter 7 of the xv6 book) to handle blocking for send/receive operations.
- Handle deadlock that may arise.

Deliverables for Task 1

1. Modified kernel files implementing the system calls (`sysproc.c`, `usys.S`, `syscall.h`, etc.).
2. New kernel files for the core logic (e.g., `shm.c`, `mbox.c`).
3. Any necessary changes to kernel headers (`defs.h`, `proc.h`).
4. At least two test programs: one to demonstrate that two processes can read and write to the same shared memory, and another to show one process sending a message to another via a mailbox.

Task 2: The Intertwined Memory Challenge

In this task, you will use the primitives created in Task 1 to build a concurrent application where a team of two processes must communicate and cooperate to navigate a shared data structure.

Objective: Apply your 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 other process for directions.

1. The Traversal Logic

This happens in 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 process B) begins at a specific starting **address** within this shared memory.
- The path is "intertwined": the value stored at process A's current location is actually the **next address** for process B.
- Conversely, the value at process B's current location is the **next address** for process A.
- Neither process knows its own path. To advance, process 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.
- Simultaneously, process B must communicate its location to process A, so process A can find its next move.
- This cycle of communication and movement continues until a process's determined next address is a special end-marker value (e.g., `0xFFFF`). A team wins only when both processes have reached their respective end-markers.

2. Program Requirements

Master Program (`master.c`)

- This program sets up the game.
- It must use `shm_create()` to allocate the shared memory page.
- It initializes the intertwined paths within the shared memory. You can hardcode the paths as shown in the example code.
- It must use `mbox_create()` to create two mailboxes for each team: one for P1 to send to P2, and another for P2 to send to P1.
- It then uses `fork()` to create the two processes. The starting positions can be passed as command-line arguments via `exec()`, or the process can have fixed starting positions.

process Program (`process.c`)

- This program contains the logic for a single process process.
- It must attach to the shared memory using `shm_get()`.
- It navigates the path step-by-step using a strict communication protocol:
    1. Send its current location to another process using `mbox_send()`.

    2. Wait and receive another process's current location using `mbox_recv()`.
    3. Use the location received from another process to look up its *own* next location in the shared memory.
    4. Update its local position and repeat the loop.
- **Deadlock Avoidance:** A deadlock will occur if both processes wait to receive a message before sending one. Your design **must** prevent this. The simplest strategy is to establish a fixed order: one designated process (e.g., the parent process) always sends first and then receives, while the other process (the child) always receives first and then sends. Document this strategy in your report.

Deliverables for Task 2

1. The source code for your `master` and `process` user-space programs.
2. An updated `Makefile` to include your new programs in `UPROGS`.
3. A section in your report that includes:
    - A brief overview of your path structure in the shared memory page.
    - A clear explanation of your communication protocol.
    - Your deadlock avoidance strategy.
    - Screenshots showing the traversal in action, with print statements demonstrating the step-by-step progress of both processes.