Wednesday, 20 August 2025

OS-344 Lab - 3

Instructions

- This assignment must be completed in a group.
- Group members should ensure that one member submits the completed assignment before the deadline.
- Each group should submit a report, with relevant screenshots/ necessary data and findings related to the assignments.
- We expect a sincere and fair effort from your side. All submissions will be checked for plagiarism through software, and plagiarised submissions will be penalised heavily, irrespective of the source and copy.
- There will be a viva associated with the assignment. Attendance of all group members is mandatory.
- Assignment code, report and viva all will be considered for grading.
- An early start is recommended; any extension to complete the assignment will not be given.
- If any required information is missing, reasonable assumptions may be made. All such assumptions must be mentioned in the report to ensure transparency and consistency.
- Before you start coding, read Chapters 3, 6 and 7 of the xv6 book.

———————————————————————

Task 3.1: Implement Kernel Primitives for IPC (Moderate)

In this assignment, you will add fundamental Inter-Process Communication (IPC) mechanisms to the xv6 kernel: counting semaphores for synchronisation and a simple shared memory system for data exchange. These primitives will be used in Task 3.2.

Objective:
—————
The default xv6 kernel has limited IPC capabilities (pipes). In this task, you will implement two core primitives that allow for more complex process coordination: semaphores for managing concurrent access to resources, and shared memory for efficient data sharing between processes.

1. Add system calls for counting semaphores:
   - int sem_init(int sem_id, int value): Initialises a semaphore with a specific ID and a starting count. You can assume a fixed number of system-wide semaphores (e.g., 100).
   - int sem_down(int sem_id): Implements the P (wait) operation. Decrements the semaphore count and blocks the process if the count is insufficient.
   - int sem_up(int sem_id): Implements the V (signal) operation. Increments the semaphore count and wakes up a waiting process, if any.
   - Implement basic error handling for invalid IDs or values.

2. Add a system call for shared memory:
   - int shm_create(char *id) → allocates a shared page (4KB) if not already created.

- void shm_get(int key) : This call allows a process to access a shared memory page. All processes that call shm_get with the same key should receive a pointer to the same physical page of memory, mapped into their respective virtual address spaces.
- For simplicity, you can implement this for a single, system-wide shared page (i.e., the key is ignored, but should still be part of the function signature).
- Inside these syscalls:
  - Use kalloc() to allocate a physical page.
  - Use mappages() to map it into the user space.

3. Ensure correctness:
   - Protect the global semaphore array and shared memory structures with appropriate locking to prevent race conditions during initialization or access.

Deliverables:
—————
1. Ensure xv6 compiles, boots, and runs as expected with your changes.
2. Submit:
   - Modified system call implementations (sysproc.c , usys.S , etc.).
   - New kernel files for semaphore and shared memory logic (e.g., sem.c , shm.c).
   - Any necessary changes to kernel headers (proc.h , defs.h , etc.).
   - At least one test program to demonstrate that your semaphores (e.g., a producer-consumer test) and shared memory (e.g., two processes writing/reading to the same page) work correctly.

Hints:
—————
- For semaphores, review the sleep and wakeup mechanisms in Chapter 7. You will need a spinlock for each semaphore to protect its count.
- For shared memory, review how page tables are managed in Chapter 3. You will need to use kalloc() to get a physical page and mappages() to map it into a process's address space.
- Remember to add your test program(s) to UPROGS in the Makefile so it is compiled automatically.

Notes:
—————
- This is a moderate-difficulty assignment that requires careful modification of the xv6 kernel.
- Focus on getting the locking correct for both primitives. An error here will cause complex bugs in Task 3.2.
- Don't forget to recompile xv6 after making your changes:

```
make clean
make qemu
```

—————————————————————————

Task 3.2: The Indian Grand Prix (Advanced)

In this task, you will use the primitives created in Task 3.1 to build a concurrent, multiplayer racing game inside xv6.

Objective:
—————
To apply your understanding of shared memory, synchronisation, and deadlock avoidance by building a complete, concurrent application. The game, "The Indian Grand Prix" involves teams of two processes that must communicate and synchronize to navigate a shared maze, while competing with other teams for limited space.

Requirements:
—————
1. Design and implement a "Game Master" program:
- This program is responsible for setting up the entire game.
- It must use shm_create() to acquire the shared memory page.
- It initializes the maze structure within the shared memory. The maze is "intertwined": the memory location for Player A contains the address of the next location for Player B, and vice-versa.
- It uses sem_init() to create a counting semaphore for each location in the maze, initializing it to a fixed capacity (e.g., 4), to limit how many processes can be at one location simultaneously.
- It launches a specified number of teams (e.g., 10), with each team consisting of two player processes.

2. Design and implement the "Player" program:
- This program contains the logic for a single-player process.
- It must attach to the shared memory using shm_get().
- It navigates the maze step-by-step. In each step, it must:
  - Use sem_down() to enter a maze location.
  - Read the value at its current location to discover its teammate's next location.
  - Communicate this information to its teammate using an agreed-upon protocol within the shared memory (e.g., a "mailbox" section).
  - Use sem_up() to leave the location.
  - Wait for and receive its next location from its teammate via the mailbox.
- A team wins only when both of its players reach the final destination. The first team to do so is the winner.

3. Design for Deadlock Avoidance:
- Your player logic must be carefully designed to avoid deadlocks. A process should not hold a resource (a spot in a maze location) while waiting for another resource (a message from its teammate).
- Document your deadlock avoidance strategy in your report.

Deliverables:
—————
1. A brief design overview:
The structure of your maze and mailboxes in the shared memory page.
- Your protocol for inter-process communication between teammates.

- Your strategy for deadlock avoidance, explaining what potential deadlocks could occur and how your design prevents them.

2. The user-space programs are gamemaster and player.
3. Any modifications made to kernel files from Task 3.1.
4. Updated Makefile to include your new game programs in UPROGS.
5. A section in your report showing the game in action with screenshots and explaining the output.

Hints:
—————
- The simplest way to implement IPC is to reserve a small, fixed part of the shared memory page as a "mailbox" for each team.
- The most common deadlock risk is holding a semaphore while waiting for teammate communication. Releasing the semaphore before waiting is a key strategy to investigate.
- Your gamemaster should fork() and exec() the player processes, passing team and player IDs as command-line arguments.

Notes:
—————
- This is an advanced assignment that combines kernel modifications with complex application-level concurrent programming.
- There is no single correct solution for the IPC protocol or game logic. What matters is that your implementation is functional, well-justified, and correctly avoids deadlocks