OS Program Assignment 3 Design Document

Author : B092040010 呂彥鋒

1. Address the following questions:

a. Which files did you change?

- **`reader_writer.c`** : The main file that initializes shared memory and semaphores, launches the writer and reader processes.
- **`writer.c`** : The writer process that writes numbers into the shared mailbox.
- **`reader.c`** : The reader process that reads numbers from the shared mailbox.
- **`psem.c`** : Implemented semaphore system calls used by the above programs.
- **Minix system files**: Added entries to system tables and headers to support the new semaphore system calls.
  - Add entries to the PM server table in `/usr/src/servers/pm/table.c` .
  - Define system call numbers in `/usr/src/include/minix/callnr.h` .
  - Define prototypes in `/usr/src/servers/pm/proto.h` .
  - Implement the system calls in `psem.c` and add it to the PM server Makefile.

b. Which functions did you use?

- **`do_psem_init`** : Initialize a semaphore.
- **`do_psem_wait`** : Wait (decrement) on a semaphore.
- **`do_psem_signal`** : Signal (increment) a semaphore.
- **`do_psem_destroy`** : Destroy a semaphore.
- **`do_proc_no`** : Get the process number.
- **`do_psem_block`** : Block the process.
- **`do_psem_wakeup`** : Wake up a blocked process.

c. Why?

These functions were used to implement and manage semaphores for synchronizing access to shared memory between the writer and reader processes. Semaphores ensure that critical sections of code are executed without interference and manage the coordination between multiple processes.

2. Explain the flow of a message from one process to another.

Flow Diagram:

```
Writer Process                    Reader Process
--------------                    --------------
1. Get user input
2. Wait on writing semaphore
3. Write to mailbox (shared memory)
4. Signal writing semaphore
5. Repeat                            1. Wait on mutex semaphore
                                     2. Increment reader count
                                     3. If first reader,
                                         wait on writing semaphore
                                     4. Signal mutex semaphore
                                     5. Read from mailbox (shared memory)
                                     6. Wait on mutex semaphore
                                     7. Decrement reader count
                                     8. If last reader,
                                         signal writing semaphore
                                     9. Signal mutex semaphore
                                     10. Repeat
```

Explanation:

1. The writer process prompts the user for input and waits on the writing semaphore to ensure no readers are reading.
2. Once it acquires the writing semaphore, it writes the user input to the mailbox (shared memory) and then signals the writing semaphore.
3. The reader processes wait on a mutex semaphore to ensure exclusive access to the reader count variable.
4. The first reader waits on the writing semaphore, ensuring no writers write during the reading process.
5. After reading, the last reader signals the writing semaphore, allowing the writer to write again.

3. Explain which traps were used in your design. Why?

Traps Used:

- **`PSEM_INIT`** : Initialize semaphores for synchronization.

- **`PSEM_WAIT`** : Implement the wait § operation on a semaphore.
- **`PSEM_SIGNAL`** : Implement the signal (V) operation on a semaphore.
- **`PSEM_DESTROY`** : Destroy semaphores after use.

Why:

These traps provide the fundamental operations required to implement semaphore-based synchronization, ensuring proper coordination between reader and writer processes.

4. Was the mailbox unidirectional or bi-directional?

The mailbox was **unidirectional**. The writer writes messages to the mailbox, and the readers read messages from the mailbox.

5. What challenges did you face when implementing the memory mailbox?

Challenges:

- Ensuring synchronization between multiple readers and the writer to prevent data corruption.
- Properly initializing and managing shared memory and semaphores.
- Debugging and handling synchronization issues, such as deadlocks and race conditions.

6. How did you address memory mailbox synchronization? Describe in detail.

Synchronization was addressed using semaphores:

- **Mutex Semaphore** (`sem_m`): Ensures exclusive access to the reader count variable.
- **Writing Semaphore** (`sem_w`): Ensures that the writer can only write when no readers are reading.

Detailed Steps:

1. **Writer Process**:
    - Waits on `sem_w` before writing to ensure no readers are accessing the mailbox.
    - Signals `sem_w` after writing to allow readers to access the mailbox.
2. **Reader Process**:
    - Waits on `sem_m` to increment the reader count safely.
    - The first reader waits on `sem_w` to block the writer during reading.
    - Signals `sem_m` after incrementing the reader count.
    - Reads the message from the mailbox.
    - Waits on `sem_m` to decrement the reader count safely.
    - The last reader signals `sem_w` to allow the writer to write again.
    - Signals `sem_m` after decrementing the reader count.

7. How did you prevent the same message being read by a single process multiple times? Describe in detail.

By using the combination of `sem_m` and `sem_w`, the synchronization mechanism ensures that:

- Only one reader can increment or decrement the reader count at a time.
- The writer can only write when no readers are accessing the mailbox.
- Each reader reads the message exactly once by coordinating through the reader count and semaphores, preventing multiple reads of the same message.

README

Intro

First, you need to follow the instuction to rebuild the system
to add the new system call for semaphore.

Then, Test the program with `make`

Rebuild the MINIX system

1. `mv minix /usr/src/`

2. Compiling the Server
   Steps for compiling the servers:
   Go to directory /usr/src/servers/
   Issue `make`
   Issue `make install`

3. Compiling the Library
   Steps to compile the new library
   Go to the directory /usr/src/lib/
   ```
   make
   make install
   ```

4. Creating a New Boot-Image Using the Updated Servers and Library
   Rebuilding the Kernel Only
   ```
   cd /usr/src/releasetools
   make hdboot
   cp /usr/sbin/kernel /boot/minix_latest/kernel
   reboot
   ```

Test the program

```
cd PA3
```

Issue `make run`