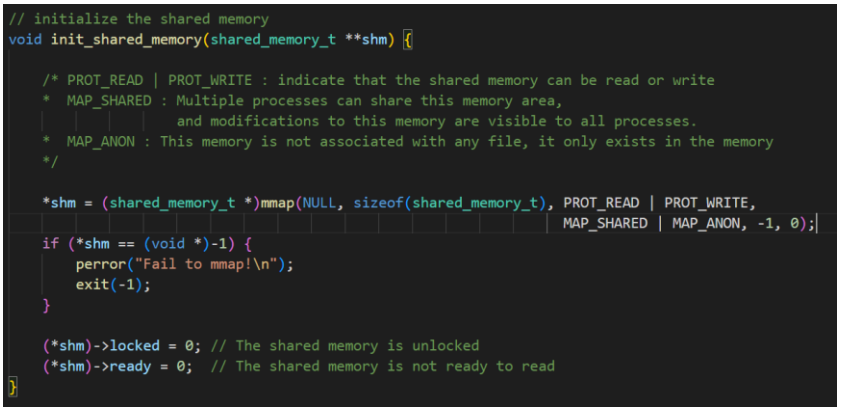
Overview

This assignment is to implement the IPC (inter-process communication). There are two ways to achieve the goat, one is through shared memory, another is through message passing.

In this assignment, I use shared memory to implement the IPC. In more detail, I use virtual memory to avoid go through the OS. The function ***mmap( )*** will generate shared memory by mapping pages read-write into virtual address space.

To reach the goals, I have decomposed the assignment into the following major parts:

- Use ***mmap( )*** to implement the shared memory :



```
// initialize the shared memory
void init_shared_memory(shared_memory_t **shm) {

    /* PROT_READ | PROT_WRITE : indicate that the shared memory can be read or write
    *  MAP_SHARED : Multiple processes can share this memory area,
                    and modifications to this memory are visible to all processes.
    *  MAP_ANON : This memory is not associated with any file, it only exists in the memory
    */

    *shm = (shared_memory_t *)mmap(NULL, sizeof(shared_memory_t), PROT_READ | PROT_WRITE,
                                   MAP_SHARED | MAP_ANON, -1, 0);
    if (*shm == (void *)-1) {
        perror("Fail to mmap!\n");
        exit(-1);
    }

    (*shm)->locked = 0; // The shared memory is unlocked
    (*shm)->ready = 0;  // The shared memory is not ready to read
}
```

    1. PROT_READ and PROT_WRITE indicate that the shared memory can be read and write.
    2. MAP_SHARED means that multiple processes can use this shared memory area.
    3. MAP_ANON means that this memory is not associated with any file, it only exists in the memory.
- Synchronization :
    1. There are N-readers and one writer. When the readers are reading, the writer can not write anything into the shared memory; meanwhile, when the writer are writing, the readers can not read the contents in the shared memory.

2. To achieve this, I define a struct called ***shared_memory_t*** to prevent the synchronization problem.

```
typedef struct {
    int locked;             // Mutex lock, 0 means unlocked, 1 means locked
    int ready;              // To check if the shared memory can be read or not, 0 means not ready
    char data[MMAP_MEM_SIZE]; // Shared memory data
}shared_memory_t;
```

- ***locked*** : It is a mutex lock, 0 means unlocked, 1 means locked.
- ***ready*** : To check if the shared memory can be read or not, 0 means the writer is writing something into the shared memory, 1 means the writer has done his work, and the readers can read the contents in the shared memory.
- ***data[MMAP_MEM_SIZE]*** : This stores the shared memory data.

Major data structures

I use ***Array*** to implement the shared memory.

Major algorithms

First the parent process will create two child processes. Then the parent process will write the user input into the shared memory. After parent process done its work, it will use pipe to inform the child processes to read the contents in the shared memory. When the readers has read the content, the program will tell the user to input another string.

```
// Create pipes for each reader process
for (int i = 0; i < NUM_READERS; ++i) {
    if (pipe(pipe_fds[i]) == -1) {
        perror("Pipe error!\n");
        exit(EXIT_FAILURE);
    }
}
```

```
// Write user input to shared memory
writer(shm);
```

```
// Create reader processes
for (int i = 0; i < NUM_READERS; ++i) {
    pid_t pid = fork();

    if (pid == 0) {
        // Close write end of the pipe in the child process
        close(pipe_fds[i][1]);
        // Wait for signal from parent process
        char buf;
        if (read(pipe_fds[i][0], &buf, 1) == -1) {
            perror("Read error!\n");
            exit(EXIT_FAILURE);
        }
        reader(i + 1, shm);
        // Close read end of the pipe in the child process
        close(pipe_fds[i][0]);
        exit(0);

    }
    else if (pid < 0) {
        perror("Fork error!\n");
        exit(-1);
    }
    else {
        pids[i] = pid;  // Store the PID of the reader process
    }
}
```

```
// Signal reader processes to read the shared memory
for (int i = 0; i < NUM_READERS; ++i) {
    if (write(pipe_fds[i][1], "r", 1) == -1) {
        perror("Write error!\n");
        exit(EXIT_FAILURE);
    }

    // Close write end of the pipe in the parent process
    close(pipe_fds[i][1]);
}

// Wait for all reader processes to finish
for (int i = 0; i < NUM_READERS; ++i) {
    waitpid(pids[i], NULL, 0);
}

// Close all pipe read ends in the parent process
for (int i = 0; i < NUM_READERS; ++i) {
    close(pipe_fds[i][0]);
}
```

In-depth analysis and implementation

Which files did I change?

- There are several files in my assignment. The file ***main.c*** is the simple user shell. It will create two child processes as readers, and use pipe to inform the child processes to read the contents in the shared memory. When the program is finished, it will destroy the shared memory.
- The file ***common.h*** defines the struct of the shared memory. Furthermore, it also defines the ***locked*** and ***ready***.
- ***shared_memory.c*** implements two functions. One is called ***init_shared_memory***, it will initialize the shared memory. Another is called ***destroy_shared_memory***, which will use ***munmap*** to destroy the shared memory.
- ***writer.c*** have two functions, one is called ***Getchar***, which will handle the user input string. Another is called ***writer***, it will write the user input string into the shared memory. When its work is done, it will print out that the parent process has written to shared memory, and set the value of ***ready*** to 1.
- The ***reader.c*** use ***putchar*** to print out the contents of the shared memory.

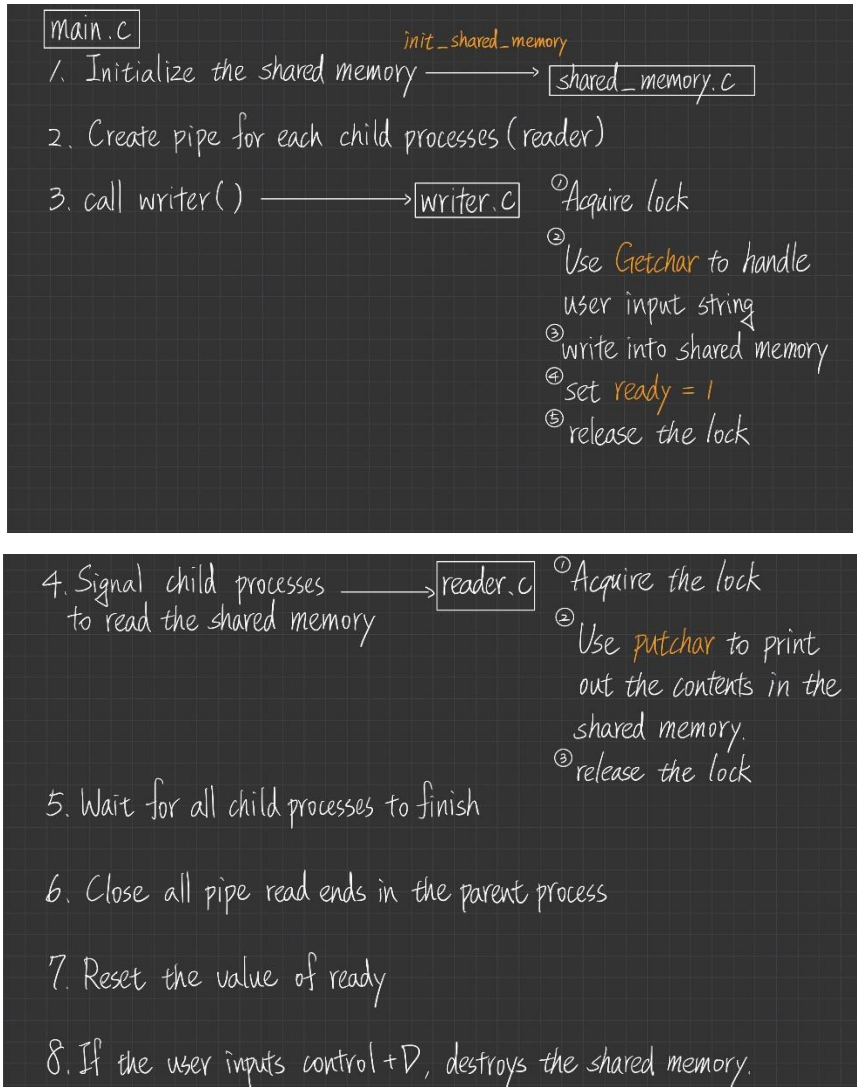
Which functions did I use?

- ***putchar*** to print out the contents of the shared memory.
- ***__sync_lock_test_and_set*** is to implement the low-level synchronization mechanisms. Its advantage is avoiding the context switch, its disadvantage is that it continuously takes up CPU time while waiting.
- ***mmap( ) is*** to create the shared memory, and use ***munmap( )*** to destroy the shared memory.
- ***__sync_lock_release*** is to release the mutex lock.

The flow of a message from one process to another

The parent process will write information into the shared memory, then use pipe to inform the child processes to read the contents in the shared memory. If user inputs control + D, the program will end, with destroying the shared memory.

The following chart is the detail of the flow of a message from one process to another.



Which traps were used in my design

1. ***mmap*** System Call

- Purpose: The ***mmap*** system call creates a new mapping in the virtual address space of the calling process. In my design, it is used to allocate

shared memory that can be accessed by multiple processes.

- Why Used: To allocate a shared memory region that is accessible by both the writer and reader processes without relying on physical files.

2. ***read*** System Call

- Purpose: The read system call reads data from a file descriptor into a buffer. In my design, it is used by the reader processes to read the signal from the pipe, indicating that new data is available in the shared memory.
- Why Used: To wait for a signal from the writer process, ensuring synchronization.

3. ***write*** System Call

- Purpose: It is used by the writer process to signal the reader processes via the pipe.
- Why Used: To notify reader processes that new data is available in the shared memory.

4. ***fork*** System Call

- Purpose: The fork system call creates a new process by duplicating the calling process. In my design, it is used to create multiple reader processes.
- Why Used: To create multiple reader processes that can read data from the shared memory concurrently.

5. ***waitpid*** System Call

- Purpose: The ***waitpid*** system call waits for a specific child process to terminate. In my assignment, it is used by the writer process to wait for all reader processes to finish reading.
- Why Used: To ensure that the writer process waits for all reader processes to finish before prompting the user for new input.

6. Atomic Operations

- Purpose: Atomic operations like ***__sync_lock_test_and_set*** and ***__sync_lock_release*** are used for synchronization. They ensure that memory updates are atomic and prevent race conditions.
- Why Used: To implement a spinlock mechanism for mutual exclusion, ensuring that only one process can access the shared memory at a time.

7. ***exit*** System Call

- Purpose: The ***exit*** system call terminates a process. In my assignment, it is

used to terminate the writer or reader processes gracefully.

- Why Used: To handle EOF or errors, ensuring that processes can exit cleanly.

Unidirectional or Bi-directional?

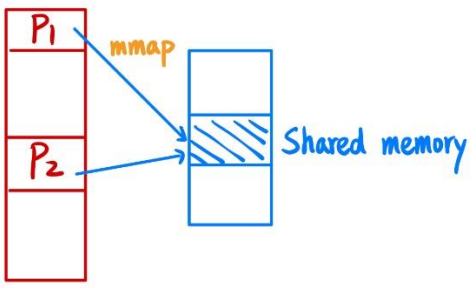
In my assignment, the mailbox is unidirectional. The parent process writes information into the shared memory, and the child processes read the contents in the shared memory.

The following figure is the demo.

```
# ./main
Process 386 launching as writer
Enter a string, control + D to exit: OSisVeryFun!
Parent process has written to shared memory: OSisVeryFun!
Process 1 named 387 launching as reader
Process 2 named 388 launching as reader
Process 387 received: OSisVeryFun!
Process 388 received: OSisVeryFun!
Process 387 has read the data from the shared memory!
Process 388 has read the data from the shared memory!
```

Challenges

- To implement IPC (inter-process communication), we should first know how to use ***mmap***. ***mmap*** is a function that will use virtual memory, and it can let two processes to communicate without going through the OS.



- Ensuring that multiple processes do not access the shared memory simultaneously, leading to data corruption or inconsistent states.

- Correctly handling the creation and termination of multiple reader processes, ensuring that all processes synchronize properly.
  Reader processes were created using fork, and their process IDs (PIDs) were stored in an array for later management. We used ***waitpid*** to wait for each reader process to finish reading and signal its completion. This ensured proper synchronization and orderly termination of processes.

address memory mailbox synchronization

There are 3 components in my assignment to achieve the synchronization, they are ***Spinlock*** (using atomic operations), ***Ready flag*** and ***Pipes for signaling***.

1. Spinlock Using Atomic Operations

The spinlock is implemented using atomic operations to ensure mutual exclusion. To ensure that only one process can access the critical section of the shared memory at a time. The atomic operation ***__sync_lock_test_and_set*** is used to acquire the lock, and ***__sync_lock_release*** is used to release the lock.

```
// Acquire lock (spinlock)
while (__sync_lock_test_and_set(&shm->locked, 1) != 0);
```

```
// Release lock
__sync_lock_release(&shm->locked);
```

- Acquire Lock: The ***__sync_lock_test_and_set*** function atomically sets the lock variable to 1 and returns its previous value. If the previous value was 1, it means the lock was already held by another process, and the current process will keep spinning (busy-waiting) until it can acquire the lock.
- Release Lock: The ***__sync_lock_release*** function sets the lock variable back to 0, making it available for other processes.

2. Ready Flag

The ready flag is used to indicate whether the data in the shared memory is ready to be read by the reader processes. It is used to prevent readers from reading stale or uninitialized data.

```
void reader(int id, shared_memory_t *shm)
    // Wait until data is ready
    while (!shm->ready);
```

```
// Signal that data is ready
shm->ready = 1;
```

- Writer Process: After writing the message to the shared memory, the writer sets the ready flag to 1, signaling that the data is ready to be read.
- Reader Process: The reader processes spin-wait (busy-wait) until the ready flag is set to 1, indicating that they can safely read the data from the shared memory.

3. Pipes for Signaling

Pipes are used to signal the reader processes when the writer process has written new data to the shared memory. It is efficiently to notify reader processes without busy-waiting.

```
// Signal reader processes to read the shared memory
for (int i = 0; i < NUM_READERS; ++i) {
    if (write(pipe_fds[i][1], "r", 1) == -1) {
        perror("Write error!\n");
        exit(EXIT_FAILURE);
    }

    // Close write end of the pipe in the parent process
    close(pipe_fds[i][1]);
}
```

```
if (pid == 0) {
    // Close write end of the pipe in the child process
    close(pipe_fds[i][1]);
    // Wait for signal from parent process
    char buf;
    if (read(pipe_fds[i][0], &buf, 1) == -1) {
        perror("Read error!\n");
        exit(EXIT_FAILURE);
    }
    reader(i + 1, shm);
    // Close read end of the pipe in the child process
    close(pipe_fds[i][0]);
    exit(0);

}
```

- Writer Process: After writing the message to the shared memory and setting

the ready flag, the writer sends a signal to each reader process via the pipes.

- Reader Process: The reader processes wait for the signal by reading from the read end of the pipe. When they receive the signal, they know that new data is available in the shared memory and proceed to read it.

prevent the same message being read multiple times

The reader process performs the following steps to ensure it reads the message only once:

1. Wait for the Signal: The reader waits for the signal from the writer using the pipe.
2. Wait for the Ready Flag: The reader waits until the ready flag is set to 1.
3. Acquire the Lock: The reader acquires the lock using ***__sync_lock_test_and_set.***
4. Read the Message: The reader reads the message from the shared memory.
5. Reset the Ready Flag: The reader resets the ready flag to 0 to indicate that the message has been read.
6. Release the Lock: The reader releases the lock using ***__sync_lock_release***.

There are also other ways to preventing multiple reads of the same message

- Pipes for Signaling: Each reader waits for a signal from the writer before attempting to read the message. This ensures that the readers do not repeatedly read the same message because they only proceed when they receive a new signal.
- Ready Flag: The ready flag is set by the writer after writing the message and reset by the readers after reading the message. This prevents the readers from reading the same message multiple times.
- Locking Mechanism: The spinlock ensures mutual exclusion, so only one reader can read and reset the ready flag at a time. This prevents race conditions where multiple readers might try to read the same message simultaneously.