北京邮电大学

计算机学院

进程管理实验

姓名: 吴清柳

学号: 2020211597

班级: 2020211323

指导老师: 杜晓峰

课程名称: 操作系统

December 16, 2022

Contents

1 实验题目

使用信号量机制，编写程序，模拟多任务下的读者优先和写者优先，能够同时稳定运行多个任务 20 分钟以上。

编写程序模拟了多个进程同时想要访问存储在共享内存中的数据的情况下, 使用读者有限和写者优先管理信号量对其进行调度的过程. 下面介绍本实验中对读者优先和写者优先的定义.

1.1 读者优先和写者优先的共同定义

此处介绍读者优先和写者优先情况下设定的共同部分.
同时只能有一个写者正在访问同一块共享内存; 同时可以有多个读者正在访问同一块共享内存.

当一个或多个读者正在访问同一块共享内存的时候, 如果队列里新增等待访问同一块共享内存的读者, 则他被允许访问, 如果队列里新增等待访问同一块共享内存的写者, 则他被阻挡在外, 直到所有读者离开.

当一个写者正在访问同一块共享内存的时候, 后面的读者或者写者都等待. 如果先来的是写者, 则写者是下一个访问共享内存的进程; 如果先来的是读者, 则读者是下一个访问共享内存的进程, 之后写者的等待和上一段所说相同.

1.2 读者优先

在读者优先的情况下, 当读者正在访问同一块共享内存的时候, 写者想要访问这一块共享内存, 当且仅当访问同一块共享内存的读者都访问完毕, 且没有其他读者在等待访问这块共享内存; 在其他情况下都是读者优先访问同一块共享内存.

当写者正在访问同一块共享内存的时候, 如果等待队列中有读者, 则下一个访问的进程将是读者, 且直到等待队列中不再有读者的时候, 等待队列里的写者才能访问同一块共享内存. 因此如果不断有读者前来访问同一块共享内存, 会导致写者饥饿.

1.3 写者优先

在写者优先的情况下, 如果一个或多个读者正在访问同一块共享内存, 新来的读者和写者在关键区外排队, 则在第一个排队的写者之前的读者能正常进入关键区来共同访问共享内存. 当第一个排队的写者前面的读者都完成对共享内存的访问后, 写者将阻挡读者在关键区之外, 直到队列中所有的写者依次完成写操作后才会让队列中的读者进入关键区. 因此, 如果队列中不断有写者进入, 则会导致读者的饥饿.

1.4 避免饥饿的限制

为了避免读者和写者的饥饿导致实验在运行一段时间后变成只读或者只写, 我在实验程序中以宏的形式指定了每个读者或写者最多访问一块共享内存的次数. 当达到最大访问次数后, 进程将在日志中记录自己的访问等待时间等信息, 并销毁自己. 日志记录在 logs/file.log 和 logs/stat.log 中. 其中, logs/file.log 详细记录了各进程的读者等待时间和写者等待时间.

2 实验过程

2.1 实现介绍

2.1.1 概述

使用 c 语言标准库的 sys/shm.h 库创建共享内存, sys/sym.h 创建和管理信号量, 使用 fork() 创建多个子进程并分化位读者和写者.

程序使用调用程序的时候传入的命令行参数来设置共享内存大小, 读者和写者的总数, 以及读者和写者的占比.

第一个参数为 peers, 表示读者和写者进程的总数. 一般不超过 3000, 否则操作系统将没有多余的进程资源可以分配给其他进程请求, 影响操作系统正常运行.

第二个参数为 rw_ratio, 表示读者和写者的占比, 范围为 $0, 1, \ldots, 100$, 含义为每个进程有 rw_ratio 的概率变成读者.

第三个参数为 shm_size, 表示共享内存的大小, 即有多少块独立的共享内存可供访问.

程序代码结构如下.

```
makefile
run.sh
src
main_reader_priority.c
main_writer_priority.c
semaphores.c
semaphores.h
shared_memory.c
shared_memory.h
```

源代码在 src 文件夹下, 其中:

- main_reader_priority.c 中是读者优先策略的主程序,
- main_writer_priority.c 中是写者优先策略的主程序.
- semaphores.[ch] 中封装了对信号量的操作, 包括对信号量的初始化和修改操作;
- shared_memory.[ch] 中封装了对共享内存的操作, 包括对共享内存的初始化和删除回收操作;

2.1.2 共享内存

共享内存定义在 shared_memory.[ch] 中, 定义了如下操作:

```
/**
* @brief Initialize shared memory.
*
* @param key
* @param size
* @return int
*/
int shm_init(key_t key, int size);

/**
* @brief Return a pointer to memory.
*
* @param shm_id
* @return shm_data*
*/
ShmData *shm_at(int shm_id);

/**
* @brief Detach memory
*
* @param shm_ptr
* @return int
*/
int shm_detach(void *shm_ptr);

/**
* @brief Deallocate memory
*
* @param shm_id
* @return int
*/
int shm_delete(int shm_id);
```

2.1.2.1 共享数据 实验中将多个共享数据结构体存储在共享内存中, 通过对多个共享数据结构体的并行访问和控制模拟真实情况. 结构体定义如下.

```
typedef struct shm_data {
int reads;
int writes;
} ShmData;
```

其中 reads 记录了写者的访问次数, writes 记录了读者的访问次数.
为了模拟更加真实, 模拟多个共享数据如下.

```
// Initialize the shared matrix.
// Allocate shared memory for ShmData.
shm_id = shm_init(597, sizeof(ShmData) * shm_size);
shmData = shm_at(shm_id);
for (int i = 0; i < shm_size; i++) {
        shmData[i].writes = 0;
        shmData[i].reads = 0;
    }
```

其中 597 是 shared_memory key, 唯一标识一块共享内存. 在此处取学号后 3 位.

2.1.2.2 读者/写者计数器 读者/写者计数器需要被多个读者或多个写者访问, 因此也存放在共享内存中, 使用信号量控制访问. 信号量的介绍在下一节中. 读者/写者计数器初始化如下.

```
// Initialize shared counter for readers
readers_counter_id = shm_init((key_t)211, shm_size * sizeof(int));
readers_counter_data = (int *)shm_at(readers_counter_id);
for (size_t i = 0; i < (size_t)shm_size; i++) {
readers_counter_data[i] = 0;
}
```

Listing 1: 读者计数器

初始化了大小为 shm_size * sizeof(int) 的空间, shm_size 位共享数据所在共享内存的结构体个数, 对每个结构体单独计数正在访问的读者数量.

为了实现写者优先, 本实验也用到了写者计数器, 类似读者计数器.

```
// Initialize shared counter for writers
writers_counter_id = shm_init((key_t)212, shm_size * sizeof(int));
writers_counter_data = (int *)shm_at(writers_counter_id);
for (size_t i = 0; i < (size_t)shm_size; i++) {
writers_counter_data[i] = 0;
}
```

Listing 2: 写者计数器

初始化了大小为 shm_size * sizeof(int) 的空间, shm_size 位共享数据所在共享内存的结构体个数, 对每个结构体单独计数正在访问的写者数量.

2.1.2.3 统计信息共享内存 为了统计所有读者/写者进程在进行预先设定好的轮数的共享内存访问时, 作为读者/写者的等待时间总和, 设计了统计信息共享内存. 其初始化过程如下.

```
/** Initialize shared memory to keep stats
* 0: Writer waiting time;
* 1: Reader waiting time;
* 2: Total waiting time;
*/
shm_stat_id = shm_init((key_t)2022, 3 * sizeof(long));
stat_data = (long *)shm_at(shm_stat_id);
for (int i = 0; i < 3; i++) {
stat_data[i] = 0;
}
```

其中 2022 为这块共享内存的 key. 每块共享内存存有 3 个 long 变量, 分别统计所有读者/写者进程的总计等待时间, 读者等待时间和写者等待时间.

注意由于程序中使用了 usleep() 来延长读者/写者占用共享内存的时间, 此时操作系统不会分配给进程 CPU clock, 因此使用 clock() 无法获取准确的休眠时间, 因此程序中使用 gettimeofday() 的差值来计算经过的时间. 这个函数的返回值和时区相关, 由于只在东八区 (+8HRS) 测试过, 所以不确定其他时区下表现是否有异常. 因此建议在东八区使用该程序.

2.1.3 信号量

共有三组信号量, 封装信号量操作如下.

```
/**
* @brief Create and initialize semaphore.
*
* @param key
* @param sem_num
* @return int
*/
int sem_init(key_t key, int sem_num);

/**
* @brief Delete semaphore.
*
* @param sem_id
* @param sem_num
* @return int
*/
int sem_del(int sem_id);

/**
* @brief Semaphore up.
*
* @param sem_id
* @param sem_num
* @return int
*/
int sem_up(int sem_id, int sem_num);

/**
* @brief Sem down.
*
* @param sem_id
* @param sem_num
* @return int
*/
int sem_down(int sem_id, int sem_num);
```

下面对三组信号量进行介绍.

2.1.3.1 第一组信号量 第一组组信号量对共享内存中每个结构体单元的访问进行控制. 具体信号量初始化如下.

```
// Initialize semaphores.
read_mutex = sem_init((key_t)10001, shm_size);
write_mutex = sem_init((key_t)20001, shm_size);
reader_counter_mutex = sem_init((key_t)30001, shm_size);
writer_counter_mutex = sem_init((key_t)30002, shm_size);
queue_mutex = sem_init((key_t)50001, shm_size);
print_mutex = sem_init((key_t)40001, shm_size);
```

其中,

- read_mutex 是读信号量, 用于控制读者进入关键区;

- write_mutex 是写信号量, 用于控制读者进入关键区;
- reader_counter_mutex 是读者计数器信号量, 用于控制对读者计数器的访问权限;
- writer_counter_mutex 是写者计数器信号量, 只在写者优先的主程序代码中出现, 用于控制对写者计数器的访问权限;
- queue_mutex 是队列互斥锁信号量, 只在写者优先的主程序代码中出现, 用于控制是否允许读者进入等待访问共享内存的关键区的队列.
- print_mutex 用于控制不同进程写 log 到文件, 保证同一时间只有一个进程正在写 log 到文件.

2.1.4 读者/写者分配

第一个参数为 peers, 表示读者和写者进程的总数. 一般不超过 3000, 否则操作系统将没有多余的进程资源可以分配给其他进程请求, 影响操作系统正常运行.

第二个参数为 rw_ratio, 表示读者和写者的占比, 范围为 $0, 1, \ldots, 100$, 含义为每个进程有 rw_ratio 的概率变成读者.

2.1.5 读者优先策略

2.1.5.1 写者进程 成为写者的进程的操作伪代码如下. 对伪代码的解释在代码下方.

```
// child become writer
entry = get random entry in shared memory.
es++;

/**
* CRITICAL SECTION OF WRITER
*/
down(write_mutex, entry);

leep to occupy symaphore.
ep(sleeptime);

ata[entry].writes++;

up(write_mutex, entry);

IT CRITICAL SECTION OF WRITER


pdate statistic data.
l_time_writer += curr_time;
l_time += curr_time;
```

首先, 随机选择共享内存中一个共享数据结构体 entry 准备访问. 全局写次数 writes+1. 之后 wait(write_mutex, entry) 等待访问 entry 的读信号量, 并记录等待时间. 获取读信号量后, 进入关键区, 使用 usleep() 来占用关键区一段时间模拟真实

使用, 然后让该块共享内存写次数 +1. 然后退出共享内存, 更新该进程总写者时间和总时间.

2.1.5.2 读者进程 成为读者的进程的操作伪代码如下. 对伪代码的解释在代码下方.

```
entry = Access a random shared data struct in shared memory.
reads++;

// Update counter.
sem_down(counter_mutex, entry);
readers_counter_data[entry]++;
if (readers_counter_data[entry] == 1) {
        // Prevent writer from entering critical section.
        sem_down(write_mutex, entry);
    }
sem_up(counter_mutex, entry);

/**
* CRITICAL SECTION OF READER
*/
sem_down(read_mutex, entry);

    // Sleep to occupy symaphore.
usleep(sleeptime);
shmData[entry].reads++;

sem_up(read_mutex, entry);
/**
* EXIT CRITICAL ZONE OF READER.
*/

// Update counter.
sem_down(counter_mutex, entry);
readers_counter_data[entry]--;
if (readers_counter_data[entry] == 0) {
        // Release writer lock.
        sem_up(write_mutex, entry);
    }
sem_up(counter_mutex, entry);

// Update timer.
total_time_reader += curr_time;
total_time += curr_time;
```

和写者进程一样, 首先随机选取一块待访问的共享内存, 然后增加进程阅读次数. 等待读者计数器信号量后增加该块共享内存读者计数器. 如果读者计数器值为 1, 则等待写者信号量, 以阻止写者进入关键区. 然后释放计数器信号量, 进入关键区. 等待读信号量, 然后使用 usleep() 函数占用该块共享内存一段时间后, 增加该块共享内存的读次数, 释放读信号量. 等待读者计数器信号量, 减少该块共享内存读者计数器值. 如果读者计数器值为 0, 释放写者信号量, 允许写者进入共享内存. 然后释放该块共享内存读者计数器信号量. 更新总读者等待时间和总等待时间.

2.1.6 写者优先策略

2.1.6.1 写者进程 与读者优先策略相比, 写者进程增加了当有写者正在访问的时候阻止读者进入关键区队列的操作. 具体伪代码和解释如下.

```
// child become writer
entry = Access a random shared data struct in shared memory.
writes++;

// Update counter.
sem_down(writer_counter_mutex, entry);
writers_counter_data[entry]++;
if (writers_counter_data[entry] == 1) {
        // Prevent the following readers from entering queue.
        sem_down(queue_mutex, entry);
    }
sem_up(writer_counter_mutex, entry);

/**
* CRITICAL SECTION OF WRITER
*/
sem_down(write_mutex, entry);

// Sleep to occupy critical section.
    usleep(sleeptime);

    shmData[entry].writes++;

sem_up(write_mutex, entry);
    /**
    * EXIT CRITICAL SECTION OF WRITER
    */
// Update counter.
sem_down(writer_counter_mutex, entry);
writers_counter_data[entry]--;
if (writers_counter_data[entry] == 0) {
        // No writers waiting. Allow the following readers to
        // enqueue.
        sem_up(queue_mutex, entry);
    }
sem_up(writer_counter_mutex, entry);

    // Update statistic data.
    total_time_writer += curr_time;
    total_time += curr_time;
```

和读者优先策略中的写者一样, 首先随机选择一块共享内存中的结构体, 并增加进程总写次数; 之后等待写者计数器信号量, 增加该共享内存块的写者计数器. 如果该共享内存块的写者计数器值为 1, 等待该共享内存块的队列信号量 queue_mutex, 以阻止后面的读者进入队列. 然后释放该共享内存块的写者计数器信号量. 之后等待写者信号量 write_mutex, 进入关键区, usleep() 等待一段随机时间 sleeptime 后增加该块共享内存的写次数, 释放写信号量, 退出关键区. 等待写者计数器信号量 writers_counter_mutex, 减少该共享内存块写者计数器计数. 如果写者计数器计数为 0, 释放队列入口信号量 queue_mutex, 允许后面的读者进入队列. 然后释放该共享内存块的写者计数器信号量.

2.1.6.2 读者进程 相比读者优先策略下的读者, 写者优先策略下的读者增加了等待进入队列的操作, 以保证写者能够优先访问关键区共享内存. 具体伪代码和解释如下.

```
// child become reader.
    entry = Access a random shared value in shared memory.
    reads++;

    // Wait for queue mutex.
    sem_down(queue_mutex, entry);

    // Update counter.
    sem_down(reader_counter_mutex, entry);
    readers_counter_data[entry]++;
    if (readers_counter_data[entry] == 1) {
            // Prevent writer from entering critical section.
            sem_down(write_mutex, entry);
        }

    // Release queue mutex.
    sem_up(queue_mutex, entry);
    // Release reader counter mutex.
    sem_up(reader_counter_mutex, entry);

    /**
    * CRITICAL SECTION OF READER
    */
    // Wait for read mutex.
    sem_down(read_mutex, entry);

    // Sleep to occupy symaphore.
    usleep(sleeptime);

    shmData[entry].reads++;

    sem_up(read_mutex, entry);
    /**
    * EXIT CRITICAL ZONE OF READER.
    */

    // Update counter.
    sem_down(reader_counter_mutex, entry);
    readers_counter_data[entry]--;
    if (readers_counter_data[entry] == 0) {
            // Release writer lock.
            sem_up(write_mutex, entry);
        }
    sem_up(reader_counter_mutex, entry);

    // Update timer.
    total_time_reader += curr_time;
    total_time += curr_time;
```

首先随机选择一块共享内存中的待访问结构体, 等待本块共享内存队列信号量 queue_mutex. 然后等待读者计数器信号量 reader_counter_mutex, 将该块共享内存的读者计数器数值增加. 如果读者计数器的值为 1, 等待信号量 writer_mutex, 阻止读者进入本块内存. 然后释放队列信号量 queue_mutex 和读者计数信号

量 reader_counter_mutex. 之后等待读信号量 read_mutex, 进入关键区, usleep() 等待一段随机时间 sleeptime 后增加本块共享内存的读次数, 然后释放读信号量 read_mutex. 之后等待读者计数器信号量, 减少本块共享内存的读者计数器数量, 释放读者计数器信号量, 如果读者计数器数值为 0, 释放写信号量 write_mutex, 允许后面的读者访问共享内存. 然后释放读者计数器信号量, 更新总读者等待时间和总等待时间.

由于只要队列中有写者到来, 队列入口信号量 queue_mutex 就会被占用, 导致之后来的读者不能进入队列, 因此队列中的写者可以优先访问共享内存, 也就实现了写者优先策略.

3 实验结果

3.1 运行环境

程序在以下环境下构建和测试.

操作系统 macOS 13.1 22C5044e arm64;

c 编译器 gcc (Homebrew GCC 12.2.0) 12.2.0;

构建和测试工具 GNU Make 3.81;

文档编译工具 XeTeX 3.141592653-2.6-0.999994 (TeX Live 2022)

文本编辑器 nvim v0.8.1

调试工具 Visual Studio Code Version: 1.74.0 (Universal)

3.2 程序运行方式

程序使用 make 进行构建. 读者优先程序和写者优先程序都会被构建在 build/目录下.

对于程序的构建, 执行如下命令.

```
make
```

对于程序的执行, 可以使用如下命令:

```
Usage: build/[reader_priority | writer_priority] $(PEERS) $(RW_RATIO)
    $(SHM_SIZE)
```

其中, $(PEERS) 是将要创建的读者和写者进程总数, 建议不超过 3000, 否则会导致操作系统没有多余进程资源而导致操作系统不稳定. $(RW_RATIO) 是读者和写者比例, 范围是 0 到 100. $(SHM_SIZE) 是共享内存大小, 即创建多少个独立的共享内存块. 在本实验程序的实现中, 对每个独立的共享内存块的访问进行独立控制.

3.3 程序日志格式

程序的日志有三种等级.

一级是最详细的日志, 包括进出关键区, 进程创建和分化为读者写者, 等待信号量耗时, 休眠时间和进程退出等信息, 随程序运行输出在标准输出中, 即 console 中. 截取部分示例如下.

```
$ ./build/reader_priority 10 90 10

    [2022-12-16T15:48:41] Child with pid: 82474
    [2022-12-16T15:48:41] Child[82474](reader) accessing shared_memory[0].
[2022-12-16T15:48:41] Child[82474](reader) waited for 0.000008 seconds.
    [2022-12-16T15:48:41] Sleep for 0.916291 seconds.
    [2022-12-16T15:48:41] Child with pid: 82477.
    [2022-12-16T15:48:41] Child with pid: 82475.
    [2022-12-16T15:48:41] Child[82475](reader) accessing shared_memory[9].
[2022-12-16T15:48:41] Child[82477](reader) accessing shared_memory[7].
[2022-12-16T15:48:41] Child[82475](reader) waited for 0.000018 seconds.
    [2022-12-16T15:48:41] Child[82477](reader) waited for 0.000028 seconds.
    [2022-12-16T15:48:41] Sleep for 1.609438 seconds.
    [2022-12-16T15:48:41] Sleep for -0.000000 seconds.
    [2022-12-16T15:48:41] Child[82475](reader) exited shared_memory[9].
...
```

二级是总结性日志, 在每个进程结束的时候写到 logs/file.log 文件中, 包括每个进程的读者和写者等待时间总和, 以及读者和写者平均等待时间. 在日志的结尾还有对所有进程的读者和写者数量, 以及运行次数判断统计. 此外, 每次输出日志的开头还会记录当前时间, 据此可以判断程序运行时间. 由于通过宏定义的形式设定了每个进程运行 LOOPS 次后就会退出, 因此要展示稳定运行 20 分钟以上的效果, 可以通过增加创建的子进程数量 $(PEERS) 和减少能够访问的共享内存块数量 $(SHM_SIZE) 来实现. 截取部分日志示例如下.

```
[2022-12-16T15:48:55] Child[82484]:
Total write time: 0.000005 seconds; average write time: 0.000005 seconds.
Total read time: 8.377258; average read time: 0.930806.
Total time: 8.377263, total average time: 0.837726.

    [2022-12-16T15:48:57] Child[82474]:
Total write time: 3.087316 seconds; average write time: 3.087316 seconds.
Total read time: 7.399225; average read time: 0.822136.
Total time: 10.486541, total average time: 1.048654.

    [2022-12-16T15:48:57] Child[82492]:
Total write time: 0.000000 seconds; average write time: 0.000000 seconds.
Total read time: 7.418502; average read time: 0.741850.
Total time: 7.418502, total average time: 0.741850.


VALIDATING RESULTS
shared[0]: readers[9], writers[0]
shared[1]: readers[11], writers[1]
shared[2]: readers[5], writers[1]
shared[3]: readers[7], writers[1]
shared[4]: readers[9], writers[0]
shared[5]: readers[8], writers[1]
shared[6]: readers[11], writers[1]
```

```
shared[7]: readers[15], writers[1]
shared[8]: readers[10], writers[0]
shared[9]: readers[8], writers[1]
--------------------
total: readers[93], writers[7]
Total should be equal to (peers * LOOPS)[100]
Result is right!
```

第三级日志很简略, 每次程序运行会追加一行到日志文件 logs/stat.log 的结尾, 分别为总等待时间, 读者等待时间和写者等待时间. 用于多轮程序运行进行统计.

3.4 运行结果

分别采用读者优先策略和写者优先策略, 创建 2000 个读者写者, 读者比例 50% 或 90%, 共享内存块数量为 10 的情况下运行, 具体运行命令如下.

```
# Build
make

# 读者优先策略，2000个读者写者，50%读者，10个共享内存块.
build/reader_priority 2000 50 10

# 读者优先策略，2000个读者写者，90%读者，10个共享内存块.
build/reader_priority 2000 90 10

# 写者优先策略，2000个读者写者，50%读者，10个共享内存块.
build/writer_priority 2000 50 10

# 写者优先策略，2000个读者写者，90%读者，10个共享内存块.
build/writer_priority 2000 90 10
```

得运行日志和读者/写者等待时间如下.

- 读者优先策略, 读者比例 50% 的运行日志为 logs/50_file_reader_priority.log, 等待时间统计如图 1.
- 读者优先策略, 读者比例 90% 的运行日志为 logs/90_file_reader_priority.log, 等待时间统计如图 2.
- 写者优先策略, 读者比例 50% 的运行日志为 logs/50_file_reader_priority.log, 等待时间统计如图 3.
- 写者优先策略, 读者比例 90% 的运行日志为 logs/90_file_reader_priority.log, 等待时间统计如图 4.

可以看到, 在均为 50% 读者情况下, 读者优先策略和写者优先策略的表现是对称的, 即优先的一方的等待时间占总等待时间的越 25%. 而均为 90% 读者的情况下, 差异则比较大. 读者优先策略下, 读者和写者的等待时间大约对半开, 而写者优先策略下, 几乎总是读者在等待, 而写者的等待时间相比之下很小. 可以预见, 如果本实验程序中没有设置当进程访问 LOOPS 次共享内存后结束自己的话, 读者将几乎不能访问共享内存.

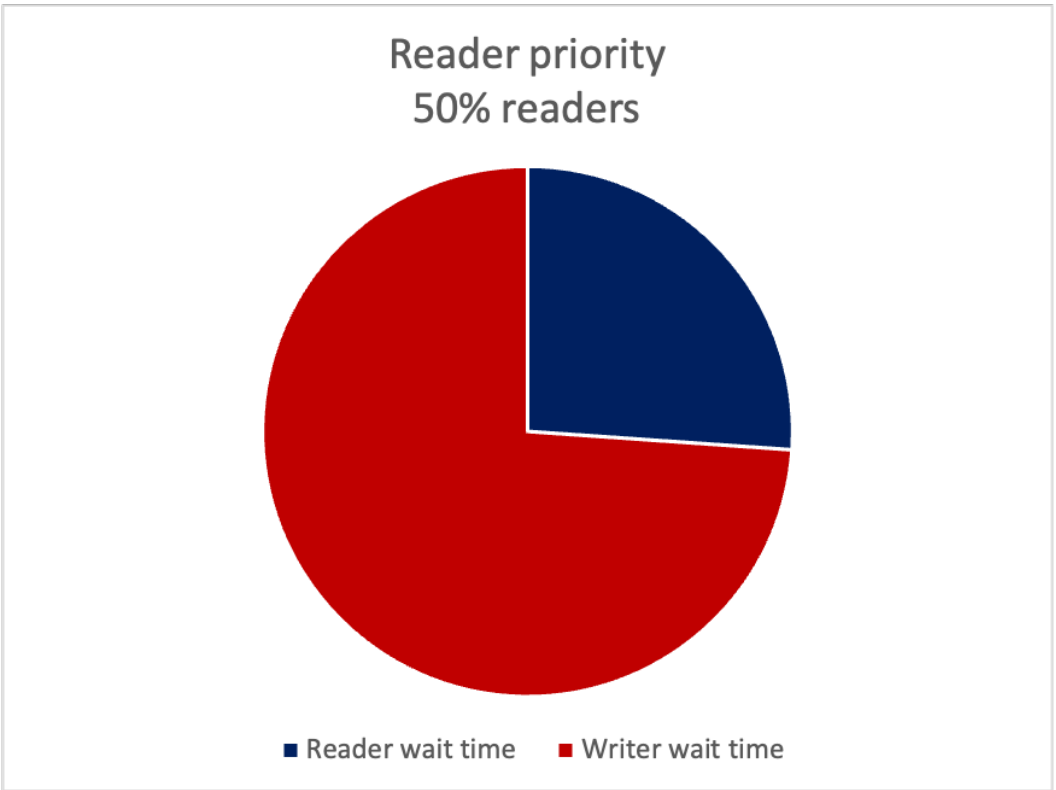

Figure 1: 读者优先策略,50% 读者

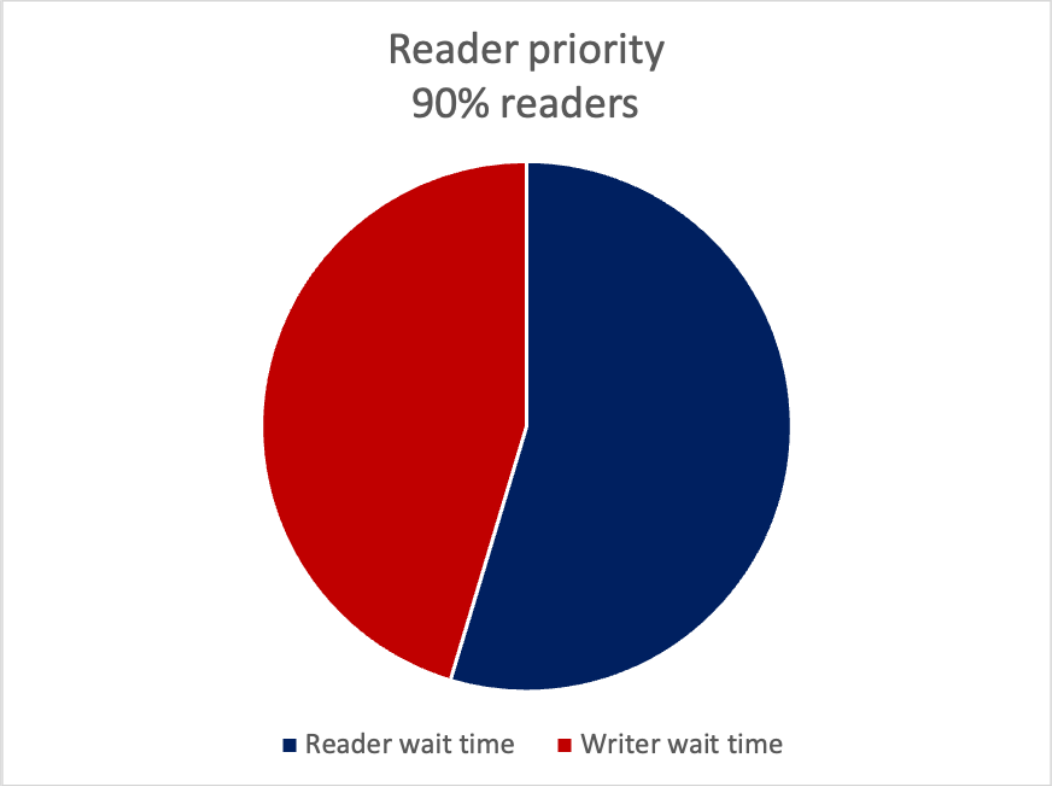

Figure 2: 读者优先策略,90% 读者

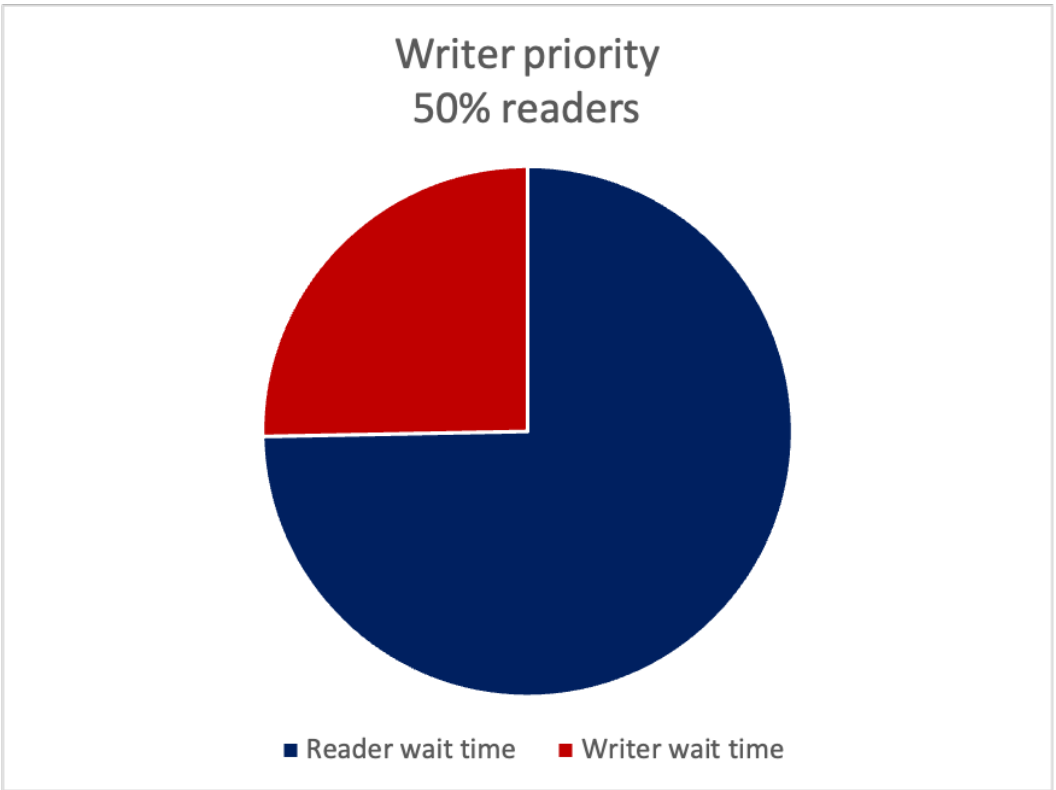

Figure 3: 写者优先策略,50% 读者

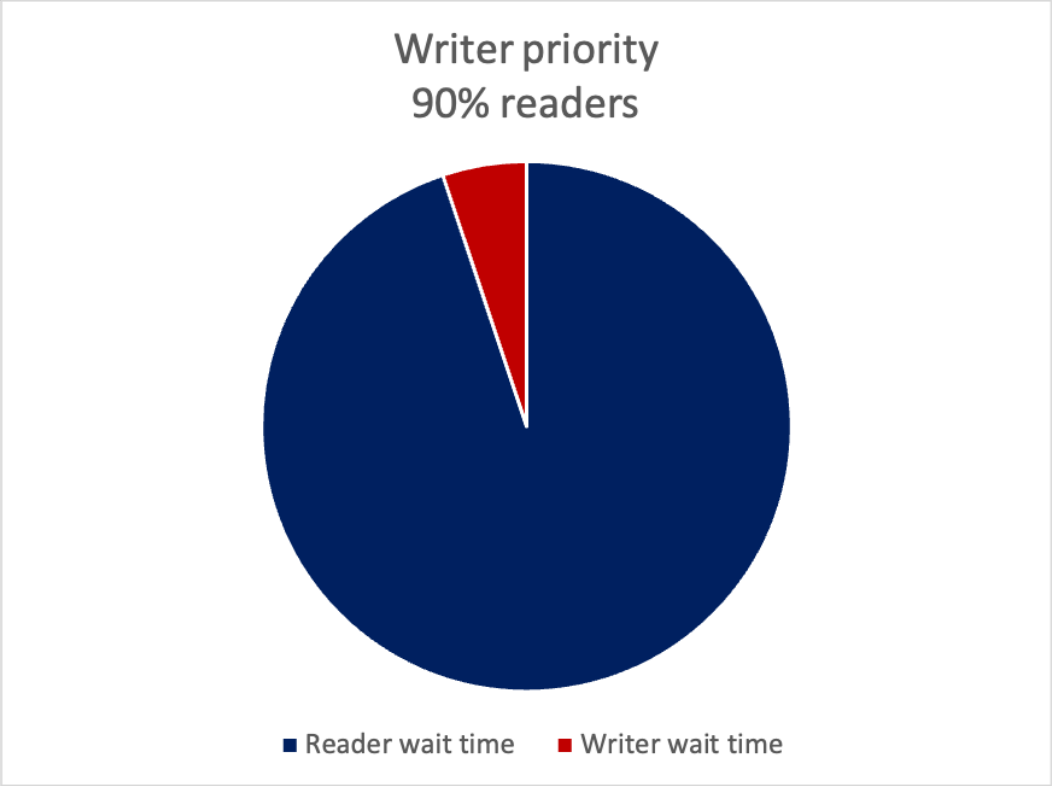

Figure 4: 写者优先策略,50% 读者

4 实验总结

通过这次实验, 我亲手实现并测试了读者优先进程调度和写者优先进程调度, 对信号量机制来控制并行进程对关键区的访问有了更深刻的认识.

在构建和管理项目的工具方面, 使用了 make 工具, 方便的实现了对程序的构建和测试; 此外, 对于文档, 也使用了 make 工具作为管理, 很大程度上方便了实验报告修改后的再次编译. 在实验报告编写方面, 练习了 latex 的使用, 对 latex 语法和使用更加熟悉.

但是, 由于时间限制, 没有来得及进行更多更详细的测试, 例如, 从 10% 读者到 90% 读者每个进行 2000 个进程运行的读者优先和写者优先策略测试等. 但是即便如此, 也已经有了丰富的收获.

总的来说, 这次实验在信号量机制进程调度之内及之外都学到了许多知识, 收获颇丰.

5 算法源代码

5.1 信号量

```c
#ifndef SEMAPHORES_H
#define SEMAPHORES_H

#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>

/**
 * @brief Create and initialize semaphore.
 *
 * @param key
 * @param sem_num
 * @return int
 */
int sem_init(key_t key, int sem_num);

/**
 * @brief Delete semaphore.
 *
 * @param sem_id
 * @param sem_num
 * @return int
 */
int sem_del(int sem_id);

/**
 * @brief Semaphore up.
 *
 * @param sem_id
 * @param sem_num
 * @return int
 */
int sem_up(int sem_id, int sem_num);
```

```
/**
 * @brief Sem down.
 *
 * @param sem_id
 * @param sem_num
 * @return int
 */
int sem_down(int sem_id, int sem_num);

#endif
```

Listing 3: semaphores.h

```
#include "semaphores.h"

int sem_init(key_t key, int sem_num) {
    int sem_id;
    // The semget() system call returns the System V semaphore set
    // identifier associated with the argument key. It may be used
    // either to obtain the identifier of a previously created semaphore
    // set (when semflg is zero and key does not have the value
    // IPC_PRIVATE), or to create a new set.
    sem_id = semget(key, sem_num, 0666 | IPC_CREAT);

    union semun sem_union;
    sem_union.val = 1;

    for (size_t i = 0; i < (size_t)sem_num; i++) {
        if (semctl(sem_id, i, SETVAL, sem_union) == -1) {
            printf("Error: semaphore init failed %d\n", sem_num);
            return 0;
        }
    }

    return sem_id;
}

int sem_del(int sem_id) {
    union semun sem_union;
    return semctl(sem_id, 0, IPC_RMID, sem_union);
}

/**
 * @brief Increase semval of semaphore sem_id by 1.
 *
 * @param sem_id
 * @param sem_num
 * @return int
 */
int sem_up(int sem_id, int sem_num) {
    struct sembuf sem_b;
    sem_b.sem_num = sem_num;
    sem_b.sem_op = 1; // V()
    sem_b.sem_flg = 0;

    if (semop(sem_id, &sem_b, 1) == -1) {
        printf("Error: semaphore up failed %d", sem_num);
        return 0;
```

```
    }

    return 1;
}

/**
 * @brief Decrease semaphore sem_id by abs(sem_op) if semval of sem_id is
 * greater or equal to abs(sem_op), else increase semcnt of sem_id and the
 * calling process will be suspended until semval of sem_id is greater or
    equal
 * to abs(sem_op), then decrease semcnt of sem_id and sustract abs(sem_op)
    from
 * semval of sem_id. Here sem_op is -1.
 *
 * @param sem_id
 * @param sem_num
 * @return int
 */
int sem_down(int sem_id, int sem_num) {
    struct sembuf sem_b;
    sem_b.sem_num = sem_num;
    sem_b.sem_op = -1; // V()
    sem_b.sem_flg = 0;

    if (semop(sem_id, &sem_b, 1) == -1) {
        printf("Error: semaphore down failed %d", sem_num);
        return 0;
    }

    return 1;
}
```

Listing 4: semaphores.c

5.2 共享内存

```
#include <sys/shm.h>

#ifndef SHARED_MEMORY_H
#define SHARED_MEMORY_H

typedef struct shm_data {
    int reads;
    int writes;
    long double avg_time;
} ShmData;

/**
 * @brief Initialize shared memory.
 *
 * @param key
 * @param size
 * @return int
 */
int shm_init(key_t key, int size);
```

```
/**
 * @brief Return a pointer to memory.
 *
 * @param shm_id
 * @return shm_data*
 */
ShmData *shm_at(int shm_id);

/**
 * @brief Detach memory
 *
 * @param shm_ptr
 * @return int
 */
int shm_detach(void *shm_ptr);

/**
 * @brief Deallocate memory
 *
 * @param shm_id
 * @return int
 */
int shm_delete(int shm_id);

#endif
```

Listing 5: shared_memory.h

```
#include "shared_memory.h"
#include <stdio.h>

int shm_init(key_t key, int size) {
    int shm_id;
    if ((shm_id = shmget(key, size, IPC_CREAT | 0666)) == -1) {
        printf("Error: shared memory failed %d \n", size);
    }

    return shm_id;
}

ShmData *shm_at(int shm_id) { return (ShmData *)shmat(shm_id, 0, 0); }

int shm_detach(void *shm_ptr) {
    if (shmdt(shm_ptr) == -1) {
        printf("Error: memory detach failed %p \n", shm_ptr);

        return 0;
    }

    return 1;
}

int shm_delete(int shm_id) {
    if (shmctl(shm_id, IPC_RMID, 0) == -1) {
        printf("Error: memory deallocation failed %d \n", shm_id);

        return 0;
    }
```

```
    return 1;
}
```

Listing 6: shared_memory.c

5.3 读者优先主程序

```
/**
 * Reader-Preference readers and writers simulation.
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/time.h>
#include <unistd.h>

#include "semaphores.h"
#include "shared_memory.h"

char time_buf[64];

char *log_time() {
    char *output = time_buf;
    time_t rawtime;
    struct tm *timeinfo;

    time(&rawtime);
    timeinfo = localtime(&rawtime);
    sprintf(output, "[%d-%d-%dT%02d:%02d:%02d]", timeinfo->tm_year + 1900,
            timeinfo->tm_mon + 1, timeinfo->tm_mday, timeinfo->tm_hour,
            timeinfo->tm_min, timeinfo->tm_sec);

    return output;
}

const int LOOPS = 10;

double u(int x) { return (double)(x + 1) / 10; }
double exp_time() {
    int x = rand() % 10;
    double sleeptime = (-(log(u(x))) * 1e6);

    return sleeptime;
}

int main(int argc, char *argv[]) {
    int rw_ratio, shm_size, peers;
    ShmData *shmData;
    int shm_id, shm_stat_id;
    int readers_counter_id;
    int *readers_counter_data;
    long *stat_data;
    pid_t pid;
    int read_mutex, write_mutex, counter_mutex, print_mutex;
```

```
if (argc != 4) {
    printf("Error: not enough arguments passed to program: Only %d/3 "
           "arguments\n ",
           argc);
}

peers = atoi(argv[1]);
// For inputs between 1 and 100 -> between 1% and 100%.
rw_ratio = atoi(argv[2]) - 1;
shm_size = atoi(argv[3]);

srand(getpid());

// Initialize the shared matrix.
// Allocate shared memory for ShmData.
shm_id = shm_init(597, sizeof(ShmData) * shm_size);
shmData = shm_at(shm_id);
for (int i = 0; i < shm_size; i++) {
    shmData[i].avg_time = 0;
    shmData[i].writes = 0;
    shmData[i].reads = 0;
}

// Initialize shared counter for readers
readers_counter_id = shm_init((key_t)211, shm_size * sizeof(int));
readers_counter_data = (int *)shm_at(readers_counter_id);
for (int i = 0; i < shm_size; i++) {
    readers_counter_data[i] = 0;
}

/** Initialize shared memory to keep stats
 * 0: Writer waiting time;
 * 1: Reader waiting time;
 * 2: Total waiting time;
 */
shm_stat_id = shm_init((key_t)2022, 3 * sizeof(long));
stat_data = (long *)shm_at(shm_stat_id);
for (int i = 0; i < 3; i++) {
    stat_data[i] = 0;
}

// Initalize array for child pid.
pid_t child_pid[peers];

// Initialize semaphores.
read_mutex = sem_init((key_t)10001, shm_size);
write_mutex = sem_init((key_t)20001, shm_size);
counter_mutex = sem_init((key_t)30001, shm_size);
print_mutex = sem_init((key_t)40001, shm_size);

// Initialize counters for statistics.
struct timeval stop, start;
double avg_time = 0;
double avg_time_writer = 0;
double avg_time_reader = 0;
long total_time_reader = 0;
long total_time_writer = 0;
```

```
long curr_time = 0;
long total_time = 0;
int reads = 0;
int writes = 0;
int entry = 0;

// File to print statistics.
char *log_file_path = "logs/file.log";
FILE *log_file = fopen(log_file_path, "w+");
char *log_stat_path = "logs/stat.log";
FILE *log_stat = fopen(log_stat_path, "a");

for (int i = 0; i < peers; i++) {
    pid = fork();
    // Child process
    if (pid == 0) {
        break;
    }
    child_pid[i] = pid;
}

// R/W processes.
if (pid != 0) {
    // parent
    printf("%s Parent with pid: %d\n", log_time(), pid);
} else {
    // child
    printf("%s Child with pid: %d\n", log_time(), getpid());
    srand(getpid());
    // counter for average time in microseconds.
    for (int i = 0; i < LOOPS; i++) {
        if (rand() % 100 > rw_ratio) {
            // child become writer
            entry = rand() % shm_size;
            writes++;
            gettimeofday(&start, NULL);

            /**
             * CRITICAL SECTION OF WRITER
             */
            sem_down(write_mutex, entry);
            printf("%s Child[%d](writer) accessing shared_memory[%d].\n",
                   log_time(), getpid(), entry);

            gettimeofday(&stop, NULL);
            curr_time = (stop.tv_sec - start.tv_sec) * 1e6 + stop.tv_usec
                -
                        start.tv_usec;
            printf("%s Child[%d](writer) waited for %lf seconds.\n",
                   log_time(), getpid(), (double)curr_time / (double)1e6);

            // Sleep to occupy symaphore.
            double sleeptime = exp_time();
            printf("%s Sleep for %lf seconds.\n", log_time(),
                   (double)sleeptime / 1e6);
            usleep(sleeptime);

            shmData[entry].writes++;
```

```
            sem_up(write_mutex, entry);
            printf("%s Child[%d](writer) exited shared_memory[%d].\n",
                   log_time(), getpid(), entry);
            /**
             * EXIT CRITICAL SECTION OF WRITER
             */

            // Update statistic data.
            total_time_writer += curr_time;
            total_time += curr_time;
        } else {
            // child become reader.
            entry = rand() % shm_size; // Access a random shared value in
                                       // shared memory.
            reads++;

            gettimeofday(&start, NULL);
            // Update counter.
            sem_down(counter_mutex, entry);
            readers_counter_data[entry]++;
            if (readers_counter_data[entry] == 1) {
                // Prevent writer from entering critical section.
                sem_down(write_mutex, entry);
            }
            sem_up(counter_mutex, entry);

            /**
             * CRITICAL SECTION OF READER
             */
            sem_down(read_mutex, entry);
            printf("%s Child[%d](reader) accessing shared_memory[%d].\n",
                   log_time(), getpid(), entry);

            gettimeofday(&stop, NULL);
            curr_time = (stop.tv_sec - start.tv_sec) * 1e6 + stop.tv_usec
                -
                        start.tv_usec;
            printf("%s Child[%d](reader) waited for %lf seconds.\n",
                   log_time(), getpid(), (double)curr_time / 1e6);

            // Sleep to occupy symaphore.
            double sleeptime = exp_time();
            printf("%s Sleep for %lf seconds.\n", log_time(),
                   (double)sleeptime / 1e6);
            usleep(sleeptime);
            shmData[entry].reads++;

            sem_up(read_mutex, entry);
            printf("%s Child[%d](reader) exited shared_memory[%d].\n",
                   log_time(), getpid(), entry);
            /**
             * EXIT CRITICAL ZONE OF READER.
             */

            // Update counter.
            sem_down(counter_mutex, entry);
            readers_counter_data[entry]--;
```

```
                if (readers_counter_data[entry] == 0) {
                    // Release writer lock.
                    sem_up(write_mutex, entry);
                }
                sem_up(counter_mutex, entry);

                // Update timer.
                total_time_reader += curr_time;
                total_time += curr_time;
            }
        }
    }

    if (pid == 0) {
        // Let parent process print statistics.
        avg_time_writer = (writes) ? ((double)total_time_writer / writes) :
            0;
        avg_time_reader = (reads) ? ((double)total_time_reader / reads) : 0;
        avg_time =
            (reads + writes) ? ((double)total_time / (reads + writes)) : 0;

        /**
         * STATISTICS PRINTER CRITICAL ZONE
         */
        sem_down(print_mutex, 0);
        fprintf(log_file, "%s Child[%d]:\n", log_time(), getpid());
        fprintf(
            log_file,
            "Total write time: %lf seconds; average write time: %lf
                seconds.\n",
            (double)total_time_writer / 1e6, (double)avg_time_writer / 1e6);
        fprintf(log_file, "Total read time: %lf; average read time: %lf\n",
                (double)total_time_reader / 1e6, (double)avg_time_reader /
                    1e6);
        fprintf(log_file, "Total time: %lf, total average time: %lf\n",
                (double)total_time / 1e6, (double)avg_time / 1e6);
        fprintf(log_file, "\n");

        // Shared statistics
        stat_data[0] += total_time_writer / 10;
        stat_data[1] += total_time_reader / 10;
        stat_data[2] += total_time / 10;

        sem_up(print_mutex, 0);
        /**
         * EXIT STATISTICS PRINTER CRITICAL ZONE
         */
    }

    // Exit processes.
    if (pid != 0) {
        // Parent process,
        // Terminate child process.
        int status;
        pid_t child_terminated_pid;

        // Close all child processes.
        for (size_t i = 0; i < (size_t)peers; i++) {
```

```
            child_terminated_pid = waitpid(child_pid[i], &status, 0);
            printf("%s Child[%d] has finished.\n", log_time(),
                   child_terminated_pid);
            if (WIFEXITED(status)) {
                printf("%s Child exited with code %d\n", log_time(), status);
            } else {
                printf("%s Child terminated abnormally %d\n", log_time(),
                       status);
            }
        }

        // Validate results.
        int sum_readers = 0;
        int sum_writers = 0;

        /**
         * ENTER STATISTIC PRINTER CRITICAL ZONE.
         */
        sem_down(print_mutex, 0);
        printf("%s Parent[%d] Writing statistics log.\n", log_time(), pid);
        fprintf(log_file, "\n VALIDATING RESULTS \n");
        for (size_t i = 0; i < (size_t)shm_size; i++) {
            fprintf(log_file, "shared[%zu]: readers[%d], writers[%d]\n", i,
                    shmData[i].reads, shmData[i].writes);
            sum_readers += shmData[i].reads;
            sum_writers += shmData[i].writes;
        }
        fprintf(log_file, "--------------------\n");
        fprintf(log_file, "total: readers[%d], writers[%d]\n", sum_readers,
                sum_writers);
        fprintf(log_file, "Total should be equal to (peers * LOOPS)[%d]\n",
                peers * LOOPS);
        if (peers * LOOPS == sum_writers + sum_readers) {
            fprintf(log_file, "Result is right!\n");
        }

        // Print statistics in log_stat.
        fprintf(log_stat, "%ld, %ld, %ld\n", (long)(stat_data[2] / 1e3),
                (long)(stat_data[1] / 1e3), (long)(stat_data[0] / 1e3));
        sem_up(print_mutex, 0);
        /**
         * EXIT STATISTIC PRINTER CRITICAL ZONE
         */
    } else {
        // Child process,
        // Exit.
        // pid = getpid();
        printf("%s Process with pid %d exitting...\n", log_time(), getpid());
        exit(0);
    }

    // Delete semaphores.
    sem_del(read_mutex);
    sem_del(write_mutex);
    sem_del(counter_mutex);
    sem_del(print_mutex);

    // Free allocated shared memory.
```

```
    shm_delete(shm_stat_id);
    shm_detach(stat_data);

    shm_delete(readers_counter_id);
    shm_detach(readers_counter_data);

    shm_delete(shm_id);
    shm_detach((ShmData *)shmData);

    // Close files.
    fclose(log_file);
    fclose(log_stat);

    return 0;
}
```

Listing 7: main_reader_priority.c

5.4 写者优先主程序

```
/**
 * Writer-Preference readers and writers simulation.
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/time.h>
#include <unistd.h>

#include "semaphores.h"
#include "shared_memory.h"

char time_buf[64];

char *log_time() {
    char *output = time_buf;
    time_t rawtime;
    struct tm *timeinfo;

    time(&rawtime);
    timeinfo = localtime(&rawtime);
    sprintf(output, "[%d-%d-%dT%02d:%02d:%02d]", timeinfo->tm_year + 1900,
            timeinfo->tm_mon + 1, timeinfo->tm_mday, timeinfo->tm_hour,
            timeinfo->tm_min, timeinfo->tm_sec);

    return output;
}

const int LOOPS = 10;

double u(int x) { return (double)(x + 1) / 10; }
double exp_time() {
    int x = rand() % 10;
    double sleeptime = (-(log(u(x))) * 1e6);
```

```
    return sleeptime;
}

int main(int argc, char *argv[]) {
    int rw_ratio, shm_size, peers;
    ShmData *shmData;
    int shm_id, shm_stat_id;
    int readers_counter_id;
    int *readers_counter_data;
    long *stat_data;
    pid_t pid;
    int read_mutex, write_mutex, reader_counter_mutex, print_mutex;

    // For writer priority.
    int writers_counter_id;
    int *writers_counter_data;
    int writer_counter_mutex;
    // Queue mutex
    int queue_mutex;

    if (argc != 4) {
        printf("Error: not enough arguments passed to program: Only %d/3 "
               "arguments\n ",
               argc);
    }

    peers = atoi(argv[1]);
    // For inputs between 1 and 100 -> between 1% and 100%.
    rw_ratio = atoi(argv[2]) - 1;
    shm_size = atoi(argv[3]);

    srand(getpid());

    // Initialize the shared matrix.
    // Allocate shared memory for ShmData.
    shm_id = shm_init(597, sizeof(ShmData) * shm_size);
    shmData = shm_at(shm_id);
    for (int i = 0; i < shm_size; i++) {
        shmData[i].avg_time = 0;
        shmData[i].writes = 0;
        shmData[i].reads = 0;
    }

    // Initialize shared counter for readers
    readers_counter_id = shm_init((key_t)211, shm_size * sizeof(int));
    readers_counter_data = (int *)shm_at(readers_counter_id);
    for (size_t i = 0; i < (size_t)shm_size; i++) {
        readers_counter_data[i] = 0;
    }

    // Initialize shared counter for writers
    writers_counter_id = shm_init((key_t)212, shm_size * sizeof(int));
    writers_counter_data = (int *)shm_at(writers_counter_id);
    for (size_t i = 0; i < (size_t)shm_size; i++) {
        writers_counter_data[i] = 0;
    }
```

```
/** Initialize shared memory to keep stats
 * 0: Writer waiting time;
 * 1: Reader waiting time;
 * 2: Total waiting time;
 */
shm_stat_id = shm_init((key_t)2022, 3 * sizeof(long));
stat_data = (long *)shm_at(shm_stat_id);
for (int i = 0; i < 3; i++) {
    stat_data[i] = 0;
}

// Initalize array for child pid.
pid_t child_pid[peers];

// Initialize semaphores.
read_mutex = sem_init((key_t)10001, shm_size);
write_mutex = sem_init((key_t)20001, shm_size);
reader_counter_mutex = sem_init((key_t)30001, shm_size);
writer_counter_mutex = sem_init((key_t)30002, shm_size);
queue_mutex = sem_init((key_t)50001, shm_size);
print_mutex = sem_init((key_t)40001, shm_size);

// Initialize counters for statistics.
struct timeval stop, start;
double avg_time = 0;
double avg_time_writer = 0;
double avg_time_reader = 0;
long total_time_reader = 0;
long total_time_writer = 0;
long curr_time = 0;
long total_time = 0;
int reads = 0;
int writes = 0;
int entry = 0;

// File to print statistics.
char *log_file_path = "logs/file.log";
FILE *log_file = fopen(log_file_path, "w+");
char *log_stat_path = "logs/stat.log";
FILE *log_stat = fopen(log_stat_path, "a");

for (int i = 0; i < peers; i++) {
    pid = fork();
    // Child process
    if (pid == 0) {
        break;
    }
    child_pid[i] = pid;
}

// R/W processes.
if (pid != 0) {
    // parent
    printf("%s Parent with pid: %d\n", log_time(), pid);
} else {
    // child
    printf("%s Child with pid: %d\n", log_time(), getpid());
    srand(getpid());
```

```
        // counter for average time in microseconds.
        for (int i = 0; i < LOOPS; i++) {
            if (rand() % 100 > rw_ratio) {
                // child become writer
                entry = rand() % shm_size;
                writes++;
                gettimeofday(&start, NULL);

                // Update counter.
                sem_down(writer_counter_mutex, entry);
                writers_counter_data[entry]++;
                if (writers_counter_data[entry] == 1) {
                    // Prevent the following readers from entering queue.
                    sem_down(queue_mutex, entry);
                }
                sem_up(writer_counter_mutex, entry);

                /**
                 * CRITICAL SECTION OF WRITER
                 */
                sem_down(write_mutex, entry);
                printf("%s Child[%d](writer) accessing shared_memory[%d].\n",
                       log_time(), getpid(), entry);

                gettimeofday(&stop, NULL);
                curr_time = (stop.tv_sec - start.tv_sec) * 1e6 + stop.tv_usec
                    -
                            start.tv_usec;
                printf("%s Child[%d](writer) waited for %lf seconds.\n",
                       log_time(), getpid(), (double)curr_time / (double)1e6);

                // Sleep to occupy critical section.
                double sleeptime = exp_time();
                printf("%s Sleep for %lf seconds.\n", log_time(),
                       (double)sleeptime / 1e6);
                usleep(sleeptime);

                shmData[entry].writes++;

                sem_up(write_mutex, entry);
                printf("%s Child[%d](writer) exited shared_memory[%d].\n",
                       log_time(), getpid(), entry);
                /**
                 * EXIT CRITICAL SECTION OF WRITER
                 */
                // Update counter.
                sem_down(writer_counter_mutex, entry);
                writers_counter_data[entry]--;
                if (writers_counter_data[entry] == 0) {
                    // No writers waiting. Allow the following readers to
                    // enqueue.
                    sem_up(queue_mutex, entry);
                }
                sem_up(writer_counter_mutex, entry);

                // Update statistic data.
                total_time_writer += curr_time;
                total_time += curr_time;
```

```
        } else {
            // child become reader.
            entry = rand() % shm_size; // Access a random shared value in
                                       // shared memory.
            reads++;

            gettimeofday(&start, NULL);

            // Wait for queue mutex.
            sem_down(queue_mutex, entry);

            // Update counter.
            sem_down(reader_counter_mutex, entry);
            readers_counter_data[entry]++;
            if (readers_counter_data[entry] == 1) {
                // Prevent writer from entering critical section.
                sem_down(write_mutex, entry);
            }

            // Release queue mutex.
            sem_up(queue_mutex, entry);
            // Release reader counter mutex.
            sem_up(reader_counter_mutex, entry);

            /**
             * CRITICAL SECTION OF READER
             */
            // Wait for read mutex.
            sem_down(read_mutex, entry);
            printf("%s Child[%d](reader) accessing shared_memory[%d].\n",
                   log_time(), getpid(), entry);

            gettimeofday(&stop, NULL);
            curr_time = (stop.tv_sec - start.tv_sec) * 1e6 + stop.tv_usec
                -
                        start.tv_usec;
            printf("%s Child[%d](reader) waited for %lf seconds.\n",
                   log_time(), getpid(), (double)curr_time / 1e6);

            // Sleep to occupy symaphore.
            double sleeptime = exp_time();
            printf("%s Sleep for %lf seconds.\n", log_time(),
                   (double)sleeptime / 1e6);
            usleep(sleeptime);
            shmData[entry].reads++;

            sem_up(read_mutex, entry);
            printf("%s Child[%d](reader) exited shared_memory[%d].\n",
                   log_time(), getpid(), entry);
            /**
             * EXIT CRITICAL ZONE OF READER.
             */

            // Update counter.
            sem_down(reader_counter_mutex, entry);
            readers_counter_data[entry]--;
            if (readers_counter_data[entry] == 0) {
                // Release writer lock.
```

```
                sem_up(write_mutex, entry);
            }
            sem_up(reader_counter_mutex, entry);

            // Update timer.
            total_time_reader += curr_time;
            total_time += curr_time;
        }
    }
}

if (pid == 0) {
    // Let parent process print statistics.
    avg_time_writer = (writes) ? ((double)total_time_writer / writes) :
        0;
    avg_time_reader = (reads) ? ((double)total_time_reader / reads) : 0;
    avg_time =
        (reads + writes) ? ((double)total_time / (reads + writes)) : 0;

    /**
     * STATISTICS PRINTER CRITICAL ZONE
     */
    sem_down(print_mutex, 0);
    fprintf(log_file, "%s Child[%d]:\n", log_time(), getpid());
    fprintf(
        log_file,
        "Total write time: %lf seconds; average write time: %lf
            seconds.\n",
        (double)total_time_writer / 1e6, (double)avg_time_writer / 1e6);
    fprintf(log_file, "Total read time: %lf; average read time: %lf\n",
            (double)total_time_reader / 1e6, (double)avg_time_reader /
                1e6);
    fprintf(log_file, "Total time: %lf, total average time: %lf\n",
            (double)total_time / 1e6, (double)avg_time / 1e6);
    fprintf(log_file, "\n");

    // Shared statistics
    stat_data[0] += total_time_writer / 10;
    stat_data[1] += total_time_reader / 10;
    stat_data[2] += total_time / 10;

    sem_up(print_mutex, 0);
    /**
     * EXIT STATISTICS PRINTER CRITICAL ZONE
     */
}

// Exit processes.
if (pid != 0) {
    // Parent process,
    // Terminate child process.
    int status;
    pid_t child_terminated_pid;

    // Close all child processes.
    for (size_t i = 0; i < (size_t)peers; i++) {
        child_terminated_pid = waitpid(child_pid[i], &status, 0);
        printf("%s Child[%d] has finished.\n", log_time(),
```

```
                child_terminated_pid);
        if (WIFEXITED(status)) {
            printf("%s Child exited with code %d\n", log_time(), status);
        } else {
            printf("%s Child terminated abnormally %d\n", log_time(),
                   status);
        }
    }

    // Validate results.
    int sum_readers = 0;
    int sum_writers = 0;

    /**
     * ENTER STATISTIC PRINTER CRITICAL ZONE.
     */
    sem_down(print_mutex, 0);
    printf("%s Parent[%d] Writing statistics log.\n", log_time(), pid);
    fprintf(log_file, "\n---------VALIDATING RESULTS---------\n");
    for (size_t i = 0; i < (size_t)shm_size; i++) {
        fprintf(log_file, "shared[%zu]: readers[%d], writers[%d]\n", i,
                shmData[i].reads, shmData[i].writes);
        sum_readers += shmData[i].reads;
        sum_writers += shmData[i].writes;
    }
    fprintf(log_file, "--------------------\n");
    fprintf(log_file, "total: readers[%d], writers[%d]\n", sum_readers,
            sum_writers);
    fprintf(log_file, "Total should be equal to (peers * LOOPS)[%d]\n",
            peers * LOOPS);
    if (peers * LOOPS == sum_writers + sum_readers) {
        fprintf(log_file, "Result is right!\n");
    }

    // Print statistics in log_stat.
    fprintf(log_stat, "%ld, %ld, %ld\n", (long)(stat_data[2] / 1e3),
            (long)(stat_data[1] / 1e3), (long)(stat_data[0] / 1e3));
    sem_up(print_mutex, 0);
    /**
     * EXIT STATISTIC PRINTER CRITICAL ZONE
     */
} else {
    // Child process,
    // Exit.
    // pid = getpid();
    printf("%s Process with pid %d exiting...\n", log_time(), getpid());
    exit(0);
}

// Delete semaphores.
sem_del(read_mutex);
sem_del(write_mutex);
sem_del(queue_mutex);
sem_del(reader_counter_mutex);
sem_del(writer_counter_mutex);
sem_del(print_mutex);

// Free allocated shared memory.
```

```
    shm_delete(shm_stat_id);
    shm_detach(stat_data);

    shm_delete(readers_counter_id);
    shm_detach(readers_counter_data);

    shm_delete(writers_counter_id);
    shm_detach(writers_counter_data);

    shm_delete(shm_id);
    shm_detach((ShmData *)shmData);

    // Close files.
    fclose(log_file);
    fclose(log_stat);

    return 0;
}
```

Listing 8: main_writer_priority.c

6 附录

代码仓库: OSLab