操作系统第一次大作业 实验报告

张灿晖 18364114 智科 2 班

一、生产者消费者问题

1. （30 分）生产者消费者问题

1. 需要创建生产者和消费者两个进程（注意：不是线程），一个 prod，一个 cons，每个进程有 3 个线程。两个进程之间的缓冲最多容纳 20 个数据。
2. 每个生产者线程随机产生一个数据，打印出来自己的 id（进程、线程）以及该数据；每个消费者线程取出一个数据，然后打印自己的 id 和数据。
3. 生产者和消费者这两个进程之间通过共享内存来通信，通过信号量来同步。
4. 生产者生成数据的间隔和消费者消费数据的间隔，按照负指数分布来控制，各有一个控制参数λp，λc
5. 运行的时候，开两个窗口，一个./prod λp，另一个./cons λc，要求测试不同的参数组合，打印结果，截屏放到作业报告里。

prod.c 代码：

在代码开头，预定义了一些常量宏、结构体、信号量和指针：

```c
#define MAX_THREADS 3 // 最大线程数设置为3
#define MAX_BUF_SIZE 20 // 缓冲区最大规模设置为20

// 缓冲区结构体
typedef struct buf
{
    int rear;
    int head;
    int buffer[MAX_BUF_SIZE];
}buf;

sem_t *full = 0; // 缓冲区满的信号量
sem_t *empty = 0; // 缓冲区空的信号量
sem_t *s_mutex = 0; // 针对缓冲区结构体的互斥锁
void *ptr = NULL; // 全局空指针，用于写入数据
```

函数 neg_exp_distribute 为负指数分布采样函数，返回值为生产者生产数据之间的间隔时间：

```c
// 返回负指数分布的采样值
double neg_exp_distribute(double arg_p){
    double res;
    do
    {
        res = ((double)rand() / RAND_MAX);
    }while((res==0) || (res==1));
    res = -(1/arg_p*log(res));
    return res;
}
```

函数 prod_func 为生产者线程核心函数。该函数先将传入的参数 arg_p_ptr 空指针转为双精度浮点数，调用函数 pthread_self 获取当前线程 id，存储于变量 temp_thread 中。

接下来，进入条件为 1 的 while 循环，先调用函数 neg_exp_distribute 获取本次写入数据的时间间隔，再调用函数 usleep 让线程休眠这段时间间隔。接下来，以函数 rand 为基础生成需要写入共享内存的随机数据，并存储于整形变量 item 中。

随后，将全局作用域中声明的指向 void 的指针 ptr 强制转型为指向共享内存区域的指针 shared_mem_ptr，准备进行数据写入。在写入之前，要等待缓冲区空的信号量 empty，并获取互斥锁 s_mutex。当准备工作就绪后，写入数据的过程也就是对循环队列数组 buffer 的尾端 rear 赋值的过程。写入完成后，打印写入的缓存区数组下标信息、线程标识符和写入的整型数据值，随后释放互斥锁并发出缓冲区满的信号量，等待消费者进程读取数据。

```
// 生产者线程核心函数
void* prod_func(void *arg_p_ptr)
{
    double arg_p = (double)(*(int*) arg_p_ptr);
    pthread_t temp_thread = pthread_self();

    while(1)
    {
        int interval = neg_exp_distribute(arg_p); // 写入数据的时间间隔
        usleep((unsigned int)(neg_exp_distribute(interval)*1e6)); // 以时
        间间隔为依据休眠对应时间
        int item = rand() % 14; // 随机产生需要写入的数据

        struct buf *shared_mem_ptr = ((struct buf*) ptr); // 空指针强制类型
        转换为缓冲区结构体指针，准备进行数据写入
        sem_wait(empty); // 等缓冲区空，准备写入数据
        sem_wait(s_mutex); // 获取缓冲区互斥锁后才能写入

        // 写入并更新循环队列尾部指针
        shared_mem_ptr->buffer[shared_mem_ptr->rear] = item;
        shared_mem_ptr->rear = (shared_mem_ptr->rear + 1) % MAX_BUF_SIZE;
        // 打印写入信息
        printf("Writing item... buffer: [%d]\tthread id: [%lu]\titem
        content: (%d)\n",shared_mem_ptr->rear, (unsigned long)
        temp_thread, item);

        sem_post(s_mutex); // 释放缓冲区互斥锁
        sem_post(full); // 数据已写满，设缓冲区状态为满
    }

    pthread_exit(0);
}
```

主函数 main 完成了变量的初始化和线程的创建工作。函数 main 的函数体之中，先创建了一个缓冲区的结构体并调用函数 memset 对其进行初始化。随后

连续三次调用函数 sem_open 打开信号量 full、empty、s_mutex，并紧接着调用函数 sem_init 对它们进行初始化。

随后，调用函数 shm_open 打开一个共享内存区域，并调用函数 ftruncate 对其进行截断，使其容量刚好为一个缓冲区结构体 buf 的规模，再调用函数 mmap 将其映射到文件中，以备后续进行写入和读取：

```
int main(int argc, char* argv[])
{
    struct buf shared_mem; // 缓冲区结构体实例
    memset(&shared_mem, 0, sizeof(struct buf)); // 初始化缓冲区，全设为0

    // sem_open的第二个参数O_CREAT表示如果不存在指定标识符的信号量就创建，第三个参数表示创建后的默认权限，但由于下列三个信号量我们已在全局声明，该函数就仅仅是初始化它们
    full = sem_open("full", O_CREAT, 0666, 0);
    empty = sem_open("empty", O_CREAT, 0666, 0);
    s_mutex = sem_open("mutex", O_CREAT, 0666, 0);

    // sem_init的第二个参数设置为1表示本进程中的线程不可共享信号量，第三个参数表示信号量需要被初始化的值
    sem_init(full, 1, 0);
    sem_init(empty, 1, MAX_BUF_SIZE);
    sem_init(s_mutex, 1, 1);

    int shared_mem_fd = shm_open("buffer", O_CREAT | O_RDWR, 0666); // 通过共享内存进行数据共享
    ftruncate(shared_mem_fd, sizeof(struct buf)); // 将其截断到一个缓冲区结构体的规模
    ptr = mmap(0, sizeof(struct buf), PROT_WRITE, MAP_SHARED,
    shared_mem_fd,0); // 将其映射到文件中以备生产者进程访问
```

接下来，main 函数要进行线程的创建，其核心就是通过 for 循环调用函数 pthread_create 并传入控制参数 p：

```
    int arg_p = atoi(argv[1]); // 读取控制参数p，并转为整形数
    pthread_t td[MAX_THREADS]; // 线程标识符数组
    int err[MAX_THREADS];
    for(int ii = 0; ii < MAX_THREADS; ii++) // 创建MAX_THREADS个线程
    {
        err[ii] = pthread_create(&td[ii], NULL, prod_func, (void*)&arg_p);
        if(err[ii] < 0)
        {
            perror("Thread create failed!\n");
            exit(-1);
        }
    }

    // 回收资源
    for (int ii = 0; ii < MAX_THREADS; ii++)
    {
        pthread_join(td[ii], NULL);
    }

    return 0;
}
```

cons.c 代码：

由于 cons.c 中预定义的数据和 main 函数的主要流程都与 prod.c 类似，这里就不再赘述，仅对消费者程序中的核心线程函数 cons_func 作出说明。

函数 cons_func 为生产者线程核心函数。该函数先将传入的参数 arg_c_ptr 空指针转为双精度浮点数，调用函数 pthread_self 获取当前线程 id，存储于变量 temp_thread 中。

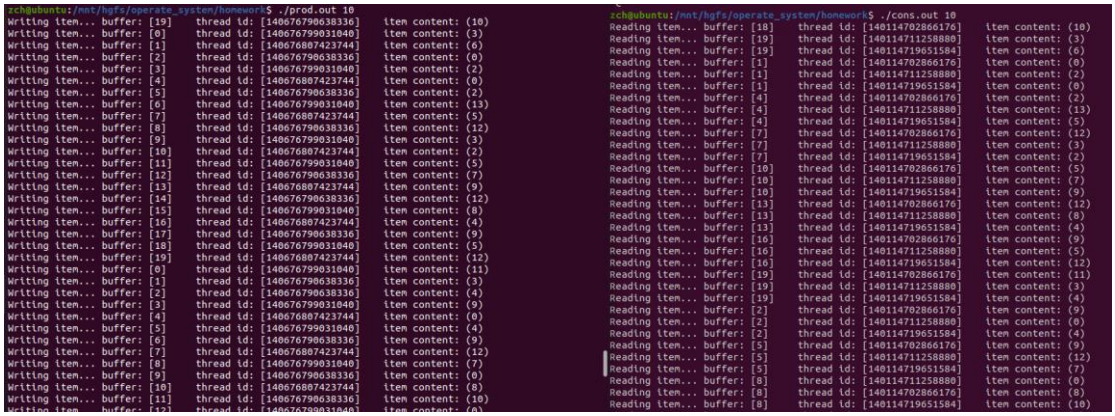
接下来，进入条件为 1 的 while 循环，先调用函数 neg_exp_distribute 获取本次写入数据的时间间隔，再调用函数 sleep 让线程休眠这段时间间隔。随后，将全局作用域中声明的指向 void 的指针 ptr 强制转型为指向共享内存区域的指针 shared_mem_ptr，准备进行数据读取。在读取之前，要等待缓冲区满的信号量 full，这说明生产者进程已经生产完毕，并获取互斥锁 s_mutex。当准备工作就绪后，读取数据的过程也就是提取循环队列数组 buffer 的头端 head 值的过程。

读取完成后，打印读取的缓存区数组下标信息、线程标识符和读取的整型数据值，随后释放互斥锁并发出缓冲区空的信号量，等待生产者进程继续生产数据。

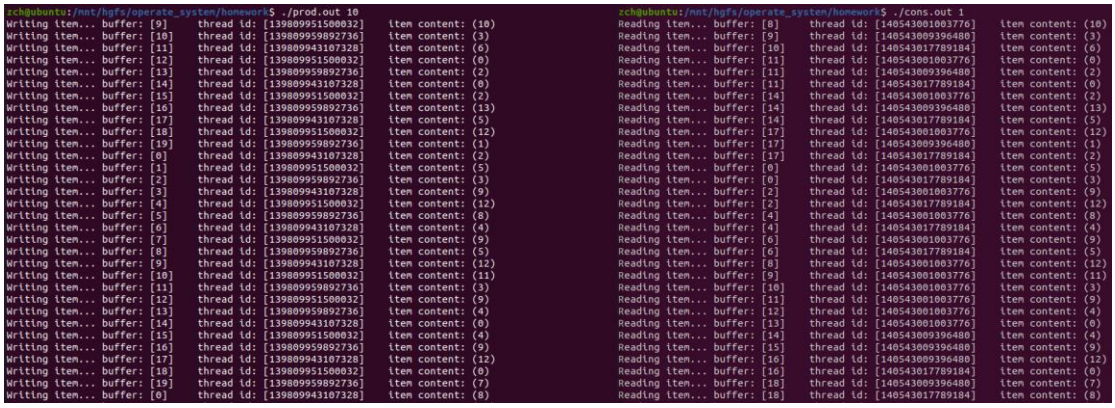
```
void *cons_func(void *arc_c_ptr)
{
    double arg_c = *((int*)arc_c_ptr);
    pthread_t temp_thread = pthread_self();

    while(1)
    {
        int interval = neg_exp_distribute(arg_c); // 读取数据的时间间隔
        sleep((unsigned int)neg_exp_distribute(interval)); // 以时间间隔为
        依据休眠对应时间

        struct buf *shared_mem_ptr = ((struct buf*)ptr);
        sem_wait(full); // 等缓冲区满，准备读取数据
        sem_wait(s_mutex); // 获取缓冲区互斥锁后才能读取

        // 从循环队列中读取数据
        int item = shared_mem_ptr->buffer[shared_mem_ptr->head];
        shared_mem_ptr->head = (shared_mem_ptr->head+1) % MAX_BUF_SIZE;

        // 打印读取到的数据
        printf("Reading item... buffer: [%d]\tthread id: [%lu]\titem
        content: (%d)\n", shared_mem_ptr->rear, (unsigned long)
        temp_thread, item);

        sem_post(s_mutex); // 释放缓冲区互斥锁
        sem_post(empty); // 数据已读完，设缓冲区状态为空


    }
    pthread_exit(0);
}
```

运行截图：

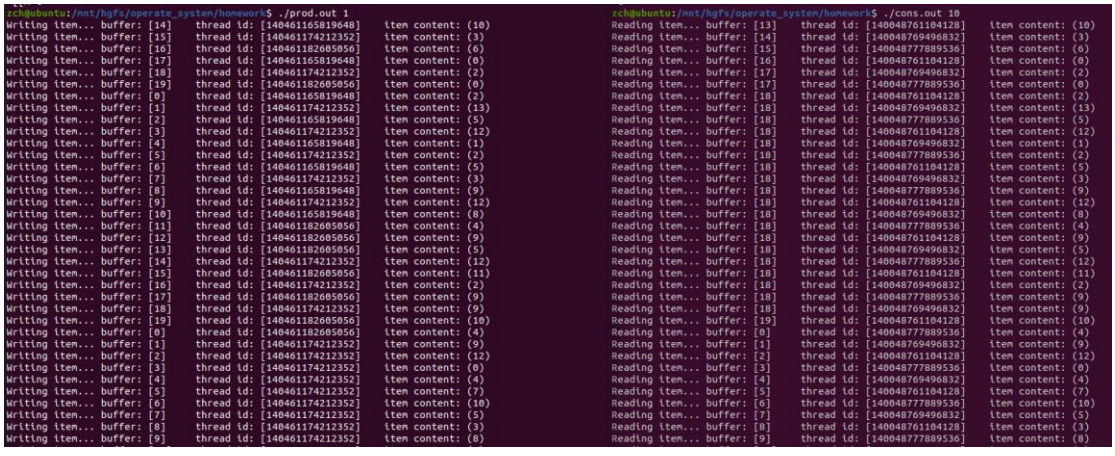
1. 当控制参数 p 等于控制参数 c 时，即生产速率等于消费速率时，生产者进程生产数据与消费者消费数据同步进行，总的效率最高：



2. 当控制参数 p 大于控制参数 c 时，即生产速率小于消费速率时，消费者进程的消费要等待生产者进程生产数据才可进行，总的效率中等：



3. 当控制参数 p 小于控制参数 c 时，即生产速率大于消费速率时，生产者进程的生产要等待消费者进程消费数据才可进行，总的效率中等：



总而言之，数据被生产和消费的间隔会影响系统的效率，需要仔细设计并调整参数，以达到最优性能。

二、哲学家就餐问题

2. （20 分）哲学家就餐问题

参考课本（第十版）第 7 章 project 3 的要求和提示

1. 使用 POSIX 实现
5. 要求通过 make，能输出 dph 文件，输出哲学家们的状态。打印结果，截屏放到作业报告中。

先定义打印宏、哲学家最大数目常量和叉子数组（由 POSIX 信号量 sem_t 组成）：

```
// 打印宏
#define PHL_DEBUG
#ifdef PHL_DEBUG
#define PHL_PRT(...) do{printf("[%s]", __func__); printf(""__VA_ARGS__);}while(0)
#else
#define PHL_PRT(...)
#endif

#define PHILOSHOPHER_NUM 5 // 定义哲学家最大数目

sem_t fork_num[PHILOSHOPHER_NUM]; // 叉子数组
```

接下来定义的函数 time_of_think_or_eat，用于生成哲学家思考或吃饭的时间，通过随机数函数 rand 为基础产生一个 5s 内的随机时间，然后 sleep 这个随机时间，以模拟思考或是吃饭。

接下来，定义了核心函数 pickup_forks，该函数先接受一个表征哲学家标识符的空指针，再在函数体中将其转化为整型变量，并取引用赋值给变量 p_id，由 p_id 我们可以定义出表征左右手叉子的变量 left_fork 和 right_fork，再定义变量 left_get 和 right_get，以表征获取左右手的叉子是否成功。

接下来，函数进入 while 条件为 1 的无限循环，先让哲学家思考一段时间，即调用 time_of_think_of_eat。在哲学家思考完毕后，进入吃饭环节，先分别尝试获取左右手的叉子，即调用 POSIX 函数 sem_trywait，尝试能否获取

```
void* pickup_forks(void* philoshopher_id)
{
    int p_id = *((int *)philoshopher_id);

    // 定义左右手变量表示哲学家左右手的叉子
    int left_fork = (p_id-1+PHILOSHOPHER_NUM)%PHILOSHOPHER_NUM;
    int right_fork = p_id;

    int left_get = 0;
    int right_get = 0;

    while(1)
    {
        // 哲学家要思考了
        PHL_PRT("philosopher %d is going to think ", p_id);
        time_of_think_or_eat();

        // 获取左右手的叉子
        left_get = sem_trywait(&fork_num[left_fork]);
        right_get = sem_trywait(&fork_num[right_fork]);
```

成功：

在上一步调用以后，我们就可以通过变量 left_fork 和 right_fork 的值判定左右手的叉子是否获取成功，倘若均为 0，则表示都获取到了，可以开始吃饭，因此打印一条哲学家准备吃饭的信息，再调用函数 time_of_think_or_eat 模拟哲学家吃饭，吃完饭后哲学家要释放资源，即把叉子放回，以免造成死锁，因此调用 POSIX 函数 sem_post 释放左右手叉子：

```
// 哲学家通过获取左右手的叉子来判断是否可以吃饭
if(left_get == 0 && right_get == 0) // 都为 0 表示 OK ,可以吃饭
{
    PHL_PRT("philosopher %d going to eat ", p_id);
    time_of_think_or_eat();

    // 放回叉子，即return_forks
    sem_post(&fork_num[left_fork]);
    sem_post(&fork_num[right_fork]);
}
```

倘若哲学家没有获取到左右手的两只叉子，而是只获得了其中之一，或者干脆两只叉子都没获取到，则情况又有所不同，下面对着三种情况统一说明。两只叉子若都没有获取到，直接运行 continue 语句，让哲学家回到循环头部，继续思考；而如果左手或者右手的叉子获取成功，那么先调用函数 sleep 等候 1ms 再调用 POSIX 函数 sem_trywait 尝试获取另一侧的叉子并将结果复制给对应 get 变量。倘若等待 1ms 后获取成功，则进行吃饭的步骤，并在吃完饭后释放两根叉子，返回循环头部继续思考。而倘若获取失败，则哲学家会释放已经获取到的左/右手叉子，让其他哲学家有吃饭的机会，代码均已贴出如下：

```
else if(left_get == 0 && right_get != 0) // 左手获取了叉子，等待1s，判断右手看是否有叉子
{
    sleep(1);
    right_get = sem_trywait(&fork_num[right_fork]);  // 再次获取右手叉子
    if(right_get == 0) // 获取成功
    {
        PHL_PRT("Philosopher %d going to eat ", p_id);
        time_of_think_or_eat();

        // 放回叉子，即return_forks
        sem_post(&fork_num[left_fork]);
        sem_post(&fork_num[right_fork]);
    }
    else
    {
        sem_post(&fork_num[right_fork]);
    }
}
```

```
        else if(right_get == 0 && left_get != 0) // 右手获取了叉子，等待 1 s，判断左手看是否有叉子
        {
            sleep(1);
            left_get = sem_trywait(&fork_num[left_fork]); // 再次获取右手叉子
            if (left_get == 0) // 获取成功
            {
                PHL_PRT("Philosopher %d is going to eat ", p_id);
                time_of_think_or_eat();

                // 放回叉子，即return_forks
                sem_post(&fork_num[left_fork]);
                sem_post(&fork_num[right_fork]);
            }
            else
            {
                sem_post(&fork_num[left_fork]);
            }
        }
        else  // 表示左右手都没有获得叉子，直接进入思考
        {
            continue;
        }
    }
}
```

接下来，我们对 main 函数进行分析，main 函数先调用函数 srand，以当前时间为参数设置随机数种子，以确保函数 time_of_think_or_eat 中的函数 rand 可以正确地产生随机数。随后，main 函数的第一个 for 循环完成了对信号量数组 fork_num 的初始化。

随后，进入第二个 for 循环，这一循环的主要工作即为调用 POSIX 函数 pthread_create 创建多个执行函数 pickup_fork 的进程，并且创建一个进程后就检查是否创建成功，如果创建失败，就要打印错误信息，并进入内层 for 循环调用 POSIX 函数 sem_destory 释放所有已创建的进程，而在释放已创建进程的过程中，如果函数 sem_destory 出现异常，则打印错误信息并直接调用函数 exit(-1)终止程序。要注意的是，在内层循环结束后要调用函数 usleep(100)，这主要是为了防止传入的 jj 值都一样。

```c
int main(int argc, char* argv[])
{
    srand((unsigned)time(NULL)); // 种下时间种子，产生随机数表示哲学家的思考和吃饭时间

    int ii = 0;
    for(ii = 0; ii < PHILOSHOPHER_NUM; ii++)
    {
        if(sem_init(&fork_num[ii], 0, 1) < 0) // 初始化信号量为 1，且共享（参数2设为0）
        {
            perror("sem_init error!\n");
            exit (-1);
        }
        else
        {
            printf("semaphore %d is created\n", ii);
        }
    }

    // 模拟思考吃饭过程
    pthread_t td[PHILOSHOPHER_NUM];
    int err[PHILOSHOPHER_NUM];
    int jj = 0, kk = 0;
    for(jj = 0; jj < PHILOSHOPHER_NUM; jj++)
    {
        err[jj] = pthread_create(&td[jj], NULL, pickup_forks, (void*)&jj);
        if(err[jj] < 0)
        {
            perror("Philosopher create failed!\n");
            for(kk = 0; kk < PHILOSHOPHER_NUM; kk++)
            {
                if(sem_destroy(&fork_num[kk]) < 0)
                {
                    perror("Semaphore error!\n");
                    exit(-1);
                }
            }
            exit(-1);
        }
        usleep(100);
    }
```

main 函数最后的部分很简单，即让主函数在条件为 1 的 while 循环中不断调用函数 sleep 休眠，让 5 个哲学家交替思考与吃饭，当计数变量 tmp 达到预设值后跳出 while 循环。最后的 for 循环调用 POSIX 函数 sem_destory 释放已创建的进程，防止资源泄漏。

```c
    int tmp = 0;
    while(1)
    {
        printf("-----------1s passed-------------\n");
        sleep(1); // 主函数睡眠
        tmp++;
        if(tmp == 1000)
        {
            break;
        }
    }

    for(ii = 0; ii < PHILOSHOPHER_NUM; ii++)
    {
        if(sem_destroy(&fork_num[ii]) < 0) // 初始化信号量为 1，且共享
        {
            perror("sem_init error!\n");
            exit (-1);
        }
        else
        {
            printf("semaphore %d is destoryed\n", ii);
        }
    }

    return 0;
}
```

dph. c 编译运行的结果截图如下：

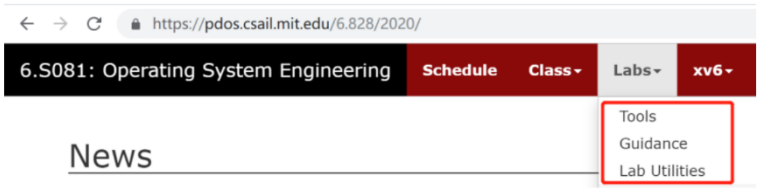
```
[pickup_forks]philosopher 0 is going to think time = 2 s
[pickup_forks]philosopher 1 is going to think time = 1 s
[pickup_forks]philosopher 2 is going to think time = 3 s
[pickup_forks]philosopher 3 is going to think time = 5 s
[pickup_forks]philosopher 4 is going to think time = 1 s
-----------1s passed-------------
[pickup_forks]philosopher 1 going to eat time = 4 s
-----------1s passed-------------
[pickup_forks]philosopher 4 going to eat time = 5 s
[pickup_forks]philosopher 0 is going to think time = 3 s
-----------1s passed-------------
-----------1s passed-------------
[pickup_forks]philosopher 2 is going to think time = 5 s
-----------1s passed-------------
[pickup_forks]philosopher 3 is going to think time = 3 s
[pickup_forks]philosopher 0 is going to think time = 4 s
[pickup_forks]philosopher 1 is going to think time = 4 s
-----------1s passed-------------
[pickup_forks]philosopher 4 is going to think time = 3 s
-----------1s passed-------------
-----------1s passed-------------
-----------1s passed-------------
[pickup_forks]philosopher 0 going to eat time = 1 s
[pickup_forks]philosopher 3 is going to think time = 1 s
[pickup_forks]philosopher 2 going to eat time = 3 s
[pickup_forks]philosopher 4 is going to think time = 5 s
-----------1s passed-------------
```

可以发现，5 位哲学家都能够轮替吃饭与思考，说明我们的程序编写成功了，使哲学家用餐问题得到了解决。

三、MIT 6.S081 课程实验

3. （50 分）MIT 6.S081 课程实验

1. （20 分）阅读 MIT 6.S081 项目介绍,如下图；完成 xv6 的安装和启动（Ctrl-a x 可退出）；完成 Lab: Xv6 and Unix utilities 中的 sleep（easy）任务，即在 user/下添加 sleep.c 文件。在报告中提供 sleep.c 的代码，并提供 sleep 运行的屏幕截图。提示：在 vmware 下安装 ubuntu20，可以较为顺利完成 xv6 安装和编译。

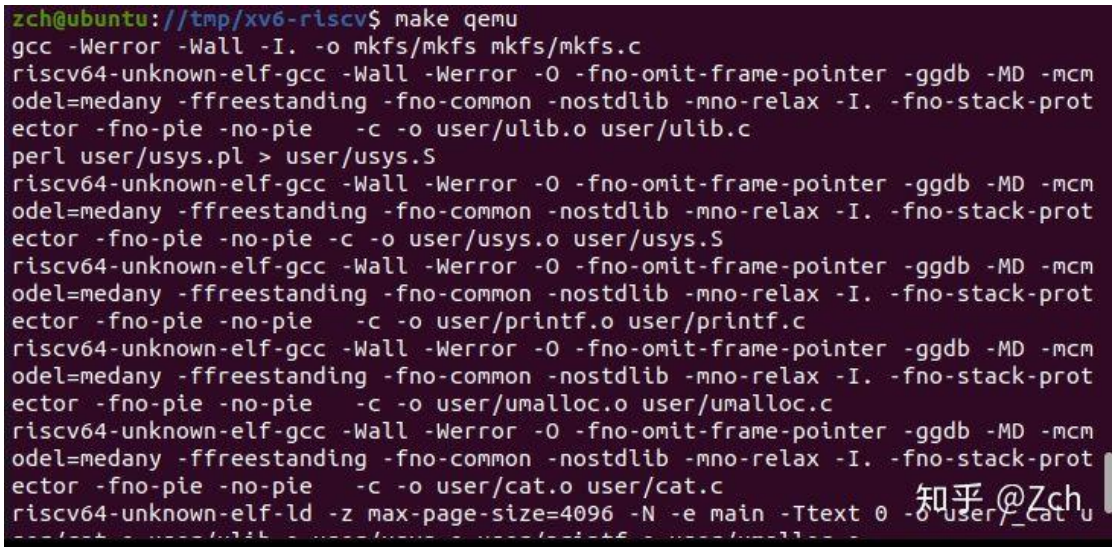


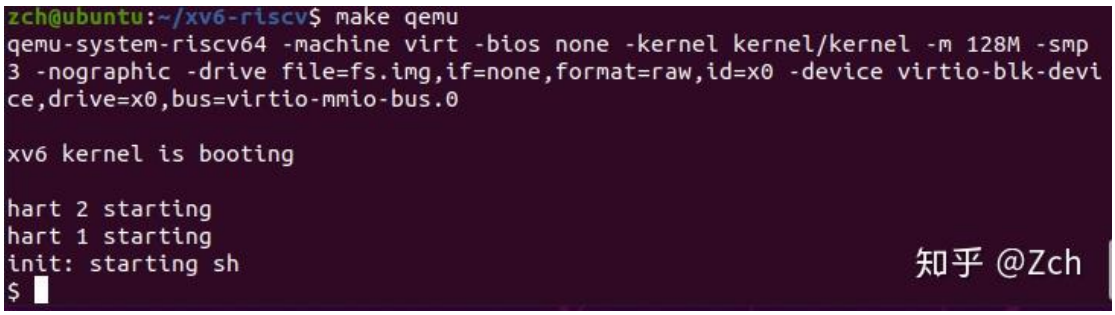
1. xv6 安装

xv6 操作系统详细的安装流程可以点击链接：

https://zhuanlan.zhihu.com/p/267159664

在我创作的知乎文章中查看细节，其中包含了从虚拟机安装 ubuntu 到从 ubuntu 安装 xv6 和 qemu 的全过程，这里仅展示安装完成并运行的截图如下：





2. sleep.c 文件

如下图，代码的核心部分如下：

```c
int main(int argc, char *argv[])
{
    if(argc <= 1)
    {
        printf("Error, received parameter is not enough!\n");
        exit(-1);
    }
    else if(argc >= 3)
    {
        printf("Error, received too many parameters!\n");
        exit(-1);
    }
    else
    {
        int sleepTime = atoi(argv[1]);
        printf("Sleep time: %d\n", sleepTime);
        sleep(sleepTime);
        printf("Wake up!\n");
        exit(0);
    }
}
```

其中，main 函数接收命令行参数以实现用户输入控制效果。main 函数接收的第一个整型参数 argc 为用户输入的命令字符串的数量，第二个字符串指针数组参数 argv 储存了用户依次输入的各个命令字符串。在这里，用户输入的正常格式应该为“sleep 数字”，如 sleep 10，即为让系统休眠 10 毫秒。

接下来，main 函数对接收的参数进行处理，接受到的参数数量应该正好为 2，超过规定数量和少于规定数量都是不符合规定的，因此前两个判定条件对参数的数量进行判断，确保参数数量刚好为 2，否则打印错误信息并调用函数 exit(-1)退出。

最后，若传入的参数数量无误，则将接收的第二个字符串 argv[1]，用函数 atoi 转换为整型数，并存储在变量 sleepTime 中，并打印休眠时间信息。接下来，调用标准函数 sleep 并传入休眠时间这一整型数作为参数，让系统“真正”休眠，在休眠结束后返回并打印“Wake up!\n”，最后调用函数 exit(0)退出，运行结果截图如下：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ sleep 20
Sleep time: 20
Wake up!
$ sleep 30
Sleep time: 30
Wake up!
$ sleep 40
Sleep time: 40
Wake up!
$ sleep 50
Sleep time: 50
Wake up!
$
```

2. （30 分）结合 xv6 book 第 1、2、7 章，阅读 xv6 内核代码(kernel/目录下)的进程和调度相关文件，围绕 swtch.S，proc.h/proc.c，理解进程的基本数据结构，组织方式，以及调度方法。提示：用 source insight 阅读代码较为方便。
   a) 修改 proc.c 中 procdump 函数，打印各进程的扩展信息，包括大小（多少字节）、内核栈地址、关键寄存器内容等，通过^p 可以查看进程列表，提供运行屏幕截图。
   b) 在报告中，要求逐行对 swtch.S，scheduler(void)，sched(void)，yield(void)等函数的核心部分进行解释，写出你对 xv6 中进程调度框架的理解。阐述越详细、硬件/软件接口部分理解越深，评分越高。
   c) 对照 Linux 的 CFS 进程调度算法，指出 xv6 的进程调度有何不足；设计一个更好的进程调度框架，可以用自然语言（可结合伪代码）描述，但不需要编码实现。

a)：

修改后的函数 procdump 代码如下：

```
void
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  struct proc *p;
  char *state;

  printf("\n");
  for(p = proc; p < &proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
    printf("pid:%d state:%s name:%s size:%d kernel:%d ra:%d sp:%d", \
      p->pid, state, p->name, p->sz, p->trapframe->kernel_satp, p->context.ra, p->context.sp);
    printf("\n");
  }
}
```

本题目中要求对函数 procdump 执行的修改主要集中在其 printf 语句中，只要将进程结构体的各个成员弄懂，打印出来这些成员的值其实很简单，直接将输出附加到 printf 语句即可。在这里我们附加打印出进程的大小 p->sz、内核堆栈地址 p->trapframe->kernel_satp 以及返回地址寄存器的地址 p->context.ra 和栈顶指针寄存器的地址 p->context.sp，运行结果如下：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
pid:1 state:sleep  name:init size:12288 kernel:557055 ra:-2147475378 sp:-8464
pid:2 state:sleep  name:sh size:16384 kernel:557055 ra:-2147475378 sp:-16768

pid:1 state:sleep  name:init size:12288 kernel:557055 ra:-2147475378 sp:-8464
pid:2 state:sleep  name:sh size:16384 kernel:557055 ra:-2147475378 sp:-16768
```

b)：

1. swtch.S

该函数的 C 接口为 void swtch(struct context* old, struct context* new)。其中，根据结构体 context 的定义，我们知道 context 保存着被调用函数必须手动保存的寄存器的值，即这一结构体的实质即为返回地址寄存器 ra、栈顶指针寄存器 sp 和 s0-s11 这 12 个寄存器的状态。知道结构体 context 的细节对我们理解函数 swtch 是很重要的，因为 xv6 使用 context 切换来完成进程切换操作。

```
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

接下来，我们回到文件 swtch.S，可以看到，语句.golbl swtch 定义了函数 swtch 为外部函数，使得外部文件可以找到该函数并通过接口对它进行调用。在下面的第一张图，即 swtch 函数主体的上半部分中，包含了 14 条 sd 语句，由于 sd 语句的含义是将数据从寄存器存储到存储器中，例如语句 sd ra, 0(a0)，其意义是将参数 a0 的寄存器 ra 的地址加上偏移量 0，再将此地址的 8 位数据存入存储器，对照结构体 context 的成员来看，不难发现，这 14 条 sd 语句将传入的第一个参数 old 的所有成员存入了存储器。

```
.globl swtch
swtch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)
```

再来看 swtch 函数主体的下半部分，包含了 14 条 ld 语句，与上半部分的结构是对称的，再结合指令 ld 的意义，我们可以知道这 14 条 ld 语句的作用为将参

数 a1 的所有成员的数据载入 14 个寄存器 ra、sp、s1-s11 中。

```
        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret
```

最后，调用指令 ret，返回调用子程序 swtch 的位置继续执行。

2. scheduler(void)

该函数是单 CPU 处理器调度函数，每个 CPU 都在设置好自身以后调用该函数，函数 scheduler 无返回值，进入函数后循环进行下列操作：首先选取一个进程，随后将 CPU 切换到该进程运行，最后通过切回 scheduler 来让该进程传导控制。本质上，函数 scheduler 运行了一个普通的循环：找到一个进程来运

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
      }
      release(&p->lock);
    }
  }
}
```

行，运行到它停止为止，然后继续循环。

进入函数后，先建立一个进程结构体 p，再通过函数 mycpu 建立一个 cpu 结构体 c，将 c 的 proc 属性置空。接下来，进入 for 循环，先调用函数

intr_on，这是由于可能进程都在等待 I/O，从而找不到一个 RUNNABLE 的进程，如果调度器一直不允许中断，I/O 就永远无法到达了，这样就会产生死锁的问题。

接下来，进入内层循环 for(p = proc; p < &proc[NPROC]; p++){…}，令先前建立的进程结构体 p 为进程数组 proc 的首元素地址，令其与进程数组 proc 的最末尾元素的后一个单元的地址&proc[NPROC]比较，倘若相等则说明数组 proc 已遍历完成。

在该内层循环 for(p = proc; p < &proc[NPROC]; p++){…}中，先调用函数 acquire 获取当前进程 p 的锁，确保该进程执行的安全性。接下来，if 语句检查当前进程 p 的状态是否为可运行 RUNNABLE，如果该条件为真，则将当前进程 p 的状态改为运行中 RUNNING，将 CPU 的 proc 域赋值为当前进程 p，即让 CPU 执行当前进程 p，再调用函数 swtch 切换到上下文执行当前进程 p 的任务，而当函数 swtch 切回时，说明任务已执行完毕，重新令 cpu 结构体 c 的 proc 域置空。需要注意的是，在函数 swtch 切回时，当前进程 p 的状态 p->state 应该已在执行结束前更新。

最后，离开 if 语句，调用函数 release 释放当前进程 p 的锁 p->lock，该语句的作用是要释放进程表锁，这是为了防止 CPU 闲置时，由于当前 CPU 一直占有锁，其他 CPU 无法调度运行而导致的死锁，如果一个闲置的调度器一直持有锁，那么其他 CPU 就不可能执行上下文切换或任何和进程相关的系统调用了，也就更不可能将某个进程标记为 RUNNABLE，然后让闲置的调度器能够跳出循环了。进入下一循环。

3.sched(void)

该函数的作用是切换到调度器，其执行的必要条件是获取执行进程 p 的锁 p->lock，同时进程的状态 proc->state 已被改变。此外，该函数还具备保存中断的作用，中断是内核线程的属性，而不是 CPU 的属性，中断的检查应查看两个变量 proc->intena 和 proc->noff，变量 proc->intena 表示在中断发生前中断是否被允许，变量 proc->noff 表示中断的深度，即中断每多一层，变量 proc->noff 就会加一。需要指出的是，中断也可能发生于一些有锁但无进程之处。

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&p->lock))
    panic("sched p->lock");
  if(mycpu()->noff != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(intr_get())
    panic("sched interruptible");

  intena = mycpu()->intena;
  swtch(&p->context, &mycpu()->context);
  mycpu()->intena = intena;
}
```

进入函数体，函数先声明了整型变量 intena 和进程结构体指针 p，p 用函数 myproc 的返回值初始化，即表示当前正在执行的进程。接下来，函数进入一系列 if 判定语句，第一句为 if(!holding(&p->lock))，即判定 CPU 是否正持有进程 p 的锁 p->lock，如果 CPU 已持有锁，则 if 语句通过，如果 CPU 尚未持有锁，则打印关于 p->lock 的信息；第二个 if 语句为 if(mycpu()->noff != 1)，即判定当前 CPU 的中断的层级是否为多层嵌套中断，如果是则打印信息，否则 if 语句通过；第三个 if 语句为 if(p->state == RUNNING)，即判定当前进程的状态是否为正在运行，如果是则打印信息，否则 if 语句通过；第四个 if 语句为 if(intr_get())，即调用函数 intr_get 判定设备中断是否已经可以进行，如果返回结果为否则打印信息，否则 if 语句通过。

最后，在函数体开头声明的 intena 变量先临时保存由函数 mycpu 返回的当前 CPU 结构体的 intena 域，即 mycpu()->intena 的值，再调用函数 swtch 切换上下文执行进程，执行完毕后再将原先存储的 intena 值赋回给 mycpu()->intena。

4.yield(void)

该函数使得某一个调度周期内，进程持有的 CPU 被放弃，当前进程会让出 CPU 资源来给其他线程执行。

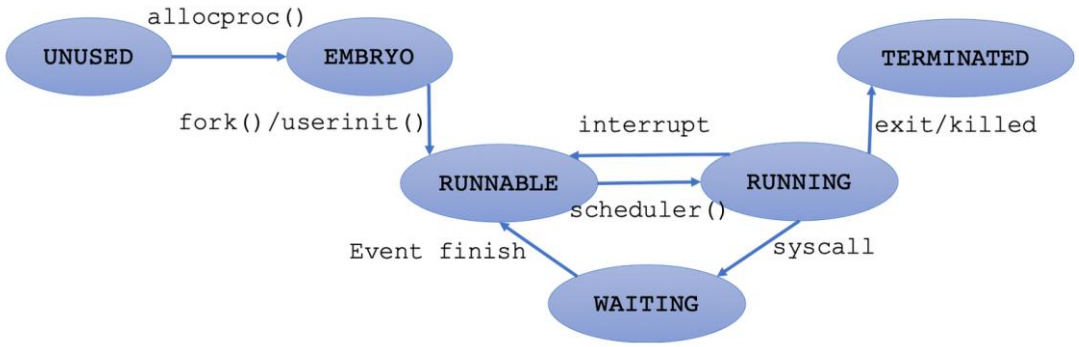
```
void
yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  sched();
  release(&p->lock);
}
```

其中，先定义进程结构体指针 p，调用函数 myproc 为它初始化，获取当前进程。接下来调用函数 acquire 获取进程 p 的锁 p->lock，再将进程 p 的状态 p->state 改为可执行，即 RUNNABLE，接下来调用上文提到的函数 sched 使进程 p 让出 CPU，切换到 CPU 原有的上下文 mycpu()->context 执行，最后释放进程 p 的锁 p->lock。

5.总结

总体来说，xv6 是支持多处理器多进程的操作系统，每个 CPU 都能够并行地运行不同的进程，同一个 CPU 也能通过不断地切换进程达到并发的效果，xv6 在时钟中断机制下，在很短的时间内完成进程切换操作使得宏观上感觉在同一个 CPU 下能出现“并行”的效果。

xv6 进程包括 UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE 这 6 种状态，即未使用态、初始态、等待态、就绪态、运行态、僵尸态。状态转换图如下：



xv6 永远不会直接从一个进程的上下文切换到另一个进程的上下文，这些都是通过一个中间的内核线程实现的：内核调度器线程。

当进程用完它的 CPU 时间片时，时钟中断会调用 yield 函数来让出 CPU 给新的进程，yield 调用 sched 函数，sched 调用 swtch 来切换到调度器线程。调度器线程从进程表中找到一个就绪进程，并初始化进程运行环境，然后调用 swtch 切换到新的进程。而调度器线程仅仅是简单地进行轮转调度，一旦找到就绪线程便切换到新的线程。

c)：

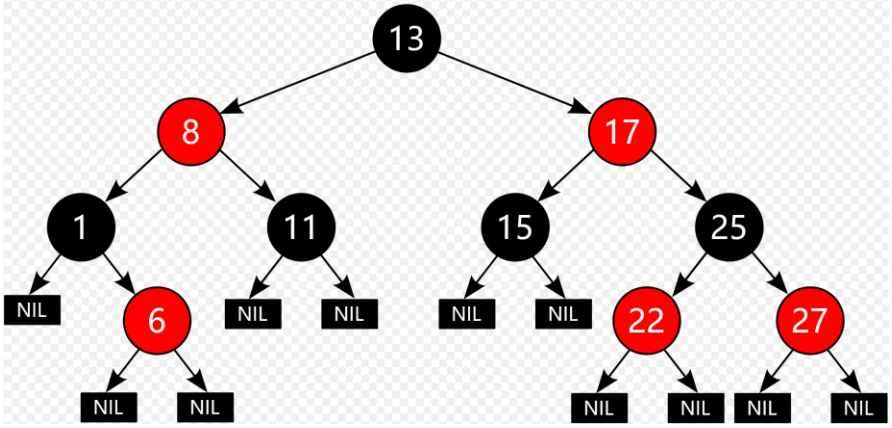
1.Linux 调度算法：

Linux 主要采取的调度算法为 CFS 算法，全称为 Completely Fair Scheduler 算法，即所谓的“完全公平调度算法”。

CFS 背后的主要想法是维护为任务提供处理器时间方面的平衡（公平性）。这意味着应给进程分配相当数量的处理器。当分给某个任务的时间失去平衡时（意味着一个或多个任务相对于其它任务而言未被给予相当数量的时间）。应给失去平衡的任务分配时间，让其运行。要实现平衡，CFS 在叫做虚拟执行时地方维持提供给某个任务的时间量。任务的虚拟执行时越小。意味着任务被同意访问 server 的时间越短，其对处理器的需求越高。CFS 还包括睡眠公平概念以便确保那些眼下没有执行的 任务（比如。等待 I/O）在其终于须要时获得相当份额的处理器。

CFS 维护了一个以时间为顺序的红黑树，红黑树是一个树，具有一些实用的属性。首先，它是自平衡的，这意味着树上没有路径比任意其它路径长两倍以上。

第二，树上的执行按 O(logn)时间复杂度发生，这意味着我们能够高速高效地插入或删除任务。



任务存储在以时间为顺序的红黑树中，对处理器需求最多的任务（最低虚拟执行时）存储在树的左侧，处理器需求最少的任务（最高虚拟执行时）存储在树的右侧。为了公平，调度器然后选取红黑树最左端的节点调度为下一个，以便保持公平性。

任务通过将其执行时间加入到虚拟执行时， 说明其占用 CPU 的时间，然后假设可执行。再插回到树中。这样，树左侧的任务就被给予时间执行了，树的内容从右侧迁移到左侧以保持公平。因此，每一个可执行的任务都会追赶其它任务以维持整个可执行任务集合的执行平衡。

2. xv6 调度改进

默认情况下，xv6 使用一个简单的时间片轮转 RR 算法来调度进程。该算法中，将一个较小时间单元定义为时间量或时间片，就绪队列为循环队列。CPU 调度程序循环整个就绪队列，为每个进程分配不超过一个时间片的 CPU。

为了实现 RR 调度，我们将就绪队列视为进程的 FIFO 队列。新进程添加到就绪队列的尾部。CPU 调度程序从就绪队列中选择第一个进程，将定时器设置在一个时间片后中断，最后分派这个进程。

接下来，有两种情况可能发生。进程可能只需少于时间片的 CPU 执行。对于这种情况，进程本身会自动释放 CPU。调度程序接着处理就绪队列的下一个进程。否则，如果当前运行进程的 CPU 执行大于一个时间片，那么定时器会中断，进而中断操作系统。然后，进行上下文切换，再将进程加到就绪队列的尾部，接着 CPU 调度程序会选择就绪队列内的下一个进程。采用 RR 策略的平均等待时间通常较长。

对 xv6 的 RR 算法的改进可以参考 CFS 调度算法，也建立完全公平运行队列、调度实体、用于储存的红黑树的数据结构，并对进程附加权重、虚拟执行时属性，通过虚拟执行时属性反映进程当前已经运行时间的多少，使各个进程公平地竞争 CPU，让优先级大的进程拥有更多的实际运行时间。