操作系统大作业 1

李宏立 19351064

一、Client-Server 问题

1.1 实验思路

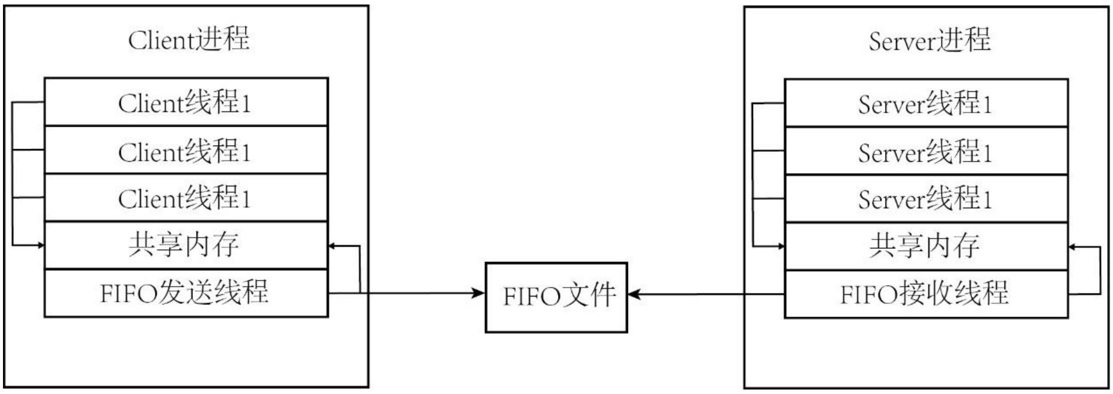
1.1.1 进程间通信

实验共有两个进程：Client 和 Server 进程。根据实验要求，两个进程之间采用管道通信，本实验采用命名管道的方法，比匿名管道的使用更加灵活，适用于任意两个不相关的进程。我们设定 Client 进程对命名管道的 FIFO 文件进行写的功能，而 Server 进程对命名管道的 FIFO 文件进行读的功能。

1.1.2 线程间通信

实验共有六个线程，Client 和 Server 进程分别派生出三个读写线程。根据实验要求，派生的线程和相关的进程之间采用共享内存的方式进行通信。线程之间是默认共享内存的，所以只需要在各个线程读写共享内存时加上互斥锁，就可以解决线程之间的同步问题。

最终可以得到所有线程、进程之间的通信逻辑。



1.1.3 控制速率

根据实验要求，需要控制线程运行的时间间隔，并使得呈负指数分布。因此需要使用数学函数库来计算负指数，负指数分布的公式如下：

$$f(x) = \begin{cases} \lambda e^{-\lambda x}, x \geq 0 \\ 0, x < 0 \end{cases}$$

1.2 关键代码

1.2.1 Client 进程

Client 进程中生产者线程执行的任务如下。

```
void *client_do(void *arg)
{
    char Buffer_write[BUFFER_SIZE];
    while(1){
```

```
5.         double gap = produce_time(lambda);
6.         usleep(1000000*gap);    //产生负指数分布的间隔
7.         sem_wait(&empty);
8.         pthread_mutex_lock(&mutex);     //加上互斥锁
9.         for(int j=0;j<BUFFER_SIZE;j++){
10.            Buffer[in][j] = 65 + rand()%26;     //产生 10 个随机大写字母作为一个 slot
11.            Buffer_write[j] = Buffer[in][j];
12.        }
13.        in=(in+1)%n;    //缓存区入指针后移
14.        printf("producer %d write %s into buffer\n", (int)arg, Buffer_write);
15.        pthread_mutex_unlock(&mutex);   //解开互斥锁
16.        sem_post(&full);
17.    }
18.    pthread_exit(0);
19. }
```

第 3 行是建立一个全局变量数组作为共享内存，结合两个出入的逻辑指针，成为循环队列。第 4 行到第 12 行生产者线程每次休眠一定时间，然后生成一串随机信息，通过互斥锁和同步变量控制同步写入共享内存。第 13 行队列的入指针后移一位。

Client 进程中管道线程执行的任务如下。

```
void *thread_fifo_write(void *arg)
{
    char write_buf[BUFFER_SIZE];
    int fd;
    //以只写的方式打开管道 fifo 文件
    fd = open("./myfifo",O_WRONLY);
    if(fd == -1)
    {
        printf("write fifo open fail....\n");
        exit(-1);
        return;
    }
    while(1)
    {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);     //加上互斥锁
        for(int j=0;j<BUFFER_SIZE;j++){
            write_buf[j] = Buffer[out][j];  //读取缓冲区的一个 slot
        }
        out = (out+1)%n;    //缓冲区出指针后移
        pthread_mutex_unlock(&mutex);   //解开互斥锁
```

```c
22.         sem_post(&empty);
23.         printf("fifo %d read from Buffer,the data is %s\n",(int)arg, write_buf);
24.         int ret = write(fd, write_buf, BUFFER_SIZE);     //将 slot 写入管道
25.         if(ret <= 0) {
26.             perror("write()");
27.             printf("Pipe blocked, try again ...\n");
28.             sleep(1);
29.         }
30.     }
31.     close(fd);       //关闭管道
32.     pthread_exit(0);//fifo 线程正常退出
33. }
```

主要思想是不停读取共享内存中的信息，然后立即写入到 FIFO 文件中。第 4 行到第 12 行是打开 FIFO 文件的函数，以只写的方式打开，如果打开失败，打印报错信息。第 15 行到第 22 行是从共享内存中读取 slot 的函数。第 24 行到第 29 行是将 slot 写入 FIFO 文件的函数。

1.2.2 Server 进程

Server 进程中管道线程执行的任务如下。

```c
void *thread_fifo_read(void *arg)
{
    char read_buf[BUFFER_SIZE];
    int fd;
    fd = open("./myfifo", O_RDONLY );
    if(fd == -1)
    {
        printf("read fifo open fail...\n");
        exit(-1);
        return;
    }
    while(1)
    {
        memset(read_buf, 0, BUFFER_SIZE);
        int ret = read(fd, read_buf, BUFFER_SIZE); //从管道中读 slot
        if(ret <= 0) {
            break;
        }
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);         //加上互斥锁
        for(int j=0;j<BUFFER_SIZE;j++){       //将一个 slot 写入缓冲区
            Buffer[in][j] = read_buf[j];
        }
        in=(in+1)%n;         //缓存区入指针后移
```

```
25.         printf("fifo %d write %s into buffer\n",(int)arg, read_buf);
26.         pthread_mutex_unlock(&mutex);       //解开互斥锁
27.         sem_post(&full);
28.     }
29.     close(fd);
30.     pthread_exit(0);
31. }
```

主要思想是读取 FIFO 文件中的信息，然后立即写入到共享内存中。第 4 行到第 11 行是打开 FIFO 文件的函数，以只读的方式打开，如果打开失败，打印报错信息。第 15 行是从 FIFO 文件中读取 slot。第 19 行到第 27 行是将 slot 写入共享内存的函数。

Server 进程中消费者线程执行的任务如下。

```
void *server_do(void *arg){
    char Buffer_read[BUFFER_SIZE];
    while(1){
        double gap = produce_time(lambda);
        usleep(1000000*gap);     //产生负指数分布的间隔
        sem_wait(&full);
        pthread_mutex_lock(&mutex);      //加上互斥锁
        for(int j=0;j<BUFFER_SIZE;j++){
            Buffer_read[j] = Buffer[out][j];
        }
        out=(out+1)%n;       //缓存区出指针后移
        pthread_mutex_unlock(&mutex);       //解开互斥锁
        sem_post(&empty);
        printf("consumer %d read from Buffer,the data is %s\n", (int)arg, Buffer_read);
    }
}
```

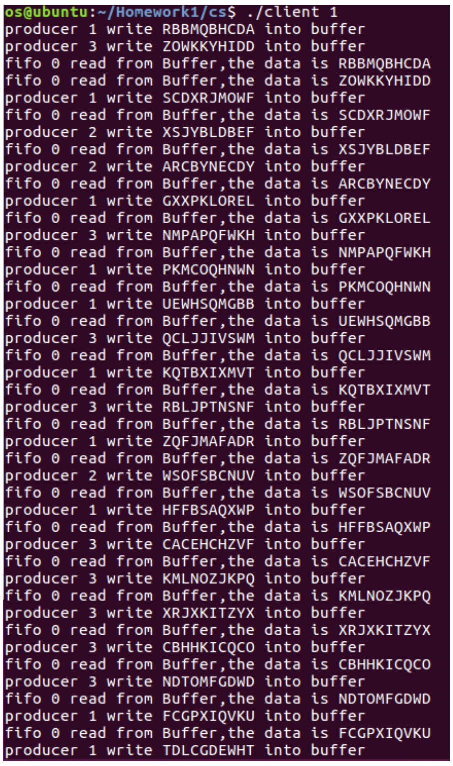
1.2.3 负指数时间间隔分布

主要思想是利用一个分布在 0-1 之内的随机数，根据指数分布的公式反推出一个指数间隔，相邻时间间隔是随机分布的，但注意到总体上是呈一定指数分布的。

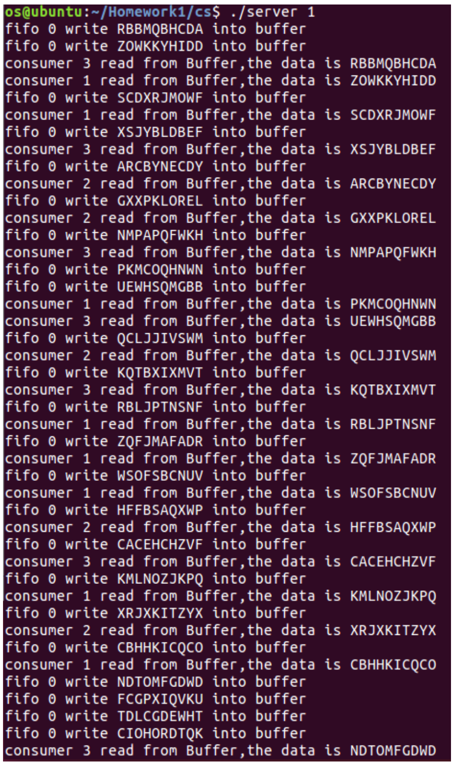
```
double produce_time(double lambda_){
    double x;
    do{
        x = ((double)rand() / RAND_MAX);
    }
    while((x==0)||(x==1));
    return (-1/lambda_ * log(x));
}
```

1.3 实验结果

1.3.1 $\lambda c = 1, \lambda s = 1$



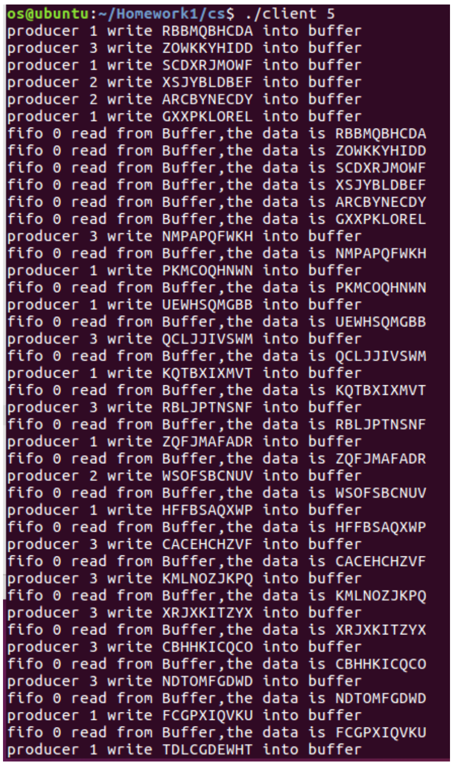
```
os@ubuntu:~/Homework1/cs$ ./client 1
producer 1 write RBBMQBHCDA into buffer
producer 3 write ZOWKKYHIDD into buffer
fifo 0 read from Buffer,the data is RBBMQBHCDA
fifo 0 read from Buffer,the data is ZOWKKYHIDD
producer 1 write SCDXRJMOWF into buffer
fifo 0 read from Buffer,the data is SCDXRJMOWF
producer 2 write XSJYBLDBEF into buffer
fifo 0 read from Buffer,the data is XSJYBLDBEF
producer 2 write ARCBYNECDY into buffer
fifo 0 read from Buffer,the data is ARCBYNECDY
producer 1 write GXXPKLOREL into buffer
fifo 0 read from Buffer,the data is GXXPKLOREL
producer 3 write NMPAPQFWKH into buffer
fifo 0 read from Buffer,the data is NMPAPQFWKH
producer 1 write PKMCOQHNWN into buffer
fifo 0 read from Buffer,the data is PKMCOQHNWN
producer 1 write UEWHSQMGBB into buffer
fifo 0 read from Buffer,the data is UEWHSQMGBB
producer 3 write QCLJJIVSWM into buffer
fifo 0 read from Buffer,the data is QCLJJIVSWM
producer 1 write KQTBXIXMVT into buffer
fifo 0 read from Buffer,the data is KQTBXIXMVT
producer 3 write RBLJPTNSNF into buffer
fifo 0 read from Buffer,the data is RBLJPTNSNF
producer 1 write ZQFJMAFADR into buffer
fifo 0 read from Buffer,the data is ZQFJMAFADR
producer 2 write WSOFSBCNUV into buffer
fifo 0 read from Buffer,the data is WSOFSBCNUV
producer 1 write HFFBSAQXWP into buffer
fifo 0 read from Buffer,the data is HFFBSAQXWP
producer 3 write CACEHCHZVF into buffer
fifo 0 read from Buffer,the data is CACEHCHZVF
producer 3 write KMLNOZJKPQ into buffer
fifo 0 read from Buffer,the data is KMLNOZJKPQ
producer 3 write XRJXKITZYX into buffer
fifo 0 read from Buffer,the data is XRJXKITZYX
producer 3 write CBHHKICQCO into buffer
fifo 0 read from Buffer,the data is CBHHKICQCO
producer 3 write NDTOMFGDWD into buffer
fifo 0 read from Buffer,the data is NDTOMFGDWD
producer 1 write FCGPXIQVKU into buffer
fifo 0 read from Buffer,the data is FCGPXIQVKU
producer 1 write TDLCGDEWHT into buffer
```



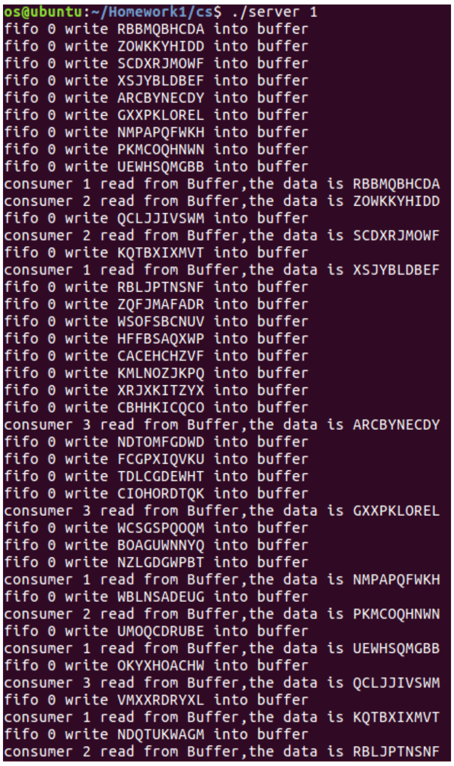
```
os@ubuntu:~/Homework1/cs$ ./server 1
fifo 0 write RBBMQBHCDA into buffer
fifo 0 write ZOWKKYHIDD into buffer
consumer 3 read from Buffer,the data is RBBMQBHCDA
consumer 1 read from Buffer,the data is ZOWKKYHIDD
fifo 0 write SCDXRJMOWF into buffer
consumer 1 read from Buffer,the data is SCDXRJMOWF
fifo 0 write XSJYBLDBEF into buffer
consumer 3 read from Buffer,the data is XSJYBLDBEF
fifo 0 write ARCBYNECDY into buffer
consumer 2 read from Buffer,the data is ARCBYNECDY
fifo 0 write GXXPKLOREL into buffer
consumer 2 read from Buffer,the data is GXXPKLOREL
fifo 0 write NMPAPQFWKH into buffer
consumer 3 read from Buffer,the data is NMPAPQFWKH
fifo 0 write PKMCOQHNWN into buffer
fifo 0 write UEWHSQMGBB into buffer
consumer 1 read from Buffer,the data is PKMCOQHNWN
consumer 3 read from Buffer,the data is UEWHSQMGBB
fifo 0 write QCLJJIVSWM into buffer
consumer 2 read from Buffer,the data is QCLJJIVSWM
fifo 0 write KQTBXIXMVT into buffer
consumer 3 read from Buffer,the data is KQTBXIXMVT
fifo 0 write RBLJPTNSNF into buffer
consumer 1 read from Buffer,the data is RBLJPTNSNF
fifo 0 write ZQFJMAFADR into buffer
consumer 1 read from Buffer,the data is ZQFJMAFADR
fifo 0 write WSOFSBCNUV into buffer
consumer 1 read from Buffer,the data is WSOFSBCNUV
fifo 0 write HFFBSAQXWP into buffer
consumer 2 read from Buffer,the data is HFFBSAQXWP
fifo 0 write CACEHCHZVF into buffer
consumer 3 read from Buffer,the data is CACEHCHZVF
fifo 0 write KMLNOZJKPQ into buffer
consumer 1 read from Buffer,the data is KMLNOZJKPQ
fifo 0 write XRJXKITZYX into buffer
consumer 2 read from Buffer,the data is XRJXKITZYX
fifo 0 write CBHHKICQCO into buffer
consumer 1 read from Buffer,the data is CBHHKICQCO
fifo 0 write NDTOMFGDWD into buffer
fifo 0 write FCGPXIQVKU into buffer
fifo 0 write TDLCGDEWHT into buffer
fifo 0 write CIOHORDTQK into buffer
consumer 3 read from Buffer,the data is NDTOMFGDWD
```

可以看到当两个进程的指数分布系数都为 1 时，两边进程的读写速率接近。client 进程中的三个生产者线程和管道线程工作频率十分稳定，server 进程中由于一些随机的指数时间，导致缓冲区偶尔发生堆积，但预计一般情况下缓冲区不会发生饱和。

1.3.2 $\lambda c = 5, \lambda s = 1$



```
os@ubuntu:~/Homework1/cs$ ./client 5
producer 1 write RBBMQBHCDA into buffer
producer 3 write ZOWKKYHIDD into buffer
producer 1 write SCDXRJMOWF into buffer
producer 2 write XSJYBLDBEF into buffer
producer 2 write ARCBYNECDY into buffer
producer 1 write GXXPKLOREL into buffer
fifo 0 read from Buffer,the data is RBBMQBHCDA
fifo 0 read from Buffer,the data is ZOWKKYHIDD
fifo 0 read from Buffer,the data is SCDXRJMOWF
fifo 0 read from Buffer,the data is XSJYBLDBEF
fifo 0 read from Buffer,the data is ARCBYNECDY
fifo 0 read from Buffer,the data is GXXPKLOREL
producer 3 write NMPAPQFWKH into buffer
fifo 0 read from Buffer,the data is NMPAPQFWKH
producer 1 write PKMCOQHNWN into buffer
fifo 0 read from Buffer,the data is PKMCOQHNWN
producer 1 write UEWHSQMGBB into buffer
fifo 0 read from Buffer,the data is UEWHSQMGBB
producer 3 write QCLJJIVSWM into buffer
fifo 0 read from Buffer,the data is QCLJJIVSWM
producer 1 write KQTBXIXMVT into buffer
fifo 0 read from Buffer,the data is KQTBXIXMVT
producer 3 write RBLJPTNSNF into buffer
fifo 0 read from Buffer,the data is RBLJPTNSNF
producer 1 write ZQFJMAFADR into buffer
fifo 0 read from Buffer,the data is ZQFJMAFADR
producer 2 write WSOFSBCNUV into buffer
fifo 0 read from Buffer,the data is WSOFSBCNUV
producer 1 write HFFBSAQXWP into buffer
fifo 0 read from Buffer,the data is HFFBSAQXWP
producer 3 write CACEHCHZVF into buffer
fifo 0 read from Buffer,the data is CACEHCHZVF
producer 3 write KMLNOZJKPQ into buffer
fifo 0 read from Buffer,the data is KMLNOZJKPQ
producer 3 write XRJXKITZYX into buffer
fifo 0 read from Buffer,the data is XRJXKITZYX
producer 3 write CBHHKICQCO into buffer
fifo 0 read from Buffer,the data is CBHHKICQCO
producer 3 write NDTOMFGDWD into buffer
fifo 0 read from Buffer,the data is NDTOMFGDWD
producer 1 write FCGPXIQVKU into buffer
fifo 0 read from Buffer,the data is FCGPXIQVKU
producer 1 write TDLCGDEWHT into buffer
```



```
os@ubuntu:~/Homework1/cs$ ./server 1
fifo 0 write RBBMQBHCDA into buffer
fifo 0 write ZOWKKYHIDD into buffer
fifo 0 write SCDXRJMOWF into buffer
fifo 0 write XSJYBLDBEF into buffer
fifo 0 write ARCBYNECDY into buffer
fifo 0 write GXXPKLOREL into buffer
fifo 0 write NMPAPQFWKH into buffer
fifo 0 write PKMCOQHNWN into buffer
fifo 0 write UEWHSQMGBB into buffer
consumer 1 read from Buffer,the data is RBBMQBHCDA
consumer 2 read from Buffer,the data is ZOWKKYHIDD
fifo 0 write QCLJJIVSWM into buffer
consumer 2 read from Buffer,the data is SCDXRJMOWF
fifo 0 write KQTBXIXMVT into buffer
consumer 1 read from Buffer,the data is XSJYBLDBEF
fifo 0 write RBLJPTNSNF into buffer
fifo 0 write ZQFJMAFADR into buffer
fifo 0 write WSOFSBCNUV into buffer
fifo 0 write HFFBSAQXWP into buffer
fifo 0 write CACEHCHZVF into buffer
fifo 0 write KMLNOZJKPQ into buffer
fifo 0 write XRJXKITZYX into buffer
fifo 0 write CBHHKICQCO into buffer
consumer 3 read from Buffer,the data is ARCBYNECDY
fifo 0 write NDTOMFGDWD into buffer
fifo 0 write FCGPXIQVKU into buffer
fifo 0 write TDLCGDEWHT into buffer
fifo 0 write CIOHORDTQK into buffer
consumer 3 read from Buffer,the data is GXXPKLOREL
fifo 0 write WCSGSPQOQM into buffer
fifo 0 write BOAGUWNNYQ into buffer
fifo 0 write NZLGDGWPBT into buffer
consumer 1 read from Buffer,the data is NMPAPQFWKH
fifo 0 write WBLNSADEUG into buffer
consumer 2 read from Buffer,the data is PKMCOQHNWN
fifo 0 write UMOQCDRUBE into buffer
consumer 1 read from Buffer,the data is UEWHSQMGBB
fifo 0 write OKYXHOACHW into buffer
consumer 3 read from Buffer,the data is QCLJJIVSWM
fifo 0 write VMXXRDRYXL into buffer
consumer 1 read from Buffer,the data is KQTBXIXMVT
fifo 0 write NDQTUKWAGM into buffer
consumer 2 read from Buffer,the data is RBLJPTNSNF
```

当 client 进程时间系数为 5 时，线程工作速度加快。server 进程时间系数保持为 1 时，消费者线程明显跟不上生产者线程的工作速度。预计在一段时间之后，server 进程中的缓冲区将会饱和阻塞。

1.3.3 $\lambda c = 1, \lambda s = 5$

```
os@ubuntu:~/Homework1/cs$ ./client 1
producer 3 write RBBMQBHCDA into buffer
producer 1 write ZOWKKYHIDD into buffer
fifo 0 read from Buffer,the data is RBBMQBHCDA
fifo 0 read from Buffer,the data is ZOWKKYHIDD
producer 3 write SCDXRJMOWF into buffer
fifo 0 read from Buffer,the data is SCDXRJMOWF
producer 2 write XSJYBLDBEF into buffer
fifo 0 read from Buffer,the data is XSJYBLDBEF
producer 2 write ARCBYNECDY into buffer
fifo 0 read from Buffer,the data is ARCBYNECDY
producer 3 write GXXPKLOREL into buffer
fifo 0 read from Buffer,the data is GXXPKLOREL
producer 1 write NMPAPQFWKH into buffer
fifo 0 read from Buffer,the data is NMPAPQFWKH
producer 3 write PKMCOQHNWN into buffer
fifo 0 read from Buffer,the data is PKMCOQHNWN
producer 3 write UEWHSQMGBB into buffer
fifo 0 read from Buffer,the data is UEWHSQMGBB
producer 1 write QCLJJIVSWM into buffer
fifo 0 read from Buffer,the data is QCLJJIVSWM
producer 3 write KQTBXIXMVT into buffer
fifo 0 read from Buffer,the data is KQTBXIXMVT
producer 1 write RBLJPTNSNF into buffer
fifo 0 read from Buffer,the data is RBLJPTNSNF
producer 3 write ZQFJMAFADR into buffer
fifo 0 read from Buffer,the data is ZQFJMAFADR
producer 2 write WSOFSBCNUV into buffer
fifo 0 read from Buffer,the data is WSOFSBCNUV
producer 3 write HFFBSAQXWP into buffer
fifo 0 read from Buffer,the data is HFFBSAQXWP
producer 1 write CACEHCHZVF into buffer
fifo 0 read from Buffer,the data is CACEHCHZVF
producer 1 write KMLNOZJKPQ into buffer
fifo 0 read from Buffer,the data is KMLNOZJKPQ
producer 1 write XRJXKITZYX into buffer
fifo 0 read from Buffer,the data is XRJXKITZYX
producer 1 write CBHHKICQCO into buffer
fifo 0 read from Buffer,the data is CBHHKICQCO
producer 1 write NDTOMFGDWD into buffer
fifo 0 read from Buffer,the data is NDTOMFGDWD
producer 3 write FCGPXIQVKU into buffer
fifo 0 read from Buffer,the data is FCGPXIQVKU
producer 3 write TDLCGDEWHT into buffer
```

```
os@ubuntu:~/Homework1/cs$ ./server 5
fifo 0 write RBBMQBHCDA into buffer
fifo 0 write ZOWKKYHIDD into buffer
consumer 1 read from Buffer,the data is RBBMQBHCDA
consumer 3 read from Buffer,the data is ZOWKKYHIDD
fifo 0 write SCDXRJMOWF into buffer
consumer 3 read from Buffer,the data is SCDXRJMOWF
fifo 0 write XSJYBLDBEF into buffer
consumer 1 read from Buffer,the data is XSJYBLDBEF
fifo 0 write ARCBYNECDY into buffer
consumer 2 read from Buffer,the data is ARCBYNECDY
fifo 0 write GXXPKLOREL into buffer
consumer 1 read from Buffer,the data is GXXPKLOREL
fifo 0 write NMPAPQFWKH into buffer
consumer 3 read from Buffer,the data is NMPAPQFWKH
fifo 0 write PKMCOQHNWN into buffer
consumer 2 read from Buffer,the data is PKMCOQHNWN
fifo 0 write UEWHSQMGBB into buffer
consumer 1 read from Buffer,the data is UEWHSQMGBB
fifo 0 write QCLJJIVSWM into buffer
consumer 3 read from Buffer,the data is QCLJJIVSWM
fifo 0 write KQTBXIXMVT into buffer
consumer 1 read from Buffer,the data is KQTBXIXMVT
fifo 0 write RBLJPTNSNF into buffer
consumer 2 read from Buffer,the data is RBLJPTNSNF
fifo 0 write ZQFJMAFADR into buffer
consumer 3 read from Buffer,the data is ZQFJMAFADR
fifo 0 write WSOFSBCNUV into buffer
consumer 1 read from Buffer,the data is WSOFSBCNUV
fifo 0 write HFFBSAQXWP into buffer
consumer 2 read from Buffer,the data is HFFBSAQXWP
fifo 0 write CACEHCHZVF into buffer
consumer 3 read from Buffer,the data is CACEHCHZVF
fifo 0 write KMLNOZJKPQ into buffer
consumer 1 read from Buffer,the data is KMLNOZJKPQ
fifo 0 write XRJXKITZYX into buffer
consumer 2 read from Buffer,the data is XRJXKITZYX
fifo 0 write CBHHKICQCO into buffer
consumer 1 read from Buffer,the data is CBHHKICQCO
fifo 0 write NDTOMFGDWD into buffer
consumer 3 read from Buffer,the data is NDTOMFGDWD
fifo 0 write FCGPXIQVKU into buffer
consumer 1 read from Buffer,the data is FCGPXIQVKU
fifo 0 write TDLCGDEWHT into buffer
```

当 client 进程时间系数为 1 时，线程工作速度稳定。server 进程的时间系数增大到 5 时，消费者工作速度加快，可以发现一旦生产者将消息发送到 server 进程时，立即会被消费，缓冲区常常处于饥饿的状态。

二、哲学家就餐问题

2.1 实验思路

我们将每个哲学家看作是一个执行的线程，而每个筷子是一个临界资源。每个线程需要两个临界资源才能正常运行。因此，我们需要考虑的问题是线程的死锁现象，而当某个线程长时间没有获得资源时，称作饥饿现象，本实验暂时没有考虑长期饥饿的问题。

本实验采用的思路是：只有一个哲学家的两根筷子都可用时，他才拿起他们。也就是说，当一个哲学家的左右相邻哲学家都没有在就餐时，他就可以就餐。

根据要求，本实验采用互斥锁和条件变量来实现以上思路。互斥锁和条件变量结合使用，允许线程以无竞争的方式等待特定的条件发生。

2.2 关键代码

关键代码是哲学家检查筷子是否可用和拿起、放下筷子的三个动作，如下所示。

```
//哲学家检查左右两边的筷子是否都可用
```

```
2. void test(int i){
3.     //判断条件：哲学家饥饿，且左右哲学家没有在用餐
4.     if(state[i]==HUNGRY && state[left(i)] != EATING && state[right(i)]!=EATING){
5.         pthread_mutex_lock(&mutex[i]);  //锁定资源
6.         state[i] = EATING;  //状态切换为用餐
7.         pthread_cond_signal(&phil_self[i]);  //唤醒对应的哲学家线程，并重新获得互斥锁
8.         pthread_mutex_unlock(&mutex[i]);
9.     }
10. }
11.
12. //哲学家拿起筷子
13. void pickup_forks(int i){
14.     state[i] = HUNGRY;
15.     test(i);
16.     pthread_mutex_lock(&mutex[i]);
17.     while(state[i] != EATING){
18.         //条件不成立的线程会进入阻塞，并释放互斥锁
19.         pthread_cond_wait(&phil_self[i], &mutex[i]);
20.     }
21.     pthread_mutex_unlock(&mutex[i]);
22. }
23.
24. //哲学家放回筷子
25. void return_forks(int i){
26.     state[i] = THINKING;
27.     test(left(i));
28.     test(right(i));
29. }
```

第 2 行到第 10 行是哲学家检查左右哲学家状态的函数，如果左右哲学家都不是用餐状态，自身是饥饿状态，那么就可以准备用餐了，首先获得互斥锁，然后切换状态，如果对应线程在之前进入阻塞，则利用条件变量唤醒。第 13 行到第 22 行，是哲学家拿起筷子的函数，哲学家的状态变为饥饿，然后检查左右哲学家状态，如果失败的话，哲学家的状态不能切换为用餐，那么对应线程就会被挂起。第 25 行到第 29 行是哲学家放下筷子的函数，哲学家把状态切换为思考，然后对左右的哲学家使用 test 函数，目的是为了唤醒相关条件成立的哲学家线程。

2.3 实验结果

我们设置每个哲学家就餐和思考的时间都是 1 秒钟，最终得到每个哲学家的状态输出如下图。

```
os@ubuntu:~/Homework1/cs$ ./dph
Philosopher 0 is thinking.
Philosopher 3 is thinking.
Philosopher 2 is thinking.
Philosopher 1 is thinking.
Philosopher 4 is thinking.
Philosopher 3 is eating.
Philosopher 0 is eating.
Philosopher 0 is thinking.
Philosopher 4 is eating.
Philosopher 3 is thinking.
Philosopher 2 is eating.
Philosopher 3 is eating.
Philosopher 2 is thinking.
Philosopher 0 is eating.
Philosopher 4 is thinking.
Philosopher 3 is thinking.
Philosopher 2 is eating.
Philosopher 0 is thinking.
Philosopher 4 is eating.
Philosopher 4 is thinking.
Philosopher 3 is eating.
Philosopher 2 is thinking.
Philosopher 1 is eating.
Philosopher 4 is eating.
Philosopher 3 is thinking.
Philosopher 2 is eating.
Philosopher 1 is thinking.
Philosopher 0 is eating.
Philosopher 3 is eating.
Philosopher 4 is thinking.
Philosopher 2 is thinking.
Philosopher 0 is thinking.
Philosopher 4 is eating.
Philosopher 3 is thinking.
Philosopher 1 is eating.
Philosopher 3 is eating.
Philosopher 0 is eating.
```

可以看到 0-4 号哲学家都能够成功用餐，之后进入思考，没有死锁和长期饥饿的问题。

三、MIT 6.S081 课程实验

3.1 Lab: Xv6 and Unix utilities

3.1.1 阅读内容

Lab1 的主要参考资料是 xv6-book 的第一章。我们首先需要理解的是 xv6 操作系统的运行方式和 Unix 相似。用户进程首先通过系统调用陷入内核，完成特定的服务之后，再从内核返回。xv6 提供的系统调用不在此列举。第一章中主要介绍了 xv6 系统的几个基础部分，其中包括：进程和内存、I/O 和文件描述、管道以及文件系统。我们在之后的实验部分分析相关内容。

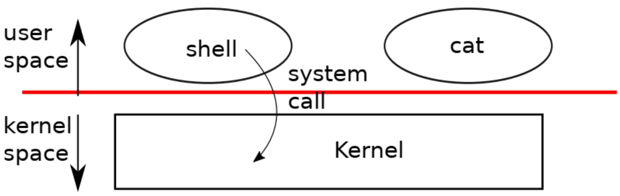


Figure 1.1: A kernel and two user processes.

3.1.2 sleep

根据实验要求，需要构造一个 sleep 函数，能够使程序中断指定的时间。除此之外，我们需要仿照其他的用户程序检查参数，通过系统调用提供的 sleep 执行。实验的主要目的是让我们熟悉 xv6 的系统调用以及代码风格。最终 sleep 实现的代码如下。

```c
1. #include "kernel/types.h"
2. #include "user/user.h"
3. int main(int argc, char **argv)
4. {
5.     // Return ASAP while there is no parameter.
6.     if (argc != 2) {
7.         fprintf(2,"sleep: wants 1 parameters, get %d parameter(s).\n",argc-1);
8.         exit(0);
9.     }
10.    int second;
11.    second = atoi(argv[1]);
12.    // If second == 0, return ASAP.
13.    fprintf(1, "Sleep %d\n", second);
14.    if (second <= 0)
15.        exit(1);
16.    sleep(second);
17.    exit(0);
18. }
```

第 3 行-第 9 行，程序首先检查参数个数是否为 2，如果不是，则输出提示信息。第 10 行-第 16 通过 atoi 将参数转化为 int 类型，并调用 sleep 执行，如果参数值为负，则异常退出。需要注意的是，程序中断和退出都有特定的系统调用：sleep 和 exit。程序运行结果如下图所示。

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ Sleep 10
exec Sleep failed
$ sleep 10
Sleep 10
```

```
os@ubuntu:~/xv6-riscv-fall20-util$ ./grade-lab-util sleep
fatal: Not a git repository (or any of the parent directories): .git
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == fatal: Not a git repository (or any of the parent
 directories): .git
sleep, no arguments: OK (2.1s)
== Test sleep, returns == sleep, returns: OK (0.6s)
== Test sleep, makes syscall == sleep, makes syscall: OK (0.7s)
```

3.1.3 pingpong

根据实验要求，需要构造两个管道，实现父子进程的全双工通信。Ping 和 pong 分别是两个管道传递的信息。

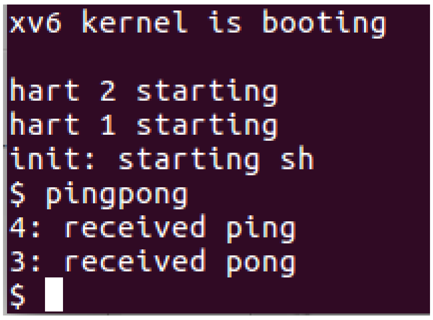
我们首先需要了解 xv6 的父子进程、文件描述符和管道特性。首先 fork 系统调用是用来创建一个新的子进程，对于父进程返回子进程的 pid，对于子进程返回 0。通然后是文件描述符，定义与 Unix 相同，有

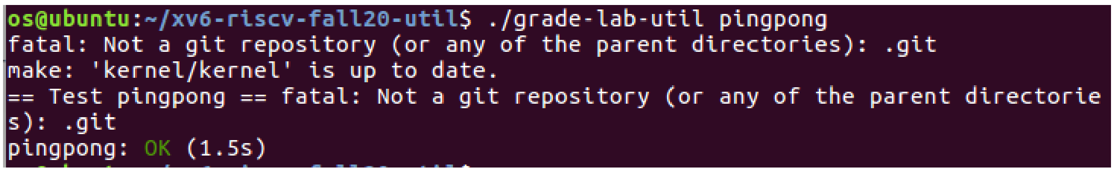
O_RDONLY、O_WRONLY、O_RDWR、O_CREATE 和 O_TRUNC 这些标识。最后通过系统调用 pipe(int p[])，我们可以创建一个匿名管道，在 p[0]放入读端的文件描述符，在 p[1]放入写端的文件描述符。最终 pingpong 代码实现如下。

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char *argv[])
{
    char c;
    int n;

    int parent_to_child[2];
    int child_to_parent[2];

    //定义两个管道
    pipe(parent_to_child);
    pipe(child_to_parent);

    if (fork() == 0)     //如果是子进程
    {
        close(parent_to_child[1]);  //关闭父进程的写端
        close(child_to_parent[0]);  //关闭子进程的读端
        //从管道中读取消息到 c
        n = read(parent_to_child[0], &c, 1);
        if (n != 1)
        {
            fprintf(2, "child read error\n");
            exit(1);
        }
        //打印子进程 pid 和提示消息
        printf("%d: received ping\n", getpid());
        write(child_to_parent[1], &c, 1);
        //程序结束前要关闭管道，避免阻塞
        close(parent_to_child[0]);  //关闭父进程的读端
        close(child_to_parent[1]);  //关闭子进程的写端
        exit(0);
    }
    else     //如果是父进程
    {
        close(parent_to_child[0]);  //关闭父进程的读端
        close(child_to_parent[1]);  //关闭子进程的写端
```

```c
40.         //从 c 中写消息到管道
41.         write(parent_to_child[1], &c, 1);
42.         n = read(child_to_parent[0], &c, 1);
43.         if (n != 1)
44.         {
45.             fprintf(2, "parent read error\n");
46.             exit(1);
47.         }
48.         //打印父进程 pid 和提示消息
49.         printf("%d: received pong\n", getpid());
50.         //程序结束前要关闭管道，避免阻塞
51.         close(child_to_parent[0]);  //关闭子进程的读端
52.         close(parent_to_child[1]);  //关闭父进程的写端
53.         wait(0);
54.         exit(0);
55.     }
56. }
```

我们分析 17 行-35 行子进程执行的任务，父进程十分相似。首先 17 行在判断语句中运行系统调用 fork()，生成子进程。fork 返回值为 0 时，表明当前进程是子进程，任务要求子进程从管道中读取消息，然后在另外一个管道中写入消息，因此需要关闭其他无关的管道端口，然后再调用相关的 write、read 等系统调用。最后结束前关闭所有管道端口。程序运行如下图。





3.2 Lab: system calls

3.2.1 system call tracing

根据实验要求，需要添加一个 trace system call 系统调用，实现对其他的系统调用的跟踪。我们首先需要了解系统调用的执行路径，根据书籍内容，我们以 exec 系统调用为例。

首先用户代码执行 exec，参数被放入寄存器 a0、a7 中，其中系统调用号被放置在 a7 中，通过 ecall 命令陷入内核。然后经过 syscall 的处理，

找到对应的系统调用，内核执行系统调用 sys_exec，再执行 exec.c 的内容。最后执行完系统调用之后，得到一个返回值，内核会将这个返回值放入 trapframe 的 a0 中，经过 userret 将返回值取出，放入 a0 寄存器中。由此完成了系统调用的全部过程。

我们仿照以上的路径，在相关的文件中编写新的系统调用 trace system call，并在关键节点输出信息。

1）相关声明和宏定义

```
#define SYS_sysinfo 23
```

kernel/syscall.h

```
entry("trace");
```

user/usys.pl

```
int trace(int);
```

user/user.h

```
char mask[23];
```

kernel/proc.h

2）具体函数实现

```
1.  uint64
2.  sys_trace(void){
3.    int n;
4.    if(argint(0, &n) < 0)
5.      return -1;
6.    struct proc *p = myproc();
7.    char *mask = p->mask;
8.    int i = 0;
9.    while (i < MASK_SIZE && n > 0){
10.     if (n % 2){
11.       mask[i++] = '1';
12.     }else {
13.       mask[i++] = '0';
14.     }
15.     n >>= 1;
16.   }
17.   return 0;
18. }
```

kernel/sysproc.c

sys_trace 主要是对结构体 proc(kernel/proc.h)进行转码，第 9 行-第 16 行将参数 n 的转化为二进制。

```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

```

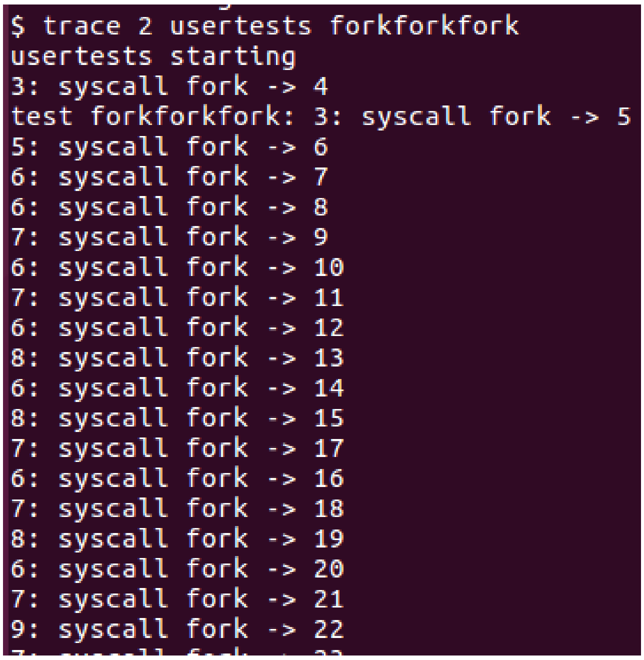
```c
7.    num = p->trapframe->a7;
8.    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
9.      p->trapframe->a0 = syscalls[num]();
10.     if(strlen(p->mask) > 0 && p->mask[num] == '1'){
11.       printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num], p->trapframe->a0);
12.     }
13.   } else {
14.     printf("%d %s: unknown sys call %d\n",
15.             p->pid, p->name, num);
16.     p->trapframe->a0 = -1;
17.   }
18. }
```

kernel/syscall.c

在 syscall 添加第 10 行-第 12 行，用于打印系统调用的相关进程号、系统调用类型以及 a0 寄存器内的参数。最终程序执行结果如下图。

程序运行命令参数需要四个及以上，其中第一个参数 trace 表示执行系统调用的跟踪，第二个参数 n 表示想要跟踪的系统调用的标识符，n 表示的形式多种多样，例如 trace(1<<SYS_fork)、trace(10b)、trace(2)都表示追踪系统调用 fork。其余参数表示执行指定的程序，例如 grep hello README 表示执行 grep 程序，hello、README 作为其参数。

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 968
3: syscall read -> 235
3: syscall read -> 0
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 968
4: syscall read -> 235
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$
```



```
$ trace 2 usertests forkforkfork
usertests starting
3: syscall fork -> 4
test forkforkfork: 3: syscall fork -> 5
5: syscall fork -> 6
6: syscall fork -> 7
6: syscall fork -> 8
7: syscall fork -> 9
6: syscall fork -> 10
7: syscall fork -> 11
6: syscall fork -> 12
8: syscall fork -> 13
6: syscall fork -> 14
8: syscall fork -> 15
7: syscall fork -> 17
6: syscall fork -> 16
7: syscall fork -> 18
8: syscall fork -> 19
6: syscall fork -> 20
7: syscall fork -> 21
9: syscall fork -> 22
```

根据实验要求总共运行了四种命令，相关命令及分析结果如下：

1）$ trace 32 grep hello README

n 为 32，对应二进制为 10000b，表示跟踪第五个系统调用即 read。之后执行 grep 程序时，当调用到系统调用 read 时，都会打印相关信息。

2）$ trace 2147483647 grep hello README

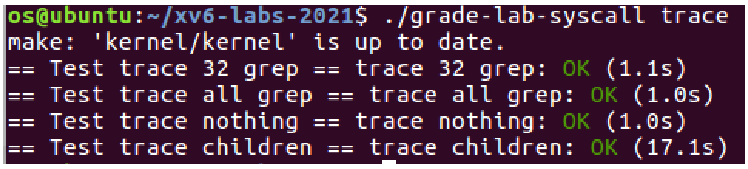
n 对应的二进制是 111 1111 1111 1111 1111 1111 1111 1111b，表示追踪所有的系统调用。

3）$ grep hello README

表示跟踪系统调用即 read 该命令只执行 grep 程序，没有追踪系统调用。因此没有输出。

4）$ grep hello README

n 为 3，对应二进制为 11b，表示跟踪系统调用即 fork。之后执行 usertests 程序时，当调用到系统调用 fork 时，都会打印相关信息。



```
os@ubuntu:~/xv6-labs-2021$ ./grade-lab-syscall trace
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.1s)
== Test trace all grep == trace all grep: OK (1.0s)
== Test trace nothing == trace nothing: OK (1.0s)
== Test trace children == trace children: OK (17.1s)
```

3.2.2 system call sysinfo

根据实验要求，需要添加一个 system call sysinfo 系统调用，直接打印出可用空间大小和可用进程数。相关的声明定义与 trace system call 相同，不再赘述，我们主要分析两个打印信息的计算。

根据提示我们将计算值赋给 sysinfo 的结构体中。首先 freemem 成员值的计算函数如下：

```c
int
freemem_size(void){
  struct run *r;
  int num = 0;
  for (r = kmem.freelist; r; r = r->next){
```

```c
    num++;
 }
  return num * PGSIZE;
}
```

kernel/kalloc.c

主要含义是计算系统可用的页表数目，然后乘上定义的页表大小，即得到可用的字节数。

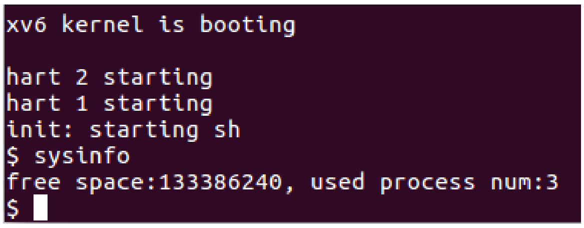
然后 nproc 成员值的计算函数如下：

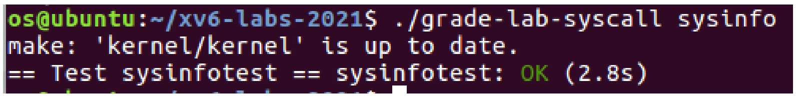
```c
int
proc_num(void){
  struct proc *p;
  uint64 num = 0;
  for(p = proc; p < &proc[NPROC]; p++) {
    if (p->state != UNUSED){
      num++;
    }
  }
  return num;
}
```

kernel/proc.c

主要含义是统计系统当前状态为 UNUSED 的进程数目。

最终运行结果如下图所示。





3.3 内核代码阅读

3.3.1 proc.h/proc.c

xv6 系统中每个进程都是一个结构体的形式，定义在 proc.h 中，主要包含了进程的属性、父进程、内存以及一些相关的其他结构体成员。

进程的工作方式主要被定义在 proc.c 中，主要有进程创建、退出、派生、等待以及杀死等等。这些工作常常需要和 cpu 或者其他进程等结构体合作。 因此，进程往往不执行具体的任务，而是操控内存和内核变量。

为了查看每个进程的扩展信息，将进程的属性和结构体中的各成员的值打印出来，我们在 procdump 函数中做了如下修改。

```c
printf("name: %s\n", p->name);
    do {
    printf("\tpid: %d\n", p->pid);
```

```c
    printf("\tstate: %s", state);
    printf("\tsize: %d", p->sz);
    printf("\taddress: %p", p->pagetable);
    printf("register:\n");
    printf("\treturn address: %p\n", p->context.ra);
    printf("\tstack pointer: %p\n", p->context.sp);
    printf("\ts0: %p    s1: %p    s2: %p    s3: %p\n", p->context.s0, p->context.s1,p->context.s2,p->context.s3);
    printf("\ts4: %p    s5: %p    s6: %p    s7: %p\n", p->context.s4, p->context.s5,p->context.s6,p->context.s7);
    printf("\ts8: %p    s9: %p    s10: %p    s11: %p\n", p->context.s8, p->context.s9,p->context.s10,p->context.s11);
    //printf("\ta0: %p    a1: %p\n", p->context.a0, p->context.a1);
    }while(0);
```

首先我们打印了进程的状态、大小、虚拟地址等信息。然后我们因此打印出 13 个寄存器的内容。

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
name: init
        pid: 1
        state: sleep    size: 12288     address: 0x0000000087f6f000register:
        return address: 0x0000000080001498
        stack pointer: 0x0000003fffffdee0
        s0: 0x0000003fffffdf10    s1: 0x0000000080009480    s2: 0x000000008000
9050    s3: 0x0000000000000001
        s4: 0x0000000000000005    s5: 0x0000000000000001    s6: 0x000000000000
0000    s7: 0x0000000000000000
        s8: 0x0000000080009068    s9: 0x0505050505050505    s10: 0x05050505050
50505    s11: 0x0505050505050505

name: sh
        pid: 2
        state: sleep    size: 16384     address: 0x0000000087f64000register:
        return address: 0x0000000080001498
        stack pointer: 0x0000003fffffbe80
        s0: 0x0000003fffffbeb0    s1: 0x0000000080009600    s2: 0x000000008000
9050    s3: 0x0000000000000001
        s4: 0x0000000000003f2f    s5: 0x0000000000000001    s6: 0x000000000000
0001    s7: 0x0000000000000004
        s8: 0xffffffffffffffff    s9: 0x000000000000000a    s10: 0x05050505050
50505    s11: 0x0505050505050505
```

3.3.2 swtch.S

swtch 是用汇编语言编写的，直接对寄存器操作。根据 risc-v 的指令集，我们可以查阅得到 sd 指令的含义是将寄存器的内容存储到存储器中，而 ld 指令是将存储器的内容存储到寄存器中。swtch.S 中用 sd 指令将 13 个寄存器的内容都存储到了存储器 a0 上，每次调用 sd 指令时，相关语句的数值都增加 8，是因为寄存器大小为 64 位，即 8 个字节，每次读取一个寄存器时都需要偏移 8 个字节。而程序使用 ld 指令又将 a1 存储器中的内容依次存储到 13 个寄存器中。这样下来，程序就完成了存储器内容的切换，即切换到一个新的进程中。