实验三 同步与通信

16281153　张天悦 安全1601

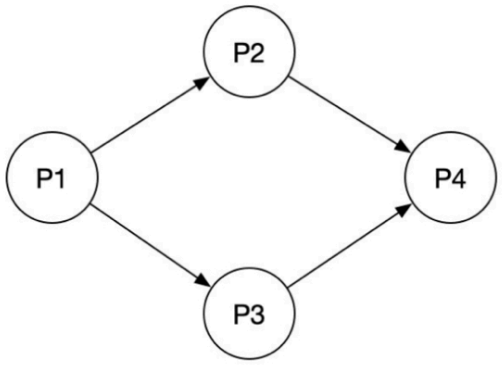
1. 实验目的

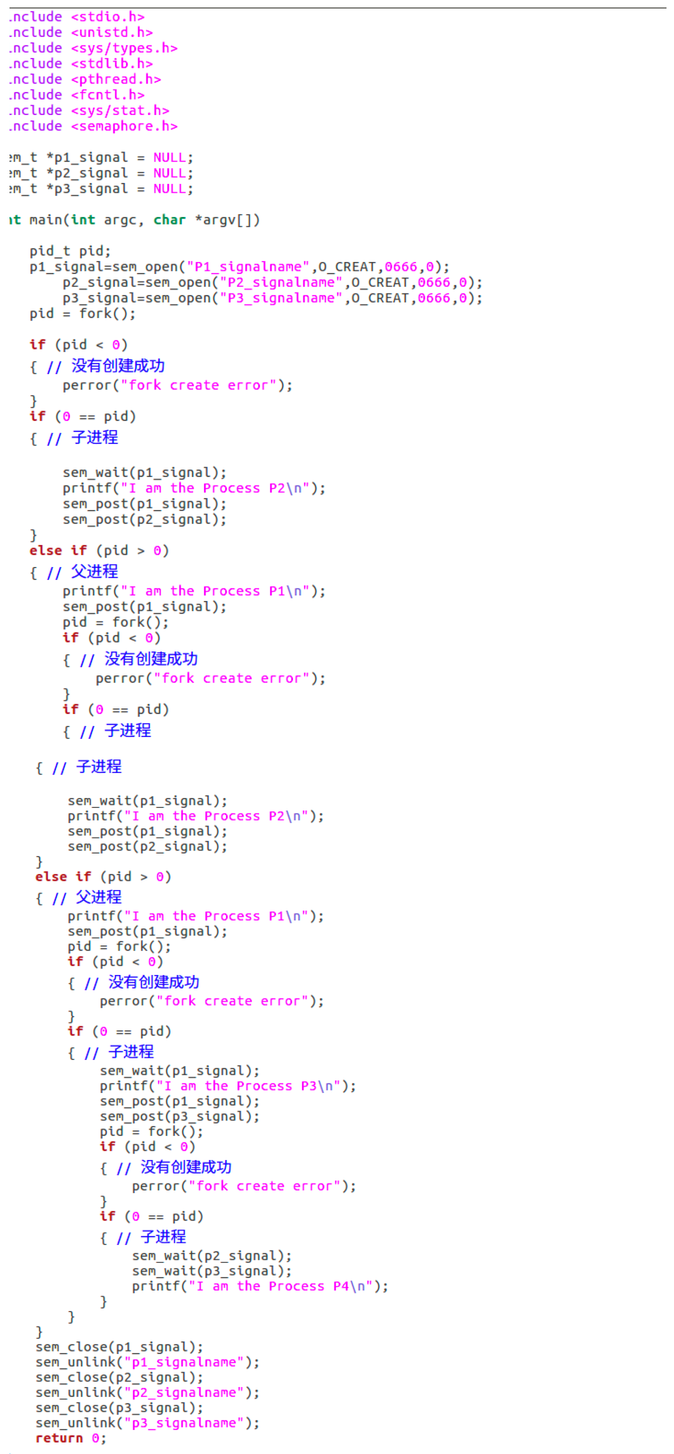
- 系统调用的进一步理解。
- 进程上下文切换。
- 同步与通信方法。

2. 实验题目

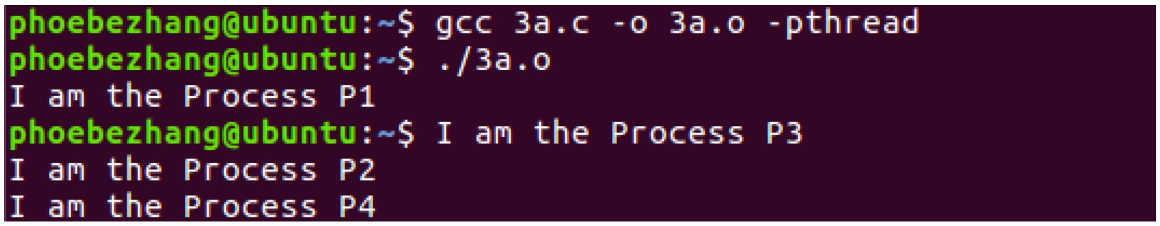
1）通过fork的方式，产生4个进程P1,P2,P3,P4，每个进程打印输出自己的名字，例如P1输出"I am the process P1"。要求P1最先执行，P2、P3互斥执行，P4最后执行。通过多次测试验证实现是否正确。

四个进程按照一定顺序运行，利用多个信号量来保证这个指定顺序
利用fork()产生多个进程
其中p1最先执行执行，利用信号量 p1_signal 初始化为0， 让进程p2，p3，p4等待p1_signal,只有p1执行完毕才会signal()
p2，p3互斥，利用wait(p1_signal)来等待p1结束，而且两者都同时等待p1_signal，通过这种方式保证互斥
p4最后一个执行，利用wait(p2_signal),wait(p3_signal)来等待所有进程都结束





```c
.nclude <stdio.h>
.nclude <unistd.h>
.nclude <sys/types.h>
.nclude <stdlib.h>
.nclude <pthread.h>
.nclude <fcntl.h>
.nclude <sys/stat.h>
.nclude <semaphore.h>

m_t *p1_signal = NULL;
m_t *p2_signal = NULL;
m_t *p3_signal = NULL;

t main(int argc, char *argv[])

  pid_t pid;
  p1_signal=sem_open("P1_signalname",O_CREAT,0666,0);
      p2_signal=sem_open("P2_signalname",O_CREAT,0666,0);
      p3_signal=sem_open("P3_signalname",O_CREAT,0666,0);
  pid = fork();

  if (pid < 0)
  { // 没有创建成功
      perror("fork create error");
  }
  if (0 == pid)
  { // 子进程

      sem_wait(p1_signal);
      printf("I am the Process P2\n");
      sem_post(p1_signal);
      sem_post(p2_signal);
  }
  else if (pid > 0)
  { // 父进程
      printf("I am the Process P1\n");
      sem_post(p1_signal);
      pid = fork();
      if (pid < 0)
      { // 没有创建成功
          perror("fork create error");
      }
      if (0 == pid)
      { // 子进程

   { // 子进程

       sem_wait(p1_signal);
       printf("I am the Process P2\n");
       sem_post(p1_signal);
       sem_post(p2_signal);
   }
   else if (pid > 0)
   { // 父进程
       printf("I am the Process P1\n");
       sem_post(p1_signal);
       pid = fork();
       if (pid < 0)
       { // 没有创建成功
           perror("fork create error");
       }
       if (0 == pid)
       { // 子进程
           sem_wait(p1_signal);
           printf("I am the Process P3\n");
           sem_post(p1_signal);
           sem_post(p3_signal);
           pid = fork();
           if (pid < 0)
           { // 没有创建成功
               perror("fork create error");
           }
           if (0 == pid)
           { // 子进程
               sem_wait(p2_signal);
               sem_wait(p3_signal);
               printf("I am the Process P4\n");
           }
       }
   }
   sem_close(p1_signal);
   sem_unlink("p1_signalname");
   sem_close(p2_signal);
   sem_unlink("p2_signalname");
   sem_close(p3_signal);
   sem_unlink("p3_signalname");
   return 0;
```



```
phoebezhang@ubuntu:~$ gcc 3a.c -o 3a.o -pthread
phoebezhang@ubuntu:~$ ./3a.o
I am the Process P1
phoebezhang@ubuntu:~$ I am the Process P3
I am the Process P2
I am the Process P4
```

2）火车票余票数 ticketCount 初始值为 1000，有一个售票线程，一个退票线程，各循环执行多次。添加同步机制，使得结果始终正确。要求多次测试添加同步机制前后的实验效果。(*说明：为了更容易产生并发错误，可以在适当的位置增加一些 pthread_yield()，放弃 CPU，并强制线程频繁切换，例如售票线程的关键代码：*

temp=ticketCount;

pthread_yield();

temp=temp-1;

pthread_yield();

ticketCount=temp;

退票线程的关键代码：

temp=ticketCount;

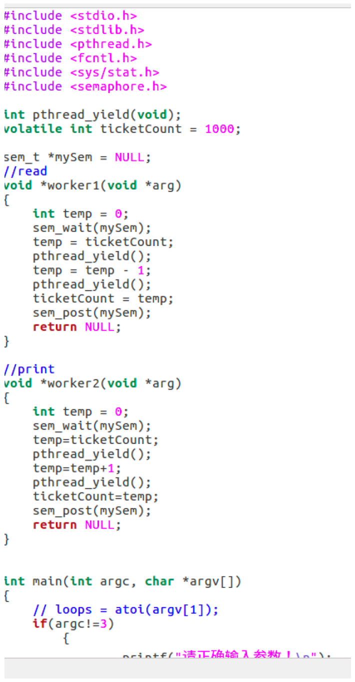
pthread_yield();

temp=temp+1;

pthread_yield();

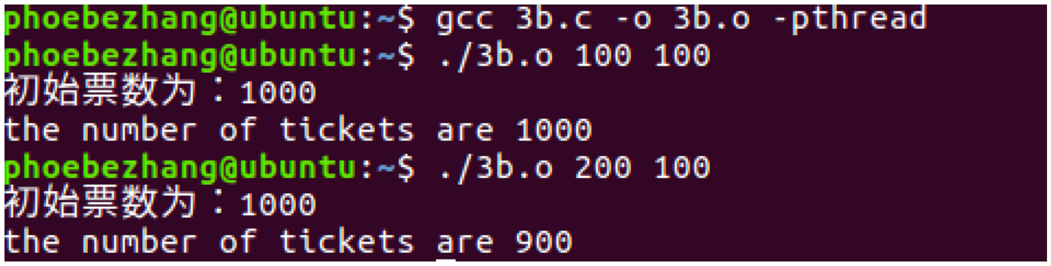
ticketCount=temp;

)



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

int pthread_yield(void);
volatile int ticketCount = 1000;

sem_t *mySem = NULL;
//read
void *worker1(void *arg)
{
    int temp = 0;
    sem_wait(mySem);
    temp = ticketCount;
    pthread_yield();
    temp = temp - 1;
    pthread_yield();
    ticketCount = temp;
    sem_post(mySem);
    return NULL;
}

//print
void *worker2(void *arg)
{
    int temp = 0;
    sem_wait(mySem);
    temp=ticketCount;
    pthread_yield();
    temp=temp+1;
    pthread_yield();
    ticketCount=temp;
    sem_post(mySem);
    return NULL;
}


int main(int argc, char *argv[])
{
    // loops = atoi(argv[1]);
    if(argc!=3)
        {
                printf("请正确输入参数！\n");
```

实验结果

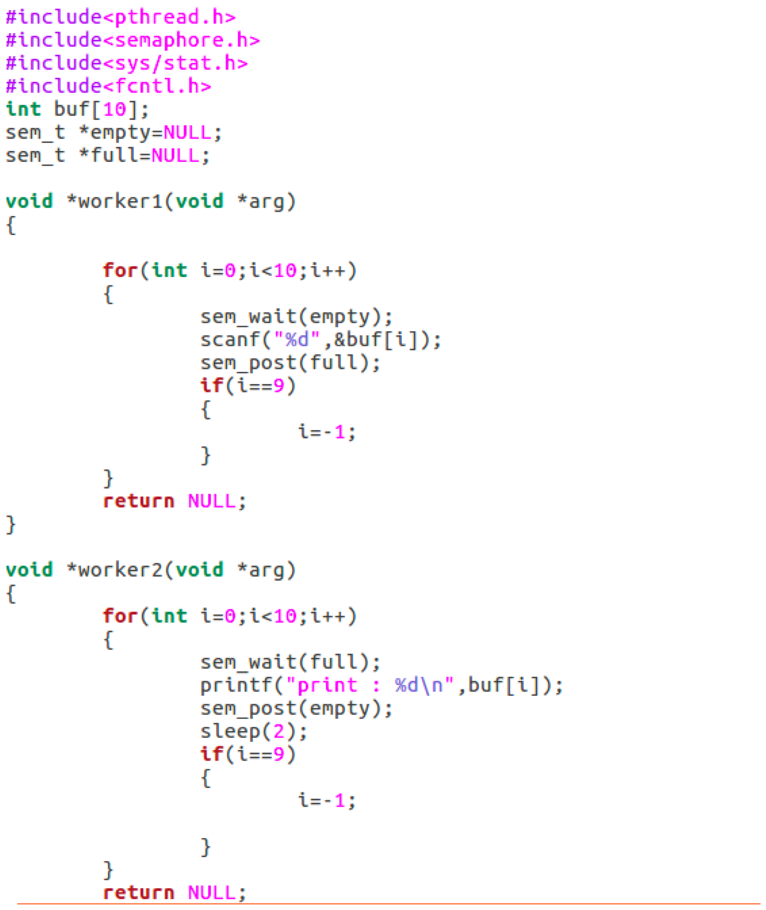


```
phoebezhang@ubuntu:~$ gcc 3b.c -o 3b.o -pthread
phoebezhang@ubuntu:~$ ./3b.o 100 100
初始票数为：1000
the number of tickets are 1000
phoebezhang@ubuntu:~$ ./3b.o 200 100
初始票数为：1000
the number of tickets are 900
```

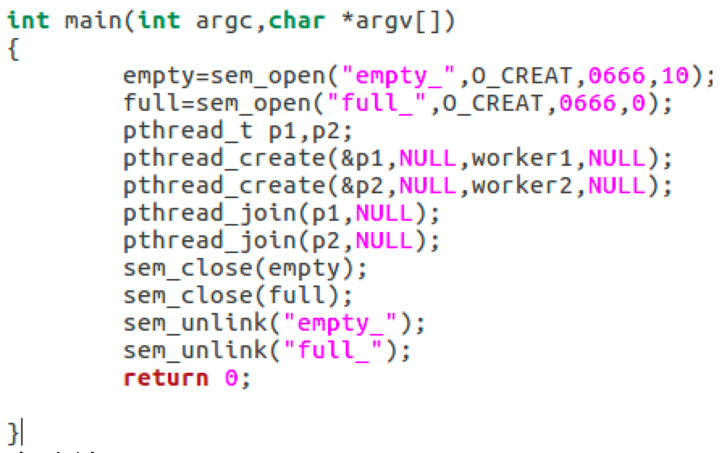
3）一个生产者一个消费者线程同步。设置一个线程共享的缓冲区， char buf[10]。

一个线程不断从键盘输入字符到 buf,一个线程不断的把 buf 的内容输出到显示器。要求输出的和输入的字符和顺序完全一致。(在输出线程中，每次输出睡眠一秒钟，然后以不同的速度输入测试输出是否正确)。要求多次测试添加同步机制前后的实验效果。

- 编写两个进程，一个输入，一个输出
- 利用一个共享的数据段进行存放数据
- 申请两个信号变量来控制输入输出，当 buff 不为满的时候才能输入
- 当 buff 不为空的时候才能输出，让输出函数用 sleep 控制速度

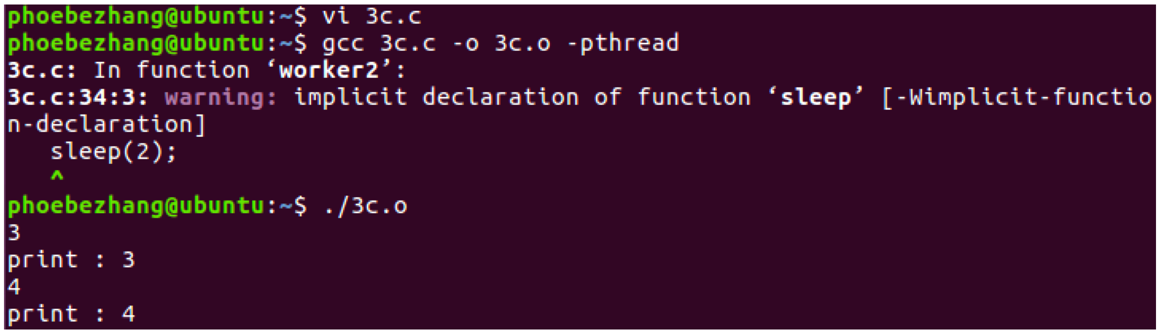


```c
#include<pthread.h>
#include<semaphore.h>
#include<sys/stat.h>
#include<fcntl.h>
int buf[10];
sem_t *empty=NULL;
sem_t *full=NULL;

void *worker1(void *arg)
{

        for(int i=0;i<10;i++)
        {
                sem_wait(empty);
                scanf("%d",&buf[i]);
                sem_post(full);
                if(i==9)
                {
                        i=-1;
                }
        }
        return NULL;
}

void *worker2(void *arg)
{
        for(int i=0;i<10;i++)
        {
                sem_wait(full);
                printf("print : %d\n",buf[i]);
                sem_post(empty);
                sleep(2);
                if(i==9)
                {
                        i=-1;

                }
        }
        return NULL;
```



```
int main(int argc,char *argv[])
{
        empty=sem_open("empty_",O_CREAT,0666,10);
        full=sem_open("full_",O_CREAT,0666,0);
        pthread_t p1,p2;
        pthread_create(&p1,NULL,worker1,NULL);
        pthread_create(&p2,NULL,worker2,NULL);
        pthread_join(p1,NULL);
        pthread_join(p2,NULL);
        sem_close(empty);
        sem_close(full);
        sem_unlink("empty_");
        sem_unlink("full_");
        return 0;

}
```

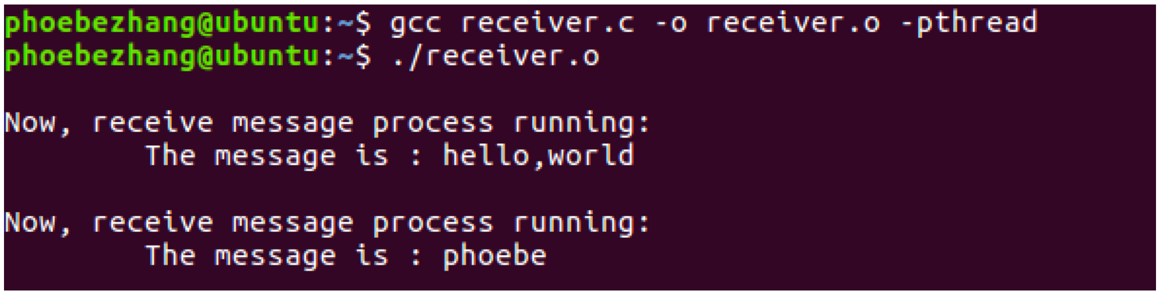
实验结果



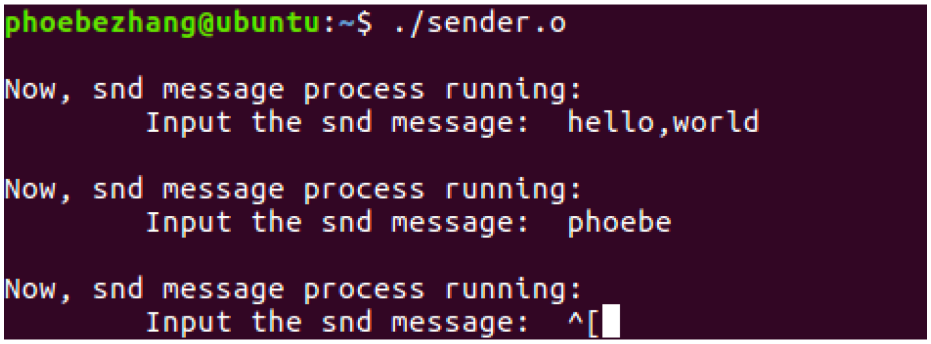
```
phoebezhang@ubuntu:~$ vi 3c.c
phoebezhang@ubuntu:~$ gcc 3c.c -o 3c.o -pthread
3c.c: In function ‘worker2’:
3c.c:34:3: warning: implicit declaration of function ‘sleep’ [-Wimplicit-functio
n-declaration]
   sleep(2);
   ^
phoebezhang@ubuntu:~$ ./3c.o
3
print : 3
4
print : 4
```

4)

4.a 通过实验测试，验证共享内存的代码中，receiver 能否正确读出 sender 发送的字符串？如果把其中互斥的代码删除，观察实验结果有何不同？如果在发送和接收进程中打印输出共享内存地址，他们是否相同，为什么？



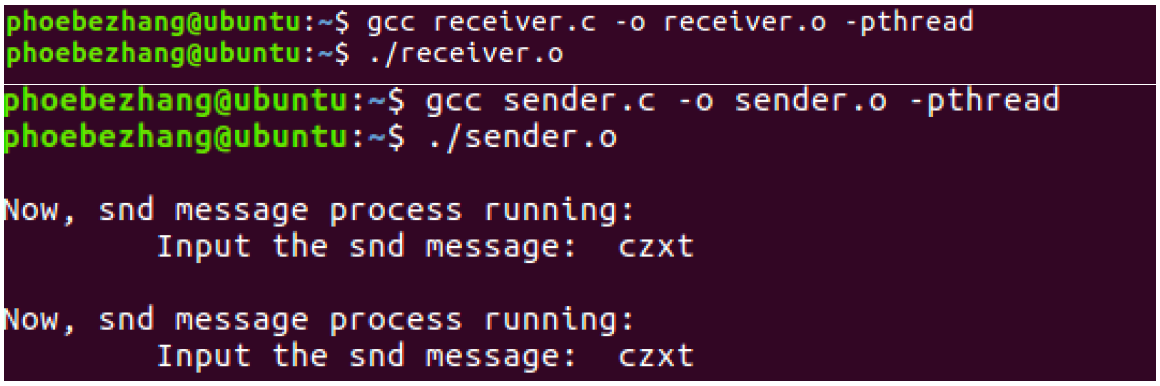
```
phoebezhang@ubuntu:~$ gcc receiver.c -o receiver.o -pthread
phoebezhang@ubuntu:~$ ./receiver.o

Now, receive message process running:
        The message is : hello,world

Now, receive message process running:
        The message is : phoebe
```



```
phoebezhang@ubuntu:~$ ./sender.o

Now, snd message process running:
        Input the snd message:  hello,world

Now, snd message process running:
        Input the snd message:  phoebe

Now, snd message process running:
        Input the snd message:  ^[
```

sender 进程发出的消息 receiver 进程均收到。

删除互斥相关访问代码

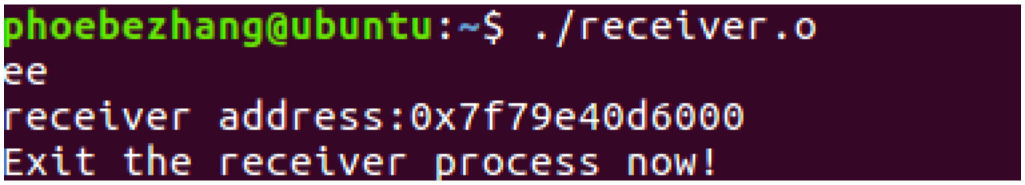


```
phoebezhang@ubuntu:~$ gcc receiver.c -o receiver.o -pthread
phoebezhang@ubuntu:~$ ./receiver.o
```

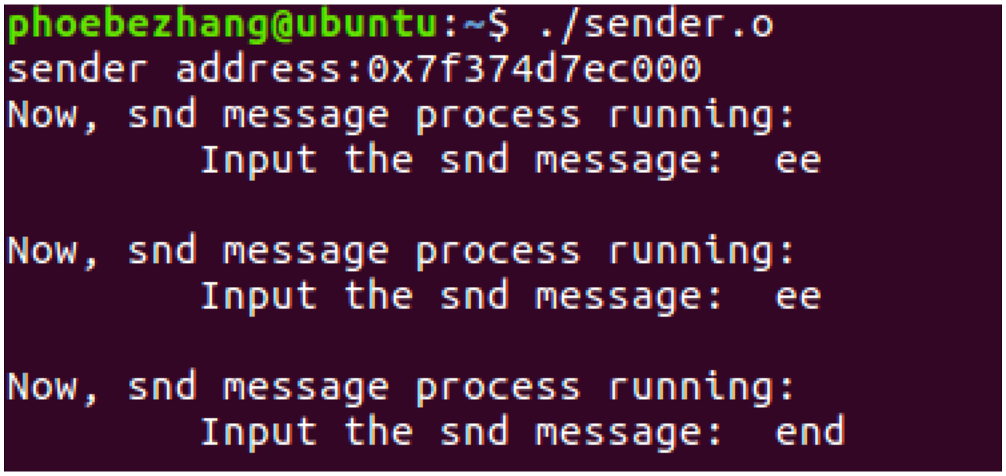
```
phoebezhang@ubuntu:~$ gcc sender.c -o sender.o -pthread
phoebezhang@ubuntu:~$ ./sender.o

Now, snd message process running:
        Input the snd message:  czxt

Now, snd message process running:
        Input the snd message:  czxt
```

当删除互斥访问之后，两个进程便没有限制的访问共享内存

打印共享内存地址



```
phoebezhang@ubuntu:~$ ./receiver.o
ee
receiver address:0x7f79e40d6000
Exit the receiver process now!
```



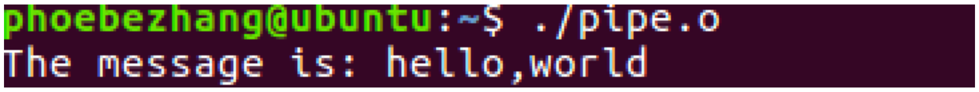
```
phoebezhang@ubuntu:~$ ./sender.o
sender address:0x7f374d7ec000
Now, snd message process running:
        Input the snd message:  ee

Now, snd message process running:
        Input the snd message:  ee

Now, snd message process running:
        Input the snd message:  end
```

在两个进程中共享内存的地址不一样

4.b&4.c

无名管道

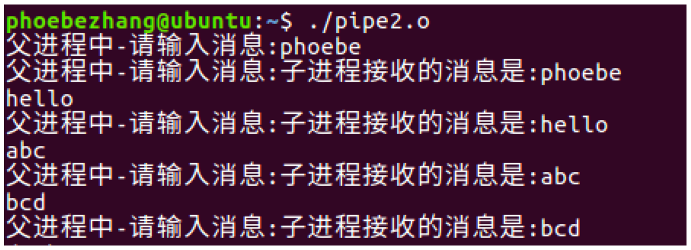
pipe.c



```
phoebezhang@ubuntu:~$ ./pipe.o
The message is: hello,world
```

无名管道同步机制验证

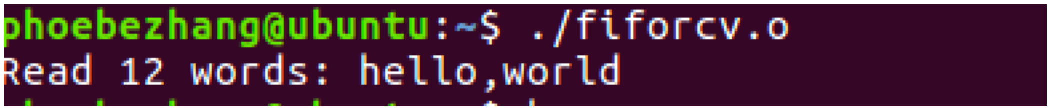
pipe2.c



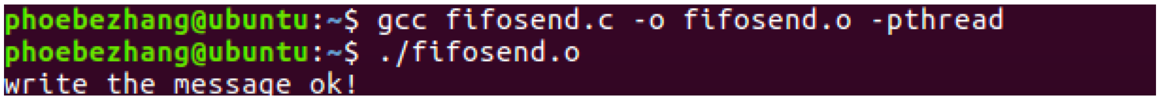
```
phoebezhang@ubuntu:~$ ./pipe2.o
父进程中-请输入消息:phoebe
父进程中-请输入消息:子进程接收的消息是:phoebe
hello
父进程中-请输入消息:子进程接收的消息是:hello
abc
父进程中-请输入消息:子进程接收的消息是:abc
bcd
父进程中-请输入消息:子进程接收的消息是:bcd
```

管道的读写通过两个系统调用 write 和 read 实现
发送者在向管道内存中写入数据之前，首
先检查内存是否被读进程锁定和内存中是否还有剩余空间，如果这两个要求都满足的话 write 函数会对内存上锁，然后进行写入数据，写完之后解锁；否则就会等待(阻塞)。
写进程在读取管道中的数据之前，也会检查内存是否被读进程锁定和管道内存中是否有数据，如果满足这两个条件，read 函数会对内存上锁，读取数据后在解锁；否则会等到(阻塞)

有名管道



```
phoebezhang@ubuntu:~$ ./fiforcv.o
Read 12 words: hello,world
```



```
phoebezhang@ubuntu:~$ gcc fifosend.c -o fifosend.o -pthread
phoebezhang@ubuntu:~$ ./fifosend.o
write the message ok!
```

当客户端不阻塞的话在客户端接受服务器端消息的时候会无限制的打印消息队列中的空消息，哪怕消息队列中没有任何消息
当服务器端没有设置阻塞的时候，服务器端会一直接受消息队列中的空消息并向客户端转发。
5)

```
/* States in a thread's life cycle. */
enum thread_status
  {
    THREAD_RUNNING,       /* Running thread. */
    THREAD_READY,         /* Not running but ready to run. */
    THREAD_BLOCKED,       /* Waiting for an event to trigger. */
    THREAD_DYING          /* About to be destroyed. */
  };
/* Thread identifier type.
   You can redefine this to whatever type you like. */
typedef int tid_t;
#define TID_ERROR ((tid_t) -1)          /* Error value for tid_t. */
```

```
/* Thread priorities. */
#define PRI_MIN 0                          /* Lowest priority. */
#define PRI_DEFAULT 31                     /* Default priority. */
#define PRI_MAX 63                         /* Highest priority. */
```

结构体的定义

tid_t **tid** :线程的线程标识符。每个线程必须具有在内核的整个生命周期内唯一的 tid。默认情况下，tid_t 是 int 的 typedef（在上面定义过了），每个新线程接收数字上的下一个更高的 tid，从初始进程的 1 开始。

enum **thread_status**：线程的状态，一共有以下四种：

- **THREAD_RUNNING**：线程在给定时间内正在运行。
- **THREAD_READY**：该线程已准备好运行，但它现在没有运行。
- **THREAD_BLOCKED**：线程正在等待某些事务，
- **THREAD_DYING**：切换到下一个线程后，调度程序将销毁该线程。

char **name[16]**：线程命名的字符串，至少前几个数组单元为字符。

uint8_t ***stack**：线程的栈指针。当线程运行时，CPU 的堆栈指针寄存器跟踪堆栈的顶部，并且该成员未使用。但是当 CPU 切换到另一个线程时，该成员保存线程的堆栈指针。保存线程的寄存器不需要其他成员，因为必须保存的其他寄存器保存在堆栈中。

int **priority**：线程优先级，范围从 PRI_MIN（0）到 PRI_MAX（63）。较低的数字对应较低的优先级，因此优先级 0 是最低优先级，优先级 63 是最高优先级。

struct list_elem **allelem**：用于将线程链接到所有线程的列表中。每个线程在创建时都会插入到此列表中，并在退出时删除。应该使用 thread_foreach（）函数来迭代所有线程。

struct list_elem **elem**：用于将线程放入双向链表：ready_list（准备好运行的线程列表）或 sema_down（等待信号量的线程列表）。上面定义过了，default 31。

uint32_t **pagedir**：页表指针，用于将进程结构的虚拟地址映射到物理地址。

unsigned **magic**：始终设置为 THREAD_MAGIC，用于检测堆栈溢出。

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */
    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

thread.c

init()函数：它为一个新建的进程指定了状态，分配进程号。调用 init_thread()分配地址。

```
void
thread_init (void)
{
  ASSERT (intr_get_level () == INTR_OFF);

  lock_init (&tid_lock);
  list_init (&ready_list);
  list_init (&all_list);

  /* Set up a thread structure for the running thread. */
  initial_thread = running_thread ();
  init_thread (initial_thread, "main", PRI_DEFAULT);
  initial_thread->status = THREAD_RUNNING;
  initial_thread->tid = allocate_tid ();
}

/* Does basic initialization of T as a blocked thread named
   NAME. */
static void
init_thread (struct thread *t, const char *name, int priority)
{
  enum intr_level old_level;

  ASSERT (t != NULL);
  ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
  ASSERT (name != NULL);

  memset (t, 0, sizeof *t);
  t->status = THREAD_BLOCKED;
  strlcpy (t->name, name, sizeof t->name);
  t->stack = (uint8_t *) t + PGSIZE;
  t->priority = priority;
  t->magic = THREAD_MAGIC;

  old_level = intr_disable ();
  list_push_back (&all_list, &t->allelem);
  intr_set_level (old_level);
}

thread_block (void)
/* Puts the current thread to sleep.  It will not be scheduled
   again until awoken by thread_unblock().

   This function must be called with interrupts turned off.  It
   is usually a better idea to use one of the synchronization
   primitives in synch.h. */
void
thread_block (void)
{
  ASSERT (!intr_context ());
  ASSERT (intr_get_level () == INTR_OFF);

  thread_current ()->status = THREAD_BLOCKED;
  schedule ();
}

/* Schedules a new process.  At entry, interrupts must be off and
   the running process's state must have been changed from
   running to some other state.  This function finds another
   thread to run and switches to it.

   It's not safe to call printf() until thread_schedule_tail()
   has completed. */
static void
schedule (void)
{
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (cur->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (cur != next)
    prev = switch_threads (cur, next);
  thread_schedule_tail (prev);
}
```

看 running_thread()的返回值，此函数嵌入了汇编代码，将 CPU 堆栈指针复制到 “esp” 中。因为 “struct thread” 总是在页面的起始，而堆栈指针位于中间的某个位置，所以它定位当前线程。因此 cur 指针就是当前运行线程的指针。

```
/* Returns the running thread. */
struct thread *
running_thread (void)
{
  uint32_t *esp;

  /* Copy the CPU's stack pointer into `esp', and then round that
      down to the start of a page.  Because `struct thread' is
      always at the beginning of a page and the stack pointer is
      somewhere in the middle, this locates the curent thread. */
  asm ("mov %%esp, %0" : "=g" (esp));
  return pg_round_down (esp);
}
```

上面 next 指针对应的函数 next_thread_to_run（），是一个返回 list_entry 中 pop 的值的函数。如果 list 为空，返回 idle_thread.

```
/* Chooses and returns the next thread to be scheduled.  Should
    return a thread from the run queue, unless the run queue is
    empty.  (If the running thread can continue running, then it
    will be in the run queue.)  If the run queue is empty, return
    idle_thread. */
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
    else
        return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
```

pre 指针为 null

调用 switch_threads (cur, next)将当前进程和下一个进程进行切换。存放在 siwth.S 中，将当前堆栈的指针保存到 cur 线程的堆栈，接着从 next 线程的堆栈中恢复当前堆栈的指针，也就是寄存器 esp 的操作。由此我们可以确定进程的保存与恢复就是利用 CPU 栈顶指针的变化进行的，进程的状态则是保存在自身的堆栈当中。

```
#include "threads/switch.h"

#### struct thread *switch_threads (struct thread *cur, struct thread *next);
####
#### Switches from CUR, which must be the running thread, to NEXT,
#### which must also be running switch_threads(), returning CUR in
#### NEXT's context.
####
#### This function works by assuming that the thread we're switching
#### into is also running switch_threads().   Thus, all it has to do is
#### preserve a few registers on the stack, then switch stacks and
#### restore the registers.   As part of switching stacks we record the
#### current stack pointer in CUR's thread structure.

.globl switch_threads
.func switch_threads
switch_threads:
        # Save caller's register state.
        #
        # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
        # but requires us to preserve %ebx, %ebp, %esi, %edi.   See
        # [SysV-ABI-386] pages 3-11 and 3-12 for details.
        #
        # This stack frame must match the one set up by thread_create()
        # in size.
        pushl %ebx
        pushl %ebp
        pushl %esi
        pushl %edi

        # Get offsetof (struct thread, stack).
```

```
.globl thread_stack_ofs
        mov thread_stack_ofs, %edx

        # Save current stack pointer to old thread's stack, if any.
        movl SWITCH_CUR(%esp), %eax
        movl %esp, (%eax,%edx,1)

        # Restore stack pointer from new thread's stack.
        movl SWITCH_NEXT(%esp), %ecx
        movl (%ecx,%edx,1), %esp

        # Restore caller's register state.
        popl %edi
        popl %esi
        popl %ebp
        popl %ebx
                  ret
.endfunc
```

schedule ();的最后一个函数：thread_schedule_tail (prev)，通过激活新线程的页表完成线程切换，如果前一个线程正在死亡，则销毁它。首先会获取当前运行进程的指针并保证此时程序不能被中断。接着其会将当其运行进程的状态改变为 THREAD_RUNNING 以及初始化其时间切片，这可以看做切换进程后对新进程的一个激活。最后的部分表示如果我们切换的线程正在死亡，销毁它的 struct 线程。而我们传入的 prev 一定为 NULL，所以在切换过程中这一部分并不会执行。

```
void
thread_schedule_tail (struct thread *prev)
{
    struct thread *cur = running_thread ();

    ASSERT (intr_get_level () == INTR_OFF);

    /* Mark us as running. */
    cur->status = THREAD_RUNNING;

    /* Start new time slice. */
    thread_ticks = 0;

#ifdef USERPROG
    /* Activate the new address space. */
```

```
  process_activate ();
#endif

  /* If the thread we switched from is dying, destroy its struct
     thread.  This must happen late so that thread_exit() doesn't
     pull out the rug under itself.  (We don't free
     initial_thread because its memory was not obtained via
     palloc().) */
  if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
    {
      ASSERT (prev != cur);
      palloc_free_page (prev);
    }
}
```