

通信机制实习报告

杨东升 1400012898
2017.6.10

目录

| | |
|-------------------------------|---|
| 内容一：总体概述..... | 3 |
| 内容二：任务完成情况..... | 3 |
| 任务完成列表（Y/N）..... | 3 |
| 具体 Exercise 的完成情况..... | 3 |
| 内容三：遇到的困难以及解决方法..... | 8 |
| 内容四：收获及感想..... | 8 |
| 内容五：对课程的意见和建议..... | 8 |
| 内容六：参考文献..... | 8 |

内容一：总体概述

本次试验要设计实现进程间的通信机制。首先调研 Linux 中的通信机制，然后我在 nachos 中实现了信号机制，并编写程序进行测试。

内容二：任务完成情况

任务完成列表 (Y/N)

| | Exercise1 | Exercise2 | Exercise3 | Exercise4 | Exercise5 |
|------|-----------|-----------|-----------|-----------|-----------|
| 第一部分 | Y | Y | Y | | |

(总之就是都完成了)

具体 Exercise 的完成情况

Exercise1 调研Linux中进程通信机制的实现

Linux中的进程通信机制有管道、FIFO、信号等。

1. 管道机制

管道也是文件的一种，在Linux系统中是一段固定大小的缓冲区，大小为1页。在Linux中，管道没有专门的数据结构，而是借用了文件系统中的inode节点和file结构。通过将两个文件指向同一个inode节点，再让这个inode节点指向一个物理页面，来实现管道。这样，就可以通过文件系统的读写操作，将读写重定向到这个物理页面中，当管道为空时读文件操作会阻塞，管道为满时写文件操作会阻塞，就是一个生产者消费者问题。管道文件操作是基于fork函数机制来实现的，因此管道文件只能让具有亲缘关系的进程之间进行通信。

2. FIFO

FIFO也叫命名管道，这个机制克服了管道中不能任意进程之间通信的障碍。FIFO也是一种特殊的文件，它在文件系统中具有自己的文件路径。如果同时有一个进程对该文件进行读操作，另一个进程对其进行写操作时，那么Linux内核就会在这两个进程之间建立管道。而FIFO实际上并不与磁盘有关，只是内核中的实现抽象，本质上也是一个FIFO队列。FIFO只是借用了文件系统的命名机制。当删除FIFO文件时，管道也就消失了。这种机制可以很好的让任意进程之间建立一个管道通信。

3. 信号

信号机制是用来让一个进程给另外一个进程发送信号来通知有事件发生。每次当一个进程被调度器调上cpu时，会检查信号，然后处理。信号本质上是在软件层次上对中断机制的模拟。当一个进程收到信号，也可以通过中断机制来处理。内核给进程发送信号，实际做法是在进程对应的信号表项上设置位，当这个进程从内核态返回到用户态时，就会开始处理信

号。等到处理完信号，才会返回到用户态。信号还可以阻塞，通过设置进程中对应的阻塞位，就可以屏蔽一些信号，即使信号到达也不会对信号进行处理。信号的处理函数还可以让程序员自己编写，只要在程序中进行对应的设置，就可以通知内核在对应的信号处理函数处理信号。一个信号的生命周期包括了：信号的诞生，信号在进程中注册，信号在进程中注销，信号处理函数执行。

Exercise 2 为 Nachos 设计并实现一种线程/进程间通信机制

基于已完善的线程、调度、同步机制部分，选择 Linux 中的一种进程通信机制，在 Nachos 上实现。

我选择实现信号机制。进程通过设置其他进程的标志位来传递信息，每个进程在被调度上 CPU 时检查自己的标志位，如有信号就执行相关操作。

涉及到的数据结构有：

1、Thread 类的成员变量：线程编号 tid，信号标志 signalList，信号标志是一个 8 位比特串，代表 8 个信号。

```
public:
    int tid ;
    char signalList ;
```

在 Thread 构造函数中初始化上述变量：

```
Thread::Thread(char* threadName)
{
    static int tidcnt = 0 ;
    tid = tidcnt ++ ;
    signalList = 0 ;
```

2、全局变量：线程池指针数组，信号量数组。

```
Thread* threadPool[128] ;
Semaphore * syc[128][8] ;
```

在 Thread 构造函数中，把 this 插入线程池与 tid 对应位置

信号量数组对应每个进程的每个信号标志位，阻塞额外的信号，防止信号丢失。在 initialize 函数中，初始化信号量的值为 1

```
for(int j = 0 ; j < 128 ; j ++ )
    for( int i = 0 ; i < 8 ; i ++ )
        syc[j][i] = new Semaphore("name", 1) ;
```

在 system.cc 中编写信号的发送函数和处理函数如下：

信号发送函数在线程池中找到对应线程，如果线程存在，就试图对它的相应信号标志进行 P 操作，如果 P 操作成功，就改写目标线程的信号标志；如果 P 操作失败，说明这个信号标志当前已经被设置，而且还没被目标线程处理，因此需要等待目标线程处理好这个信号，V 之后才能再设置这个信号标志。

```

void Thread::Signal(int sig, int tid)
{
    printf("signal %d %d\n", sig, tid) ;
    if( threadPool[tid] != NULL )
    {
        syc[tid][sig]->P() ;
        threadPool[tid]->signalList |= ( 1 << sig ) ;
    }
}

```

信号接收函数检查自身的信号标志，如果某个 bit 被置 1，就需要执行对应的操作，然后把此 bit 归零，并且 V 对应的信号量。

```

void Thread::HandleSignal()
{
    if((signalList & 1) == 1)
    {
        signalList -= 1 ;
        currentThread->Finish() ;
        syc[tid][0]->V() ;
    }
    if((signalList & 2) == 2)
    {
        signalList -= 2 ;
        currentThread->Sleep() ;
        syc[tid][1]->V() ;
    }
    if((signalList & 4) == 4)
    {
        signalList -= 4 ;
        currentThread->Yield() ;
        syc[tid][2]->V() ;
    }
}

```

信号 0 把第一个 bit 置 1，代表结束当前线程。

信号 1 把第二个 bit 置 1，代表当前线程睡眠。

信号 2 把第三个 bit 置 1，代表当前线程下 CPU

在 schedual.cc 的 run 中，当线程完成切换后，调用 HandleSignal 函数，检查并处理信号。也就是说信号只会在线程切换时被接受和处理。

```

SWITCH(oldThread, nextThread);

```

```

DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());
currentThread->HandleSignal() ;

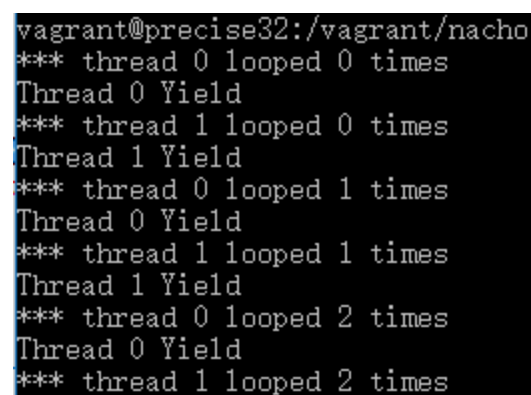
```

Exercise 3 为实现的通信机制编写测试用例

在 threadtest.cc 中编写测试函数。创建两个 SimpleThread 线程，tid 分别是 0 和 1，二者循环交替执行并输出循环次数。

```
void
SimpleThread(int which)
{
    int num;
    for (num = 0; num < 3; num++) {
        printf("*** thread %d looped %d times\n", currentThread->tid,
            currentThread->Yield());
    }
}
```

运行结果如下：



```
vagrant@precise32:/vagrant/nacho
*** thread 0 looped 0 times
Thread 0 Yield
*** thread 1 looped 0 times
Thread 1 Yield
*** thread 0 looped 1 times
Thread 0 Yield
*** thread 1 looped 1 times
Thread 1 Yield
*** thread 0 looped 2 times
Thread 0 Yield
*** thread 1 looped 2 times
```

之后修改代码，令 tid 为 1 的线程向 tid 为 0 的线程发送信号。所有版本包括：

发送 Yield 信号，

发送 Finish 信号，

发送 Sleep 信号，

两次发送 Yield 信号。下面是其中一个版本：

```
void
SimpleThread(int which)
{
    int num;
    if( which == 1)
    {
        currentThread->Signal(2, 0) ;
        currentThread->Signal(2, 0) ;
    }
    for (num = 0; num < 3; num++) {
        printf("*** thread %d looped %d times\n", currentThread->tid,
            currentThread->Yield());
    }
}
```

运行结果如下：

发 Finish 的运行结果:

```
*** thread 0 looped 0 times
Thread 0 Yield
signal 0 0
*** thread 1 looped 0 times
Thread 1 Yield
Thread 0 Finish
Thread 0 Sleep
```

发 Sleep 的运行结果:

```
vagrant@precise32:/vagrant/na
*** thread 0 looped 0 times
Thread 0 Yield
signal 1 0
*** thread 1 looped 0 times
Thread 1 Yield
Thread 0 Sleep
*** thread 1 looped 1 times
Thread 1 Yield
*** thread 1 looped 2 times
```

发 Yield 的运行结果:

```
*** thread 0 looped 0 times
Thread 0 Yield
signal 2 0
*** thread 1 looped 0 times
Thread 1 Yield
Thread 0 Yield
*** thread 1 looped 1 times
Thread 1 Yield
*** thread 0 looped 1 times
Thread 0 Yield
*** thread 1 looped 2 times
Thread 1 Yield
*** thread 0 looped 2 times
Thread 0 Yield
```

发两次 Yield 的运行结果:

```
*** thread 0 looped 0 times
Thread 0 Yield
signal 2 0
signal 2 0
Thread 1 Sleep
Thread 0 Yield
*** thread 0 looped 1 times
Thread 0 Yield
*** thread 1 looped 0 times
Thread 1 Yield
Thread 0 Yield
*** thread 1 looped 1 times
Thread 1 Yield
*** thread 0 looped 2 times
Thread 0 Yield
*** thread 1 looped 2 times
Thread 1 Yield
```

线程 1 向线程 0 发送两个连续的 Yield,
第二次发送时 P 被阻塞, 所以线程 1 Sleep
线程 0 先 Yield 一次, 然后 V 唤醒线程 1
线程 1 发送第二个 Yield 成功
线程 0 再 Yield 一次
之后正常运行

内容三：遇到的困难以及解决方法

困难 1：防止信号丢失

如果目标线程还没有处理上个信号，下个相同信号就来了，那么就会发送信号丢失，即信号只会被接受一遍。为了避免这个问题，我采用信号量来进行同步。每次发信号需要 P 相应的信号量，每次接受信号要 V 相应的信号量，这样信号就不会丢失了。

困难 2：交叉编译

由于文件的编译有先后关系，所以不能让依赖关系成环。在实现 `syc` 这个信号量数组时，我发现不能把它放在 `Thread` 类中，否则无法编译通过，所以只好把和信号量有关的函数和数据结构都放在了 `system.cc` 中。

内容四：收获及感想

通信机制是内核中的桥梁，如何兼顾效率和功能是一个比较困难的问题。我选用的信号机制开销比较低，但是只能在线程切换时才能接受信号，所以功能上有一定的局限性。因此，我认识的通信机制的多样性是很重要的，`linux` 之所以设计了很多种通信机制，就是为了满足不同场景的需求。这是我通过做这次 lab 才意识到的。

内容五：对课程的意见和建议

我认为最后的 challenge 难度都不大，没有 challenge 的感觉，应该把前面比较难的 lab 当作 challenge。

内容六：参考文献

【1】现代操作系统 陈向群