操作系统实验报告 2

林祥，PB16020923

1. 主要步骤及核心代码解释说明

（1） Fork 的实现

a) 在 code/userprog/syscall.h 中添加

```
#define SC_Fork    16
ThreadId Fork();
```

说明：定义 Fork 的系统调用号，提供调用接口

b) 在 code/test/start.S 中添加

```
    .globl Fork
    .ent   Fork
Fork:
    addiu $2,$0,SC_Fork //在 r2 寄存器中放系统调用号
    syscall  //调用"系统中断"调用
    j   $31  //跳转到中断返回寄存器
    .end  Fork
```

说明：start.S 是一段汇编代码，它在编译时被加入用户程序，实现了用户程序使用系统调用时跳转到内核进行处理，并且把调用的参数放到寄存器中，其中 r2 寄存器存放系统调用号，r31 是中断返回寄存器。

c) 在 code/userprog/exception.cc 内的 ExceptionHandler 函数中的 switch(type)下的 case SC_Fork 添加：

```
    int id = SysFork();
    kernel->machine->WriteRegister(2, id);
    kernel->machine->PCplusPlus();
```

说明：这里进行中断入口处理，使用内核提供的函数 SysFork 来实现系统调用，得到子进程 id 后，通过 r2 寄存器传回。

d) 在内核中实现系统调用函数 SysFork

在 code/userprog/ksyscall.h 中添加：

```
int SysFork(){
    Thread *t=new Thread((char *)"New Thread");
    t->space=new AddrSpace();
    t->space->copyMemory(t->space,kernel->currentThread->space);
    t->SaveUserState();
    t->parent=kernel->currentThread;
```

```
    t->Fork((VoidFunctionPtr)forked,(void *)t);
    return t->tid;
}
void forked(int arg){
    Thread *t=(Thread *)arg;
    t->RestoreUserState();
    t->space->RestoreState();
    kernel->machine->WriteRegister(2,0);
    kernel->machine->PCplusPlus();
    kernel->machine->Run();
}
```

说明：这一步相当于在内核中来实现 SysFork 函数

i. 首先创建一个子进程，也就是创建一个 Thread 类对象 t（相当于 PCB，保存进程各种信息），其构造函数传入参数为进程名。

ii. 为这个子进程创建进程空间，t 的 space 指针指向一个新创建的 Addrspace 对象，Addrspace 类定义在 addrspace.h 和 addrspace.cc 中，然后使用 Addrspace 类中的 copyMemory 方法，复制父进程（也就是当前进程，即 kernel 类中的 currentThread）的进程空间到子进程的进程空间。

iii. 保存虚拟机的寄存器状态到 t 中，使用 Thread 类的 SaveUserState 方法可以实现，寄存器信息会被保存在 t 的 userRegisters 数组中

iv. 指定进程 t 的 parent 参数为父进程，建立父子关系。（为了实现这一步，应该在 Thread 类中添加一个成员 parent，类型为 Thread *）

v. 为子进程分配栈空间，通过使用 Thread 类的 Fork 方法实现，该方法有两个参数，一个是某个函数的指针，一个是函数的参数，Fork 方法会调用 StackAllocate 方法为子进程分配栈空间。这里函数指针传入 forked()作为子进程创建后的执行体，第二个参数可以随意，这里传入了子进程 t。在 forkerd()中，子进程 t 使用 Thread 类的 RestoreUserState 方法恢复 userRegisters 数组中的信息到虚拟机寄存器中，使用 Addrspace 类的 RestoreState 方法恢复内存页表信息，然后向虚拟机寄存器写入返回值 0，PC+4 后，使用 machinel 类的 Run 方法开始执行用户空间代码。

vi. 父进程返回子进程 id。

（2） Exec 的实现

a) 前 3 步添加系统调用与 Fork 类似，其中应该注意的是，在 exception.cc 中应该如下这样写。从寄存器读出的文件名地址不是 Linux 中的内存地址，而是 Nachos 中的，**应该用 Machine 类的 ReadMem 方法逐个字符读出到一个 buffer 中**。

具体地，在 code/userprog/exception.cc 内的 ExceptionHandler 函数中的 switch(type)下的 case SC_Exec 添加：

```
char exec_filename[128];
int exec_filename_addr; //address in nachos system,not in linux system
exec_filename_addr=(int)kernel->machine->ReadRegister(4);
for(int i1=0;i1<128;i1++){//copy memory from nachos to linux
    kernel->machine->ReadMem(exec_filename_addr++,1,(int*)&exec_filename[i1]);
    if(exec_filename[i1]==0) break; //end of text
}
SysExec(exec_filename);
```

b) 在内核中实现系统调用函数 SysExec

在 code/userprog/ksyscall.h 中添加：

```
void SysExec(char *fileName) {
    kernel->currentThread->setName(fileName);
    kernel->currentThread->space->reset();
    if(kernel->currentThread->space->Load(fileName)){
         kernel->currentThread->space->Execute();
    }
    else{
        system(fileName);
        kernel->currentThread->Finish();
    }
}
```

说明：这一步相当于在内核中来实现 SysExec 函数

i. 要载入新的程序之前，重置该进程的内存空间，使用 Addrspace 类的 reset 方法实现。

ii. 根据文件名将程序加载到内存，使用 Addrspace 类的 Load 方法实现。

iii. 如果加载成功，就使用 Addrspace 类的 Execute 方法开始执行进程，如果失败，就用 Thread 类的 Finish 方法结束进程。

（3） Join 的实现

a) 前 3 步添加系统调用与 Fork 类似，不过应该注意的是，**并没有一个 SysJoin 的内核函数，调用 Join 的对象应该是当前进程**，所以从虚拟机寄存器读出子进程 id 之后，应该调用当前进程的 Join 方法实现。

具体地，在 code/userprog/exception.cc 内的 ExceptionHandler 函数中的 switch(type) 下的 case SC_Join 添加：

```
    // 通过调用当前进程(父进程)的 join 函数，来等待指定的子进程
    int join_id;    //child id
    join_id=(int)kernel->machine->ReadRegister(4);
    kernel->currentThread->join(join_id);
    kernel->machine->PCplusPlus();
```

b) 这一步实现 Thread 类的 Join 方法

在 code/threads/thread.cc 中添加：

```
void Thread::join(int tid){
    Thread *childt=kernel->getThreadByID(tid);
    if(childt!=NULL){
        Semaphore *sem=new Semaphore((char*)"New Sem",0);
        this->joinSemMap_insert(tid,sem);
        sem->P();
        this->joinSemMap_remove(tid);
    }
}
void Thread::Finish(){
    …
    Thread *parent=this->parent;
    if(parent!=NULL){
        Semaphore *sem=parent->joinSemMap_getSemByID(this->tid);
        if(sem!=NULL) sem->V();
    }
    …
}
```

说明：

i. 为了方便操作，在 Kernel 中添加一个数据结构 ThreadMap，类型为 map<int，Thread *>，该数据结构将进程 id 和进程对象的指针一一对应，然后实现 3 个方法 addThread，removeThread，getThreadByID 分别用于插入，删除，查询一个键值对。

ii. 类似地，在 Thread 中添加一个数据结构 joinSemMap，类型为 map<int, Semaphore*>，也实现 3 个方法，方便查询父进程所有 join 的子进程对应的信号量。

iii. 在 Thread::join 中，首先根据子进程 id 获取子进程的 Thread 对象，值得注意的是，**应该判断该对象是否为 NULL（表明要等待的子进程不存在）**，如果不为 NULL，创建一个信号量，将信号量加入子进程 id 和信号量的键值对中，然后父进程等待该信号量。

iv. 在 Thread::Finish 中，该进程即将结束，通过 parent 指针获取父进程 Thread 对象进而获取信号量键值对，通过本进程 tid 查询对应信号量，释放该信号量。

（4） Shell 的实现

a) 这一步实现一个 Nachos 系统的 Shell，可以根据命令执行指定程序，根据 Shell.c 中已经实现的功能（获取输入，解析命令内容），应该在 main 的 while 大循环中添加

```
for(i = 0; i < cmdNum; i++){
    childID[i] = Fork();
    if(childID[i] == 0)
        Exec(cmdLine[i]+2);
}
for(i = 0; i < cmdNum; i++){
    Join(childID[i]);
}
```

说明：

i. 这里根据命令个数循环调用 Fork，判断返回值，如果为 0，表明为子进程，调用 Exec 载入指定程序。

ii. 循环调用 Join，等待所有子进程完成后 Shell 继续执行。

iii. 值得注意的是，这里实现并行执行程序，因此**并不能将 Join 和 Exec 写在同一个循环中**，否则实现的效果是执行一个程序，Shell 等待其结束后才执行下一个程序。

（5） 动态优先级调度的实现

a) 这一步实现动态优先级调度，需要补充的是 FindNextToRun 和 flushPriority 的实现，在 code/threads/scheduler.cc 中添加：

```
Thread *Scheduler::FindNextToRun(){
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    if (readyList->IsEmpty()) return NULL;
    else{
        kernel->currentThread->setPriority(kernel->currentThread->getPriority() -
(kernel->stats->totalTicks - lastSwitchTick) / 100);
        flushPriority();
        Print(); //打印当前进程状态。
        if(kernel->stats->totalTicks - lastSwitchTick; < MinSwitchPace)
            return NULL; // 间隔太小，返回 NULL （避免过分频繁地调度）
        if(readyList->IsEmpty())// 就绪队列为空，返回 NULL （没有就绪进程可以调度）
            return NULL;
       if(readyList->Front()->getPriority()>kernel->currentThread->getPriority()){
            lastSwitchTick = kernel->stats->totalTicks;
            return readyList->RemoveFront();
        }// 找到优先级最高的就绪态进程 t
        return NULL;
    }
}
```

说明：

i. FindNextToRun 实现寻找下一个可以调入 CPU 执行的程序

ii. 先判断就绪队列是否为空，不为空则更新当前进程的优先级（减去距离上一次调度的时间的 100 分之一），调用 flushPriority 更新其他就绪进程的优先级（加上一个固定的时间 AdaptPace），flushPriority 的具体实现如下：

```
void Scheduler::flushPriority(){
    ListIterator<Thread*> *iter = new ListIterator<Thread*>(readyList);
    for (; !iter->IsDone(); iter->Next()) {
        iter->Item()->setPriority(iter->Item()->getPriority() + AdaptPace);
    }
}
```

iii. 就绪队列是一个 SortedList 类型（继承了 List 类并实现有序插入）的，使用 List 类提供的迭代器更新每个就绪进程的优先级。

iv. 计算距离上一次调度的间隔，如果太小，返回 NULL，避免调度过于频繁。

v. 判断就绪队列中第一个进程的优先级是否高于当前进程，如果是，将该进程从就绪队列中移除，作为下一个可执行的进程。

vi. 关于动态优先级调度实现的其他相关说明：FindNextToRun 只是实现寻找下一个可以调入 CPU 执行的程序以及更新进程的优先级，一般地（例如在 Sleep 中），在调用了 FindNextToRun 之后，判断返回值是否为 NULL（表明当前不需要或者没有可以调度的进程），若不是 NULL，调用 ReadyToRun，置当前进程的状态为 READY，加入就绪队列尾部，然后调用 Run 执行下一个可以执行的进程。

（6） tid 的维护

i. 为了辨别每一个进程，在 Thread 类中添加 tid 成员，并在 thread.cc 的 Thread::Thread()中添加如下代码，表示创建进程时，先找到一个可用的 tid，然后向 kernel 的 threadMap 中添加键值对。

```
for(int i=0;i<MAX_THREAD_NUM;i++){
    if(kernel->getThreadByID(i)==NULL){
        this->tid=i;
        kernel->addThread(i,this);
        ThreadCount++;
        break;
    }
}
```

ii. 在 Thread::~Thread()中，调用 removeThread 方法移除键值对。

2. 实验运行结果截图及分析说明

a) Fork 测试程序执行结果

```
lin@ubuntu:~/nachos/NachOS-4.0/code/build.linux$ ./nachos -x ../test/fork

1. init var = "parent"
2. i am child. i change var = "child"
3. i am parent. my var = "parent"

Machine halting!

Ticks: total 335158, idle 0, system 33550, user 301608
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

解释说明：fork.c 的 main 函数中，先设置一个变量 var，值为“parent”，调用了 Fork 创建一个子进程，判断返回值，为 0 时表示子进程，置 var 为“child”，不为 0 时为父进程，不做什么事（实际上进行了大量空循环，只是为了拖延时间，等待子进程完成），因此，会出现上图的结果。

b) Exec 测试程序执行结果

```
lin@ubuntu:~/nachos/NachOS-4.0/code/build.linux$ ./nachos -x ../test/exec

1. i am child. i am runing 'add'.
   42 + 23 = 65
2. i am parent. i finished after my child

Machine halting!

Ticks: total 1823624, idle 0, system 182400, user 1641224
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

解释说明：exec.c 的 main 函数中，调用了 Fork 创建一个子进程，判断返回值，为 0 时表示子进程，输出一个子进程的信息，调用 Exec 载入”add”程序，实现一个加法运算；返回值不为 0 时为父进程，进行了大量空循环，拖延时间（确保子进程已经完成），输出一个父进程的信息，因此，会出现上图的结果。

c) Join 测试程序执行结果

```
lin@ubuntu:~/nachos/NachOS-4.0/code/build.linux$ ./nachos -x ../test/join

1. i am parent. i am waitting my childID=2
2. i am child. i am runing 'add'. please wait...
   42 + 23 = 65
3. parent finished

Machine halting!

Ticks: total 2447651, idle 0, system 244810, user 2202841
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

解释说明：join.c 的 main 函数中，调用了 Fork 创建一个子进程，判断返回值，为 0 时表示子进程，进行大量空循环，拖延时间（确保父进程已经输出信息），然后输出一个子进程的信息，调用 Exec 载入”add”程序，实现一个加法运算；返回值不为 0 时为父进程，输出一个父进程的信息后调用 Join，会等待子进程完成后才继续执行，输出第三条信息，因此，会出现上图的结果。

d) Shell 执行结果和动态优先级调度信息

如下图，shell.c 实现了按输入命令执行指定程序，多个程序并行时，按动态优先级调度，取 0schedule_info.txt 中的一段如下图

```
lin@ubuntu:~/nachos/NachOS-4.0/code/test$ ../build.linux/nachos -x shell

nachos >> ./2
i am testshell

nachos >> ./add
42 + 23 = 65

nachos >> ./sub
23 - 42 = -19

nachos >> ./2;./add;./sub;
23 - 42 = -19
42 + 23 = 65
i am testshell

nachos >>
```

```
35 -------------------------one switch-------------------------------
36 running: tid=3   name=2               status=RUNNING     pri=1157
37
38 Ready list contents:
39          tid=2   name=2               status=READY       pri=1159
40          tid=4   name=2               status=READY       pri=1157
41
42
43 -------------------------one switch-------------------------------
44 running: tid=2   name=2               status=RUNNING     pri=1158
45
46 Ready list contents:
47          tid=4   name=2               status=READY       pri=1159
48          tid=3   name=2               status=READY       pri=1159
49
50
51 -------------------------one switch-------------------------------
52 running: tid=4   name=2               status=RUNNING     pri=1158
53
54 Ready list contents:
55          tid=3   name=2               status=READY       pri=1161
56          tid=2   name=2               status=READY       pri=1160
57
58
59 -------------------------one switch-------------------------------
60 running: tid=3   name=2               status=RUNNING     pri=1160
61
62 Ready list contents:
63          tid=2   name=2               status=READY       pri=1162
64          tid=4   name=2               status=READY       pri=1160
```

分析说明：图中第一部分，显示了当前运行进程 tid=3，优先级为 1157，就绪队列中有 tid=2 和 tid=4 的进程，优先级分别为 1157 和 1159，由于 1159>1157，下一次调度时，tid=2 的进程被调入执行，并且 tid=3 的进程被加入就绪队列尾部。下面的过程同理。

3．实验过程中遇到的问题及解决方法

（1）遇到问题：

执行测试程序 fork 时提示了

```
lin@ubuntu:~/nachos/NachOS-4.0/code/build.linux$ make
g++ -g -Wall -I../network -I../filesys -I../userprog -I../threads -I../machine -I../lib -I- -DFILESYS_
STUB -DRDATA -DSIM_FIX -Dx86 -DLINUX -DCHANGED -m32 -c ../userprog/exception.cc
cc1plus: note: obsolete option -I- used, please use -iquote instead
In file included from ../userprog/exception.cc:27:0:
../userprog/ksyscall.h: In function ‘int SysFork()’:
../userprog/ksyscall.h:88:29: error: invalid conversion from ‘void (*)(int)’ to ‘VoidFunctionPtr {aka
void (*)(void*)}’ [-fpermissive]
     t->Fork(forked,(void *)t);
                             ^
In file included from ../threads/kernel.h:14:0,
                 from ../threads/main.h:13,
                 from ../userprog/exception.cc:25:
../threads/thread.h:112:8: error:   initializing argument 1 of ‘void Thread::Fork(VoidFunctionPtr, voi
d*)’ [-fpermissive]
   void Fork(VoidFunctionPtr func, void *arg);
        ^
```

解决方法：

Fork 原型为 void Fork(VoidFunctionPtr func, void *arg)

故第一个参数函数指针应该加上强制类型转换(VoidFunctionPtr)

（2）遇到问题：

执行测试程序 exec 时提示了 Unable to open file add

```
lin@ubuntu:~/nachos/NachOS-4.0/code/build.linux$ ./nachos -x ../test/exec

1. i am child. i am runing 'add'.
   Unable to open file add
exec cmd by Linux
sh: 1: add: not found
2. i am parent. i finished after my child

Machine halting!
```

分析问题：

经查，是因为测试程序 exec 中使用的文件路径为“add”，这是默认当前目录为测试程序所在目录 code/test，而我执行测试程序用的命令是“./nachos -x ../test/exec”，当前目录是在 code/build.linux 下，故找不到 add 文件。

解决方法：

只要进入 code/test 下用命令“../build.linux/nachos -x exec”即可或者修改 exec.c 的“add”为“../test/add”。

（3）遇到问题：

执行测试程序 join 时，出现加法结果后始终停留于此，线程没有结束

```
lin@ubuntu:~/nachos/NachOS-4.0/code/build.linux$ ./nachos -x ../test/join

1. i am parent. i am waitting my childID=2
2. i am child. i am runing 'add'. please wait...
   42 + 23 = 65
```

分析问题：

一开始我将信号量的 up 操作（Nachos 中即 sem->V()）写在了 Thread::~Thread()中，即 Thread 类对象被销毁时。后来我想起来 Nachos 中线程结束时并不销毁自己的内存空间，而由下一个执行的线程来销毁，所以这种写法将导致父进程一直等待子进程 up 信号量，而子进程的~Thread 需要等到切换到父进程才会执行，故子进程永远不能结束。

解决方法：

应该将信号量的 up 操作写在 Thread::Finish()中

（4）遇到问题：

某个 shell 命令执行后，不继续出现提示符（“nachos >>”），界面停留。

分析问题：

不继续出现提示符，说明某处出现了死循环，最有可能是 shell.c 中 for 循环里的 Join。检查 thread.cc 中的 void Thread::join(int tid)的实现，发现通过 getThreadByID 取得线程指针后，没有判断是否为空指针就设置了信号量，由于空指针表示子线程不存在，这将导致无限等待该信号量。出现空指针的原因是 shell.c 中 for 循环按输入命令的顺序 Join 每个子线程，但是有可能在执行到某个 Join 之前，对应的那个子线程已经结果，故实际上无需 Join，getThreadByID 将得到空指针。

解决方法：

增加判断 getThreadByID 是否返回空指针，不是才设置信号量。