

OS Assignment – PA4 Implementation of System Calls in Nachos

Name	SUID	Email
Rakshith Nallahally Shivanna:	928670713	rnallaha@syr.edu
Ravi Karan Anand:	648496556	ranand03@syr.edu

List of your files with directory name that you modified or created for this PA

Exception.cc, addspace.cc.

Brief description:

User programs invoke system calls by executing the MIPS syscall instruction, which causes the Nachos kernel exception handler to be invoked. The kernel must first tell the processor where the exception handler is by calling ExceptionHandler. The default Kernel exception handler reads the value of the processor's cause register, determines the current process, and invokes handleException on the current process, passing the cause of the exception as an argument.

Write System Call: To implement Write system call first we read the register value from MIPS registers, register 4 holds the address, register 5 contains length of the write, register 6 contains the console (0 for read, 1 for write). Kernel allows only getchar, putchar functionality to read and write from the console, these are implemented under synchconsole.h and synchconsole.cc in /userprog. We loop through the length read in the register, we read the data from mainmemory (if page table not implemented), readmem if pagetable is implemented and write to the console using putchar. Finally, we return the length of the write to the output register (r2). We increment the program counter to point to the next instructions.

case SC_Write:

```
DEBUG(dbgSys, "syscall:write: Hello: pid=" << kernel->currentThread->pid << ", args={" << kernel->machine->ReadRegister(4) << ", " << kernel->machine->ReadRegister(5) << ", " << kernel->machine->ReadRegister(6) << " }\n");
```

```
{
```

```
int t2=0, tchi;
```

```
char tch2;
```

```
addr_write = (int) kernel->machine->ReadRegister(4); //addr of string to write
```

```
len_write = (int) kernel->machine->ReadRegister(5); //no of charaters to write
```

```
write_id = (int) kernel->machine->ReadRegister(6); //file descriptor
```

```
DEBUG(dbgSys, "syscall:write: Entering Write Systemcall" << " \n");
```

```
for(t2=0; t2<len_write; t2++ ) {
```

//tch2=kernel->machine->mainMemory[addr_write+t2]; // perhaps will change when pagetables is implemented, and use of readmem and writemem

```
kernel->machine->ReadMem(addr_write+t2,1,&tchi);
```

```
tch2=(char)tchi;
```

```
//IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
```

```
kernel->synchConsoleOut->PutChar(tch2);
```

```
//kernel->interrupt->SetLevel(oldLevel);
```

```
}
```

```
kernel->machine->WriteRegister(2,len_write);
```

```
/* Modify return point */
```

```
{
```

```
/* set previous programm counter (debugging only)*/
```

```
kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
```

```
/* set programm counter to next instruction (all Instructions are 4 byte wide)*/
```

```
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
```

```
/* set next programm counter for brach execution */
```

```
kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg) + 4);
```

```
}
```

```
DEBUG(dbgSys, "syscall:write: Leaving Write Systemcall" << " \n");
```

```
}
```

```
return;
```

```
ASSERTNOTREACHED();
```

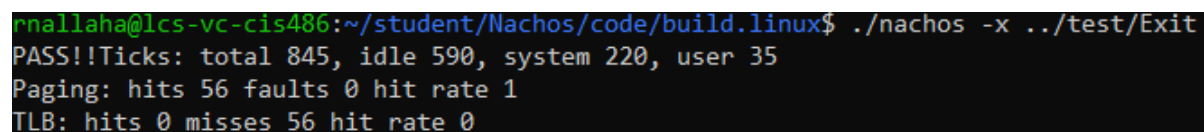
```
break;
```

```
rnallaha@lcs-vc-cis486:~/student/Nachos/code/build.linux$ ./nachos -x ../test/Write
PASS!!Ticks: total 850, idle 590, system 220, user 40
Paging: hits 63 faults 0 hit rate 1
TLB: hits 0 misses 63 hit rate 0
```

Exit System Call:

case SC_Exit: We call `currentThread->Finish()` to end the execution of the current process. If there are more than two processes in the process table we yield the process to give control back to the scheduler for future execution and relinquish the resources in use currently.

```
    DEBUG(dbgSys, "syscall:exit: Entering Exit Systemcall" << " \n");
    {
        int exitretval = (int) kernel->machine->ReadRegister(4); // exit status
        if ( kernel->currentThread->pid == 0 ) {
            // usually 2 threads are left :main-thread and postal-worker-thread
            while(kernel->ProcessTable->NumInList(>2) {
                DEBUG(dbgSys,"syscall:exit: main process waiting on other threads #threads=" << kernel->ProcessTable->NumInList() << "\n");
                kernel->currentThread->Yield(); // calls Sleep(true)
            }
            kernel->stats->Print();
        }
        DEBUG(dbgSys, "syscall:exit: Calling Thread->Finish() on pid=" << kernel->currentThread->pid << " \n");
        kernel->currentThread->Finish(); // calls Sleep(true)
        kernel->machine->WriteRegister(2, exitretval);
        DEBUG(dbgSys, "syscall:exit: Leaving Exit Systemcall" << " \n");
    }
    return;
    ASSERTNOTREACHED();
    break;
```



```
rnallaha@lcs-vc-cis486:~/student/Nachos/code/build.linux$ ./nachos -x ../test/Exit
PASS!!Ticks: total 845, idle 590, system 220, user 35
Paging: hits 56 faults 0 hit rate 1
TLB: hits 0 misses 56 hit rate 0
```

Read System call: Similar to write system call we read the data related to address, length and console to use from MIPS registers. We loop through the length of data read read the characters from the console using `getchar()` and write it to `mainMemory`, we then write the number of characters read to register 2.

```

case SC_Read:

    DEBUG(dbgSys, "syscall:read: Hello2: args={" << kernel->machine->ReadRegister(4) << ", " <<
kernel->machine->ReadRegister(5) << ", " << kernel->machine->ReadRegister(6) << "} \n");

    {

        int t1 ; //, sizeread=0 ; //read(arg3,buf,arg2);

        char tch1;

        addr_read = (int) kernel->machine->ReadRegister(4); //addr of string to store
        len_read = (int) kernel->machine->ReadRegister(5); //no of charaters to read
        read_id = (int) kernel->machine->ReadRegister(6); //file descriptor

        DEBUG(dbgSys, "Entering Read Systemcall" << " \n");

//    DEBUG(dbgSys, "Read: Hello2: " << arg1 << ", " << arg2 << ", " << arg3 << " \n");

        t1=0;

        while(t1<len_read && tch1 != '\n' ) {

            tch1=kernel->synchConsoleIn->GetChar();

            kernel->machine->mainMemory[addr_read+t1] = tch1; // when TLB functionaility is done,
readmeme, writemem can be used

            //buf[sizeread++]=tch;

            t1++;

        }

        kernel->machine->mainMemory[t1]='\0';

        if(t1<0) DEBUG(dbgSys,"Error reading from file\n");

        DEBUG(dbgSys, "Read: Hello: " << t1 << " \n");

        kernel->machine->WriteRegister(2,t1);

        {

            /* set previous programm counter (debugging only)*/

            kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));

            /* set programm counter to next instruction (all Instructions are 4 byte wide)*/

            kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);

            /* set next programm counter for brach execution */

```

```

kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg) + 4);
}

DEBUG(dbgSys, "syscall:read: Leaving Read Systemcall" << " \n");

}

return;

ASSERTNOTREACHED();

break;

```

```

rnallaha@lcs-vc-cis486:~/student/Nachos/code/build.linux$ ./nachos -x ../test/Read
>>Syracuse University CIS657
PASS!!  Reading is Syracuse University CIS657

Ticks: total 232950594, idle 232946131, system 3670, user 793
Paging: hits 1052 faults 0 hit rate 1
TLB: hits 0 misses 1052 hit rate 0

```

Exec System Call:

case SC_Exec: In exec function call, we are loading the executable provided in command line to the current thread, before that we are deleting the current executable by deleting currentThread->space, once the new executable is loaded into the thread space, we execute it using currentThread->space->Execute().

```

DEBUG(dbgSys, "syscall:exec Entering Exec SystemCall" << " \n");

{

char buf[40], tch=1;

int i;

for (i=0; (i<40) && (tch!='\0') ;i++)

{

tch = kernel->machine->mainMemory[kernel->machine->ReadRegister(4)+i];

buf[i]= tch;

}

buf[i] = '\0';)

//delete the current thread pointed to address space

delete kernel->currentThread->space;

```

```

//call load function to allocate new space and load new binary

kernel->currentThread->space = new AddrSpace ;

kernel->currentThread->space->Load(buf);

kernel->currentThread->space->RestoreState(); // set the kernel page table

DEBUG(dbgSys, "Hello 2 " << kernel->machine->ReadRegister(4) << " \n");

{
/* set previous programm counter (debugging only)*/
kernel->machine->WriteRegister(PrevPCReg, 0);

/* set programm counter to next instruction (all Instructions are 4 byte wide)*/
kernel->machine->WriteRegister(PCReg, 4);

/* set next programm counter for brach execution */
kernel->machine->WriteRegister(NextPCReg, 4);
}

DEBUG(dbgSys, "syscall:exec: Calling addrspace->execute " << " \n");

kernel->currentThread->space->Execute();

DEBUG(dbgSys, "syscall:exec: Leaving Exec Systemcall" << " \n");

}

return;

ASSERTNOTREACHED();

break;

```

```

rnallaha@lcs-vc-cis486:~/student/Nachos/code/build.linux$ ./nachos -x ../test/Exec
PASS!!Ticks: total 879, idle 590, system 220, user 69
Paging: hits 115 faults 0 hit rate 1
TLB: hits 0 misses 115 hit rate 0

```

Fork System call:

To implement fork system call, we have to create a new thread, create its own space for that we have implemented the copy constructor which finds the free page in the physical memory and adds it to the threads page table. We then setup the registers for the child thread and save the state, including the return value. We then call the simulators thread fork which will take two arguments one of which is the function from which the new thread starts this is implemented in **stacktop function**, the job of stacktop function is to restore the register states for the thread before calling machine->run which will run the MIPS instructions in the thread. The logic to find the free page in the kernel page table is written in copy

constructor. We are doing this by creating a static array of length 128, initialize it to flag 0, whenever the page is allocated flag is set as 1, when the thread finished the pages are deallocated and corresponding flags are set to 0, this is done by the destructor of the thread. Once we call the fork function scheduler will pickup the child thread from its process table. Both parent and child process execute concurrently and write the appropriate return values(pid) to the output register 2.

```
case SC_SysFork:
    DEBUG(dbgSys, "SysFork Initiated" << " \n");
    {
        IntStatus oldLevel;
        int newthpid=1;
        //kernel->currentThread->SaveUserState();
        DEBUG(dbgSys,"syscall:fork: A: parent pid=" << (int)kernel->currentThread->pid << "\n");
        DEBUG(dbgSys,"syscall:fork: A: parent space=" << (int)kernel->currentThread->space << "\n");
        DEBUG(dbgSys,"syscall:fork: A: " << " REGISTER: PC=" << kernel->machine->ReadRegister(PCReg) <<
"\n");
        DEBUG(dbgSys,"syscall:fork: A: " << " REGISTER: Stack=" << kernel->machine->ReadRegister(StackReg)
<< "\n");
        DEBUG(dbgSys,"syscall:fork: A: " << " REGISTER: RetAddr=" << kernel->machine-
>ReadRegister(RetAddrReg) << "\n");
        DEBUG(dbgSys,"syscall:fork: A: " << " REGISTER: [2]=" << kernel->machine->ReadRegister(2) << "\n");
        /* set previous programm counter (debugging only)*/
        kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
        /* set programm counter to next instruction (all Instructions are 4 byte wide)*/
        kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
        /* set next programm counter for brach execution */
        kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg) + 4);
        Thread* newthread = new Thread("Child Thread");
        //Allocate an address space for the child thread
        newthpid=newthread->pid;
        newthread->space = new AddrSpace(*kernel->currentThread->space);
        DEBUG(dbgSys,"syscall:fork: child pid=" << newthread->pid << "\n");
```

```

    DEBUG(dbgSys,"syscall:fork: child space=" << newthread->space << "\n" );

    kernel->machine->WriteRegister(2, 0);

    //save register states of the child process so that it can restore it later

    newthread->SaveUserState(); // save the registers as part of the thread for later loading

    kernel->machine->WriteRegister(2, newthpid);

    //kernel->currentThread->SaveUserState();

    DEBUG(dbgSys,"syscall:fork: B: parent pid=" << (int)kernel->currentThread->pid << "\n");

    DEBUG(dbgSys,"syscall:fork: B: parent space=" << kernel->currentThread->space << "\n");

    DEBUG(dbgSys,"syscall:fork: B: " << "REGISTER: PC=" << kernel->machine->ReadRegister(PCReg) <<
"\n");

    DEBUG(dbgSys,"syscall:fork: B: " << "REGISTER: Stack=" << kernel->machine->ReadRegister(StackReg)
<< "\n");

    DEBUG(dbgSys,"syscall:fork: B: " << "REGISTER: RetAddr=" << kernel->machine-
>ReadRegister(RetAddrReg) << "\n");

    DEBUG(dbgSys,"syscall:fork: B: " << "REGISTER: [2]=" << kernel->machine->ReadRegister(2) << "\n");

    newthread->Fork( (VoidFunctionPtr)stacktopfunc, (void *)newthpid);

    DEBUG(dbgSys,"syscall:fork: Leaving Fork SystemCall pid=" << kernel->currentThread->pid << "\n");

    }

    return;

    ASSERTNOTREACHED();

    break;

void stacktopfunc(void *arg) {

    DEBUG(dbgSys,"stacktopfunc: Entered void stacktopfunc(void *) \n");

    kernel->currentThread->RestoreUserState(); // restore machine registers

    kernel->currentThread->space->RestoreState(); // restore kernel->pageTable

    DEBUG(dbgSys,"stacktopfunc: arg=" << (int)arg << "\n");

    DEBUG(dbgSys,"stacktopfunc: pid=" << kernel->currentThread->pid << "\n");

    DEBUG(dbgSys,"stacktopfunc: space=" << kernel->currentThread->space << "\n");

    DEBUG(dbgSys,"stacktopfunc: PageTable[0].physicalPage=" << kernel->currentThread->space-
>getPageTable()[0].physicalPage << "\n");

```



```

    DEBUG(dbgSys,"stacktopfunc: " << "REGISTER: PC=" << kernel->machine->ReadRegister(PCReg) <<
"\n");

    DEBUG(dbgSys,"stacktopfunc: " << "REGISTER: Stack=" << kernel->machine->ReadRegister(StackReg)
<< "\n");

    DEBUG(dbgSys,"stacktopfunc: " << "REGISTER: RetAddr=" << kernel->machine-
>ReadRegister(RetAddrReg) << "\n");

    DEBUG(dbgSys,"stacktopfunc: " << "REGISTER: [2]=" << kernel->machine->ReadRegister(2) << "\n");

    DEBUG(dbgSys,"stacktopfunc: " << "Entering machine->Run()" << "\n");

    kernel->machine->Run();

    DEBUG(dbgSys,"stacktopfunc: " << "Returned from machine->Run()" << "\n");
}

```

```

AddrSpace::AddrSpace()

```

```

{
    static int initonce=0;

    DEBUG(dbgAddr,"AddrSpace: constructor");

    if (initonce==0){
        for (int i = 0; i < 128; i++) {
            pagetracker[i] = 0;
        } // zero out the entire address space

        initonce=1;
    }

    numPages=0;
    pageTable=NULL;

    //bzero(kernel->machine->mainMemory, MemorySize);
}

```

```

//-----
// AddrSpace::~AddrSpace
//    Dealloate an address space.
//-----

```

```

AddrSpace::~~AddrSpace()
{
    DEBUG(dbgAddr,"AddrSpace: destructor");
    for (int i =0; i < this->getNumPage(); i++)
        pagetracker[table[i].physicalPage] = 0 ;
    delete pageTable;
}

int
AddrSpace::allocatefreepage(int start){
    for (unsigned int i = start ; i < NumPhysPages ; i++){
        DEBUG(dbgAddr, "allocatefreepage: pagetracker[" << i << "] " << pagetracker[i] << "\n" );
        if (pagetracker[i] == 0)
        {
            DEBUG(dbgAddr, "allocatefreepage: Free page found:" << i << "\n" );
            return i;
        }
    }
    //panic
    DEBUG(dbgAddr, "allocatefreepage: Error found: PANIC: Free page nor found:" << "\n" );
    printf("allocaefreepage: Error found : Free page not found\n");
    return -1;
}

AddrSpace::AddrSpace(const AddrSpace& copiedItem) { // copy constructor
    int f_page=0;
    const TranslationEntry * oldpageTable = copiedItem.getPageTable();
    int phys_page_new_addr, phys_page_old_addr;
    DEBUG(dbgAddr,"AddrSpace: constructor (Copy Constructor)\n");

```

```

numPages=copiedItem.getNumPage();
pageTable = new TranslationEntry[numPages];
for (int i = 0; i < numPages; i++) {
    DEBUG(dbgAddr, "AddrSpace: allocating page << " << i << "\n");
    pageTable[i].virtualPage = i;
    f_page = allocatefreepage(f_page);
    if(f_page== -1)
        DEBUG(dbgAddr, "AddrSpace: Error: Page not found \n");
    pageTable[i].physicalPage = f_page; //now holds a free page
    pagetracker[f_page]=1; // note that page is no longer available in pagetracker
    DEBUG(dbgAddr, "AddrSpace: pagetracker[" << f_page << "]=" << pagetracker[f_page] << "\n");
    f_page++;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;

    // copy the contents of the page
    phys_page_new_addr = pageTable[i].physicalPage;
    phys_page_old_addr = oldpageTable[i].physicalPage;
    DEBUG(dbgAddr, "copy constructor: page from " << phys_page_old_addr << " to " <<
phys_page_new_addr << "\n");
    for (int j = 0 ; j < 128 ; j++)
    {
        // kernel->machine->mainMemory[phys_page_new_addr+j] = kernel->machine-
>mainMemory[phys_page_old_addr+j]; // << BUG!! WRONG

        kernel->machine->mainMemory[phys_page_new_addr*PageSize+j] = kernel->machine-
>mainMemory[phys_page_old_addr*PageSize+j];
    }
}
}

```

```
DEBUG(dbgAddr, "AddrSpace: copy construtor DONE\n");
```

```
rnallaha@lcs-vc-cis486:~/student/Nachos/code/build.linux$ ./nachos -x ../test/Fork
parent processchild processTicks: total 3783, idle 1356, system 2320, user 107
Paging: hits 195 faults 0 hit rate 1
TLB: hits 0 misses 195 hit rate 0
```