CSCI 447 - Operating Systems, Winter 2020
Assignment 4: User-Level Processes

Due Date: Friday, February 7, 2020
Points: 150

1 Overview and Goal

In this assignment, you will explore user-level processes. You will create a single process, running in its own address space. When this user-level process executes, the CPU will be in ”user mode.”

The user-level process will make system calls to the kernel, which will cause the CPU to switch into ”system mode.” Upon completion, the CPU will switch back to user mode before resuming execution of the user-level process.

The user-level process will execute in its own ”logical address space.” Its address space will be broken into a number of ”pages” and each page will be stored in a frame in memory. The pages will be resident (i.e., stored in frames in physical memory) at all times and will not be swapped out to disk in this assignment. (Contrast this with ”virtual” memory, in which some pages may not be resident in memory.)

The kernel will be entirely protected from the user-level program; nothing the user-level program does can crash the kernel.

2 Download New Files

The files for this assignment are available in:
`https://facultyweb.cs.wwu.edu/~phil/classes/w20/447/a4`

First, make sure you are on branch master. Then get copies of the following files. For files with the same name as you currently have on your master branch, just copy the files over the current version. Once you have all these files copied, commit them all. The following files remain the same from assignment 3:

```
BitMap.h
BitMap.k
List.h
List.k
Main.h
Runtime.s
Switch.s
Syscall.h
Syscall.k
```

```
System.h
System.k (except change HEAP_SIZE to 20000)
```

The following file was distributed for assignment 3, but you need a new copy for assignment 4:

```
Main.k
```

The following files are new for assignment 4:

```
MyProgram.h
MyProgram.k
TestProgram1.h
TestProgram1.k
TestProgram2.h
TestProgram2.k
UserRuntime.s
UserSystem.h
UserSystem.k
FileStuff.h (does not stand on its own, do not add)
FileStuff.k (does not stand on its own, do not add)
```

Kernel.h and Kernel.k are not provided. You start with the final version from your last assignment. It should be the version on your master branch.

3 Merging New ”File Stuff” Code

For this assignment, we are distributing additional code which you should add to the Kernel package. It is found in the files “FileStuff.h” and “FileStuff.k” Please add the material in FileStuff.k to the end of file Kernel.k. It should be inserted directly before the final endCode keyword. Also, please add the material in FileStuff.h to the end of file Kernel.h. It should be inserted directly before the final endHeader keyword. Do not commit the FileStuff.k/h files to your repo other than in your Kernel.k/h files.

This code adds the following classes:

```
DiskDriver
FileManager
ToyFs
InodeData
FileControlBlock
OpenFile
```

You will use these classes, but you should not modify them yet.

There will be a single DiskDriver object (called diskDriver) which is created and initialized at start-up time. There will be a single ToyFS object (called fileSystem) which is created and initialized at start-up time. There will be a single FileManager object (called fileManager) which is created and initialized at start-up time. The new main function contains statements to create and initialize the diskDriver, fileSystem and the fileManager objects.

FileControlBlock and OpenFile objects will be handled much like Threads and ProcessControlBlocks. They are a limited resource. A limited supply is created at start-up time and then they are managed by the fileManager. There is a free list of FileControlBlock objects and a free list of OpenFile objects. The fileManager oversees both of these free lists. Threads may make requests and may return resources, by invoking methods in the fileManager.

The diskDriver object encapsulates all the hardware specific details of the disk. It provides a method that allows a thread to read a sector from disk into a memory frame and it provides a method that writes a frame from memory to a sector on disk. The InodeData class is used by the file system in connection with files. For this assignment, the file system has enough functionality to allow your kernel to execute user programs from the DISK.

4 Other Changes To Your Kernel Code

Please make the following changes to your copy of Kernel.h:

Change

```
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 35            -- for testing only
```

to:

```
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 100           -- for testing only
```

Change

```
--diskDriver: DiskDriver
--fileManager: FileManager
--fileSystem: ToyFs
```

to:

```
diskDriver: DiskDriver
fileManager: FileManager
fileSystem: ToyFs
```

Add a function prototype for the function StartUserProcess. You can add it after the other function prototypes:

Change

```
ProcessFinish (exitStatus: int)
InitFirstProcess ()
```

to:

```
ProcessFinish (exitStatus: int)
InitFirstProcess ()
StartUserProcess (arg: int)
```

Also, add an empty function body for StartUserProcess to Kernel.k. You will complete this function as part of creating your first user process.

For the final change to Kernel.k, change the DiskInterruptHandler function from:

```
FatalError ("DISK INTERRUPTS NOT EXPECTED IN PROJECT 4")
```

to:

```
currentInterruptStatus = DISABLED
-- print ("DiskInterruptHandler invoked!\n")
if diskDriver.semToSignalOnCompletion
   diskDriver.semToSignalOnCompletion.Up()
endIf
```

5 Task 1: First user-level process

Your first task is to load and execute the user-level program called MyProgram. Since the user-level program must be read from a file on the BLITZ disk, you'll first need to understand how the BLITZ disk works, how files are stored on the disk, and how the FileManager and ToyFs code works.

"MyProgram" invokes the SystemShutdown syscall, which you'll need to implement.

6 Task 2: Syscall Handlers

Modify all the syscall handlers so they print the arguments that are passed to them. In the case of integer arguments, this should be straightforward, but the following syscalls take a pointer to an array of char as one of their arguments.

```
Exec
Open
```

This pointer is in the user-program's logical address space. You must first move the string from user-space to a buffer in kernel space. Only then can it be safely printed.

Also, some of the syscalls return a result. You must modify the handlers for these syscalls so that the following syscalls return these values. (These are just arbitrary values, to make sure you can return something.)

`Fork`	`1000`
`Join`	`2000`
`Exec`	`3000`
`Open`	`5000`
`Read`	`6000`
`Write`	`7000`
`Seek`	`8000`

For this task, you should modify only the handler methods (e.g., Handle_Sys_Fork, Handle_Sys_Join, etc.) You should not modify SyscallTrapHandler or the wrapper functions in UserSystem.

7 Task 3: Exec syscall

Implement the Exec syscall. The Exec syscall will read a new executable program from disk and copy it into the address space of the process which invoked the Exec. It will then begin execution of the new program. Unless there are errors, there will not be a return from the Exec syscall.

8 Task 4: Command line arguments

Implement command line arguments to the Exec syscall. (This documented near the end of this write-up.)

9 The User-Level View

First, let's look at our operating system from the users' point of view. User-level programs will be able to invoke the following system calls:

```
GetError
Exit
Shutdown
Yield
Fork
```

```
Join
Exec
Open
Read
Write
Seek
Close
```

(This is some of the grand plan for our OS; most of these system calls will not be implemented in this assignment.)

These syscalls are quite similar to kernel syscalls of the same names in Unix. We describe their precise functionality later.

A user-level program will be written in KPL and linked with the following files:

```
UserSystem.h
UserSystem.k
UserRuntime.s
Syscall.h
Syscall.k
```

We are providing a sample user-level program in MyProgram.h and MyProgram.k.

The UserSystem package includes a wrapper (or "jacket") function for each of the system calls. Here are the names of the wrapper functions. There is a one-to-one correspondence between the system calls and the wrapper functions.

System call	Wrapper function name
GetError	Sys_GetError
Exit	Sys_Exit
Shutdown	Sys_Shutdown
Yield	Sys_Yield
Fork	Sys_Fork
Join	Sys_Join
Exec	Sys_Exec
Open	Sys_Open
Read	Sys_Read
Write	Sys_Write
Seek	Sys_Seek
Close	Sys_Close

(In Unix, the wrapper function often has the same name as the syscall. All wrapper functions have names beginning with Sys_ just to help make the distinction between wrapper and syscall.)

Each wrapper function works the same way. It invokes an assembly language routine called DoSyscall, which executes a "syscall" machine instruction. When the kernel call finishes, the DoSyscall function simply returns to the wrapper function, which returns to the user's code.

Arguments may be passed to and from the kernel call. In general, these are integers and pointers to memory. The wrapper function works with DoSyscall to pass the arguments. When the wrapper function calls DoSyscall, it will push the arguments onto the stack. The DoSyscall will take the arguments off the stack and move them into registers. Since it runs as a user-level function, it places them in the "user" registers. (Recall that the BLITZ machine has a set of 16 "system registers" and a set of 16 "user registers.")

Each wrapper function also uses an integer code to indicate which kernel function is involved. Here is the enum giving the different codes. For example, the code for "Fork" is 7.

```
enum SYSCALL_EXIT = 1,
     SYSCALL_GETERROR,
     SYSCALL_SHUTDOWN,
     SYSCALL_YIELD,
     SYSCALL_GETPID,
     SYSCALL_GETPPID,
     SYSCALL_FORK,
     SYSCALL_JOIN,
     SYSCALL_EXEC,
     SYSCALL_OPEN,
     SYSCALL_READ,
     SYSCALL_WRITE,
     SYSCALL_SEEK,
     SYSCALL_CLOSE, ...
```

These code numbers are used both by the user-level program and by the kernel. These numbers are defined in the file Syscall.h. Syscall.k contains a single function we will use in a future assignment.

As an example, here is the code for the wrapper function for "Read." It simply invokes DoSyscall and returns whatever DoSyscall returns.

```
function Sys_Read (fileDesc: int,
                   buffer: ptr to char,
                   sizeInBytes: int) returns int
    return DoSyscall (SYSCALL_READ,
                      fileDesc,
                      buffer asInteger,
                      sizeInBytes,
                      0)
  endFunction
```

Here is the function prototype for DoSyscall:

```
external DoSyscall (funCode, arg1, arg2, arg3, arg4: int) returns int
```

The DoSyscall routine is set up to deal with up to 4 arguments. Since the Read syscall only needs 3 arguments, the wrapper function must supply an extra zero for the fourth argument.

DoSyscall treats all of its arguments as untyped words (i.e., as int), so the wrapper functions must coerce the types of the arguments if they are not int. Whatever DoSyscall returns, the wrapper function will return.

DoSyscall is in UserRuntime.s, which will be linked with all user programs. The code is given next.

It moves each of the 4 arguments into registers r1, r2, r3, and r4. It then moves the function code into register r5 and executes the syscall instruction. It assumes the kernel will place the result (if any) in r1, so after the syscall instruction, it moves the return value from r1 to the stack, so that the wrapper function can retrieve it.

```
DoSyscall:
       load     [r15+8],r1        ! Move arg1 into r1
       load     [r15+12],r2       ! Move arg2 into r2
       load     [r15+16],r3       ! Move arg3 into r3
       load     [r15+20],r4       ! Move arg4 into r4
       load     [r15+4],r5        ! Move funcCode into r5
       syscall  r5                ! Do the syscall
       store    r1,[r15+4]        ! Move result from r1 onto stack
       ret                        ! Return
```

Some of the kernel routines require no arguments and/or return no result. As an example, consider the wrapper function for Yield. The compiler knows that DoSyscall returns a result, so it insists that we do something with this value. The wrapper function simply moves it into a variable and ignores it.

```
function Sys_Yield ()
    var ignore: int
    ignore = DoSyscall (SYSCALL_YIELD, 0, 0, 0, 0)
  endFunction
```

Here is a list of some of the wrapper functions, including their arguments and return types.

```
Sys_Exit (returnStatus: int)
Sys_GetError () returns int
Sys_Shutdown ()
Sys_Yield ()
```

```
    Sys_Fork () returns int
    Sys_Join (processID: int) returns int
    Sys_Exec (filename: String, args: ptr to array of ptr to array of char)
             returns int
    Sys_Open (filename: String, flags, mode: int ) returns int
    Sys_Read (fileDesc: int, buffer: ptr to char, sizeInBytes: int)
             returns int
    Sys_Write (fileDesc: int, buffer: ptr to char, sizeInBytes: int)
              returns int
    Sys_Seek (fileDesc: int, newCurrentPos: int) returns int
    Sys_Close (fileDesc: int)
```

In addition to the wrapper functions, the UserSystem package contains a few other routines that support the KPL language. These are more-or-less duplicates of the same routines in the System package. Likewise, some of the material from Runtime.s is duplicated in UserRuntime.s. This duplication is necessary because user-level programs cannot invoke any of the routines that are part of the kernel.

For example the functions print, printInt, nl, etc. have been duplicated at the user level so the user-level program has the ability to print.

[Note that, at this point, all printing is done by cheating, using a "trapdoor" in the emulator. Normally, a user-level program would need to invoke syscalls (such as Sys_Write) to perform any output, since user-level programs can't access the I/O devices directly. However, since we are not yet ready to address questions about output to the serial device, we are including these cheater print functions, which rely on a trapdoor in the emulator.]

Every user-level program needs to "use" the UserSystem package and be linked with the UserRuntime.s code. For example:

```
        MyProgram.h
        header MyProgram
          uses UserSystem
          functions
            main ()
        endHeader

        MyProgram.k
        code MyProgram
          function main ()
              print ("My user-level program is running!\n")
              Sys_Shutdown ()
            endFunction
        endCode
```

Here are the commands to prepare a user-level program for execution. The makefile has been modified to include these commands.

```
asm UserRuntime.s
kpl UserSystem -unsafe
asm UserSystem.s
kpl MyProgram -unsafe
asm MyProgram.s
kpl Syscall
asm Syscall.s
lddd UserRuntime.o UserSystem.o MyProgram.o Syscall.o -o MyProgram
```

Note that there is no connection with the kernel. The user-level programs are compiled and linked independently. All communication with the kernel will be through the syscall interface, via the wrapper functions.

This is exactly the way Unix works. For user-level programs, library functions and wrapper functions are brought into the "a.out" file at link-time, as needed. This explains why a seemingly small "C" program can produce a rather large "a.out" executable. One small use of printf in a program might pull in, at link-time, more output formatting and buffering routines than you can possibly imagine.

When an OS wants to execute a user-level program, it will go to a disk file to find the executable. Then it will read that executable into memory and start up the new process.

In order to execute MyProgram, we need to introduce the BLITZ "disk." The disk is simulated with a Unix file called "DISK." After the user-level program is compiled, it must be placed on the BLITZ disk with the following Unix commands:

```
toyfs -i -n num_inodes -s num_sectors
toyfs -a -x MyProgram MyPRogram
```

The first command creates an empty file system on the a file called DISK. The arguments "num_inodes" and "num_sectors" should be actual numbers in the command. For example, for this assignment the makefile creates the DISK with the command:

```
toyfs -i -n10 -s250
```

The second command copies a file from the Unix file system to the BLITZ disk and makes it an executable file. It creates a directory entry and moves the data to the proper place on the simulated BLITZ disk. The makefile has the commands to create the BLITZ disk.

Once the kernel is running, it will read the file from the simulated BLITZ disk and copy it into memory.

10 The Syscall Interface

In our OS, each process will have exactly one thread. A process may also have several open files and can do I/O via the Read and Write syscalls. The I/O will go to the BLITZ disk. For now, there is no serial (i.e., terminal) device.

Next we describe each syscall in more detail.

```
function Sys_Exit (returnStatus: int)
```

This function causes the current process and its thread to terminate. The returnStatus will be saved so that it can be passed to a Sys_Join executed by the parent process. This function never returns.

```
function Sys_GetError () returns int
```

This function returns a system error code. When a system call detects an error, it stops running the system code and returns an error value from the system call. (System calls like Sys_Shutdown and Sys_Yield do not have any error cases and so won't set an error code.) If a system call returns an error value, the user code may ask what went wrong by calling the Sys_GetError system call.

```
function Sys_Shutdown ()
```

This function will cause an immediate shutdown of the kernel. It will not return.

```
function Sys_Yield ()
```

This function yields the CPU to another process on the ready list. Once this process is scheduled again, this function will return. From the caller's perspective, this routine is similar to a "nop."

```
function Sys_Fork () returns int
```

This function creates a new process which is a copy of the current process. The new process will have a copy of the virtual memory space and all files open in the original process will also be open in the new process. Both processes will then return from this function. In the parent process, the pid of the child will be returned; in the child, zero will be returned.

```
function Sys_Join (processID: int) returns int
```

This function causes the caller to wait until the process with the given pid has terminated, by executing a call to Sys_Exit. The returnStatus passed by that process to Sys_Exit will be returned from this function. If the other process invokes Sys_Exit first, this returnStatus will be saved until either its parent executes a Sys_Join naming that process's pid or until its parent terminates. If an error occurs, this function returns -1.

```
    function Sys_Exec (filename: String, args: ptr to array
                       of ptr to array of char) returns int
```

This function is passed the name of a file. That file is assumed to be an executable file. It is read into memory, overwriting the entire address space of the current process. Then the OS will begin executing the new process. Any open files in the current process will remain open and unchanged in the new process. Normally, this function will not return. If there are problems, this function will return -1. Initially, you will ignore the args parameter and just run the new executable file. (At the end of the assignment you will be asked to implement the args parameter and we will discuss that argument.)

```
    function Sys_Open (filename: String, flags, mode: int ) returns int
```

This function opens a file. The flags, found in Syscall.h, represent how the file must be opened and the file's permissions must match the requested open flags. O_READ requests to read, O_WRITE requests to write. (O_RDWR has both O_READ and O_WRITE). if neither O_CREATE or O_MAYCREATE are specified, the file must exist. If O_CREATE is specified, the file must not exist and a new file is created. If O_MAYCREATE is specified, a file may be created, but if a file exists with that file name, open the existing file. If all is OK, this function returns a file descriptor, which is a small, non-negative integer. It errors occur, this function returns -1.

```
    function Sys_Read (fileDesc: int, buffer: ptr to char,
                       sizeInBytes: int) returns int
```

This function is passed the fileDescriptor of a file (which is assumed to have been successfully opened), a pointer to an area of memory, and a count of the number of bytes to transfer. This function reads that many bytes from the current position in the file and places them in memory. If there are not enough bytes between the current position and the end of the file, then a lesser number of bytes are transferred. The current file position will be advanced by the number of bytes transferred.

If the input is coming from the serial device (the terminal), this function will wait for at least one character to be typed before returning, and then will return as many characters as have been typed and buffered since the previous call to this function.

This function will return the number of characters moved. If there are errors, it will return -1.

```
   function Sys_Write (fileDesc: int, buffer: ptr to char,
                       sizeInBytes: int) returns int
```

This function is passed the fileDescriptor of a file (which is assumed to have been successfully opened), a pointer to an area of memory, and a count of the number of bytes to transfer. This function writes that many bytes from the memory to the current position in the file.

If the end of the file is reached, the file's size will be increased.

The current file position will be advanced by the number of bytes transferred, so that future writes will follow the data transferred in this invocation.

The output may also be directed to the serial output, i.e., to the terminal.

This function will return the number of characters moved. If there are errors, it will return -1.

```
function Sys_Seek (fileDesc: int, newCurrentPosition: int)
                 returns int
```

This function is passed the fileDescriptor of a file (which is assumed to have been successfully opened), and a new current position. This function sets the current position in the file to the given value and returns the new current position.

Setting the current position to zero causes the next read or write to refer to the very first byte in the file. If the file size is N bytes, setting the position to N will cause the next write to append data to the end of the file.

The current position is always between 0 and N, where N is the file's size in bytes.

If -1 is supplied as the new current position, the current position will be set to N (the file size in bytes) and N will be returned.

It is an error to supply a newCurrentPosition that is less than -1 or greater than N. If so, -1 will be returned.

```
function Sys_Close (fileDesc: int)
```

This function is passed the fileDescriptor of a file, which is assumed to be open. It closes the file, which includes writing out any data buffered by the kernel.

11 Error Codes

The Sys_GetError system call is already implemented, but as you implement other system calls, you need to set the proper error codes when your code finds an error. The error codes are defined in the file Syscall.h and they are values of an enum. They include error codes like E_Bad_Address, E_Bad_Value and E_No_Child. The error code is stored in the PCB for the process in a field named "lastError".

When your code discovers and error, your code must set the proper error code in the "lastError" field for the current process. If your code calls other functions, like CopyBytesFromVirtual (a method in ADdrSpace) and it returns an error, you code must not set the "lastError" as that code has already set it. The idea here is that the code that discovers the error sets the "lastError" value and then returns an error code.

12 Asynchronous Interrupts

From time-to-time an asynchronous interrupt will occur. Consider a DiskInterrupt as an example. When this happens, an assembly routine called DiskInterruptHandler in Runtime.s will be jumped to. It begins by saving the system registers (after all, a Disk Interrupt might occur while a kernel routine is executing and we'll need to return to it). Then DiskInterruptHandler performs an "upcall" to the function named DiskInterruptHandler in Kernel.k. Perhaps it is a little confusing to have an assembly routine and a KPL routine with the same name, but, oh well...

The high-level DiskInterruptHandler routine simply signals a semaphore and returns to the assembly DiskInterruptHandler routine, which restores the system registers and returns to whatever code was interrupted. All the time while these routines are running, interrupts are disabled and no other interrupts can occur.

Also note that the interrupt handler uses space on the system stack of whichever thread was interrupted. It might be that some unsuspecting user-level code was running. Although the interrupt handler will use the system stack of that thread, the thread will be none-the-wiser. While the interrupt handler is running, it is running as part of some more-or-less randomly selected thread. The interrupt handler is not a thread on its own.

13 Error Exception Handling

When a runtime error is detected by the CPU, the CPU performs exception processing, which is similar to the way it processes an interrupt. (This is not the same thing as an error in a system call that sets the "lastError" field.) Here are the sorts of runtime errors that can occur in the BLITZ architecture:

```
Illegal Instruction
Arithmetic Exception
Address Exception
Page Invalid Exception
Page Read-only Exception
Privileged Instruction
Alignment Exception
```

As an example, consider what happens when an Alignment Exception occurs. (The others are handled the same way.)

The CPU will consult the interrupt vector in low memory (see Runtime.s) and will jump to an assembly language routine called AlignmentExceptionHandler. The assembly routine first checks to see if the interrupted code was executing in system mode or not. If it was in system mode, then the assumption is that there is a bug in the kernel, so the assembly routine prints a message and halts execution.

However, if the CPU was in user mode, the assumption is that the user-level program has a bug. The OS will need to handle that bug without itself stopping. So the assembly AlignmentExceptionHandler routine makes an upcall to a KPL routine with the same name.

The high-level AlignmentExceptionHandler routine simply prints a message and terminates the process. Process termination is performed in a routine called ProcessFinish, which is not yet written. (For now, we'll assume that user-level programs do not have any bugs.)

When ProcessFinish is implemented in a later assignment, it will need to return the ProcessControlBlock (PCB) to the free pool. It will also need to free any additional resources held by the process, such as OpenFile objects. Of course, any open files will need to be closed first. Finally, ProcessFinish will call ThreadFinish and will not return.

(Note that a Thread object cannot be added back to the free thread pool by the thread that is running. Instead, in ThreadFinish the thread is added to a list called threadsTobeDestroyed. Later, after another thread begins executing (in Run) the first thing it will do is add any threads on that list back to the free pool by calling threadManager.FreeThread.)

14 Syscalls

When a user-level thread executes a syscall instruction, the assembly routine SyscallTrapHandler in Runtime.s will be invoked. The assembly routine will then call a KPL routine with the same name.

The assembly routine does not need to save registers because the interrupted code was executing in user mode and the handler will be executed in system mode.

Recall that just before the syscall, the DoSyscall routine placed the arguments in the (user) registers r1, r2, r3, and r4, with an integer indicating which kernel function is wanted in register r5. The SyscallTrapHandler assembly routine takes the values from the user registers. Since it is running in system mode, it must use a special instruction called "readu" to get values from the user registers. It pushes them on to the system stack so that the high-level routine can access them. Then it calls the high-level SyscallTrapHandler routine. When the high-level routine returns, it takes the returned value from the stack and moves it into user register r1, using an instruction called "writeu," and then executes a "reti" instruction to return to the interrupted user-level process. Execution will resume back in DoSyscall directly after the "syscall" instruction.

The high-level routine called SyscallTrapHandler simply takes a look at the function code. Depending on the function code, it can handle it immediately as the GetError call or it calls the appropriate routine to finish the work. For function code values other than the GetError code, there is a corresponding "handler routine" in the OS.

System call	Handler function in the kernel
Exit	Handle_Sys_Exit
Shutdown	Handle_Sys_Shutdown
Yield	Handle_Sys_Yield

```
        Fork            Handle_Sys_Fork
        Join            Handle_Sys_Join
        Exec            Handle_Sys_Exec
        Open            Handle_Sys_Open
        Read            Handle_Sys_Read
        Write           Handle_Sys_Write
        Seek            Handle_Sys_Seek
        Close           Handle_Sys_Close
```

It is these routines that you will need to implement, in this and other assignments. Also, in later assignments you will be adding more system call handlers.

Note that interrupts will be disabled when the SyscallTrapHandler routine begins. The first thing the high-level routine does is set the global variable currentInterruptStatus to DISABLED so that it is accurate. In fact, all the interrupt and exception handlers begin by setting currentInterruptStatus to DISABLED for this reason.

Also note that after the handler routines return to the interrupted routine, interrupts will be re-enabled. Why? Because the Status Register in the CPU will be restored as part of the operation of the reti instruction, restoring the interrupt (and paging and system mode) status bits to what they were when the interrupt occurred. (Note that we do not bother to change currentInterruptStatus to ENABLED before returning to user-level code, because any re-entry to the kernel code must be through SyscallTrapHandler, or an interrupt or exception handler, and each of these begins by setting currentInterruptStatus.)

Implementing the Shutdown syscall is straightforward. The handler should call FatalError with the following message:

```
        Syscall 'Shutdown' was invoked by a user thread
```

15 The BLITZ Disk

The BLITZ computer includes a disk which is emulated using a file called DISK on the host computer. In other words, a write to the BLITZ disk will cause data to be written to a Unix file and a read from the BLITZ disk will cause a read from the Unix file. The emulator will simulate the delays involved in reading, by taking account of the current (simulated) disk head position. When the I/O is complete-that is the simulated time when the emulator has calculated the disk I/O will have completed-the emulator causes a DiskInterrupt to occur.

To interface with the BLITZ disk, we have supplied a class called DiskDriver, which makes it unnecessary for you to write the code that actually reads and writes disk sectors. You can just use the code in the class DiskDriver. There is only one DiskDriver object; it is created and initialized at startup time.

```
  class DiskDriver
```

```
    superclass Object
    fields
      ...
      semToSignalOnCompletion: ptr to Semaphore
      semUsedInSynchMethods: Semaphore
      diskBusy: Mutex
    methods
      Init ()
      SynchReadSector  (sectorAddr, numberOfSectors, memoryAddr: int)
      StartReadSector  (sectorAddr, numberOfSectors, memoryAddr: int,
                        whoCares: ptr to Semaphore)
      SynchWriteSector (sectorAddr, numberOfSectors, memoryAddr: int)
      StartWriteSector (sectorAddr, numberOfSectors, memoryAddr: int,
                        whoCares: ptr to Semaphore)
  endClass
```

This class provides a way to read and write sectors synchronously as well as a way to read and write sectors asynchronously.

To perform a disk operation without blocking the calling thread, you can call StartReadSector or StartWriteSector. These methods are passed the number of the sector on the disk at which to begin the transfer, the number of sectors to transfer and the location in memory to transfer the data to or from. These methods are also passed a pointer to a Semaphore; upon completion of the operation (possibly in error!) this semaphore will be signaled with an Up() operation. This is exactly the semaphore that is signaled whenever a DiskInterrupt occurs. So to perform asynchronous I/O, the caller will invoke StartReadSector (or StartWriteSector) giving it a Semaphore. Then the caller can either do other stuff, or wait on the Semaphore.

Since it may be a little tricky to manage asynchronous I/O correctly, the DiskDriver class also provides a couple of methods to make it easy to do I/O synchronously.

When you call SynchReadSector or SynchWriteSector, the caller will be suspended and will be returned to only after a successful completion of the I/O. These routines will deal with transient errors by retrying the operation until it works. Other errors (such as a bad sectorAddr or bad memoryAddr) will be dealt with by a call to FatalError.

In order to implement these methods, the DiskDriver contains a mutex called diskBusy and a semaphore called SemUsedInSynchMethods. Each synch method makes sure the disk is not busy with I/O from some other thread and, if so, waits until it is completed. This is the purpose of the diskBusy mutex. After acquiring the lock, each synch method will call StartReadSector (or StartWriteSector) supplying the semaphore. The synch method will then wait until the disk operation is complete. The calling thread will remain blocked for the duration.

16 The “ToyFs” File System

In most operating systems, file system code is an important part of the OS. The original BLITZ projects used a contiguous allocation, single directory, non-reusable space simple file system. To make a more realistic file system, a ”Toy” file system has been designed for CSCI 509 students. It has since been use for CSCI 447 students also, although CSCI 447 students do not implement the full ToyFs system.

The ToyFs file system is a small Unix-like file system. For now, you are supplied with a very minimal minimal implementation of the ToyFs file system. This ”Toy” file system uses ”inodes” and block allocation rather than contiguous allocation. By adding this file system to the BLITZ project, there are also new system calls needed to manipulate the ToyFs file system. Also, the ToyFs code will need blocks of memory, so ToyFs will be the user of the FrameManager methods GetAFrame() and PutAFrame().

16.1 ToyFs Disk Structures

First, you must understand the on-disk format of the ToyFs to be able to write correct code to manipulate the file system. First, since Blitz has frame sizes of 8k bytes, ToyFs uses ”sector” sizes of 8k. (The disk driver for your OS also uses 8k sector sizes.) The disk consists of one ”super block”, one or more ”inode blocks” and a number of ”data blocks”. The ToyFs will be stored in a regular UNIX file and will conform to the BLITZ ”disk specification”. Specifically, the first 4 bytes of the file contain ”BLZd” and then the remaining part of the file is an integral number of ”sectors” of 8192 bytes each. For ToyFs, these sectors contain the following data:

1. Sector 0:

   - 4 bytes: Magic Number: TyFs
   - 4 bytes: Size of disk in sectors
   - 4 bytes: Number of inodes (Multiple of 128)
   - 4 bytes: padding
   - 4 bytes: number of words of inode bitmap
   - inode bitmap (4 * number just before this)
   - 4 bytes: number of words of data bitmap
   - data bitmap (4 * number just before this)

   The total size of the inode and data bitmaps must be 8192 - 24, the number of specified ints in the sector.

2. Sector 1 to (number of inodes)/128+1: Each of these sectors contains 128 inodes each.

3. Sector (number of inodes)/128 + 2 to the end of the disk: These sectors are the data sectors.

16.2 Inode Structure

Each inode is 64 bytes of data. This yields 128 inodes per sector. Each inode contains the following (in this order):

- 2 bytes: number of links
- 2 bytes: mode (file type, r, w, x)
- 4 bytes: size of file in bytes
- 4 bytes: blocks allocated
- 4 bytes: (Kpl Array Size)
- 4 bytes * 10 direct links (Kpl Array)
- 4 bytes: single indirect (4 bytes)
- 4 bytes: double indirect (4 bytes)

The Inode structure is defined in the code you are adding to Kernel.h as found in FileStuff.h:

```
type diskInode = record
                    nlinksAndMode: int
                    fsize: int
                    balloc: int
                    direct: array [10] of int
                    indir1: int
                    indir2: int
                  endRecord
```

It is the current intention for students to deal only with the single indirect pointer blocks and to not implement double indirect blocks. The second indirect pointer was added to allow big files if wanted and to make the diskInode structure exactly 64 bytes.

File types are directory and file. Mode types are read, write and execute. Each of these are represented in the "mode" field with bits. The bit values are as follows:

```
FILE      0x10
DIRECTORY 0x20
READ      0x04
WRITE     0x02
EXECUTE   0x01
```

16.3 Directory Files

Directory files contain ¡inode,name¿ entries. The file size is limited to 10 sectors of data so directory files do not need to use indirect blocks. In the directory file, each of the 10 sectors is structured independently. Each sector is composed of entries of the following structure:

- 4 bytes, inode number
- 4 bytes, name length "n" (max of 255 even though it has 4 bytes)
- n bytes of the file name
- pad bytes so the file name and pad bytes is a multiple of 4 bytes

The last entry in each sector will have the inode number either a 0 or a -1. The inode number value of 0 says "no more entries in this sector, more in the next sector". The inode number value of -1 says "no more entries in the directory". When adding file names to a directory, code should add it to the first sector in which there is room for the entry and still include 4 bytes for the "inode number" of the next entry. When the inode number is -1 or 0, the remainder of the entry does not exist. (The inode number and the name length are integers and are stored in big-endian byte order.)

16.4 System Calls

The ToyFs has a number of system calls to deal with the file system and previous system calls may interact with the ToyFs. System calls like Sys_Read will need to deal with ToyFs. Sys_Open will need to open any kind of file, including a file in ToyFs.

The file system related system calls are:

```
Sys_Stat (name: String, stat_ptr: ptr to StatInfo) returns int
Sys_ChMode (filename: String, mode: int) returns int
Sys_Link (srcName, newName: String) returns int
Sys_Unlink (name: String) returns int
Sys_Mkdir (name: String) returns int
Sys_Rmdir (name: String) returns int
Sys_Chdir (name: String) returns int
Sys_OpenDir (name: String) returns int
Sys_ReadDir (dirFd: int, entPtr: ptr to DirEntry) returns int
```

Both StatInfo and DirEntry are records declared in Syscall.h. These ToyFs system calls are already included in the Syscall.h definitions. You will need to add some syscall structure to Kernel.* in future assignments. Also, the operation of these will be documented in future assignments. While a substantial part of the ToyFs file system is provided for you, important parts are left to the student to implement. For 447, not all of these system calls will be implemented.

16.5 The "toyfs" Tool

The tool called toyfs can be used to create a ToyFs file system on the BLITZ disk, to add files to the disk, to remove files, create directories, and to print out the directory as well as other things. The first option to toyfs specifies the operation:

- -i Initialize a new ToyFs disk.
- -a Add host files to the ToyFs disk.
- -g Get a file on the ToyFs disk and copy it to the host.
- -l List a directory on the ToyFs disk.
- -m Make a new directory.
- -c Change the "mode" on a file or directory.
- -h Print a help message. (-? also does this.)

The -i flag takes two more options, -n number_of_inodes and -s number_of_sectors. The -a flag may also have the -x added to show that the file needs to be added with execute permission. All commands may have a name of a disk specified using -d diskname. The makefiles of the projects have been set up to build the needed disks for the specified projects.

17 The FileManager

There is only one FileManager object; it is created and initialized at startup time.

We are supplying several methods to help you access files on the file system; these methods are located in this class. You'll need to know how to access files in order to create the first user-level process. You'll need to open the executable file, read the bytes from disk, then close the file. You'll also need to use the fileManager when you implement the Exec syscall.

Some of the following material pertains more to the next assignment than this assignment. Read it all now to get familiar with the framework. You may want to review it again during the next assignment.

Associated with the FileManager class, there are two other classes called FileControlBlock and OpenFile. These two classes contain fields, but do not contain many methods of their own (besides Init() and Print() methods). Instead, most of the work associated with the file system is done by the FileManager and ToyFs methods.

The FileControlBlock (FCB) objects and the OpenFile objects are limited resources. The FileManager maintains a free list for each of these, as well as code to allocate new FCB objects and new OpenFile objects and maintain the free lists.

The FileManager also deals with opening files. This involves finding the file in the file system, that is, determining the file's location on disk. In the ToyFs file system this is pretty simple by using the NameToInodeNum method of the FileManager. Once the inode number has been looked up, the file system gets a FileControlBlock for the inode. Note, the FileManager manages all files in the OS. ToyFs files are just one kind of file managed. In the final assignment, some files will not have an associated Inode.

17.1 FileControlBlock (FCB) and OpenFile

The semantics of files in the kernel you are building will be similar to the semantics of files in Unix.

Consider the case where one process has opened a file and does a kernel call to read, say, 10 bytes. The kernel must read the appropriate sector, extract the 10 bytes out of that sector, and finally copy those 10 bytes into the process's virtual memory space. This requires the kernel to maintain a frame of memory to use as a buffer; the sector will be read into this buffer by the OS.

If the 10 bytes happen to span the boundary between sectors, the kernel must read both sectors in order to complete the Read syscall. And of course, during the I/O operations other threads must be allowed to run.

Now consider what happens when a process wants to write, say, 20 bytes to a file. The kernel will need to bring in the appropriate sector and copy the 20 bytes from the process's virtual address space to the buffer. Should the kernel write the buffer back to disk immediately? No; it is likely that the process will want to write some more bytes to that very same sector, so it is more efficient to leave the sector in memory.

When should the kernel write the sector back to disk? When the process closes the file, the kernel must write it back. Also, other I/O operations on the file may need different sectors, so the kernel should write the sector back to disk when the buffer is needed for another sector. However, if the buffer has not been modified, then there is no need to write it back to the disk. Therefore, we associate a Boolean called bufferIsDirty with each buffer frame. When a buffer is first read in from disk, it is considered to be "clean," but after any operation modifies the buffer, it should be marked "dirty."

Next consider the case in which two processes have both opened the same file. (Let's call them processes "A" and "B.") Any update by process A must be immediately visible to process B. If process A writes to a file and B reads from that same file, even before A has closed the file, then B should see the new data. Since the kernel may not actually write to the disk for a long time after process A does the write, it means that processes A and B must share the buffer.

Also, when one process finally closes a file, the buffer must be written back to the disk. The guarantee the kernel makes is that once we return from a call to Sys_Close, the disk has been updated. The program can stop worrying about failures, etc., and can tell the user that it has completed its task. Any changes the program has made-even if the system crashes in the next instance-will be permanent and will not be lost. After a Sys_Close, the kernel must not return to the user-level program until the buffer (or all buffers, if there are more than one) is written to the disk successfully.

The purpose of a FileControlBlock (FCB) is to record all the data associated with a single file. This includes the buffer and the bufferIsDirty bit. Here is the definition of FCB:

```
class FileControlBlock
    superclass Listable
    fields
      inode: InodeData
      numberOfUsers: int              -- count of OpenFiles pointing here
      bufferPtr: int                  -- addr of a page frame
      relativeSectorInBuffer: int     -- or -1 if none
      bufferIsDirty: bool             -- Set to true when buffer is modified
    methods
      Init ()
      Print ()
      ReadSector (newSector: int) returns bool
      Flush ()
   endClass
```

A small number FCBs are preallocated and kept in a table called fcbTable, which is maintained by the FileManager. The FileManager is responsible for allocating new FileControlBlock objects and for returning unused FileControlBlock objects to a free pool called fcbFreeList.

The inode data tells where the file is located on the disk. The ReadSector method uses the inode data to locate the sector, a number relative to the start of the file, and read it into the buffer. A single memory frame is allocated for each FCB at kernel startup time and bufferPtr is set to point to that memory region. relativeSectorInBuffer tells which sector of the file is currently in the buffer and is -1 if there is no valid data in the buffer. The method Flush writes out a dirty buffer to the correct sector on disk.

Next consider a process "A" that has opened a file. All of the "read" and "write" operations that the user-level process executes are relative to a "current position" in the file. Several processes may have the same file open. All processes that have file "F" open will share a single FCB. However, they will each have a different "current position" in the file.

To handle the current position, we have the class OpenFile, which is defined as:

```
  class OpenFile
    superclass Listable
    fields
      kind: int                       -- FILE, TERMINAL, or PIPE
      currentPos: int                 -- 0 = first byte of file
      fcb: ptr to FileControlBlock    -- null = not open
      numberOfUsers: int              -- count of Processes pointing here
      addPos: int                     == byte address, for adding entries
    methods
      Print ()
      NewReference () returns ptr to OpenFile
      ReadBytes (targetAddr, numBytes: int) returns bool     -- true=All Okay
      ReadInt () returns int
      LoadExecutable (addrSpace: ptr to AddrSpace) returns int -- -1=problems
      Lookup ( filename: String, fcbPtr: ptr to FileControlBlock)
               returns ptr to dirEntry
      GetNextEntry (newSize: int) returns ptr to dirEntry
      AddEntry (inodeNum: int, filename: String) returns bool
  endClass
```

Like the FCBs, there is a preallocated pool of OpenFile objects, which are created at system startup time. The FileManager is responsible for allocating new OpenFile objects and for returning unused OpenFile objects to a free pool called openFileFreeList.

When process ”A” opens a file, a new OpenFile object must be allocated and made to point to an FCB describing the file. If there is already an FCB for that file, then the OpenFile should be made to point to it; otherwise, we'll have to get a new FCB, check the directory, and set up the FCB.

When do we return an FCB to the free pool? When there are no more OpenFiles using it. This is the reason we have a field called numberOfUsers in the FCB. This field is a ”reference counter.” It tells the number of OpenFile objects that point to the FCB. When a new OpenFile is allocated and made to point to an FCB, the count must be incremented. When an OpenFile is closed, the count should be decremented. When the count becomes zero, the FCB must be returned to the free pool.

When a process is terminated, for example due to an error such as an AlignmentException, the kernel must close any and all OpenFiles the process is using. The process may explicitly close an OpenFile with the Close syscall. Once a file is closed, the process should attempt no further I/O on the file and if the process does, the kernel should catch it and treat it as an error (by returning an error code from the Sys_Read or Sys_Write kernel call).

Our file I/O will follow the semantics of Unix. When a process is cloned with the Fork syscall, all open files in the parent process must be shared with the child process. Consider what happens when a parent and a child are both writing to the same file, which was originally opened in the parent. Since both processes share the OpenFile object, they will share the current position. If the child writes 5 bytes, the current position will be incremented by 5. Then, if the parent writes 13 bytes, these 13 bytes will follow the 5 bytes written by the child.

In order to implement these semantics, it will be possible for several PCBs to point to the same OpenFile object. We need to maintain a reference count for the OpenFiles, just like the reference count for the FCBs. Whenever a process opens a file, we need to allocate a new OpenFile object and set its count to 1. Whenever a process forks, we'll need to increment the count. When a process closes a file (either by invoking the Close syscall or by dying), we'll need to decrement the count. If the count goes to zero, we'll need to return the OpenFile to the free pool and decrement the count associated with the FCB.

User-level processes must not be allowed to use pointers into kernel memory and cannot be allowed to touch kernel data structures such as OpenFiles and FCBs. So how does a user process refer to an OpenFile object? Indirectly, through an integer. Here's how it works.

Each Process will have a small array of pointers to OpenFiles called fileDescriptor.

```
class ProcessControlBlock
  ...
  fields
    ...
    fileDescriptor: array [MAX_FILES_PER_PROCESS] of ptr to OpenFile
  methods
    ...
endClass
```

When a process invokes the Open syscall, a new OpenFile will be set up. Then the kernel will select an unused position in this array and make it point to the OpenFile. For example, positions 0, 1, and 2 might be in use, so the kernel may assign a file descriptor of 3 for the newly opened file. The kernel must make fileDescriptor[3] point to the OpenFile and should return "3" as the fileDescriptor to the user-level process.

When the user-level process wants to do an I/O operation, such as Read, Write, Seek, or Close, it must supply the fileDescriptor. The kernel must check that (1) this number is a valid index into the array, and (2) the array element points to a valid OpenFile. When closing the file, the kernel will need to decrement the reference count for the OpenFile object and also set fileDescriptor[3] to null. Then, if the user process attempts any future I/O operations with file descriptor 3, the kernel can detect that it is an error.

Since user-level file I/O will not be implemented in this assignment, you will not need to worry about fileDescriptors yet.

When a user-level program does a Read or Write syscall-in Unix or in our OS-the data may be transferred from/to either

```
        - a file on the disk
        - an I/O device such as a keyboard or display (these are
              called "special files" in Unix)
        - another process, via a "pipe"
```

In all three cases, an OpenFile object will be used. The field called kind tells whether the object corresponds to a FILE, the TERMINAL, or a PIPE. In this assignment, we will only use OpenFiles to perform the Exec syscall, so the kind will be only FILE (and not TERMINAL or PIPE).

17.2 To Read in an Executable File

To read in an executable file from disk, your code will need to:

```
        - Open the file
        - Invoke LoadExecutable to do the work
        - Close the file
```

Read through the code for FileManager.Open:

```
     method Open (filename: String,  dir: ptr to OpenFile, flags,
                  mode: int) returns ptr to OpenFile
```

Open is passed a four arguments as seen above, “filename” is a ptr to array of char, the name of the file on the BLITZ disk that you want to open, “dir” is a directory that is the starting place for relative file names, and finally, “flags” and “mode” are the same values as passed to the Open system call. FileManager.Open will allocate a new OpenFile object and a new FCB object and set them up. Then it will return a pointer to the OpenFile object, which you’ll use when calling LoadExecutable. If anything goes wrong, Open returns null. The only real danger is getting the filename wrong.

In BLITZ, like Unix, executable files have rather complex format. For details, you can read through the document titled ”The Format of BLITZ Object and Executable Files.” So that you don’t have to write all this code, we are providing a method called OpenFile.LoadExecutable:

```
        method LoadExecutable (addrSpace: ptr to AddrSpace) returns int
```

Look through LoadExecutable; it will

```
        - Create a new address space (by calling frameManager.GetNewFrames)
        - Read the executable program into the new address space
        - Determine the starting address (the initial program counter, also
            called the "entry point")
        - Return the entry point
```

If there are any problems with the executable file, this method will return -1. Otherwise it will return the entry point of the executable. This is the address (in the logical address space) at which execution should begin. Normally, this will be 0x00000000.

18 User-Level Processes

Each user-level process will have a single thread which will normally execute in User mode, with ”paging” turned on and interrupts enabled.

Each user-level process will have a logical address space, which will consist of

```
- A Page for "environment" data
- Pages for the text segment
- Pages for the data segment
- Pages for the BSS segment
- Pages for the user’s stack
```

These are shown in order, with the stack pages in the highest addresses of the logical address space.

The environment page will sit at address 0 and could contain information that the OS wishes to pass to a new user-level process. such as userID, working directory, etc. We will not use an environment page, so the text pages will begin at address 0.

Kernel.h contains this:

```
const
  NUMBER_OF_ENVIRONMENT_PAGES = 0
  USER_STACK_SIZE_IN_PAGES = 1
  MAX_PAGES_PER_VIRT_SPACE = 20
```

The text pages contain the program and any constant values.

The data pages will contain the static (global) program variables.

The BSS pages will contain space for uninitialized program variables (such as large arrays). The OS will set all bytes in the BSS pages to zero. Most KPL programs do not use a BSS segment, so there will usually be zero BSS pages.

The user-level program will have a stack, which will grow downward. Each logical address space will have a predetermined small number of pages (in our case, this is one page) set aside for its stack. In Unix, if a user process’s stack grows beyond its initial allocation, more stack pages would be added. In our OS, if a user process’s stack grows beyond this, it will begin overwriting the BSS and data pages, and the program will probably get an error of some sort soon thereafter.

As an example, a program might use:

```
0 environment pages
2 text pages
1 data page
0 BSS pages
1 stack page
```

This process's logical address space will have 4 pages. Each page has PAGE_SIZE bytes (8 Kbytes), so the entire address space will be 32 Kbytes. Any address between 0x00000000 and 0x00007FFF (which is 32K-1 in hex) would be legal for this program. If the program tries to use any other address, a PageInvalid Exception will occur.

In Unix, the environment and text pages would be marked read-only and any attempt to update bytes in those pages would cause an exception. In this assignment, all pages of the logical address space will be read-write, so our OS will not be able to catch that sort of error in the user-level program.

Each page in the logical address space will be stored in one frame in memory. The frames do not have to be contiguous and the pages may be stored in pretty much any order. However, all pages will be in memory throughout the process's lifetime.

The page table will keep track of where each page is kept. While the process is executing, "paging" will be turned on so that the memory management unit (MMU) will translate all logical addresses into physical addresses. Our example program will not be able to read or write anything outside of its 4 pages.

There may be several processes in the system at any time. Each ProcessControlBlock contains an AddrSpace, which tells how many pages the process's address space has and which frame in physical memory holds each page.

When some process (call it "P") is ready to be scheduled and given a time-slice, the MMU will be need to be set up so that it points to the page table for process P. You can do this with the method:

```
        AddrSpace.SetToThisPageTable ()
```

which calls an assembly routine to load the MMU registers. This method must be invoked before paging is turned on. When paging is turned off (i.e., whenever kernel code is being executed), the MMU registers are ignored.

Note that each thread will have two stacks: a user stack and a system stack. We have already seen the system stack; it is used when one kernel function calls another kernel function. The user stack will be used when the thread is running in user mode. The system stack, which is fairly small, normally contains nothing while the user-level program is running. In other words, the system stack is completely empty.

After the user-level program begins executing, execution can re-enter the kernel only through exception processing. That is, the only ways to get back into the kernel are:

```
        - an interrupt,
        - a program exception, or
        - a syscall
```

In each of these cases, the exact same thing happens: some information is pushed onto the system stack, the mode is changed to system mode, paging is turned off, and a jump is made to a kernel "handler" routine.

The BLITZ computer has two sets of registers: one for user-mode code and one for system-mode code. Thus, the user registers do not need to be saved, unless the kernel will switch to another thread. This is done in the Run method, which contains this code:

```
if prevThread.isUserThread
  SaveUserRegs (&prevThread.userRegs[0])
endIf
...
Switch (prevThread, nextThread)
...
if currentThread.isUserThread
  RestoreUserRegs (&currentThread.userRegs[0])
  currentThread.myProcess.addrSpace.SetToThisPageTable ()
endIf
```

If the kernel handler code wishes to return to the same user-level code that was interrupted, it can merely return to the assembly language handler routine, which will perform a "reti" instruction. The user registers and the MMU registers will (presumably) be unchanged, so when the mode reverts to "user mode" and the paging reverts to "paging enabled," the user-level program will resume execution with the same values in the user registers and the same logical address space.

18.1 Creating a User-Level Process

The main function calls function InitFirstProcess, which you must implement. The first thing you'll need to do is get a new thread object by invoking GetANewThread. Since the InitFirstProcess function should return, you cannot use the current thread. Next you'll need to initialize the thread and invoke Fork to start it running. (You can name this new thread something like "UserProgram," but the name is only used in the debugging printouts.)

The new thread should execute the StartUserProcess function, which will do the remainder of the work in starting up a user-level process. InitFirstProcess can supply a zero as an argument to StartUserProcess and can return after forking the new thread.

The first thing you'll need to do in StartUserProcess is allocate a new PCB (with GetANewProcess) and connect it with the thread. So initialize the myThread field in the PCB and the myProcess field in the current thread.

Next, you'll need to open the executable file. It is acceptable to "hardcode" the filename (e.g., "TestProgram1") into the call to Open, although changing the name of the initial process will require a recompile of the kernel. If there are problems with the Open, this is a fatal, unrecoverable error and the kernel startup process will fail. Note, the Kernel.h as distributed to you had a constant called INIT_NAME. You should use this constant for the executable file name because it is easier than finding the file name in the StartUserProcess function.

Next, you'll need to create the logical address space and read the executable into it. The method OpenFile.LoadExecutable will take care of both tasks. If this fails, the kernel cannot start up. LoadExecutable returns the entry point, which you might call initPC.

Don't forget to close the executable file you opened earlier, or else a system resource will be permanently locked up.

Next, you'll need to compute the initial value for the user-level stack, which you might call InitUserStackTop. It should be set to the logical address just past the end of the logical address space, since the initial push onto the user stack will first decrement the top pointer. The logical address space starts at zero. The logical address space contains

```
addrSpace.numberOfPages
```

pages. Each page has size PAGE_SIZE bytes.

The StartUserProcess function will end by jumping into the user-level program. This is a one way jump; execution will never return. (Instead, if the user-level program needs to re-enter the kernel, it will execute a syscall). As such, nothing on the system stack will ever be needed again. We want to have a full-sized system stack available for processing any syscalls or interrupts that happen later, so you need to reset the system stack top pointer, effectively clearing the system stack.

You might call the new value initSystemStackTop. You'll need to set it to:

```
& currentThread.systemStack[SYSTEM_STACK_SIZE-1]
```

Next, you'll need to turn this thread into a user-level thread. This involves these actions:

1. Disable interrupts
2. Initialize the page table registers for this logical address space
3. Set the isUserThread variable in the current thread to true
4. Set system register r15, the system stack top
5. Set user register r15, the user stack top
6. Clear the System mode bit in the condition code register to switch into user mode
7. Set the Paging bit in the cond. code register, causing the MMU to do virtual memory mapping
8. Set the Interrupts Enabled bit in the cond. code register, so that future interrupts will be handled
9. Jump to the initial entry point in the program

Recall that every thread begins life with interrupts enabled, so your StartUserProcess function will be executing with interrupts enabled. The first step is to disable interrupts, since there are possible race conditions with steps (2) and (3).

[What is the race problem? Consider what happens if a context switch (i.e., timer interrupt) were to occur between setting the page table registers and setting isUserThread to true. Look at the Run method. The MMU registers would be changed for the other process; then when this thread is once again scheduled, the code in Run will see isUserThread==false so it will not restore the MMU registers. Merely swapping the order of steps (2) and (3) results in a similar race condition.]

The first 3 steps can be done in high-level KPL code, but steps (4) through (9) must be done in assembly language.

Read through the BecomeUserThread assembly routine in the file Switch.s., which will take care of steps (4) through (9). StartUserProcess should end with a call to this routine:

```
BecomeUserThread (initUserStackTop, initPC, initSystemStackTop, argPtr)
```

The BecomeUserThread will change the mode bits and perform the jump "atomically." This must be done atomically since the target jump address is a logical address space. (The way it does this is a little tricky: it pushes some stuff onto the system stack to make it look like syscall or interrupt has occurred, and then executes a "reti" instruction.)

BecomeUserThread jumps to the user-level main routine and never returns. You should use null in your calls for argPtr until your ar asked to implement command line arguments.

19 Approach to Implementing the Exec Syscall

The sequence of steps in InitFirstProcess and StartUserProcess is very similar to what you'll need when implementing the Exec syscall. You should be able to copy much of this code when implementing Sys_Handle_Exec. (Don't forget about the final BecomeUserThread step in StartUserProcess. It needs to be done for Exec.)

One difference is that during an Exec, you already have a process and a thread, so you will not need to allocate a new ProcesControlBlock, allocate a new Thread object, or do a fork. However, you will have to work with two virtual address spaces. The LoadExecutable method requires an empty AddrSpace object; it will then allocate as many frames as necessary and initialize the new address space.

Unfortunately, LoadExecutable may fail and, if so, your kernel must be able to return to the process that invoked Exec (with an error code, of course). So you better not get rid of the old address space until after the new one has been initialized and you can be sure that no more errors can occur.

One approach is to create a local variable of type AddrSpace. Don't allocate it on the heap, just use something like:

```
    var newAddrSpace: AddrSpace = new AddrSpace
```

Then, after the new address space has been set up, you can copy it into the ProcessControlBlock, e.g.,

```
    currentThread.myProcess.addrSpace = newAddrSpace
```

Don't forget to free the frames in the previous address space first, or else valuable kernel resources will remain forever unavailable and the kernel will eventually freeze up!

Another tricky thing is copying the filename string from a virtual address space into the kernel address space where it can be used. The filename argument is a virtual address, but since the kernel is running in Handle_Sys_Exec, paging will be turned off. (Initially, you should ignore the args argument to Handle_Sys_Exec. You will deal with this argument later.)

You'll need to copy the characters into an array variable, not something newly allocated on the heap. It is okay to put a maximum size on this array and then check that it is not exceeded. In fact, there is a constant in Kernel.h for this purpose:

```
    const
      MAX_STRING_SIZE = 20
```

(In a real OS, the maximum string size would be much larger or even nonexistent. Here, we use a small size to make testing the limits easier.)

Note that the filename pointer is virtual address, which must be translated into a physical address; you can't just use it, as is. This requires some code to perform the page table lookup in software. Furthermore, since the filename string is in virtual space, it may cross page boundaries. (In fact, the test program contains cases where this happens!)

Dealing with the filename is fairly complex, but it turns out that we are giving you a method

```
    GetStringFromVirtual (kernelAddr: String, virtAddr, maxSize: int)
         returns int
```

which will do most of the work. (GetStringFromVirtual calls CopyBytesFromVirtual to do the copying.) The GetStringFromVirtual method can be used like this:

```
  var
    strBuffer: array [MAX_STRING_SIZE] of char
  ...
  i = currentThread.myProcess.addrSpace.GetStringFromVirtual (
            & strBuffer,
            filename asInteger,
            MAX_STRING_SIZE)
  if i < 0
    ...error...
  endIf
```

You might think of allocating a temporary buffer on the heap, but remember that we do not want to allocate anything on the heap after kernel start-up.

[Recall that the "alloc" expression in KPL always allocates bytes on the heap. Once the kernel has booted and is running, you must avoid further allocations. Why? One problem is automatic garbage collection like you see in Java; we can't use automatic garbage collection since it would produce unpredictable delays and might cause the kernel to miss interrupts or, in the case of a real-time system, miss deadlines. Also, there is the possibility that the heap might fill up, and dealing with a "heap full" error in the kernel is difficult. Another option might be to try to manage the heap without automatic garbage collection, but years of C++ experience has taught everybody that this is very difficult to do correctly. This explains why we have gone to the trouble to create classes like ThreadManager and ProcessManager, instead of simply allocating new Thread and ProcessControlBlock objects. Real operating systems implement some kind of kernel memory management so that they can be more dynamic than our OS.]

19.1 AllocateRandomFrames

The main function includes a function named AllocateRandomFrames, which is aimed only at catching bugs in the kernel. This function will allocate every other frame in the physical memory and never release them, creating a "checkerboard pattern" in memory. Henceforth, no two pages will ever be allocated to contiguous page frames.

Large, multi-byte chunks of data in the user-level process's address space will occasionally span page boundaries. Since these pages may not be in adjacent frames, your kernel will have to be careful about moving data to and from user space. What may appear to the user-level program as a string of adjacent bytes may in fact be spread all over memory.

Some of the user-level syscalls pass pointers to the kernel. For example, Open passes a pointer to a string of characters. Keep in mind that this pointer is a logical address, not a physical address. As such, you cannot simply use the pointer as is. Take a look at these methods AddrSpace:

```
CopyBytesFromVirtual (kernelAddr, virtAddr, numBytes: int) returns int
CopyBytesToVirtual (virtAddr, kernelAddr, numBytes: int) returns int
GetStringFromVirtual (kernelAddr: String, virtAddr, maxSize: int)
        returns int
```

An invocation of AllocateRandomFrames has been added just after the FrameManager is initialized. Please leave this in and do not modify the AllocateRandomFrames routine.

19.2 Command Line Arguments

The following definition in UserSystem.h is a way to pass command line arguments to a exec-ed process:

```
        var  cmdArgs: ptr to array of ptr to array of char
```

For the final 25 points of this assignment, modify the Exec syscall to process the second argument, args, which is of the same type as the above variable. This will allow programs to pass command line arguments to the Exec-ed program.

The old process space has an array of strings that is passed to the kernel as virtual address of the start of the array. The top level array is just an array of virtual addresses. To process the args argument, you must copy the strings from the old process space to the new process space and create the array of pointers to the strings using the virtual addresses of the new address space rather than the original address space.

The place to deal with command line arguments is after the new process address space has been built and before the previous address space has been freed. This will require multiple copies from and address spaces. You may have a fixed maximum limit on the number of command line arguments. It should be 100 or greater. Exceeding that number should cause Exec to fail. Don't forget to free the new address space if an error should occur.

The next question is where to store the arguments. One possibility is the environment page, except we do not have any environment pages. We could change the constant from zero to one, but that would require us to relink every user level program. To keep the linking the same, we will use the stack to save these strings. Currently, you should start the user stack just after the last page of user memory. We will change that so that the argument strings are packed into the high end of the stack page and then the stack will start after the command line arguments. We have a couple of choices. First, if we have too many command line arguments, we could end up with a real small stack. So, since we have 8k pages, we could limit the size of the arguments to 4K so we have at minimum 4k user stack. If we want to guarantee a minimum of 8k for the user stack, we could just change the number of stack frames from 1 to 2 and then say that the maximum size of the command line arguments would be 8k. But, for this assignment, we want you to limit the command line arguments to 2k.

For this assignment, you should build the arguments on the bottom of the stack. To do that, you need to understand KPL arrays. Arrays are represented the following way in KPL. First, the storage is aligned on a 4 byte boundary. The first 4 bytes are an unsigned size value, followed by the array, with the correct number of elements. Since these command line arguments are accessed as a KPL array, you must build the command line arguments correctly in memory so a KPL program can process them correctly. You must report an error if command line arguments requires more than 2048 bytes of memory. The error for wanting to use more than 2048 bytes is E_No_Space.

In the process of building the command line arguments for the new address space, you will need to know both the virtual address of the stack in the new address space and the physical address of the stack. Consider the following possible algorithm:

1. Get the number of arguments from the old address space.

2. Place the new array of strings at the bottom of the new address space. (Highest addresses in the stack page.) Calculate the placement of this array and save the pointer to the array as a virtual address in the new address space to pass to BecomeUserThread.
3. For each string in the args do:
   (a) Get the size of the string
   (b) Increase it to the next 4 byte aligned size. (Remember, a KPL array must start on a 4 byte aligned address.)
   (c) Calculate where the string array should be placed in the new address space.
   (d) Save the virtual address of that location in the top level array of strings at the proper location.
   (e) Copy the string using the old virtual address of the string in the old address space and the physical address space of the copy in the new address space. You need only move the characters once using GetStringFromVirtual.

Once you have copied all command line strings, you then know where to start the user stack pointer so it starts above the command line strings on the stack. (This would be the address just less than the last command line byte.) If you had no errors in copying the command line arguments and creating the new structure in the new address space, it is then time to free the old address space. It is now time to BecomeUserThread.

20 What to Hand In

Once you have completed this assignment, please create an "a4" branch. (You *do not* need an "a4" directory.) Capture a script of your runs for both MyProgram and TestProgram1 and commit them to your a4-branch as "a4-script" in the top directory of your project. Please turn in a cover sheet. If you want to comment on your assignment, use the cover sheet for your comments.

21 Coding Style

For all assignments, please follow our coding style. Make your code match our code as closely as possible.

The goal is for it to be impossible to tell, just by looking at the indentation and commenting, whether we wrote the code or you wrote the code.

Please use our convention in labeling and commenting functions:

```
    -----------------------------  Foo  ---------------------------------
```

```
function Foo ()
  --
  -- This routine does....
  --
  var x: int
    x = 1
    x = 2
    ...
    x = 9
  endFunction
```

Methods are labeled like this, with both class and method name:

```
----------  Condition . Wait  ----------

method Wait (mutex: ptr to Mutex)
  ...
```

Note that indentation is in increments of 2 spaces. Please be careful to indent statements such as if, while, and for properly.

If you follow these conventions, it will be obvious in the diff output which methods you changed or added.

22 Sample Output

If your program works correctly, you should see something similar, but not necessarily exactly as follows. (There are a few lines that are wrapped to the next line here but are full length lines in the output from blitz.)

```
==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 100
Initializing Disk Driver...
Initializing Serial Driver...
Serial handler thread running...
Initializing File Manager...
Loading initial program...
User-level program 'TestProgram1' is running...
```

```
***** Testing Syscall Parameter Passing *****

***** About to call Sys_Yield...
***** Should print:
*****     Handle_Sys_Yield invoked!

Handle_Sys_Yield invoked!

***** About to call Sys_Fork...
***** Should print:
*****     Handle_Sys_Fork invoked!

Handle_Sys_Fork invoked!

***** About to call Sys_Join...
***** Should print:
*****     Handle_Sys_Join invoked!
*****     processID = 1111

Handle_Sys_Join invoked!
processID = 1111

***** About to call Sys_Open...
***** Should print:
*****     Handle_Sys_Open invoked!
*****     virt addr of filename = 0x0000BFF8
*****     filename = MyFileName

Handle_Sys_Open invoked!
virt addr of filename = 0x0000BFF8
filename = MyFileName

***** About to call Sys_Read...
***** Should print:
*****     Handle_Sys_Read invoked!
*****     fileDesc = 2222
*****     virt addr of buffer = 0x0000B0A8
*****     sizeInBytes = 3333

Handle_Sys_Read invoked!
fileDesc = 2222
virt addr of buffer = 0x0000B0A8
```

```
sizeInBytes = 3333

***** About to call Sys_Write...
***** Should print:
*****     Handle_Sys_Write invoked!
*****     fileDesc = 4444
*****     virt addr of buffer = 0x0000B0A8
*****     sizeInBytes = 5555

Handle_Sys_Write invoked!
fileDesc = 4444
virt addr of buffer = 0x0000B0A8
sizeInBytes = 5555

***** About to call Sys_Seek...
***** Should print:
*****     Handle_Sys_Seek invoked!
*****     fileDesc = 6666
*****     newCurrentPos = 7777

Handle_Sys_Seek invoked!
fileDesc = 6666
newCurrentPos = 7777

***** About to call Sys_Close...
***** Should print:
*****     Handle_Sys_Close invoked!
*****     fileDesc = 8888

Handle_Sys_Close invoked!
fileDesc = 8888

***** About to call Sys_Exit...
***** Should print:
*****     Handle_Sys_Exit invoked!
*****     returnStatus = 9999

Handle_Sys_Exit invoked!
returnStatus = 9999

***** Syscall Test Complete *****

***** Testing Exec Syscall *****
```

```
***** About to call Sys_Exec with a non-existant file...
***** Should print:
*****     Okay

Okay

***** About to call Sys_Exec with an overly long file name...
***** Should print:
*****     Okay

Okay

***** About to call Sys_Exec with bad argument pointers...
***** Should print:
*****     Okay

Okay

***** About to call Sys_Exec with too many argument characters...
***** Should print:
*****     Okay

Okay

***** About to perform a successful Exec and jump to TestProgram2...
***** Should print:
*****     User-level program 'TestProgram2' is running!

User-level program 'TestProgram2' is running!
Command line arguments:
   arg[0] is Arg0
   arg[1] is This is string longer than 20 chars
   arg[2] is arg2
   arg[3] is arg3
   arg[4] is arg4 with more

***** Testing Sys_Exec with a null argument list.
*****  TestProgram2 should run a second time.

User-level program 'TestProgram2' is running!
Command line arguments:
   None provided.
```

```
***** About to call Sys_Shutdown...
***** Should print:
*****   FATAL ERROR in UserProgram: "Syscall 'Shutdown' was invoked by a
        user thread" -- TERMINATING!

FATAL ERROR in UserProgram: "Syscall 'Shutdown' was invoked by a
        user thread" -- TERMINATING!

(To find out where execution was when the problem arose, type 'st' at the emulator
      prompt.)

===================  KPL PROGRAM TERMINATION  ===================
```