Lab 5:
User-Level Processes

Due date: March 4, 2019, Monday. Your solutions should be submitted by **11:59 p.m.** Late submissions will not be accepted beyond 8 hours after the submission deadline. Within the eight-hour grace period, a mark penalty of 10% will be applied.

Objectives

In this lab, 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 **virtual 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 lab.

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

Setting up

Working in ECF Lab Machines

Similar to Lab 4, if you choose to work in the ECF lab machines directly (by remotely logging into one of them), we recommend that you start by creating a new directory for the files you will need for this lab. You can then copy all the files from:

```
/share/copy/ece353s/lab5
```

Even though some of the files have the same names, do not reuse files from the previous labs as they are considered out of date. The same set of files are also available in the Docker image that you can download from the link provided in the course website. Useful commands on using Docker containers have been introduced previously in the handouts in Lab 1 – 4.

For this lab, you should get the following files:

```
Switch.s
Runtime.s
System.h
System.c
```

```
List.h
List.c
BitMap.h
BitMap.c
makefile
FileStuff.h
FileStuff.c
Main.h
Main.c
DISK
UserRuntime.s
UserSystem.h
UserSystem.c
MyProgram.h
MyProgram.c
TestProgram1.h
TestProgram1.c
TestProgram2.h
TestProgram2.c
```

The following files are unchanged from the last lab and you should not modify them:

```
Switch.s
Runtime.s
System.h
System.c — except HEAP_SIZE has been modified
List.h
List.c
BitMap.h
BitMap.c
```

The following files are not provided; instead, **you will modify what you created in Lab 4**. **Copy** these files to your own **lab5** directory, so that you keep the previous **lab4** versions in your **lab4** directory, and modify the new copies.

```
Kernel.h
Kernel.c
```

Merging New "File Stuff" Code

For this lab, we are distributing additional code which you should add to the **Kernel** package. Please add the material in **FileStuff.c** to the end of file **Kernel.c.** 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.

This code adds the following classes:

DiskDriver
FileManager
FileControlBlock
OpenFile

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

There will be a single **DiskDriver** object (called **diskDriver**) which is created and initialized at start-up time. There will also 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** 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.

Other Changes To Your Kernel Code

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

Change
```
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 27              -- for testing only
```
to:
```
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 100             -- for testing only
```

Change
```
--diskDriver: DiskDriver
--fileManager: FileManager
```
to:
```
diskDriver: DiskDriver
fileManager: FileManager
```

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

Change
```
    ProcessFinish (exitStatus: int)
```
to:
```
    ProcessFinish (exitStatus: int)
```

```
InitFirstProcess ()
```

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

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
```

In this lab, you are required to complete three tasks. Please first read the following outlines of these tasks, and then read the remainder of this document. After that, come back and complete these tasks.

Task 1

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 will first need to understand how the BLITZ disk works, how files are stored on the disk, and how the **FileManager** code works.

MyProgram invokes the **Shutdown** syscall, which you will also need to implement.

Task 2

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
Create
Open

This pointer is in the user program's logical (virtual) 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
Create	4000
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**.

Task 3

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.

The User-Level View

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

Exit
Shutdown
Yield
Fork
Join
Exec
Create
Open
Read
Write
Seek
Close

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

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

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

```
UserSystem.h
UserSystem.c
UserRuntime.s
```

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

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
Exit	Sys_Exit
Shutdown	Sys_Shutdown
Yield	Sys_Yield
Fork	Sys_Fork
Join	Sys_Join
Exec	Sys_Exec
Create	Sys_Create
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. In BLITZ, all wrapper functions have names beginning with **Sys_** just to help make the distinction between a wrapper and a 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 4.

```
enum SYSCALL_EXIT = 1,
     SYSCALL_SHUTDOWN,
     SYSCALL_YIELD,
     SYSCALL_FORK,
     SYSCALL_JOIN,
     SYSCALL_EXEC,
     SYSCALL_CREATE,
     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. Consequently, there is an identical copy of this **enum** in both **Kernel.h** and **UserSystem.h**. (You should not change the system call interface, but if one were to change these code numbers, it would be critical that both **enum**s were changed identically.)

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 all the wrapper functions, including their arguments and return types.

```
Sys_Exit (returnStatus: int)
Sys_Shutdown ()
Sys_Yield ()
Sys_Fork () returns int
Sys_Join (processID: int) returns int
Sys_Exec (filename: String) returns int
Sys_Create (filename: String) returns int
Sys_Open (filename: String) 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.c:

```
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
lddd UserRuntime.o UserSystem.o MyProgram.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 file on the disk 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:

```
diskUtil -i
diskUtil -a MyProgram MyProgram
```

The first command creates an empty file system on the disk. The second command copies a file from the Unix file system to the BLITZ disk. It creates a directory entry and moves the data to the proper place on the simulated BLITZ disk. Commands to initialize the BLITZ disk have also been added to the **makefile**.

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

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_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" (no operation).

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 logical address 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.

function Sys_Exec (filename: String) returns int

This function is passed the name of a file. That file is assumed to be an executable file. It is read in to 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.

function Sys_Create (filename: String) returns int

This function creates a new file on the disk. If all is fine, it returns 0, otherwise it returns a non-zero error code. This function does not open the file; so the caller must use **Sys_Open** before attempting any I/O.

function Sys_Open (filename: String) returns int

This function opens a file. The file must already exist. If all is fine, 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.

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.c** (Perhaps it is a little confusing to have an assembly routine and a KPL routine with the same name.)

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.

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. 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, and will be implemented in the next lab. For now, we will assume that user-level programs do not have any bugs.

When **ProcessFinish** is implemented in the next lab, 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**.

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 and calls the appropriate routine to finish the work. For every kind of syscall, 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**
Create	Handle_Sys_Create
Open	Handle_Sys_Open
Read	Handle_Sys_Read
Write	Handle_Sys_Write
Seek	Handle_Sys_Seek
Close	Handle_Sys_Close

You will need to provide the full implementation for the routines marked in semibold red in this lab. For the rest of them, simply implement the minimum required functionality as shown in Task 2 on page 4-5.

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 (as well as 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
```

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.

The "Stub" File System

In this lab, you are supplied with a very minimal file system, called the "stub" file system. In Unix, directories are structured in a tree shape and there are lots of complexities concerning how files are stored on the disk.

In the stub file system, the disk will contain only one directory, and several files. The directory is limited in size to one sector and is kept in sector 0 of the disk. The exact number of files that can be accommodated depends on how long the file names are.

Each file has a name and a file length (in bytes). Each file is stored on disk in a sequence of consecutive sectors. Once a file is placed on the disk, and more files are added after it, it is impossible to increase the size of the file. Each file is allocated an integral number of sectors. (Since the last sector in each file may be only partially full, it would be possible to increase the size of a file up to the next sector boundary. However, it is not worth the effort. Instead, the solution is to design a better file system.)

For now, the directory is read-only, so files may not be created and the size of files may not be changed.

The classes **FileManager**, **FileControlBlock**, and **OpenFile** are provided for you, to make it easier to use the file system from within the kernel.

The "diskUtil" Tool

The BLITZ tool called **diskUtil** can be used to create a file system on the BLITZ disk, to add files to the disk, to remove files, and to print out the directory.

The BLITZ DISK is organized as follows. The disk contains a single directory and this is kept in sector 0. The files are placed sequentially on the disk, one after the other. Each file will take up an integral number of sectors. Each file has an entry in the directory. Each entry contains

(1) The starting sector
(2) The file length, in bytes (possibly zero)
(3) The number of characters in the file name
(4) The file name

The directory begins with three numbers:

(1) Magic Number (0x73747562 = "stub")
(2) Number of files (possibly zero)
(3) Number of the next free sector

These are followed by the entries for each file.

Once created, a BLITZ file may not have its size increased. When a file is removed, the free sectors become unusable; there is no compaction or any attempt to reclaim the lost space.

Each time the **diskUtil** program is run, it performs one of the following functions:

Initialize	set up a new file system on the BLITZ disk
List	list the directory on the BLITZ disk
Create	create a new file of a given size
Remove	remove a file
Add	copy a file from Unix to BLITZ
Extract	copy a file from BLITZ to Unix
Write	write sectors from a Unix file to the BLITZ disk

The following command line options tell which function is to be performed by **diskUtil**:

-h
Print help info.

-d DiskFileName
The file used to emulate the BLITZ disk. If missing, "DISK" will be used.

-i
Initialize the file system on the BLITZ "DISK" file. This will effectively remove all files on the BLITZ disk and reclaim all available space.

-l
List the directory on the BLITZ disk.

-c BlitzFileName SizeInBytes
Create a file of the given size on the BLITZ disk. The BLITZ disk must not already contain a file with this name. Only the directory will be modified; the actual data in the file will be whatever bytes happened to be on the disk already.

-r BlitzFileName
Remove the file with the given name from the directory on the BLITZ disk.

-a UnixFilename BlitzFileName
Copy a file from Unix to the BLITZ disk. If **BlitzFileName** already exists, it must be large enough to accommodate the new data.

-e BlitzFileName UnixFileName
Extract a file from the BLITZ disk to Unix. This command will copy the data from the BLITZ disk to a Unix file. The Unix file may or may not already exist; its size will be shortened or lengthened as necessary.

-w UnixFileName SectorNumber

The **UnixFileName** must be an existing Unix file. The **SectorNumber** is an integer. The Unix file data will be written to the BLITZ disk, starting at sector **SectorNumber**. The directory will not be modified.

We are providing a DISK file which should be large enough, but if you want, you may create a new BLITZ disk file of a different size. The new disk file must also be initialized properly; it can be created and initialized with the **format** command in the BLITZ emulator. For example:

```
% blitz
...
> format
...
The name of the disk file is "DISK".
The file "DISK" did not previously exist.  (It could not
                                        be opened for reading.)
Enter the number of tracks (e.g., 1000; type 0 to abort):
3
...
Initializing sectors 0 through 47...
Successful completion.
```

Next, we use **diskUtil** to create a file system, add several files, and print the directory, by typing these commands at the Unix prompt:

```
% diskUtil –i
% diskUtil -a temp1 MyFileA
% diskUtil -a temp2 MyFileB
% diskUtil -a MyProgram MyProgram
% diskUtil –l
   StartingSector   SizeInSectors    SizeInBytes        FileName
   ==============   =============    ===========    ====================
        0               1               8192            < directory >
        1               1               8192            MyFileA
        2               3               17000           MyFileB
        5               8               60264           MyProgram
```

The FileManager

There is only one **fileManager** object of the **FileManager** class; it is created and initialized at startup time.

We are supplying several methods to help you access files on the "stub" file system; these methods are located in this class. You will 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 Lab 7 than this lab. Read it all now to get familiar with the framework. You may want to review it again during Lab 7.

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** 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 "stub" file system this is pretty simple since there is only one directory and it fits into a single sector. The **FileManager**—as programmed now—reads the directory sector (sector 0) into a frame as part of the **FileManager.Init** method. Subsequent attempts to open a file require no disk accesses. (Of course, for a real file system, things won't be so simple.)

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 address 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 as "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
      fcbID: int
      numberOfUsers: int              -- count of OpenFiles pointing here
      startingSectorOfFile: int       -- or -1 if FCB not in use
      sizeOfFileInBytes: int
      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 ()
    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 **startingSectorOfFile** tells where the file is located on the disk. Since all the sectors in a file are contiguous, the starting address and the length are all we need. The meaning of **sizeOfFileInBytes** is... well, obvious. (Descriptive variable names like we tend to use are a HUGE help in understanding and reading code!) 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.

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
  methods
    Print ()
    ReadBytes (targetAddr, numBytes: int) returns bool        -- true = All Okay
    ReadInt () returns int
    LoadExecutable (addrSpace: ptr to AddrSpace) returns int  -- -1 = problems
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 ProcessControlBlocks (PCBs) to point to the same **OpenFile** object. This implies that 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 will need to increment the count. When a process closes a file (either by invoking the **Close** syscall or by dying), we will need to decrement the count. If the count goes to zero, we will 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 **OpenFile**s and **FCB**s. So how does a user process refer to an **OpenFile** object? Indirectly, through an integer. Here is 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 lab, you will not need to worry about **fileDescriptors**. However, it is important to understand how this mechanism works in BLITZ.

When a user-level program does a **Read** or **Write** syscall—in Unix or in the BLITZ 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 lab, we will only use **OpenFiles** to perform the **Exec** syscall, so the kind will be only **FILE** (and not **TERMINAL** or **PIPE**).

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) returns ptr to OpenFile
```

Open is passed a ptr to array of char; this is the name of the file on the BLITZ disk that you want to read from. It 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 will 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 a rather complex format. For details, you can read through the document titled "*The Format of BLITZ Object and Executable Files*." So that you do not have to write all this code, we have provided a method called **OpenFile.LoadExecutable** to you:

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

Read through **LoadExecutable in the class OpenFile**. 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 virtual address space) at which execution should begin. Normally, this will be 0x00000000.

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 virtual 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 virtual address space. The text segment will contain the instructions in the program and any constant values, the data segment will contain all static (global) variables that are not initialized to zeros, and the BSS ("Block Started by Symbol") segment will contain all static (global) variables that are not initialized explicitly in the program (such as large arrays as global variables), and therefore should be initialized to zeros.

OpenFile.LoadExecutable will initialize the entire BSS segment to all zeros. Most KPL programs do not use a BSS segment, so there will usually be zero BSS pages.

The environment pages, if any, will reside at address 0 and will contain information that the OS wishes to pass to a new user-level process. This includes 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 user-level program will have a stack, which will grow downward. Each virtual 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 the BLITZ 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 virtual 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 lab, all pages of the virtual 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 virtual 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 virtual 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 the 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 virtual address space.

Creating a User-Level Process

The **main** function calls function **InitFirstProcess**, which you must implement. The first thing you will 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 will 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 will 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 will 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.

Next, you will need to create the virtual 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 **initUserPC**.

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

Next, you will need to compute the initial value for the user-level stack, which you might call **InitUserStackTop**. It should be set to the virtual address just past the end of the virtual address space, since the initial push onto the user stack will first decrement the top pointer. The virtual address space starts at zero. The virtual 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 will 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 virtual 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 status register, causing the MMU to do virtual memory mapping;
8. Set the Interrupts Enabled bit in the status 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) above.

[What is the potential race condition 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 provided to you in the file **Switch.s**, which will take care of steps (4) through (9). In this case, **StartUserProcess** should end with a call to this routine:

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

BecomeUserThread will change the mode bits and perform the jump "atomically." This must be done atomically since the target jump address is a virtual address space. The way it does this is a little tricky: it pushes a fake exception info word, the new status register (with interrupt and paging enabled), and the PC onto the system stack, just as if a syscall or interrupt has occurred, and then executes a **reti** instruction.

BecomeUserThread jumps to the user-level **main** routine and never returns.

Approach to Implementing the Exec Syscall

The sequence of steps in **InitFirstProcess** and **StartUserProcess** is very similar to what you will need when implementing the **Exec** syscall. You should be able to copy much of this code when implementing **Sys_Handle_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 **ProcessControlBlock**, 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**. Do not 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
```

Do not 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 problem 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.

You will 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 not 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, and for this reason we provide you with 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
...
ret = currentThread.myProcess.addrSpace.GetStringFromVirtual (
            &strBuffer,
            filename asInteger,
            MAX_STRING_SIZE)
if ret < 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 not 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 through the trouble of creating classes like **ThreadManager** and **ProcessManager**, instead of simply allocating new **Thread** and **ProcessControl-Block** objects.]

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 physical 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 virtual address, not a physical address. As such, you cannot simply use the pointer as is. Take a look at these methods in **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.
Do not modify any other files except **Kernel.h** and **Kernel.c.** Do not create global variables (except for testing purposes). Do not modify the methods we have provided.

What to Submit

Please submit **Kernel.c** and **Kernel.h** using the command:

```
submitece353s 5 Kernel.c Kernel.h
```

Please do not modify any files except **Kernel.c** and **Kernel.h**. Do not create global variables (except for testing purposes).

Grading for this Lab

Your submitted solution will be marked by an autograder using test cases, and checked for potential plagiarism as well. The maximum possible mark for this lab assignment is 10. Task 1 will account for 2 marks, Task 2 will account for 4 marks, and Task 3 will account for 4 marks. Some of the test cases have been provided to you, in the form of **TestProgram1.c**, which calls **Sys_Exec** to load **TestProgram2.c**.

Sample Output from Provided Test Cases

Provided in the file called **lab5-sample-output.txt**, which you can download from the course website.