Blitz OS Overview
A4 Documentation

1 Docoument Overview

This document provides essential information about Blitz OS. We highly recommend reviewing it before starting your work on the assignments. While memorization isn't required, you should refer to this document as needed while completing the tasks in A4 and later assignments.

2 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
GetDiskInfo
Shutdown
GetPid
GetPPid
Yield
Exit
Fork
Join
Exec
Open
Close
Read
Write
Seek
Pipe
Dup
Stat
ChDir
OpenDir
ReadDir
ChMode (This and the following are Extra Credit)
Link
Unlink
Mkdir
Rmdir
```

(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 similar 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  (A "library" of useful routines.)
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, for example: (not all system calls are listed.)

System call	Wrapper function name
GetError	Sys_GetError
GetDiskInfo	Sys_GetDiskInfo
Shutdown	Sys_Shutdown
Yield	Sys_Yield
Exit	Sys_Exit
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. For our OS, the 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. Notice the "asInteger" conversion from a pointer to an integer. 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. The full list is, of course, found in "UserSystem.h" with their implementations in "UserSystem.k".

```
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 before assignment 7 is completed.

[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 User-Runtime.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. (Most current UNIX-like systems avoid large binaries by using "dynamic loading" and shared libraries.)

When an OS wants to execute its first 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 a new process in which to run the program. Once the system is running, we will create a new process using the "Fork" system call and run a new program in an existing process by using the "Exec" system call. The "Exec" system call and running a first process are related but different in that starting the first process we need to do both create a process and select a program to run in that 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, to execute "MyProgram", it will read the file from the simulated BLITZ disk and copy it into memory and create a process in which the program will run. The details of how to do this will be given in the section "User-Level Processes".

3 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. **ToyFs** files will be stored on the BLITZ disk, a terminal will control the serial device and a pipe will facilitate inter-process communication. All of these are done via the Read and Write system calls using an open file. For this assignment we will be working only with **ToyFs** files.

Next we describe each syscall in more detail. The following are the required system calls. The extra credit ones will be described when they are available to implement.

```
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." ("nop" is short for "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 original process's virtual memory space. 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 new process (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. If processID is not a child of the current process, that is an error.

```
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, replacing the entire address space of the current process with a new address space. 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 is an error, this function will return -1. The "args" argument is a pointer to an array of Strings.

```
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. The "mode" argument is used if a new file is created and the new file receives that value as the mode for the file. If all is OK, this function returns a file descriptor, which is a small, non-negative integer. It an error occurs, this function returns -1. (The create functionality of this system call is extra credit.)

```
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 read up to sizeInBytes characters from the terminal and will wait, if needed, until sizeInBytes characters have been read or a newline has been read.

This function is also used to read data from a pipe. More about the operation of read on pipes will be given in a future assignment.

This function will return the number of characters moved into user space. 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 or a pipe.

This function will return the number of characters moved. If there is an error, 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.

It is a special case if fileDesc refers to a directory. In that case, the only accepted value of newCurrentPosition is 0 and that resets the location for ReadDir to the beginning of the directory.

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

This function is passed the file descriptor of a file, which is assumed to be open. It closes the file, which includes writing out any data buffered by the kernel. If the file descriptor is invalid, nothing is done.

```
function Sys_GetPid () returns int
function Sys_GetPPid () returns int
```

These two functions returns the Process ID (Pid) of the calling process and the Pid of the parent process of the calling process. GetPid will always return a number 1 or greater. GetPPid may return -1 if the parent of calling process has terminated.

```
function Sys_Dup  (fd: int) returns int
```

This function copies the open file on the requested fd (file descriptor) to a new fd. If the fd is not open or the process is out of space for a new fd a -1 is returned.

```
function Sys_Pipe (fds: ptr to array [2] of int) returns int
```

This function opens a pipe. A return of 0 says a pipe was created and the array "fds" has the two file descriptors needed to access the pipe. fds[0] is the "read end" of the pipe and fds[1] is the "write end" of the pipe. If a pipe was not created, a -1 is returned.

```
function Sys_Stat (filename: String, statPtr: ptr to statInfo) returns int
```

This function returns information about the file named in the filename argument into the statInfo record. A return value of 0 says the system call was successful. A return value of -1 says some error happened.

```
function Sys_Chdir (filename: String) returns int
```

This function sets the working directory of the process. A return value of 0 reports success, a -1 reports an error and the working directory has not been changed.

```
function Sys_Opendir (dirname: String) returns int
```

This function opens a directory so that Sys_Readdir can return the entries found in the directory. It returns a non-negative file descriptor number if the directory was successfully opened or a -1 if the directory was not opened.

```
function Sys_Readdir (dfd: int, entPtr: ptr to dirEntry) returns int
```

This function returns the next directory entry in the open directory referenced by dfd. A return value of 0 if an entry is successfully copied to the dirEntry or a -1 if a directory entry could not be read.

```
function Sys_GetDiskInfo ( diskData: ptr to diskInfo) returns in
```

This function returns the information on the simulated DISK into a diskInfo record. If the diskData pointer is invalid a -1 will be returned. A 0 will be returned if the disk information was successfully stored.

4 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 "error".

When your code discovers and error, your code must set the proper error code in the "error" 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 "error" as that code has already set it. The idea here is that the code that discovers the error sets the "error" value and then returns an error code.

5 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.

6 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 "error" 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 processManager.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.)

7 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) reg-

isters 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, GetPid and GetPPid codes, there is a corresponding "handler routine" in the OS. Here is a partial list.

```
System call     Handler function in the kernel
GetDiskInfo     Handle_Sys_GetDiskInfo
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
```

Implementing these routines as well as implementing or adding code to support methods in this OS will be your work for this and remaining assignments in this project. These "Handle_Sys_*" routines have three primary jobs. These jobs are to validate the arguments passed by the user, copy any file name passed as an argument into a kernel array (String) and then call the proper method in the OS to do the actual work. Thus, most of the "Handle" functions are relatively short.

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. This assignment **DOES NOT** change the actual interrupt status. That can only be done from KPL by calling the function "SetInterruptsTo".

Also note that after the handler routines return to the interrupted routine, interrupts will be reenabled. 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.)

Some system calls need to run with interrupts disabled, but most of the system calls should re-enable interrupts before doing the work of the system calls. In "real OSes" it is very important to run for a minimum of time with interrupts disabled.

Several system calls have already been implemented for you as an example of how to implement system calls. First, GetError, GetPid and GetPPid are implemented in the SyscallTrapHandler and do not have an associated "Handle" routine because their implementation is just to return an easily accessible integer associated with the process. The system calls GetDiskInfo, Shutdown and Yield have also been implemented. Shutdown is implemented as:

```
function Handle_Sys_Shutdown ()
    -- Time to Stop the system!
    print ("Syscall 'Shutdown' was invoked by a user thread\n")
    RuntimeExit ()
  endFunction
```

Before we continue with the system calls, there are some other parts of BLITZ you need to know about.

8 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.

9 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. The implementation of the **ToyFs** is distributed across three classes in KPL, **ToyFs**, **InodeData** and **FileControlBlock**. These are where the core functionality of the **ToyFs** is implemented. Also, the **ToyFs** code will need blocks of memory, so **ToyFs** will be the user of the FrameManager methods GetAFrame() and PutAFrame().

9.1 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.

9.2 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 Kernel.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.

ToyFs supports two types of “files”, a directory and regular file. Mode types (how an file may be accessed) are read, write and execute. Both the file type and the mode are represented in the ”mode” field, the least significant 16 bits of the “nlinkdAndMode” field, with the following values bitwise ored together:

```
FILE      0x10
DIRECTORY 0x20
```

```
READ       0x04
WRITE      0x02
EXECUTE    0x01
```

9.3 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.)

9.4 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_Seek (fileDesc: int, newCurrentPos: int) returns int
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
```

Sys_Open, Sys_Close, Sys_Read and Sys_Write are not listed here, but they do interact with the file system. They also interact with other files in the system, specifically terminals and pipes.

For argument definitions, **StatInfo** and **DirEntry** are records declared in Syscall.h. These **ToyFs** system calls are already included in the Syscall.h definitions. 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, but will be offered as extra credit.

9.5 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.

10 The FileManager, OpenFile and FileControlBlock Objects

The **FileManager** is a class that manages **OpenFile** and Pipe objects as well as providing the non-file system specific parts of the system calls "Open", "Close" and "Pipe." There is only one

FileManager object; it is created and initialized at startup time. The Init function gets ready a collection of **OpenFile** and "Pipe" objects. The definition is:

```
class FileManager
  superclass Object
  fields
    fileManagerLock: Mutex
    openFileTable: array [MAX_NUMBER_OF_OPEN_FILES] of OpenFile
    anOpenFileBecameFree: Condition
    openFileFreeList: List [OpenFile]
    serialTerminalFile: OpenFile
    pipeTable: array [MAX_NUMBER_OF_PIPES] of Pipe
    pipeList: List [Pipe]
  methods
    Init ()
    Print ()
    GetAnOpenFile (block: bool) returns ptr to OpenFile
    GetAPipe () returns ptr to Pipe
    PutAPipe (pipePtr: ptr to Pipe)
    Open (localName: String, flags, mode: int, isDir: bool) returns int
    Close (open: ptr to OpenFile)
    Pipe (fdArray: ptr to array of int) returns int
endClass
```

This class is partially implemented as you will need to access files in order to create the first user-level process. Full implementation of files and the file system will be done in later assignments. But you will still need to know how to open a file to be able to execute the program in the file. To open an executable file, code will use the "Open" method in **ToyFs** to open a file and get an **OpenFile** object. The "ToyFs.Open" method requests an **OpenFile** from the **FileManager** and returns it ready to use to the caller. When done using an **OpenFile** object, code must call the "fileManager.Close" method to close the **OpenFile** object and add it back to the list of available **OpenFile** objects. As a note, the Open system call calls the fileManager "Open" method that then calls the **ToyFs** "Open" method if a **ToyFs** file is requested.

(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 several assignments.)

The design of this OS is such that there needs to be a "high level file representation." There are four kinds of files that will be supported by your OS at the end of this project. These are directory files and regular files as provided by the **ToyFs** subsystem. There are also pipes and a terminal file. Both of these will be implemented in assignment 7, but we need the mechanism to be able to talk about a file regardless of which kind of a file it is. This is done by the **OpenFile** object, which

we have already said is managed by the **FileManager** class. The class is defined as:

```
class OpenFile
  superclass Listable
  fields
    kind: int                       -- FILE, TERMINAL, or PIPE
    flags: int                      -- How opened, O_READ / O_WRITE
    numberOfUsers: int              -- count of Processes pointing here
    currentPos: int                 -- 0 = first byte of file
    addPos: int                     == byte address, for adding entries
    fcb: ptr to FileControlBlock    -- If this is a FILE or a DIRECTORY
    pipePtr: ptr to Pipe            -- If this is a PIPE
  methods
    Print ()
    Init (fKind: int, fFcb: ptr to FileControlBlock, openFlags: int)
    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
    Seek (newPos: int) returns int
endClass
```

When an **OpenFile** references a **ToyFs** directory or file, the **OpenFile** has a reference to another class provided by the **ToyFs** subsystem called a **FileControlBlock**." This is the "fcb" field. This **FileControlBlock** is used as the "point of contact" with the actual directory or file in the **ToyFs**. The **FileControlBlock** objects are managed by the **ToyFs** class and contain **ToyFs** specific data and methods. An **OpenFile** may reference a **FileControlBlock** but does not have to reference one. For example, a "Pipe" is also represented by an **OpenFile**, but instead of a **FileControlBlock** it references a "Pipe" object. This is the "pipePtr" field. Both the **OpenFile** and the **FileControlBlock** are needed when working with **ToyFs** directories and directories.

The purpose of a **FileControlBlock** (FCB) is to record all the data associated with a single **ToyFs** file. This includes a buffer to store up to a sector of file contents and the bufferIsDirty bit that says the buffer needs to be written back to the file as the buffer has new data. Here is the definition of FCB:

```
class FileControlBlock
    superclass Listable
```

```
 fields
   inode: InodeData
   fcbLock: Mutex
   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 ()
   Flush ()
   Release ()
   ReadSector (newSector: int) returns bool
   SynchRead (targetAddr, bytePos, numBytes: int) returns bool
   SynchWrite (sourceAddr, bytePos, numBytes: int) returns bool
endClass
```

The **FileControlBlock** (FCB) objects and the **OpenFile** objects are limited resources. The **FileManager** maintains a free list of **OpenFile** objects and the **ToyFs** maintains a free list of **FileControlBlock** objects.

The **ToyFs** file system and these three objects provide the core of the processing for files. 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 data 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. These two reads will use the same buffer, so this will require two copies of kernel data to the user virtual memory space. 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 read in the appropriate sector into the buffer so the write changes only those specific 20 bytes 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. It is also important to remember which sector is in the buffer.

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 have a bufferIsDirty field in the **FileControlBlock**. When a buffer is first read in from disk, it is considered to be ”clean” and the “bufferIsDirty” is set to false. After any operation modifies the buffer (e.g. write), it should be marked ”dirty”, that is to set “bufferIsDirty to true.

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. This then means that A and B also share access to the **FileControlBlock** and thus there is a filed “fcbLock” that allows A and B to synchronize their access to the FCB.

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 is written to the disk successfully.

The inode data (**inode** field) tells where the file is located on the disk. The FCB 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. The field “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 the same **ToyFs** file open will share a single FCB. However, they will each have a different ”current position” in the file. This is why we have both the class **OpenFile** and the **FileControlBlock**.

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 **ToyFs** 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 and set it up for file access.

When do we return an FCB to the free pool? When there are no more **OpenFile**s 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 **OpenFile**s 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).

Similar to UNIX, when a process is cloned with the Fork syscall, all open files in the parent process must be shared with the child process. This is done by sharing references to the **OpenFile** between the two processes. 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 **OpenFile**s, 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 **OpenFile**s 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 **OpenFile**s called fileDescriptor.

```
class ProcessControlBlock
  ...
  fields
    ...
    fileDescriptor: array [MAX_FILES_PER_PROCESS] of ptr to {\bf 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 **OpenFile**s to perform the Exec syscall, so the kind will be only FILE (and not TERMINAL or PIPE).

10.1 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 fileSystem.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 and must be a string in kernel memory, “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. fileSystem.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.

11 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

```
- 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.

(Note, the kernel codes still contains a reference to an "environment" page, but we will not be using it so the Kernel.h lists the number of environment pages at 0.)

Kernel.h contains this:

```
const
  USER_STACK_SIZE_IN_PAGES = 1
  MAX_PAGES_PER_VIRT_SPACE = 25
```

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:

```
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 text pages would be marked read-only and any attempt to update bytes in those pages would cause an exception. For our OS, we will also mark the text pages as read-only and if a write to the text pages is tried, it will also cause an exception.

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 loaded into memory at execution time and will be in memory throughout the run of that program.

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 onCpuThread.isUserThread
  currProc = onCpuThread.myProc
  RestoreUserRegs (&onCpuThread.userRegs[0])
  currProc.addrSpace.SetToThisPageTable ()
else
  currProc = null
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.