CSCI 447 - Operating Systems
Assignment 4: User-Level Processes

Points: 140

1 Overview and Goal

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

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

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

The kernel should be entirely protected from the user-level program; nothing the user-level program does should be able to crash the kernel.

2 Download New Files

Download the files provided on Canvas for this assignment. First, make sure you are on branch “main”. (“git checkout main”) (If you don’t have a “main” branch, try “master”.) Then get copies of the following files. For files with the same name as you currently have on your “main” branch, just copy the files over the current version. Once you have all these files copied, commit them all.

The following files remain the same from assignment 3 and will not have a copy in the assignment directory:

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

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

The following files were in assignment 3, but you need a new copy for assignment 4:

```
        Main.k
        makefile
```

The following files are new for assignment 4:

```
        MyProgram.h
        MyProgram.k
        TestProgram1.h
        TestProgram1.k
        TestProgram2.h
        TestProgram2.k
        UserRuntime.s
        UserSystem.h
        UserSystem.k
```

Kernel.h and Kernel.k are not provided. You start with the final version from assignment 3. It should be the version on your “main” branch. Remember to add and commit all of these files to your gitlab project.

3 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 = 140            -- for testing only
```

Next, in Kernel.k, change the code in DiskInterruptHandler function. Remove the line:

```
    FatalError ("DISK INTERRUPTS NOT EXPECTED IN PROJECT 3")
```

and uncomment the code:

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

4 Task 1: First user-level process (40 points)

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

"MyProgram" invokes the Shutdown syscall, which is already implemented for you as an example of a very simple system call.

4.1 Creating a User-Level Process

This is your first task, to load and execute the user-level program called "MyProgram".

The main function calls "processManager.InitFirstProcess()", which you must implement. The first thing you'll need to do is get a new thread object by invoking GetANewThread. Since the InitFirstProcess function should return, you cannot use the current thread. Next you'll need to initialize the thread and invoke Fork to start it running. Kernel.h has a constant **INIT_NAME** that is currently defined as **INIT_NAME = "MyProgram"**. You must use this definition to specify the first program your OS should start running. You should also use it to initialize the thread name.

The new thread should execute the **CallStartUserProcess** function, which will call "processManager.StartUserProcess". This indirection is necessary as KPL does not allow a method to be passed as a parameter. InitFirstProcess can supply a zero as an argument to CallStartUserProcess and can return after forking the new thread.

The function **CallStartUserProcess** has been provided, but unimplemented, just before the ProcessManager methods. The sole job of this function is to call "processManager.StartUserProcess". StartUserProcess will do the remainder of the work in starting up a user-level process.

The first thing you'll need to do in **StartUserProcess** is allocate a new PCB (with GetANewProcess) and connect it with the thread. So initialize the myThread field in the PCB and the myProcess field in the current thread. (You will need to declare local variables in the StartUserProcess method.) Since you are now running this method, you are running in the new thread and the variable "onCpuThread" is the the thread object to be associated with this PCB.

Next, you'll need to open the executable file. You must use **INIT_NAME** as the file name in the call to **fileSystem.Open**. To change which program you run, you simply change the definition in Kernel.h and recompile the kernel. That makes is much easier to change than if you had to find this code every time you need to change the initial program. If there are problems with the **Open**, this is a fatal, unrecoverable error and the kernel startup process must fail. **Open** will return a pointer to an **OpenFile**. A "null" value returned will indicate a problem.

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

Don't forget to close the executable file you opened earlier, or else a system resource will be permanently locked up. This is just a call to the **fileManager.Close** method.

Now is the time to set the "current working directory" of the new process. Since this is the first process, the working directory will be the root of the file system. The file system root is stored in the fileSystem as "fileSystem.rootDirectory". This is a pointer to an **OpenFile** object. Anytime in this system, when you do a "long term" copy of an OpenFile object, there are two things required. First, when you copy the pointer, you must do it using the **OpenFile.NewReference()** method. Second, when you are done using the new copy of the pointer to an open file object, you need to close it using the **fileManager.Close** method. This allows the OpenFile object to know how many pointers exist in the system to the OpenFile object and when it can return the OpenFile object to the appropriate manager when the last "user" has closed the OpenFile.

You will be copying the root directory pointer to the PCB object and save that reference as the "current working directory". Therefore, you need to use the "NewReference" method to copy it. It is going to stay in the PCB for the new process and thus is a "long term copy". Later we will see when you need to Close the working directory, but that is later.

Notice, the "Open" call for opening the executable above returns a pointer to a new OpenFile so you should **not** use "NewReference" with your use of "Open". But you **must** Close this OpenFile before you finish **StartUserProcess**.

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

```
        addrSpace.numberOfPages
```

pages. This value is set by the **LoadExecutable** method you called earlier and so setting **InitUserStackTop** depends on a successful **LoadExecutable**. Each page has size PAGE_SIZE bytes. Using those two numbers it is easy to calculate the first address after the end of the user's logical address space.

The **StartUserProcess** method will end by jumping into the user-level program. This is a one way jump; execution will never return. (Instead of returning, 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:

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

Do not forget the "&". It is a very common mistake to forget to use this address operator. You want the **address** of the last location in the systemStack, not the **data** in the stack.

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

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

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

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

The first 4 steps can be done in high-level KPL code, but steps (5) through (10) must be done in assembly language.

Read through the BecomeUserThread assembly routine in the file Switch.s., which will take care of steps (5) through (10). **StartUserProcess** should end with a call to this routine:

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

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

BecomeUserThread jumps to the user-level main routine and never returns. You should use 0 for "argPtr" in this call to **BecomeUserThread**.

4.2 Testing The First Process

At this point, you should be able to implement your **InitFirstProcess** and get it running. Your output should look like:

```
====================  KPL PROGRAM STARTING  ====================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 140
Initializing Disk Driver...
Initializing File Manager...
Initializing ToyFs...
Initializations finished, starting first process.

My user-level program is running!!!
Syscall 'Shutdown' was invoked by a user thread

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

5 Task 2: Syscall Handlers (20 points)

Modify several of 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 have at least one argument that is a pointer to an array of char. The system calls are:

```
Exec
Open
Read
Write
OpenDir
Stat
ChDir
```

There are five system calls that will be offered as extra credit and all five of these also take a pointer to an array of char. They are:

```
ChMode
Link
UnLink
Mkdir
Rmdir
```

The arguments that are “a pointer to an array of char” is actually a value that is a pointer in the user-program’s logical address space. You can not directly use this pointer because it is a “virtual address”. You must first move the string from user-space using the virtual address to a buffer in kernel space using a kernel space pointer. 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 and will be changed in future assignments.)

Fork	100
Join	200
Open	300
Read	400
Write	500
Seek	600
Stat	700
ChDir	800

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. Some of the system call handle functions are not modified in this part of the assignment.

5.1 Implementing the Handle_Sys_* Functions

Your next job is to start working with the "Handle_Sys_*" functions. Again, you need to know the style for these functions. As said before, they need to validate arguments and copy any string passed as an argument to a kernel variable and then call the functions in the rest of the kernel to do the functionality. As an example, the system call GetDistInfo has been implemented for you. It is:

```
-----------------------------  Handle_Sys_GetDiskInfo  -------------------------

  function Handle_Sys_GetDiskInfo (buffPtr: ptr to diskInfo) returns int
    var oldIntStat: int
        oldIntStat = SetInterruptsTo(ENABLED)

        if ! Valid_User_Pointer (buffPtr asInteger, 20, true)
           return -1
        endIf

        return fileSystem.GetDiskInfo (buffPtr)

      endFunction
```

In this case, it needs to validate the the "buffPtr" argument. This is **NOT** a kernel pointer and can not be used directly. This is a pointer valid only in the user address space. Thus, it needs to be checked to be sure it is a valid pointer in the user address space. So, because this is a common check, you are to implement a helper function "Valid_User_Pointer" that makes sure this pointer is valid. It not only has to be a valid pointer, but the entire buffer must be valid. In this case, it is known that a "diskInfo" structure requires 20 bytes to store it, so to be valid, the pointer must be valid and the pointer plus 20 bytes must also be valid. Finally, the kernel will be copying data into the user virtual address space and thus the user data buffer may not be in "user read-only memory". The kernel can copy anywhere, but we want to make sure that the user can't ask us to put data into a read-only page. That is why the third argument is "true".

The definition of ToyFs.GetDiskInfo is also given for you and is:

```
    --------------- ToyFs . GetDiskInfo  ----------------
    method GetDiskInfo (buffPtr: ptr to diskInfo) returns int
    var di: diskInfo
        rv: int
    di.diskSize = fssize
    di.totalInodes = numInodes
    di.freeInodes = i_bitmap.NumberOfClearBits()
    di.totalDblocks = numDblocks
    di.freeDblocks = d_bitmap.NumberOfClearBits()
    rv = currProc.addrSpace.
           CopyBytesToVirtual (buffPtr asInteger, (&di) asInteger, 20)
    if rv != 20
       return -1
    else
       return 0
    endIf
  endMethod
```

Some things to notice. First, this function has a local copy of the diskInfo and it uses kernel data to fill out the fields of this **local** record. It then uses the method “CopyBytesToVirtual” of the current processes AddrSpace to copy the record from kernel space to the diskInfo of the user process. This is a common pattern for kernel routines that need to put data into the user’s memory.

5.1.1 AllocateRandomFrames

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

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

As you have seen above, some of the user-level syscalls pass pointers to the kernel. For example, Open passes a pointer to a string of characters. Keep in mind that this pointer is a logical address, not a physical address. As such, you cannot simply use the pointer as is. So to help you copy data to and from user space in system calls, the following methods in AddrSpace have been implemented. Take a look at these methods AddrSpace:

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

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

There a few things to notice. First, if any errors are found these function set the error code properly and then returns an error value. Thus, if your code calls one of these and it returns an error, your code **must not** set an error code. The next thing to notice is that this code turns user virtual address space addresses into kernel space addresses so the data may be copied. The final thing to notice is the validation of user space addresses. You will need to do something similar when you implement the "Valid_User_Pointer" function. All virtual addresses must in the range 0 to PAGE_SIZE times the number of user pages. Also, for all pointers passed into a system call, they may not be 0 which is the value for a null pointer.

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

5.1.2 Handle functions

This is the time to add code to several of the "Handle_Sys_*" functions as listed above and make sure they print the required information and return the test values. Not all of the functions are included as the remaining functions are very similar to the ones you are required to complete. Read "TestProgram1.k" to see what you are required to print and the return values are specified earlier in this document and are also checked in the test program. To be able to print the names passed into some "Handle" functions, your code will need to copy the string from the user address space into a local array. Notice, you do not need to validate filename arguments as you will use "GetStringFromVirtual" (an AddrSpace method) to copy the string to a kernel array of characters. As you should know, "GetStringFromVirtual" will detect errors and return an error value. Also, you do not need to use the KPL "new" feature for these arrays as "GetStringFromVirtual" will do the "new" operation for you. You will need to declare an array in the "Handle" function when needed.

You might wonder how big an array you need in your "Handle" function for a file name. Well, "Kernel.h" has a constant:

```
    const  MAX_STRING_SIZE = 20
```

You should use this constant as it will change in a later assignment. Example code looks like:

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

You might think of allocating a temporary buffer on the heap, but remember that we do not want to allocate anything on the heap after kernel start-up. Also, with only 1000 bytes of system stack, where the above “strBuffer” is allocated, we can’t have big arrays in kernel functions.

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

Back to implementing your “Handle” functions. The first step is to change the “INIT_NAME” definition to run “TestProgram1”. Each assignment will have its own unique user level test program or programs. Now, start adding code for the handle functions requested, that is Fork, Join, Open, Read, Write, Seek, Stat, and Chdir. In all of them, you should write code to validate any pointers passed or copy any file names to a kernel array. (Note: “String” is not an array, it is just a pointer to an array and need the storage in your Handle functions.)

In this step, you should get the file names when needed. If “GetStringFromVirtual” returns an error code, your “Handle” function should return a -1. The error code will have been set properly by “GetStringFromVirtual”. To turn a pointer into an integer as needed, use the KPL “toInteger” cast. An example might be: “printInt(buffer asInteger)”. Also, don’t forget the “printIntVar” and “printHexVar” functions provided by the KPL runtime.

TestProgram1 has a function called “TestSyscallParameterPassing” in the “main” function. As distributed, the functions “TestSyscallErrors” and “TestExec” are commented out. You should get your program working first with those commented out.

5.1.3 Error Checking in the Handle Functions

Your next step is to implement the “Valid_User_Pointer” function and make sure you check the buffer pointers in the Read, Write, Stat and Chdir “Handle” functions with the “Valid_User_Pointer” function. It is done this way to make you write less code and to make sure you understand that helper functions are very important. Again, a valid pointer in the user space is a number between 1 and the number of pages in the address space times the page size. Also, you will need to figure out how to determine if the user page of a given address is a read-only page or a read-write page. This is part of the functionality of the AddrSpace class. You should read "CopyBytesToVirtual" to help you figure out if a user page is writeable. Also, do not forget that you need to validate the entire buffer. That means that the address passed is the lowest address of the buffer and that plus the size of the buffer is the highest address you need to check. Also, you need to check every user page between those two addresses.

Uncomment "TestSyscallErrors" to test the error checking in your "Handle" functions.

6 Task 3: Exec syscall (45 points)

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.

6.1 Implementing the Exec Syscall

Once you have the "Handle" functions and the error checking complete, you should then move to implementing the Exec System call. Notice, it was not part of the functions you were requested to implement and your first job will be to get the file name copied to from the user space. It is now time to uncomment the function "TestExec" in the "TestProgram1" main function.

The sequence of steps in StartUserProcess is very similar to what you'll need when implementing the Exec syscall. Your "Handle" function needs call "processManager.ExecNewProgram" to complete your "Handle" function. Then you should be able to copy much of the code from "StartUserProcess" to implement "ExecNewProgram". (Don't forget about the final BecomeUserThread step in StartUserProcess. It needs to be done for Exec.)

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

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

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

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

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

```
    currProc.addrSpace = newAddrSpace
```

Don't forget to free the frames in the previous address space first, or else valuable kernel resources will remain forever unavailable and the kernel will eventually freeze up! Also, re-enable interrupts as soon as you think it is safe to do so in the code. For Exec, that is very early in the process, clearly before you Open the new executable as Open may need to read the disk and that requires interrupts.

You should get exec working before starting command line arguments. Use the value of 0 for the "argPtr" parameter to **BecomeUserThread** function. The system call Exec **MUST** work for you to continue work on your OS. You do not need command line arguments working but is highly recommended because command line arguments are very important in assignment 8.

7 Task 4: Command line arguments (25 points)

Implement command line arguments to the Exec syscall.

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

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

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

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

The place to deal with command line arguments is after the new process address space has been built and before the previous address space has been freed. To make your code more readable, it would be best to write a helper method to copy the command line arguments from the old address space to the new address space. This copying will require multiple copies from the old address space to the kernel and he new address space. You may have a fixed maximum limit on the number of command line arguments. It should be 100 or greater. Exceeding that number should cause Exec to fail. Don't forget to free the new address space if an error should occur.

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

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

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

1. Get the number of arguments from the old address space.
2. Place the new array of strings at the bottom of the new address space. (Highest addresses in the stack page.) Calculate the placement of this array and save the pointer to the array as a virtual address in the new address space. This value will be an initial value for **InitUserStackTop** but as you copy arguments, you will decrese this value.
3. Initialize the new args array by storing the size to the new array. (This can be done by using the "CopyBytesToVirtual" method or by calculating a physical address and casting that pointer to a pointer to an integer and storing the size through the pointer.)

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

Once you have copied all command line strings, you then know where to start the user stack pointer so it starts above the command line strings on the stack. This pointer would be the minimum pointer of the args array. (Typically the last string copied will be placed at the lowest address.) If you had no errors in copying the command line arguments and creating the new structure in the new address space, it is then time to free the old address space and finish the exec process that ends with BecomeUserThread.

Hint: Since all data in the new address space is contained in one physical page/frame, using CopyBytesToVirtual is not required. In fact, you can generate a kernel address pointer to the new args array and use that pointer as an array. You will need to be computing physical addresses in the new stack page so you can so you can use GetStringFromVirtual and not copy the strings twice.

8 What to Hand In

Once you have completed this assignment, please create an “a4” branch. **You should not have an “a4” directory.** Capture a script of your runs for both MyProgram and TestProgram1 with all the tests running and commit them to your a4-branch as “a4-script” in the top directory of your project. Please turn in a pdf cover sheet on canvas that includes a link to your branch. List the most challenging aspect of the assignment and any functions that you were not able to implement correctly. If you want comments on your code, print your code to the .pdf you turn in. Please include only functions/methods that you wrote.

9 Coding Style

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

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

Please use our convention in labeling and commenting functions:

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

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

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

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

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

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

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

10 Sample Output

If your program works correctly, you should see something that looks like the file “A4-DesiredOutput.txt” that is on the assignment page.