CSCI 509 - Operating Systems Internals
Assignment 5: Multiprogramming with Fork

Points: 150

1 Overview and Goal

In this assignment, you will implement the following syscalls: Fork, Join, and Exit. After completing the previous assignment, the OS could support a single user-level process. In this assignment, you will add the necessary functionality to run many user-level processes at the same time.

2 Download New Files

The files for this assignment are available on the canvas assignment page. The following files are new to this assignment:

```
TestProgram3.h
TestProgram3.k
TestProgram3a.h
TestProgram3a.k
```

The following files have been modified from the last assignment:

```
makefile
```

The makefile has been modified to compile TestProgram3 along with other changes to support assignment 5. All remaining files are unchanged from the last assignment and there is a file named “Desired-output.txt” that shows the correct output of the test program.

The following are the tasks for this assignment:

1. Implement the Fork syscall. (70 points)

2. Implement the Exit syscall. (20 points
3. Implement the Join syscall. (35 points)

3 Changes to Kernel.h and Kernel.k

Please change Kernel.h from:

```
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 140
```

to:

```
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
```

Also change INIT_NAME in kernel.h to "TestProgram3".

4 Approach to this Assignment

First, you should read this entire document before trying to implement anything. Then you should follow the list of tests in the "TestProgram3", starting with "SysExitTest()". Implement the functions it requires and then make sure you pass the test. Then you can move on to the next one in the list. Fork, Exit and Join are all really interconnected, but you can get some working before others. It may require an iterative approach to implementing the functions and methods required.

The TestProgram3 main function looks like:

```
function main ()

    SysExitTest ()
    -- BasicForkTest ()
    -- YieldTest ()
    -- ForkTest ()
    -- JoinTest1 ()
    -- JoinTest2 ()
```

```
    -- JoinTest3 ()
    -- JoinTest4 ()
    -- ManyProcessesTest1 ()
    -- ManyProcessesTest2 ()
    -- ManyProcessesTest3 ()
    -- ArgsWithForkTest()
    -- ErrorTest ()

    Sys_Exit (0)
  endFunction
```

The points for the work in this assignment are as follows: Fork, Exit and Join are 125 points (as specified above) with 25 for all the other jobs you need to do including your time log.

5 The Fork Syscall

When a user-level process wishes to create another process, it invokes the **Fork** syscall. The kernel will then create a new process and assign it a process ID (a "pid"). This will involve creating a new logical address space, loading this address space with bytes from the current address space, and creating a single thread to run in the new space. The kernel must also copy the CPU machine state (i.e., the registers) of the current process and start the new process off running with this machine state. Thus, the newly created process is an exact clone of the first process.

While we don't have file I/O yet, the **Fork** syscall must also ensure that all files that are open in the parent will also be open in the child. You will need to add this functionality to **Fork** in a later assignment when file I/O is implemented.

The initial "parent" process is the one invoking the **Fork**. Its thread will do the work of creating the new process. The new process and its thread are immediately runnable, so the new thread should be placed on the ready list.

In the parent, after creating the new process, the thread returns to the user-level code. It returns the pid of the child process. In the child process, since it has exactly the same state, once it runs, it too should return from the **Fork** syscall. However, the value returned should be zero.

If we make an exact copy of the machine state, an exact copy of the system stack, an exact copy of the virtual address space, and an exact copy of the user stack, it is easy to resume the execution of the child process. However, it will do exactly what the parent process does

unless there is a way for the processes to easily determine which one is the new process. This can be done by returning 0 to the new process.

5.1 The Parent-Child Relationship

Each process will have a process ID and these are unique across all processes, past, present and future. (Of course, with a finite-sized integer, there is the possibility that the counter will wrap around; this would be disastrous, but we'll ignore the possibility.) A new pid should have been assigned in the ProcessManager.GetANewProcess method you wrote earlier.

Each time a process does a **Fork**, it creates a child process. A single process may create zero, one, or many children. The children may go on to create other children, which are called "descendants" of the original "ancestor." The parent may terminate before its children, or some or all of its children may terminate first.

Given a process, you will occasionally need to know which process is its parent. The ProcessControlBlock contains a field called parentsPid, which you can use. The simplest approach is to search the processManager.processTable array, looking for a process with that pid. This linear search will take a little while, but not too long. Likewise, if you want the child of a process P, you can do a linear search over the array looking for a process whose parentsPid is P. (Note, there is "processManager.FindProcess(pid:int)" method as a helper method. It is not implemented and implementing it will help when you need to find a process.)

In general, linear searches are to be avoided in OS kernels, but in our simplified OS, a linear search of PCBs is acceptable. (And expected.)

You might think that it would be smarter to store a pointer to the PCB of the parent right in the PCB of the child.

```
class ProcessControlBlock
  fields
    ...
    parent: ptr to ProcessControlBlock     -- an idea???
  ...
endClass
```

Unfortunately, there is a problem with this approach. PCBs are recycled when a process terminates and are then used for other processes. If we store a pointer to a PCB in the parent field, when we need the parent and we follow this pointer, we might get a PCB that now holds a completely unrelated process.

Although the linear search approach is just fine, a better design would be to store both a pointer to the parent's PCB and the parent's pid. Then you can follow the pointer to a PCB and then check to make sure that pid of this PCB is the pid of the parent.

Our OS will differ a little from Unix/Linux in how we deal with a parent which has terminated. In our OS, the parent (P) of process (C) may have terminated, and you will need to check for this possibility. In Unix/Linux, when a process P terminates, all of its children processes are given a new parent. In particular, all children are "reparented" to become children of process 1, the "init" process. Thus, in Unix/Linux, every process will always have a parent. In our OS, when a process terminates, you should set the parent PID to be -1 to show that the process has no parent. This allows a process to terminate quickly and not become a zombie if it has no parent. Also, in most Unix/Linux systems, if process number 1 terminates, the OS terminates with an error. That will not be a feature of our OS.

5.2 Implementing Handle_Sys_Fork

The code that implements the **Fork** syscall needs to do (more-or-less) the following:

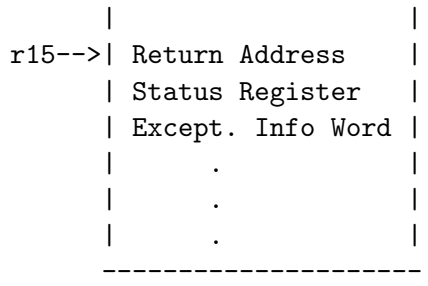
1. Get the return address from the user stack to use as the initPC for the child.
2. call processmanager.ForkNewProcess sending in the return address.
3. return the PID of the new child. This should be the return value of ForkNewProcess.

This design is in line with trying to have the "Handle" functions just check for errors. Since "Fork" takes no parameters, no checking is needed. But, in preparation for starting the child at the proper place so it returns from the "Fork" function, we must let the new thread know where (in the user-level address space) to resume execution. This must be determined by inspecting the parent process. We will call the User Space the "bottom-half" and the kernel space the "top-half". So, we need to get this "return address" from the bottom-half.

To approach this, ask how execution returns to the bottom-half after any syscall? In the case of a **Fork** syscall, how will the current thread (the parent process) perform its return to some instruction in the parent's virtual address space?

When the syscall instruction was executed, the assembly language SyscallTrapHandler was called. It invoked the high-level SyscallTrapHandler function, which in turn called Handle_Sys_Fork. When we are ready to return, the Handle_Sys_Fork function will return to the high-level Syscall-Traphandler, which will return to the assembly SyscallTrapHandler, which will execute the "reti" instruction.

At the very beginning of the **Fork** processing, a ”sycall” instruction was executed. When the ”syscall” instruction was executed, the CPU pushed an ”exception block” onto the system stack. Directly before the CPU jumped to the assembly SyscallTrapHandler, the system stack looked like this:



Subsequently, more stuff was pushed onto the stack when the high-level SyscallTrapHandler was called and more stuff was pushed when the Handle_Sys_Fork function was called, but by the time we return to the point directly before the ”reti” instruction, everything that got pushed will have been popped. The ”reti” instruction will then pop the 3 words of the exception block and will use the ”return address” to determine where to resume user-mode execution.

You’ll need to get that return address from the system stack and you’ll need to get it from within Handle_Sys_Fork. Unfortunately, it will be buried somewhere below the top of the system stack.

Fortunately, there is a function called GetOldUserPCFromSystemStack in Switch.s, which will do exactly what you need. Yes, it was written in assembly language for this purpose. It also assumes that are calling this function from a “Handle” function. This function is defined as:

```
external GetOldUserPCFromSystemStack () returns int
```

Here is the assembly code:

```
!
! ===============  GetOldUserPCFromSystemStack  ===============
!
! external GetOldUserPCFromSystemStack () returns int
!
! This routine is called by the kernel after a syscall has
! occurred.  It expects the assembly SyscallTrapHandler to have
```

```
! called the high-level SyscallTrapHandler, which then called
! Handle_Sys_Fork.  It expects to be called from Handle_Sys_Fork,
! and will not work properly otherwise.
!
! This routine looks down into stuff buried in the system stack
! and finds the exception block that was pushed onto the stack
! at the time of the syscall.  From that, it retrieves the user-mode
! PC, which points to the instruction the kernel must return to
! after the syscall.
!
GetOldUserPCFromSystemStack:
        load      [r14],r1        ! r1 = ptr to frame of SyscallTrapHandler
        load      [r1+28],r1      ! r1 = pc from interrupt block
        store     r1,[r15+4]
        ret
```

You'll need to call this function from the Handle routine to get the value. Let's call the value it returns the oldUserPC. You'll need to pass this address to the processManager function "ForkNewProcess" so when everything is set up and your code is ready to resume the child, that code can pass this return value to the function "Finish_Fork_For_Child" allowing the child can use it.

5.3 Implementing ForkNewProcess

Now the job has been passed to the processManager to fork a new process. The steps involved in this are:

1. Allocate and set up new Thread and ProcessControlBlock objects.
2. Make a copy of the address space.
3. Invoke Thread.Fork to start up the new process's thread.
4. Return the child's pid.

The newly started thread needs the following:

1. Set currProc to the new process.
2. Initialize the user registers.

3. Initialize the user and system stack pointers.
4. Invoke BecomeUserThread to jump into the new user-level process.

Let's call the initial function to be executed by the newly created thread, "Finish_Fork_For_Child. Yes, there is such a function in Kernel.k ready for you to complete it. The "ForkNewProcess" method will perform steps 1 through 4 of the first list above, which will include creating a new thread. The new thread will begin by executing the "Finish_Fork_For_Child", which will perform steps 1 through 4 of the second list, completing the operation of starting the new process. Notice, that the oldUserPC needs to be passed along the "resume" function.

There are many details, so next, we will go through these steps more carefully.

The first thing to do in "ForkNewProcess" is to obtain a new ProcessControlBlock and a new Thread object.

Next, you'll need to initialize the following fields in the PCB: myThread and parentsPid. (The pid field should have been initialized in GetANewProcess.)

You'll also need to initialize the following fields in the new Thread object: name, status, and myProc.

Recall that a user-level program was executing and it did a **Fork** syscall. At that point, the state of the user-level process was contained in the user registers. These registers have not changed since the system call. (Well, maybe the call to GetANewThread or GetANewProcess caused this thread to be suspended for a while. But during any intervening process switches, the user registers would have been saved and subsequently restored.)

Recall that the BLITZ CPU has 2 sets of registers: system and user registers. Some of the time, threads run in system mode (when handling an exception or a syscall) and some of the time, they run in user mode. Sometimes we talk about the top-half and the bottom-half of a thread. The top-half is the kernel part, the part that runs in system mode. The bottom-half is the part of the thread that runs in user mode.

Next you must grab the values in the user registers and store a copy of them in the new Thread object. You can use SaveUserRegs to do this.

Recall that all syscall handlers begin running with interrupts disabled. After getting the user registers, it would be a good idea to re-enable interrupts so that other threads can share the CPU.

We are in the middle of starting a new thread and this new thread will need a system stack. In terms of the bottom-half, the new thread must be a duplicate of the bottom-half of the

current thread, but the top-half need not be the same. In a few instructions, we are going to do a **Fork** on this Thread. We don't need anything from the system stack of the current thread. So there is no reason to copy the system stack. The systemStack array has already been initialized so there is no need to do that again. You can simply leave the contents (left over from some previous thread) in the array. All you'll need to do is initialize the stackTop pointer, which should be initialized to point to the very last word in the systemStack array, just as it was in the Thread.Init method.

```
newThrd.stackTop = & (newThrd.systemStack[SYSTEM_STACK_SIZE-1])
```

There is no reason to initialize the system registers for the new top-half. They can just pick up leftover values from some previous thread.

In this assignment, we have not yet implemented the file-related syscalls (Open, Read, Write, Close, etc.), so we don't have to do anything related to open files. However, in a future assignment, each process will have some open files. The child process should share the open files of the parent. In other words, if a file is open in the parent before the **Fork**, it should be open in the child after the **Fork**.

So at this point in "ForkNewProcess", it is recommended that you add the following line:

```
-- Don't forget to copy the fileDescriptor array here...
```

This comment will make sense later.

Next, you'll need to make a copy of the parent's virtual address space. You'll need to see how many pages are in the parent's address space and call frameManager.GetNewFrames. Then you'll need to run through each and copy the page. You can use MemoryCopy to do this efficiently. You can use AddrSpace.ExtractFrameAddr to determine where the frames are in physical memory. You'll also need to set the "writable" bit in the child's frame to whatever it was set to in the parent's frame. (See AddrSpace.IsWritable, AddrSpace.SetWritable, and AddrSpace.ClearWritable.)

You are now ready to fork the new thread. As with any fork, you need to provide a pointer to a function and a single integer argument. As an argument, you can pass oldUserPC.

```
newThrd.Fork (Finish_Fork_For_Child, oldUserPC)
```

Once you have called Thread. Fork, the new thread will finish the work of returning to the child process and will become the thread of the newly created process. The parent thread is now ready to return from "ForkNewProcess" to the parent user-level process.

Next, let's look at what you'll need to do in the "Finish_Fork_For_Child" function.

The key piece of info "Finish_Fork_For_Child" needs (besides the info stored in the Thread) is the address in the user program to return to. This is just the value returned from GetOldUserPCFromSystemStack which was passed as an argument to "Finish_Fork_For_Child".

Basically, "Finish_Fork_For_Child" needs to switch into user mode and jump to this address. Fortunately, we have an assembly routine that does just this: BecomeUserThread.

Notice that "Finish_Fork_For_Child" will bear a strong resemblance to the code in StartUserProcess.

Every thread begins with interrupts enabled. Since you will need to do things that might involve race conditions, you should begin by disabling interrupts.

Now, you need to update the new process and thread. This is one place where you should set currProc. You have switched threads but this thread was not listed as a user thread so currProc was not set. But it does have an associated PCB. So now is the time to set currProc to the PCB associated with the onCpuThread.

Next, you'll need to initialize the page table registers to point to the page table for the child process, so invoke AddrSpace.SetToThisPageTable.

Then, you'll need to set the user registers before returning to user mode. You have the values of the registers (stored in the Thread object) but you need to copy these values into the registers. This is exactly what the RestoreUserRegs function does.

You'll also need to set isUserThread to true. [The isUserThread field in Thread is used for one thing: to determine whether the user registers should be saved and restored every time a context switch occurs. This variable is consulted only in the Run function just before and after calling Switch.]

Once you begin executing the user-level code, you'll want an empty system stack. You can compute the initial value for the system stack top just as you did in StartUserProcess.

You next need a value for initUserStackTop. Because the parent was already running and was using the stack, this can not be the "bottom of the user stack" as it was in Exec. The parent's value for the user stack top is user register r15, which was stored in the Thread object by "ForkNewProcess". You need to get that value for your initUserStackTop as your forked process needs to look like it is running in the same place and with the same user stack as the parent process.

Finally, you can jump into the user-level process with the following:

```
BecomeUserThread (initUserStackTop,     -- Initial Stack
                  initPC,               -- Old User PC
                  initSystemStackTop,   -- Initial System Stack
                  0)                    -- No Arguments
```

Recall that we need to return pid=0 to the child process. To see how to do this, we need to see how any value is returned from any syscall.

Look at it from the user-level code's point-of-view. The user-level code executed a "syscall" instruction and, after the kernel has returned, the user code will expect the return value to have been placed in user register r1. See DoSyscall in UserRuntime.s for details.)

When you invoke BecomeUserThread, a jump will be made to the instruction immediately after the syscall. All the registers and the entire address space will be identical to what they were before the syscall was executed (in the parent), so the child will see this as a normal "return" from a kernel syscall. The instructions just after the syscall will fetch the value in r1 and return it to the high-level KPL Sys_Fork function. So all you have to do in the kernel is make sure the right value (namely, zero) is in user register r1.

Luckily for you, BecomeUserThread just happens to store a zero in user register r1 right before making the jump into user-land. (Do you suppose this was a coincidence???) This will cause the Sys_Fork user-level function to get a returned result of zero, which it returns as a pid to distinguish the child from the parent.

The Args value should be zero since we are not starting at the "entry" to the program, but returning from Sys_Fork. The final argument for the Args vector sets user r2 just before the return from interrupt.

Next, let's talk about two subtleties in the implementation of the "ForkNewProcess".

First, consider this race possibility: Let's say your code in "ForkNewProcess" ends by invoking **Fork** to start the new thread running and then returning newThrd.pid. So the last lines in "ForkNewProcess" might be something like this:

```
newThrd.Fork (Finish_Fork_For_Child, oldUserPC)
return newPCB.pid
```

Now suppose that right after **Fork** is invoked, the child gets scheduled before the parent gets to do the return. What if the child starts up, finishes the syscall, returns to the user program, and then the user-level child process runs all the way to completion and the child process terminates altogether? Could the PCB be returned to the free pool, then reallocated to some

other process and have a new pid value stored in it, before the parent gets to execute its return statement? When the parent finally resumes, might it grab the pid out of the PCB (getting a wrong value!) and return that (wrong) pid to the user-level code?

No, this cannot happen. The child may terminate before the parent fetches the pid out of the PCB but child PCB will become a zombie and will not be given to another process until after the parent terminates. See the discussion of Zombies below for details.

Second, consider in more detail the call to RestoreUserRegs and the assignment of true to isUserThread.

Once you set isUserThread to true, any time there is a context switch, the user registers will be copied to the Thread object (overwriting anything stored there!) as part of a switch from one thread to another. Therefore, you must call RestoreUserRegisters before setting isUserThread to true, or else a timer interrupt might cause a thread switch, which would wipe out the user register data stored in the Thread object, if it happened to occur before the call to RestoreUserRegs completed.

One the other hand, if you call RestoreUserRegs before setting isUserThread to true, it is possible that a context switch could occur after initializing the user registers but before you set isUserThread to true. The thread scheduler will see that isUserThread is false and will not save the user registers. Any intervening processes might change the user registers. Again, the register values get lost!

It seems that both orders are subject to a race bug, but there is a simple solution.

Recall that every thread initially begins with interrupts enabled. The solution is to disable interrupts in "Finish_Fork_For_Child". Then there is no possibility of a context switch between the call to RestoreUserRegs and setting isUserThread to true. (This is very important. If you don't set this to true, really weird bugs happen!) Also, recall that BecomeUserThread will re-enable interrupts as part of the process of resuming execution in user mode.

6 The Semantics of Join and Exit: Why Zombies are Needed

Just as in Unix, when a process terminates, it provides an exit code. This is provided in the call to Sys_Exit. For a "normal" termination, the convention is that zero is returned. The PCB contains a field to contain this value, called exitStatus. Your kernel must keep the PCB around to hold this value until it is no longer needed.

Once a process terminates, the kernel can release all of its resources, such as the Thread object and any OpenFile or FCB objects that are no longer needed. Only the PCB needs to

remain around. The PCB then becomes a "zombie." A zombie is a creature that has stopped living but is not quite dead.

When can a zombie PCB be freed? Whenever (1) its parent either dies or becomes a zombie itself, or (2) its parent executes a join and takes the exit status, or (3) its parent doesn't even exist.

The following approach is recommended: First, ignore the ProcessManager.FreeProcess method. In other words, either get rid of it or just don't ever call it. (We asked you to write it so it could be used to test ProcessManager.GetANewProcess.)

[By the way, whenever you have a routine you wish to keep but will not ever use, I recommend placing a line like this as the first statement:

```
method FreeProcess (p: ptr to ProcessControlBlock)
    FatalError ("Never called")
    ...
  endMethod
```

This way, you still have the code in case you change your mind, but it is clear when reading your code that it is not used. Also, if you make a mistake and try to use the routine, you'll get an immediate error.]

Second, complete the three methods in ProcessManager: TurnIntoZombie, WaitForZombie and ProcessFinish. (ProcessFinish is the first method that needs to work to successfully complete the first test function in TestProgram3.)

6.1 method ProcessFinish (exitStatus: int)

This method is called when a process is to be terminated. This function is called by the process's thread. It will free all resources held by this process and will terminate the current thread. The PCB will be turned into a zombie. This method will have to do the following...

1. Save the exitStatus in the PCB.
2. Disable interrupts.
3. Disconnect the PCB and the Thread, i.e., set myProc and myThread to null. Also, set isUserThread to false.
4. Re-enable interrupts.

5. Close any open files. (For the next assignment. Add a comment to remind you about this need.)
6. return all page frames to the free pool, by calling frameManager.ReturnAllFrames.
7. Invoke TurnIntoZombie on this PCB.
8. Invoke ThreadFinish.

One final note, once the thread has isUserThread set to false, if a process switch happens when it comes back to this function currProc will be null. So, you must save the currProc pointer so you have it even is a process switch happens.

6.2 method TurnIntoZombie (p: ptr to ProcessControlBlock)

This method is passed p, a pointer to a process; It turns it into a zombie - dead but not gone! - so that its exitStatus can be retrieved if needed by its parent.

1. Lock the process manager since you'll be messing with other PCBs.
2. Identify all children of this process who are zombies; These children are now no longer needed so for each zombie child, change its status to FREE and add it back to the PCB free list. Don't forget to signal the aProcessBecameFree condition variable for each zombie child, since other threads may be waiting for free PCBs.
3. Identify p's parent. (Note, the parent may have already terminated, so there might not be a parent.)

   If p's parent is ACTIVE, then this method must turn p into a zombie. Execute a Broadcast on the aProcessDied condition variable, because the parent of p may be waiting for p to exit.

   Otherwise (i.e., if our parent is a zombie or is non-existent) then we do not need to turn p into a zombie, so just change p's status to FREE, add it to the PCB free list and signal the aProcessBecameFree condition variable.
4. Unlock the process manager.

6.3 method WaitForZombie (proc: ptr to ProcessControlBlock) returns int

This method is passed a pointer to a process; It waits for that process to turn into a zombie. Then it saves its exitStatus and adds the PCB back to the free list. Finally, it returns the exitStatus.

1. Lock the process manager.
2. Wait until the status of proc is ZOMBIE, using a while loop and the aProcessDied condition variable.
3. Fetch proc's exitStatus.
4. If the child was killed by an error, set the last error to E_User_Process
5. Change proc's status to FREE, add it to the PCB free list, and signal the aProcessBecameFree condition variable.
6. Unlock the process manager and return the exit status.

6.4 Exit and Join System Calls

Handle_Sys_Exit is now straightforward to implement: just call ProcessFinish, which will store the exit status in the PCB and free all resources except the PCB. ProcessFinish will also free the PCB too if it is not needed.

Handle_Sys_Join is also fairly straightforward. First, you have to identify the child process and make sure that the pid that is passed in is the pid of a valid process and that it is truly a child of this process. (If not, the kernel should return -1 to the caller.) Then you can call WaitForZombie and return whatever it returns. This return should be the value passed to ProcessFinish when ProcessFinish was called. Notice, that unlike UNIX, this is a full 32-bit integer and so the user can call Sys_Exit with a -1. So the question can be asked: Is the -1 return from Sys_Join mean an error occurred or that the child process returned a -1? This could be hard to answer correctly if we didn't have the Sys_GetError system call. A true error should have an error value other than E_No_Error.

Notice, at this point, an error would be that the calling process is not the parent of the process with the given pid. But what if the user process was terminated with an error? The first question is what happens to make a process terminate with an error. The second would be how would the kernel know the user process terminated with an error.

When the user process has a KPL detected error, like accessing a null pointer, the OS does not detect the error. The user process just does a Sys_Exit with the value of -2. But when the user process generates a page fault due to a bad pointer value, the OS will know there is a user process error because of "Address Exception". There are several other kinds of exceptions that the kernel will catch that also indicate errors in the user process. These include "Page Invalid", "Page Readonly", "Alignment" and "Privileged Instruction". If we were doing full paging, "Address", "Page Invalid" and "Page Readonly" may not be errors, but given that our programs are completely loaded into memory and are never removed, these are all errors in the user program.

When a user process has one of these errors, the CPU throws the proper exception and the kernel starts running in the proper Exception Handler. Read the code and see that all of these exception handlers call "ErrorInUserProcess". This is where we deal with setting an error value and then call the function to finish the process. The exit value must be -1.

The complicating factor shows up in Sys_Join. The parent process calls Sys_Join and if the child process was terminated with an exception error, Sys_Join must return -1 and the error value must be E_User_Process. This must be communicated from the "ErrorInUserProcess" which is called with the child process running to the parent process calling the Join. The only way to communicate this is via setting a value in the child PCB (not the exitStatus) and in the processing done by Sys_Join.

6.5 TestProgram3

Again, your kernel will be tested by the user program "TestProgram3". Now that you have read this far, it is time to implement the code for Exit, Fork and Join. It would be best to attempt to implement all the above functions as a group since they all work together. Then, using the TestProgram3, verify your code works as explained before, starting with "SysExitTest" to make sure it works correctly.

Remember, errors can happen through this process and you need to clean up and correctly set the last error code when returning -1 as the value from Stat. Again, your code sets the error values only when your code discovers the problem.

7 Enhancement, Not Required

Modify the exception handlers (such as AddressExceptionHandler) so that they print the pid of the offending process and the name of the process. Also, upgrade the Exec system call so that the name of the process is the name of the executable file that was used to load the process address space. This can be accomplished by having a name array in each Thread instead of having a pointer.

8 The User-Level Programs

As mentioned above, there is a user-level program called "TestProgram3". There is a companion program called "TestProgram3a". These programs constitute the assignment test suite.

TestProgram1 and TestProgram2 will not be used in this assignment and may no longer function correctly due to changes in the system calls as you implement them. TestProgram3* should remain working for the remainder of your work on the Blitz OS. You should use TestProgram3* unmodified.

9 What to Hand In

By now you should know that TestProgram3 has a number of individual tests that you run by uncommenting them. In this program, only one test will work at a time. You can not uncomment more than one test and expect them all to work like you did in the last assignment.

Do not change TestProgram3*, except to uncomment one of the lines in the main function.

During your testing, it may be convenient to modify the tests as you try to see what is going on and get things to work. Before you make your final test runs, please recopy TestProgram3* from the assignment directory, so that you get a fresh, unaltered version. (Or checkout a fresh copy from your git repository.) Your kernel will be graded using the distributed copies of TestProgram3*

TestProgram3a is provided to test command line arguments, and to use GetPid and GetPPid to make sure the pid values are correct.

After you have finished the assignment and checked in your working copies of Kernel.h and Kernel.k, create an "a5" branch. Then run each test in TestProgram3.k once capture the output by doing these tests using the UNIX script program. Put all your test output in a file named "a5-script". Use the same Kernel code to execute all tests. Commit this a5-script to your a5 branch in the repository root directory.

Turn in a pdf file that contains a link to your repository. List the most challenging aspect of the assignment and any functions that you were not able to implement correctly. Add any code on which you want comments. Make sure you submit a single .pdf document to canvas.

10 Sample Output

If your kernel works correctly, you should see something like what is in the file Desired-output.txt that is posted on the canvas assignment page. That is a run done with the reference implementation. Your output may not look identical but should be very close to what is in that file. The file is an edited script of running all tests in the TestProgram3.k in the order

listed in the main function.