Lab 6:
Multiprogramming with Fork

Due date: March 15, 2018, Thursday, in the lab.

Submission deadline: Your solutions will also need to be submitted electronically by 11:59 p.m. of the day when your lab is due. For this lab, it will be due on March 15, 2018, 11:59 p.m.

Objectives

In this lab, you will implement the following system calls: **Fork**, **Join**, **Exit**, and **Yield**. After completing the previous lab, the BLITZ OS is able to support a single user-level process. In this lab, you will add the necessary functionality to run multiple user-level processes at once.

Files for this Lab

All the files for this lab are available at the following directory on the ECF workstations:

/share/copy/ece353s/lab6 (after logging into your account in the ECF UNIX lab)

The following files are new to this lab:

```
TestProgram3.h
TestProgram3.c
```

The following files have been modified from the last lab:

```
makefile
DISK
```

The **makefile** has been modified to compile **TestProgram3**. The **DISK** file has been enlarged, since the previous version was too small to accommodate **TestProgram3**.

All remaining files are unchanged from the last lab.

You have the following two tasks in this lab. Read the rest of the handout before attempting the tasks.

Task 1:

Implement the **Fork** syscall.

Task 2:

Implement the following syscalls:

Join
Exit
Yield

Changes to Kernel.h and Kernel.c:

Copy your **Kernel.h** and **Kernel.c** from the previous lab into this lab.

Please change **Kernel.h:**

from:

```
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 100
```

to:

```
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
```

Please change your **InitFirstProcess** routine in **Kernel.c:**

from:

```
th.Fork (StartUserProcess, "TestProgram1" asInteger)
```

to:

```
th.Fork (StartUserProcess, "TestProgram3" asInteger)
```

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 virtual 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.

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 did. It is rather difficult to make it do anything different, such as returning zero instead of returning the **pid**.

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 will ignore the possibility.) A new **pid** should have been assigned in the **ProcessManager.GetANewProcess** method you implemented in Lab 4.

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.

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

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 that has terminated. In our OS, the parent (P) of a 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 whichever process was previously the parent of P. In other words, the "grandparent" takes over as parent when the parent dies. Thus, in Unix/Linux, every process will always have a parent.]

Detail of implementing Handle_Sys_Fork

Implementing Handle_Sys_Fork

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

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 to do the following:

A. Initialize the user registers.
B. Initialize the user and system stacks.
C. Figure out where in the user's address space to "return" to.
D. Invoke **BecomeUserThread** to jump into the new user-level process.

Let us call the initial function to be executed by the newly created thread, **ResumeChildAfterFork**. This is a kernel function you'll need to add to **Kernel.c**. The **Handle_Sys_Fork** function will perform steps 1 through 4 above, which will include creating a new thread. The new thread will begin by executing the **ResumeChildAfterFork**, which will perform steps A through D, completing the operation of starting the new process.

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

The first thing to do in **Handle_Sys_Fork** is to obtain a new **ProcessControlBlock** and a new **Thread** object.

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

You will also need to initialize the following two fields in the new **Thread** object: **status**, and **myProcess**.

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.)

Also recall that the BLITZ CPU has two 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 do not 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 will 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 lab, you are not asked to implement the file-related syscalls (**Open**, **Read**, **Write**, **Close**, etc.), so we do not have to do anything related to open files.

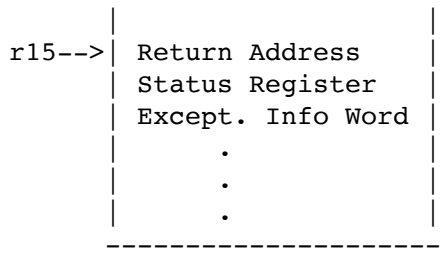
Next, you will need to make a copy of the parent's virtual address space. You will need to see how many pages are in the parent's address space and call **frameManager.GetNewFrames.** Then you will need to run through each and copy the page. You can use **MemoryCopy** to do this efficiently. You can 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**.)

At this point, you are almost ready to invoke **Thread.Fork** to start the new thread, but there is one more number you will need first. The new thread needs to know where (in the user-level address space) to resume execution and the parent's top-half must determine that value.

To approach this, ask how does execution return 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 **SyscallTrapHandler**, which will return to the assembly **SyscallTrapHandler**, which will execute the "reti" instruction.

At the very beginning of the Fork processing, a "syscall" 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 will need to get that return address from the system stack and you will 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.

```
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 exception block
store   r1,[r15+4]
ret
```

You will need to call this function; let us call the value it returns **oldUserPC**. You will need to pass this address to the **ResumeChildAfterFork** function so that the child can use it.

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 (ResumeChildAfterFork, 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 **Handle_Sys_Fork** to the parent user-level process.

Next, let us look at what you will need to do in the **ResumeChildAfterFork** function.

The key piece of info **ResumeChildAfterFork** 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 **ResumeChildAfterFork**.

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

Notice that **ResumeChildAfterFork** 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.

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

Then, you will 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 will 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 will want an empty system stack. You can compute the initial value for the system stack top just as you did in **StartUserProcess**.

The initial value for the user stack top has been stored in user register **r15**, which was stored in the **Thread** object by **Handle_Sys_Fork**.

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

```
BecomeUserThread (initUserStackTop,     -- Initial Stack
                  initUserPC,           -- Initial User PC
                  initSystemStackTop)   -- Initial System Stack
```

Recall that we need to return a pid of 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.

Next, let us discuss two subtleties in the implementation of the **Handle_Sys_Fork**.

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

```
newThrd.Fork (ResumeChildAfterFork, 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 with the support of "zombies." The child may terminate before the parent fetches the pid out of the PCB, but the 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.

On 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 **ResumeChildAfterFork**. Then there is no possibility of a context switch between the call to **RestoreUserRegs** and setting **isUserThread** to true. Also recall that **BecomeUserThread** will reenable interrupts as part of the process of resuming execution in user mode.

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. (In real life, zombies roam the earth at night terrorizing teenagers staying in remote cabins.)

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 do not 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 ("should never be 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 will get an immediate error.]

Second, add two new methods to **ProcessManager**: **TurnIntoZombie** and **WaitForZombie**. Also, you will need to implement the **ProcessFinish** function.

function ProcessFinish (exitStatus: int)

This function 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...

First, save the **exitStatus** in the PCB.

Next, disable interrupts.

Next, disconnect the PCB and the Thread, i.e., set **myProcess** and **myThread** to null. Also, set **isUserThread** to false.

Next, re-enable interrupts (in fact, it is more appropriate to restore interrupt status to the previous one).

Next, return all page frames to the free pool, by calling **frameManager.ReturnAllFrames**.

Next, invoke **TurnIntoZombie** on this PCB.

Finally, invoke **ThreadFinish**.

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.

First, lock the process manager since you will be messing with other PCBs.

Next, 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, since other threads may be waiting for free PCBs.

Next, 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.

Finally, unlock the process manager.

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**.

First, lock the process manager.

Next, wait until the status of **proc** is ZOMBIE, using a while loop and the **aProcessDied** condition variable.

Next, fetch **proc**'s **exitStatus**.

Next, change **proc**'s status to **FREE**, add it to the PCB free list, and signal the **aProcessBecameFree** condition variable.

Finally, unlock the process manager and return the exit status.

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.

Implementing Handle_Sys_Yield

There is not really a need for a **Yield** syscall in any OS that has preemptive scheduling, but it is helpful in testing, to make sure that other processes are really running.

Within **Handle_Sys_Yield**, you can simply invoke **Thread.Yield** on the current thread and return.

When executed, the scheduler will be invoked and other threads will get a chance to run. Sometime later, this thread will run again (when the call to **Yield** returns) and a return will be made to the user-level code.

The User-Level Programs

The **lab6** directory contains the following user-level programs:

MyProgram	-- For you to use during debugging
TestProgram1	-- Do not modify
TestProgram2	-- Do not modify
TestProgram3	-- Do not modify

You may modify **MyProgram** any way you wish while testing.

The remaining three programs constitute the test suite. **TestProgram1** and **TestProgram2** are from the previous lab, while **TestProgram3** contains new tests for this lab.

The **TestProgram3** main function looks like this:

```
function main ()

   --SysExitTest ()
   --BasicForkTest ()
   --YieldTest ()
   --ForkTest ()
   --JoinTest1 ()
   --JoinTest2 ()
   --JoinTest3 ()
   --JoinTest4 ()
   --ManyProcessesTest1 ()
   --ManyProcessesTest2 ()
   --ManyProcessesTest3 ()
   --ErrorTest ()
   Sys_Exit (0)

  endFunction
```

An individual test can be run by uncommenting the appropriate line, compiling the code, and running it. For example, to run the "basic fork" test, uncomment the line that calls **BasicForkTest**, compile, and run. To run another test, go back and re-comment the **BasicForkTest** and uncomment some other line. Note that to run the **BasicForkTest**, the **Exit** system call must have already been implemented.

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

What to Submit

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

```
submitece353s 6 Kernel.c Kernel.h
```

Please do not modify any files except **Kernel.c** and **Kernel.h**.

Grading for this lab

When grading, the TA will ask you to run your solutions in a live demonstration session, in which you will run the 12 test methods in **TestProgram3.c**, each worth 1 mark, for a total of 12 marks. Optionally, the TA may also ask you a question to see if you have actually read this handout, followed the steps outlined in this lab, and understood your own solutions. 4 marks will be deducted if your answer is not satisfactory.

Sample Output

If your program works correctly, you should see something like this:

```
==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

SysExitTest running.

About to terminate the only process; should cause the OS to stop on a 'wait' instruction.

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****




==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

BasicForkTest running.
```

```
I am the parent
I am the child

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****




==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

YieldTest running.

This test involves calls to Fork, Yield, and Exit.

RUN ONE: You should see 10 'compiler' messages and 10 'OS' messages.

Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Designing compilers is fun!
Designing compilers is fun!
Designing compilers is fun!
Designing compilers is fun!
Designing compilers is fun!
Designing compilers is fun!
Designing compilers is fun!
Designing compilers is fun!
Designing compilers is fun!
Designing compilers is fun!

RUN TWO: You should see the same 20 messages, but the order should be different, due to the presence of
'Yield's.

Designing compilers is fun!
Designing compilers is fun!
Writing OS kernel code is a blast!
Designing compilers is fun!
Designing compilers is fun!
Writing OS kernel code is a blast!
Designing compilers is fun!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Designing compilers is fun!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
```

```
Designing compilers is fun!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Writing OS kernel code is a blast!
Designing compilers is fun!
Designing compilers is fun!
Designing compilers is fun!

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****




==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

ForkTest running.

This test involves calls to Fork, Yield, and Exit.
There should be 26 columns (A-Z) printed.  Each letter should be printed 5 times.
A
A
 B
 B
A
  C
  C
 B
   D
A
  C
   D
 B
   D
  C
A
 B
   D
    E
  C
    E
    E
   D
     F
     F
```

...A BUNCH OF STUFF, DELETED IN THIS DOCUMENT...

```
                    W
                    W
                  U
```

```
                    V
                     W
                      X
                      X
                    V
                     W
                      X
                       Y
                       Y
                     W
                      X
                       Y
                        Z
                        Z
                      X
                       Y
                        Z
                       Y
                        Z
                        Z

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****




==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

JoinTest 1 running.

This test involves calls to Fork, Yield, and Exit.
Running first test...
This line should print first.
This line should print second.
Done.
Running second test...
This line should print first.
This line should print second.
Done.

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****




==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
```

```
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

JoinTest 2 running.

This test involves calls to Fork, Yield, and Exit.
Creating 5 children...
Child 1 running...
Child 2 running...
Child 3 running...
Child 4 running...
Waiting for children in order 1, 2, 3, 4, 5...
Child 5 running...
Creating 5 more children...
Child 1 running...
Child 2 running...
Child 3 running...
Child 4 running...
Waiting for children in order 5, 4, 1, 3, 2...
Child 5 running...
Done.

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****




==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

JoinTest3 running.

This test involves 5 illegal calls to Sys_Join, waiting on non-existent children.
In each case, it prints the return code, which should be -1.
Return code from 1st call = -1
Return code from 2nd call = -1
Return code from 3rd call = -1
Return code from 4th call = -1
Return code from 5th call = -1
Done.

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****




==================  KPL PROGRAM STARTING  ==================
```

```
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

JoinTest4 running.

   This test forks a child process and then waits on it twice.
   The first call to Sys_Join should return its error code; the
   second call to Sys_Join should return -1.

The PID of the child = 2
This should print first.
This should print second.
Okay (1).
Okay (2).
This should print first.
The PID of the child = 3
This should print second.
Okay (3).
Okay (4).

   In the next test, we create 2 children, and each creates 2 children,
   giving 7 processes in all.  Then each process attempts a Sys_Join on
   every process except its own children, to make sure the result is -1.
   Finally, each process with children waits on them.

A is running...
  My first  child is A.B   pid1 = 4
  My second child is A.C   pid2 = 5
---------------
A.B.D is running...
---------------
A.C.F is running...
---------------
A.C is running...
  My first  child is A.C.F   pid1 = 7
  My second child is A.C.G   pid2 = 9
---------------
A.C.G is running...
---------------
A.B is running...
  My first  child is A.B.D   pid1 = 6
  My second child is A.B.E   pid2 = 8
---------------
A.B.E is running...
---------------
A done with error tests...
A.C.F done with error tests...
A.C done with error tests...
A.C.G done with error tests...
A.B.E done with error tests...
A.B done with error tests...
A.B.D done with error tests...
-----------------------------------A is waiting on A.B     pid1 = 4
-----------------------------------A.C is waiting on A.C.F     pid1 = 7
-----------------------------------A.B is waiting on A.B.D     pid1 = 6
A.C.F is done.
```

```
----------------------------------A.C is waiting on A.C.G     pid2 = 9
A.C.G is done.
A.C is done.
A.B.E is done.
A.B.D is done.
----------------------------------A.B is waiting on A.B.E     pid2 = 8
A.B is done.
----------------------------------A is waiting on A.C     pid2 = 5
A is done.

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****




==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

ManyProcessesTest1 running.

This test should create 100 child processes.
It should print 100 lines of output.
Child 1
Child 2
Child 3
Child 4
Child 5
```

...A BUNCH OF STUFF, DELETED IN THIS DOCUMENT...

```
Child 92
Child 93
Child 94
Child 95
Child 96
Child 97
Child 98
Child 99
Child 100
Done.

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****




==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
```

```
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

ManyProcessesTest2 running.

This test attempts to create 9 new processes.
It should print a line for each process and then it should print 123.
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
Process 8
Process 9
Final return value = 123
Done.

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****




==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

ManyProcessesTest3 running.

This test attempts to create 10 new processes.
It should run out of resources and hang.
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
Process 8
Process 9

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****
```

```
==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...
AllocateRandomFrames called.  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
Initializing Disk Driver...
Initializing Serial Driver...
Initializing File Manager...
Serial handler thread running...
Loading initial program...

ErrorTest running.

Should print "User Program Error: Attempt to use a null pointer!"...

User Program Error: Attempt to use a null pointer!  Type 'st' to see stack.

Okay.

Should print "An AddressException exception has occurred while in user mode"...

**********  An AddressException exception has occurred while in user mode  **********

  ProcessControlBlock
```

...A BUNCH OF STUFF, DELETED IN THIS DOCUMENT...

```
  Thread "UserProgram"
```

...A BUNCH OF STUFF, DELETED IN THIS DOCUMENT...

```
Okay.

Should print "A PageReadonlyException exception has occurred while in user mode"...

**********  A PageReadonlyException exception has occurred while in user mode  **********

  ProcessControlBlock
```

...A BUNCH OF STUFF, DELETED IN THIS DOCUMENT...

```
  Thread "UserProgram"
```

...A BUNCH OF STUFF, DELETED IN THIS DOCUMENT...

```
Okay.

Done.

*****  A 'wait' instruction was executed and no more interrupts are scheduled... halting emulation
*****
```