PROJECT 1:
FORK & EXEC

Minimum Requirements: 5 public tests

TEST DISTRIBUTION

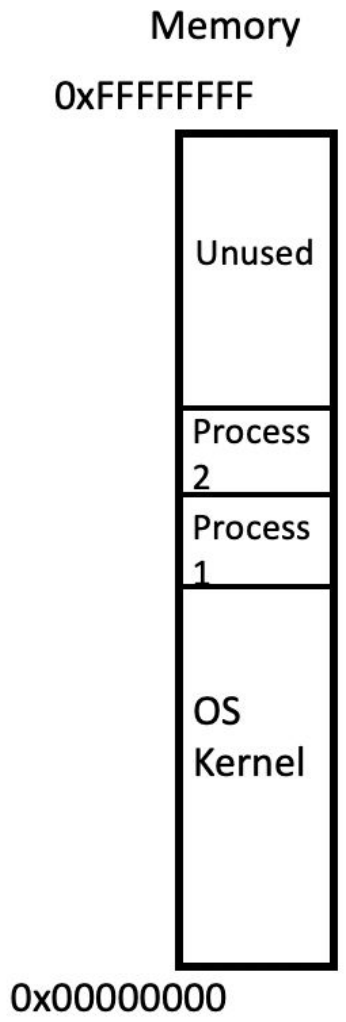
Public tests – 6 tests | 27 points

Release tests – 3 tests | 15 points

Secret tests – 7 tests | 35 points

PROCESS AND THREAD

- Process
  - The instance of a computer program that is being executed. It contains not only the executable itself but also the resources assigned to that process.
- Thread
  - A smallest unit that can be scheduled onto CPU, it shares with other threads belonging to the same process its memory and other OS resources.
- What's the difference?
  - The most important difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.



PROCESS CONTROL BLOCK

- Process Control Block (PCB)
  - A data structure in the operating system kernel containing the information needed to manage the processes.
- Why is PCB needed
  - Scheduling, context switch ...

PCB IN GEEKOS

- struct UserContext (user.h)
  - Memory related data, file_descriptor_table …
- struct KernelThread(kthread.h)
  - Kernel stack, scheduling information, parent information ...
  - A user process has its corresponding kernel thread.

REGISTERS, KERNEL STACK AND INTERRUPT STATE

- When a user process calls a system call, its registers are pushed into its kernel stack and enters kernel mode.
  - Record the context: when it returns to user mode, the values are popped out to restore the values of registers.
  - Passing arguments: Arguments for syscalls are written into registers first and then they will be written into kernel stack. Then syscall functions will be able to get this arguments by looking at its kernel stack.
- Interrupt_State is a pointer pointing to the top of the kernel stack right after we enter the syscalls. This is why Interrupt_State is passed in as an argument for all system calls, so that the syscalls will be able to get the arguments.

USEFUL FUNCTIONS

Go through the definitions of the following functions.
Make sure the function is visible to other files if you need them.

- struct User_Context *Create_User_Context(ulong_t size) //userseg.c
  - Create a new user context of given size. Ignore the LDT part, it will be mentioned in the following projects.
- struct Kernel_Thread *Create_Thread(int priority, bool detached) //kthread.c
  - Create an instance of kthread. Pay attention to how stackPage(kernel stack) is allocated. Look at Init_Thread in the function to see how stack pointer (esp) is initialized.
- void Attach_User_Context(struct Kernel_Thread *kthread, struct User_Context *context) //user.c
  - Associate the given user context with a kernel thread.
- void Detach_User_Context(struct Kernel_Thread *kthread) //user.c
  - Get rid of the old user context and destroy the UserContext.
- void Destroy_User_Context(struct User_Context *userContext) //userseg.c
  - Destroy an instance of User_Context

USEFUL FUNCTIONS

- void Setup_User_Thread(struct Kernel_Thread *kthread, struct User_Context *userContext) //kthread.c
  - Attach the userContext to the kthread. Push initial values to the kernel stack so that when they are popped out to fill in the values of the registers, the values of the registers will be initialized properly to start a new user process (e.g. instruction pointer).
- void Make_Runnable_Atomic(struct Kernel_Thread *kthread) //sched.c
  - Add the kthread to the running queue using an atomic operation so that it can be scheduled to run.
- Copy_user_string and get_path_from_registers //syscall.c
  - Can be used in Sys_Execl. Copy the path of the executable from the address given by the registers.

CREATE A NEW PROCESS: FORK()

- After a Fork() call, the caller(the parent) creates a new process(the child) that is identical to itself.
- Fork returns different values to the parent and the child. It returns the child's PID to the parent, and 0 to the child.
- If the creation of a child process was unsuccessful, Fork() returns a negative integer (error code).
- After Fork() returns, both the parent and the child will continue executing the next instruction that follows Fork().

START A NEW PROGRAM: EXEC()

- A process replaces the program it was running by calling Exec().
- Load the program into the current process space and runs it from the entry point.
- Exec does not return on success, returns error code if any unexpected error occurs.

FUNCTIONS TO IMPLEMENT/MODIFY

- In syscall.c
  - Sys_Fork
  - Sys_Execl
  - Sys_Exit
- In vfs.c
  - Close
- Make modifications to struct File (in vfs.h) to support multiple processes.
- Probably modify previous pipe implementation (most likely you don't need to modify anything).

SYS_FORK()

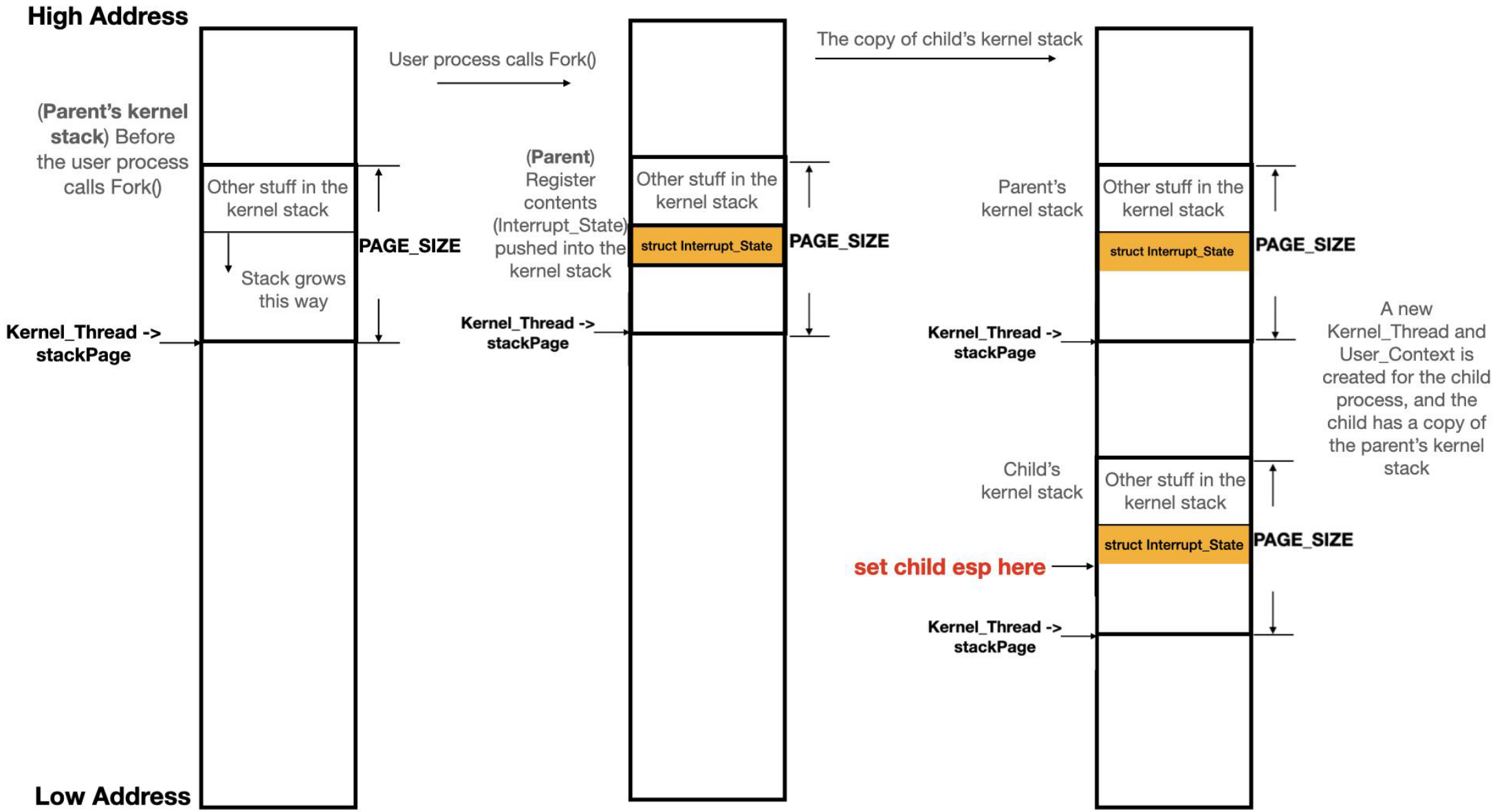
- Create a new user context (User_Context) and a new thread (Kernel_thread) for the child process.
- Copy the address space from parent to child. To get the current thread, use the macro CURRENT_THREAD. Fork() creates an exact duplicate process (with some differences such as the new process having its own a0ddress space), so transfer data from parent process to child process appropriately (necessary components in struct User_Context, but leave ldtselector, csselector, dsselector, entryAddr, argBlockAddr, stackpointerAddr, etc. alone).
- Copy the FDT. Files can be referenced by multiple processes, so keep track of reference counts for each file.
  - Think where the reference count should be initialized and incremented.
  - Might want to use mutexes to make increments atomic.
- Copy the kernel stack from parent to child and fix the position of Kernel_Thread::esp.
- Fork() returns the child's pid for the parent by directly returning the value. Returns 0 for the child, this return value needs to be stored in the EAX register of the child process.
- Add the created thread to the ready queue so the scheduler can schedule. (Refer to Start_User_Thread in kthread.c to see how it is done, but you should not use it because it also does something else rather than just adding the thread to the queue.)
- Check for appropriate error conditions throughout (mostly ENOMEM).

SYS_FORK()

- X86 uses EAX register to return values, the EAX register of the child process needs to be changed to the returning value of the child process
- When a process enters kernel mode, all registers are pushed into the kernel stack of its corresponding kernel thread. When it returns to user mode, the registers are recovered from the stack.
- To return the appropriate value for child process. The stack of the parent process needs to be copied to the child process and the corresponding position storing EAX should be modified.
- Refer to Setup_Kernel_Thread to see how the stack looks like when a newly kernel thread is set up.

CHILD ESP

- There are two ways to set the child esp.
  1. Follow the setup_user_thread, but instead of pushing dummy/initial values, retrieve the values from parent and put them on the child's stack page.
  2. The lazy way: copy over the entire parent kernel stack and calculate where the child esp should be and set it correctly. Refer to the next page.


High Address
(Parent's kernel stack) Before the user process calls Fork()
Other stuff in the kernel stack
PAGE_SIZE
Stack grows this way
Kernel_Thread -> stackPage
Low Address
User process calls Fork()
(Parent) Register contents (Interrupt_State) pushed into the kernel stack
Other stuff in the kernel stack
struct Interrupt_State
PAGE_SIZE
Kernel_Thread -> stackPage
The copy of child's kernel stack
Parent's kernel stack
Other stuff in the kernel stack
struct Interrupt_State
PAGE_SIZE
Kernel_Thread -> stackPage
A new Kernel_Thread and User_Context is created for the child process, and the child has a copy of the parent's kernel stack
Child's kernel stack
Other stuff in the kernel stack
struct Interrupt_State
PAGE_SIZE
set child esp here
Kernel_Thread -> stackPage

RETURN VALUE OF FORK

- As mentioned earlier, Fork returns 0 to child and child's PID to parent.
- X86 uses eax to hold the return values.
- Two ways to find eax:
  - Use esp and relative position
  - Typecasting esp, then use ->eax

SYS_EXECL()

- Copy the executable path and the command string that user provided to kernel space (refer to other syscalls to see how to do this). Read the comments of Sys_Execl() to see what registers are used.
- Refer to Spawn() system call that GeekOS currently uses on how to create a new process.
- Get rid of old user context and follow Spawn() (in user.c) code to see what needs to be done.
- Look at appropriate man pages to read about components that are preserved across a call to exec (FDT).
- Set esp to the initial position (think about what the initial position should be).
- Use Setup_User_Thread to attach the user context, init the stack. (Think about why we don't need to put it in the running queue)
- Check for appropriate error conditions throughout.

SYS_EXIT()

- Make sure your code works and make a submission before start working on this function.
- You can either implement the following in Sys_Exit or Exit (kthread.c).
- Iterate through available threads and check if owner is the current thread. If so, take care of it. Follow through the logic of reaping a thread and appropriately determine conditions to mop children off (hint: use Detach_Thread).
- Refer to Lookup_Thread(kthread.c) to see how to iterate through all the threads.
- Note that GeekOS does not allow lock re-entrant.

CLOSE() IN VFS.C

- Decrease the reference count
- When the reference count is 0, close the file.

LOCKS

- Mutex s_vfsLock in vfs.c
- Spin_Lock_t kthreadLock in syscall.c