QEMU Threading

Ethan Miller and Curtiss Davis

CSSE332-04 Group A

Contents

Milestone 1

Theory

Page Table Contents

In xv6, each process that is created gets its own address space. Thus, each process will get its own page table to be able to navigate the address space. The starting virtual address that the process can access is 0 and can get up to the value of MAXVA. As seen in the diagram below (taken from the xv6 book), the address space contains all the necessary pages to be able run a process: including spaces for the process code, the stack, and the heap.

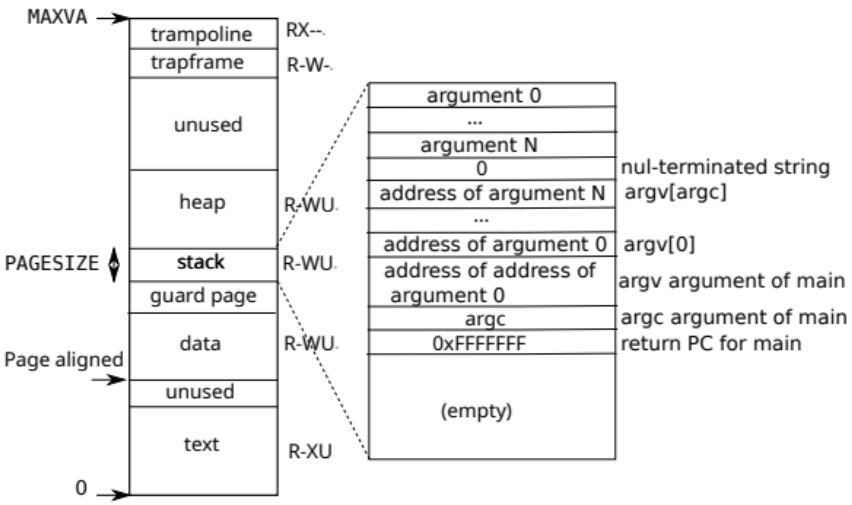


Figure 3.4: A process's user address space, with its initial stack.

Each page in the diagram above is wrapped with its own permissions, which are shown as characters to the right of the page. There are four possible permissions that a page can have within a process page table: read (R), write (W), execute (X), and user (U). These permissions are set according to the needs of specific pages. For example, the text page will not have the write permissions, to prevent a process from being able to edit its own code. In addition, the data page will not have the execute permission to prevent data objects from being treated as code and being executed.

The stack is a single page that can be accessed through the page table. The arguments for the process along with their addresses are stored at the top of the stack. Just under, are the arguments (argv, argc) that can be used to run the main function of the process. And just below this is the return value for the PC once the process is complete. The guard page, that is positioned below the stack in the address space, will be hit if the process tries to allocate more space in the stack than is available. Attempting to write to the guard page will then cause a hardware exception to be thrown and will prevent any stack overflow.

Trapframe and Trampoline

When a new page table is created, the top of the address space will contain a page for trampoline and another page for the trapframe. The process will use these two pages to transition between the kernel process and the new process. Each one then serves a different purpose to make the transition possible and smooth.

The trampoline page will always be at the top of the address space, at address TRAMPOLINE. As a result, it will be above the program memory and will not easily be affected by the process. In addition, since it will be at the same address in the kernel table, when the switch is made from the process to the kernel, the trap handler will be able to continue executing its code. When the trampoline page is called and run, it will run a function (uservec) which will handle the process of saving all 32 register values into the trapframe, so that they will be able to be restored when the process continues running.

The trapframe page is located just below the trampoline at address TRAPFRAME. It contains the address of the process's kernel stack, the address of the function usertrap, and the address of the kernel's page table. It also contains the register values that were backed up by the uservec function in the trampoline. The usertrap function is used to catch and determine the cause of a trap and then determine what the next step should be, such as killing a faulty process. The trapframe can also be accessed through the process, to its physical address so that the kernel can use the trapframe through its page table.

Context Switching

The process of context switching in the xv6 involves switching from one user process to another. There are a couple examples, but in general they are: a system call or interrupt to an old thread, a switch to a new kernel's thread, and a trap return to the user process. This dedicated thread that is passed around is saved per CPU for security. Imagine using the same stack on different cores, then you would have similar issues to sharing page tables.

When switching from one thread to another, the kernel thread has a function called *swtch*. This function simply saves and stores the registers of the process and calls them *contexts*. At the time that a context switch is happening, the process will call *swtch* which will store the current contexts into a struct labeled old and loads the new processes registers into the struct called new. *Swtch* also saves the return address for the process that had the call from.

When *swtch* returns, it uses that *ra, return address,* register to return to the instructions at that point, not the exact process it was called in before. It retains the new thread's stack, however.

Sharing Page Tables

Each process has its own separate page table. The common data between the page tables will be the same based on the physical address accessed and if the created thread is a parent or

child. This is here for a lot of reasons, but a main one would be protection again crazy functionality between two different processes. Let's say a process goes to access and change some data on its stack. If another process was sharing that page table, then that first process could change the functionality of a function existing in that page table. When process two goes to call that function, the function now does something completely different. By splitting up the Page Tables for every process, we add a level of protection to the user space. It does cause a lot of waste in the design since not all page tables will be used fully, however the tradeoff for protected execution is far too important to try sharing page tables between processes.

Usage of uvmcopy

The use of uvmcopy is what the current xv6 designers decided for. This means, they chose to make a copy of the current page table in one point of a physical address and then copy it over to another for the child processes. This is not a bad design since it ensures access protection between the parent and child when they both go to write to their table. However, it also adds a lot more wasted memory, since it creates a whole page table with the exact data as the parent. There are ways about it where using complicated PTE key switching, we can allow the parent and child to use the same physical address. The issue with this design is that it is fundamentally flawed since having this shared memory and locking it with flags will cause not multiple accesses to the memory, which we want the user to be able to do. Therefore, we will be developing a uvmcopy that will take the data in the parent process and passing it to a new physical address location for the child to access whenever they want.

The Design

How to Create a New Thread

A pthread being created will either need to create and access a point at physical memory (using kalloc) and establish a pagetable_t for digital memory or find an address that was passed in and create a pagetable_t there. The main function that will be needed for creating a pthread is mappapges. This will pull the data from the physical address associated with the given address or will process the needed data into a newly created physical address spot. This function installs PTEs for new mappings to the physical addresses created. Copyto and copyfrom are important in these thread creations since they are how we will copy data to the threads. Using the pagetable_t object that we will be creating on the initialization of the memory used, we will set a PTE_V flag which specifies if a PTE is present in the physical memory that was accessed/created there. To track the execution order of the process, we will have an array of linked lists in the scheduler which will connect the needed child threads to the parent thread in a linked list and create a new array object for the single new thread that does not have a connection to children yet.

How to Join Threads

The basic use of joining a thread is to wait for all other active threads to finish their operations. This will be done by calling a join function from the parent thread that created the new child thread. When the parent thread calls the *join* function, it will be blocked until the child process has terminated. This will be accomplished by going through the linked list of the parent process and while looping checking for the child's status to switch to zombie. Once this is completed, it will signal to the parent that the join has finished, and it can continue execution.

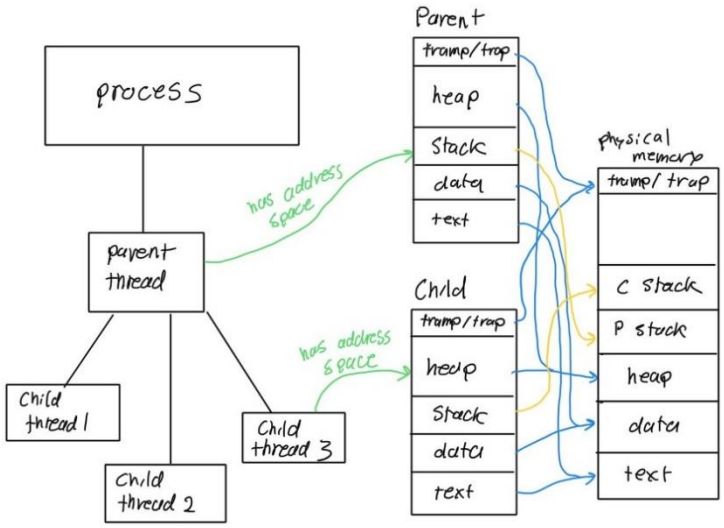
In a situation where multiple threads are created and need to be joined, the parent thread will first run through a loop to create the child threads. After all child threads are created it will then enter another loop with join functions. When the first join function is hit, the parent will be blocked from executing until the first child thread has terminated. When the first child has terminated, the join will unblock the parent thread and return the status of the terminated thread. The loop will then iterate to another join function that will wait for the second child thread to terminate. This loop continues until the loop exits and the parent is free of all join functions.

Thread Relationships

Threads relationship can be tracked from the context data. There is a *ra* register for each context that tracks the address of the original call. This context also holds data for old contexts and new contexts which will help the threads to have separate data properly. This execution order will help each child and parent processes to execute in order if we wish them to. Since we still want other processes to process in parallel, using contexts is secure since each process executes from its own CPU register, which is only shared on parent and child processes and not separate ones. Sched is an important function since it will help the scheduler to switch the process correctly.

Thread Stacks

To allow different threads to have access to different stacks, the page table that each thread sees will be different. When a thread is created, it will create a new page table and allocate more space in physical memory for a new stack. The new page table will then map the locations of the trampoline, trapframe, heap, data, and text to the locations of those from the parent thread. The stack of the child thread will then be mapped to the new space that was allocated. A new guard page will also need to be created and mapped to the page table. The overall result is a child thread that will be able to access all the same parts of the process as the parent thread, except it will have access to a completely different stack and respective guard page.



Thread Arguments

When a process is created, the top of the new stack contains the process arguments. When a new child thread is created, as stated above it will get a newly allocated stack. This new stack will have all the thread arguments placed at the top of the stack. This allows the arguments to be accessed by the thread at an easily accessible location. It will also contain several arguments for usability. Unlike the parent, however, it will not have the return PC value, as the child thread should not be responsible for returning to the main function or higher caller.

Termination

When a child thread has completed all the functions that it desired to complete, it will return, and the parent thread must be able to detect when this happens. To do so, when the child is finished running it will call the *exit* function. This function will first handle the closing of all open files, then it will place the processes into a Zombie state. It will also release the *lock* on the process, a process which will signal to the parent that the child has finished. Or in the chance of it being the main process, it will finish its execution normally. It uses the sched function after all that, just like we talk about in Thread Relationships, because it will load the next process to run/finish the old process in the context/a new CPU register.

The Implementation

Kernel Side

The first area where we applied changes was in the kernel area. Specifically, we targeted the proc.c file. We both created new functions and edited existing ones. The table below shows the new functions that were created, and the ones that were edited, and all their details.

Function Name	Arguments	Returns	Description
procclone	func pointer, void pointer arg, void pointer stack	32-bit integer	Function responsible for allocating a new process for the child thread and then mapping the pagetable and setting the registers and flags.
join	Int tid	32-bit integer	Function responsible for awaiting the input thread id and then telling its parent its finished.

System Calls

To be able to run the functions from the kernel, we had to implement system calls from the defs.h, syscall.c, syscall.h, sysproc.c, user.h, and usys.pl files. With these files, we had to tell the user side what function it could call and then connect that will the kernel functions that we want to pass to the user. Thus, allowing the separation between the kernel and user. This is also how we designated the arguments for each of the APIs we designed, and which addresses they correlated to.

User Side

Created Functions

Function Name	Arguments	Returns	Description
thread_create	thread_t addr, func pointer, void pointer	32-bit integer	Function responsible for adding the thread to the process array of threads and marking the thread as runnable. Doing so, the scheduler can now run the thread. It also contains a pointer to the argument, which is added to the stack of the thread.
thread_join	thread_t	32-bit integer	Function responsible for freezing the mother thread, and making it wait until the given child thread has completed its actions.
thread_kill	thread_t	32-bit integer	Function responsible for stopping a thread regardless of completion status.

New Data Structures

Threads must have certain information stored to be used effectively by the process. Thus, the data structure “thread_t” is created in the threads.c file. The table below shows the different values that will be stored in this new data structure.

Variable Name	Type	Description
ID	32-bit integer	A unique ID for the thread within the process. Threads from different processes may have the same ID. Threads with ID=0 will always be the mother threads.
Busy	32-bit Integer	A Boolean identifier which tracks whether the thread is in use or not.
Stack Pointer	void addr	A pointer to the stack of the pagetable for the specific thread_t object.

Testing process

A create a multi-step test plan was created to be sure that the threads that we created, namely their creation and joining them. If all tests pass, then the thread API is working as expected.

All testing is stored in the threadtesting.c file within the user space. To call the file and run the tests, type “threadtesting” into the console once qemu has booted. The table below shows the process of the test along with descriptions of the test.

Test Num	Variables	Testing Aspects	Description
1		Create	Tests if a thread can be created. Runs the thread_create function with a pointer to a function that the child should run. If the result of thread_create is below zero, then the test fails. Otherwise, the thread was successfully created. This will also test to ensure that the main thread is still running even after the creation of the child thread.
2	x	Join	This tests if the join functionality works: the main function waits until the child function has finished its process. Using the thread that was created in test 1, runs the join command. If the test passes, then the returned value from the join function was zero. In addition, the function the thread runs should have altered a global variable. This test checks that global variable to make sure that it changed; showing that all the threads have a shared heap.
3	array	Create/Join	This test puts create and join through a stress test to check to see if multiple threads can be created and if they all can be joined back together. Using a for loop, six threads are created, each one is given an integer as their id. The function that they will run will put their id at the index of the id’s value in the global array. After creating, it will loop through again and join all the created threads. A final loop will ensure that all the values in the array are the correct values.