OPERATING SYSTEMS – ASSIGNMENT 2

KERNEL LEVEL THREADS, SYNCHRONIZATION AND MEMORY MANAGEMENT

In this assignment we will extend xv6 to support kernel level threads, synchronization primitives and *Copy On Write (COW)* optimization for the fork system call.

You can download xv6 sources for the current work by executing the following command:

```
git clone https://github.com/mit-pdos/xv6-public
```

Task 1: Kernel level threads (KLT)

Kernel level threads (KLT) are implemented and managed by the kernel. Our implementation of KLT will be based on a new struct you must define called "thread". One key characteristic of threads is that all threads of the same process share the same virtual memory space. You will need to decide (for each field in the *proc struct*) if it will stay in *proc* or will move to the new *struct thread*. Be prepared to explain your choices in the frontal check. Thread creation is very similar to process creation in the fork system call but requires less initialization, for example, there is no need to create a new memory space.

1. Add a file named kthread.h which contains some constants, as well as the prototypes of the KLT package API functions.
2. Understand what each of the fields in *proc struct* means to the process.

To implement kernel threads as stated above you will have to initially transfer the entire system to work with the new thread structure. After doing so every process will have 1 initial thread running in it. Once this part is done you will need to create the system calls that will allow the user to create and manage new threads.

It is very important to read the entire task before starting. Don't be lazy here, the work will be much easier if you read and understand the entire work in advance.

1.1. Moving to threads:

This task includes passing over a lot of kernel code and understanding any access to the *proc* field. Once this is done, you should add a support for a new global variable thread. Looking at *proc.h* you can see 2 such global variables: a pointer to the *cpu struct* that contains CPU information and another pointer to a *proc struct* which is how you access the current running process. You must extend current implementation so that you will be able to access the *thread struct* of the currently running thread in addition to *proc struct*.

In order to maintain current thread you need to perform following steps:

1. Add a pointer to current thread (of type *struct thread*) right after the pointer to proc struct (see struct cpu at proc.h).
2. Update function seginit (see vm.c) so that the line 31:

```
c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
```

   will be replaced with following line:

```
c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 12, 0);
```

3. Add a global pointer to current thread by inserting following line to proc.h:

```
extern struct thread *thread asm("%gs:8");
```

Note that additional changes may be required (e.g., the scheduler, for example, must update thread in addition to proc).

In your implementation each process should contain its own static array of threads. Add a constant to kthread.h called ***NTHREAD = 16***. This will be the maximum number of threads each process can hold.

As you have seen in assignment 1, XV6 implements scheduling policy that goes over the list of processes and chooses the next ***RUNNABLE*** process in the array. You are expected to change the scheduling so it will run over all the available threads of a specific process before proceeding to the next process. This is similar to the original scheduling policy, just over threads.

Supporting multiple threads per process means that you will have to think about handling shared resources information. For example: growproc is a function in proc.c which is responsible for retrieving more memory when the process asks for it. If not protected, 2 running threads of the same process calling this function may override the sz parameter and cause issues in the system. Consider synchronizing all shared fields of the thread owner when they are accessed. Be prepared to explain why you did or didn't synchronize those accesses.

Expected behavior of existing system calls:

1. **Fork** – should duplicate only the calling thread, if other threads exist in the process they will not exist in the new process
2. **Exec** – should start running on a single thread of the new process. Note that in a multi-threaded environment a thread might be running while another thread of the same process is attempting to perform exec. The thread performing exec should “tell” other threads of the same process to destroy themselves and only then complete the exec task. Hint: You can review the code that is performed when the proc->killed field is set and write your implementation similarly.
3. **Exit** – should kill the process and all of its threads, remember while a single threads executing exit, others threads of the same process might still be running.

1.2. Thread system calls:

In this part of the assignment you will add system calls that will allow applications to create kernel threads. Implement the following new system calls:

int kthread_create(void*(*start_func)(), void* stack, int stack_size);

Calling *kthread_create* will create a new thread within the context of the calling process. The newly created thread state will be ***RUNNABLE***. The caller of *kthread_create* must allocate a user stack for the new thread to use (it should be enough to allocate a single page i.e., 4K for the thread stack). This does not replace the kernel stack for the thread. start_func is a pointer to the entry function, which the thread will start executing. Upon success, the identifier of the newly created thread is returned. In case of an error, a non-positive value is returned.

- The kernel thread creation system call on real Linux does not receive a user stack pointer. In Linux the kernel allocates the memory for the new thread stack. You will need to create the stack in user mode and send its pointer to the system call in order to be consistent with current memory allocator of xv6.

int kthread_id();

Upon success, this function returns the caller thread's id. In case of error, a non-positive error identifier is returned. Remember, thread id and process id are not identical.

void kthread_exit();

This function terminates the execution of the calling thread. If called by a thread (even the main thread) while other threads exist within the same process, it shouldn't terminate the whole process. If it is the last running thread, process should terminate. Each thread must explicitly call kthread_exit() in order to terminate normally.

int kthread_join(int thread_id);

This function suspends the execution of the calling thread until the target thread (of the same process), indicated by the argument thread_id, terminates. If the thread has already exited (or not exists), execution should not be suspended. If successful, the function returns zero. Otherwise, -1 should be returned to indicate an error.

It is recommended that you write a user program to test all the system calls stated above.

Task 2: Synchronization primitives

In this task we will implement a synchronization primitive - **mutex**. The **mutex** will be implemented in the kernel. Mutex is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. A mutex is designed to enforce a mutual exclusion concurrency control policy.

Before implementing this task you should do the following:

1. Read about mutexes in POSIX. Our implementation will emulate the behavior and API of pthread_mutex. A nice tutorial is provided by Lawrence Livermore National Laboratory.
2. Examine the implementation of spinlocks in xv6's kernel (for example, the scheduler uses a spinlock). Spinlocks are used in xv6 in order to synchronize kernel code. Your task is to implement the required synchronization primitives as a kernel service to applications, via system calls. Locking and releasing can be based on the implementation of spinlocks to synchronize access to the data structures which you will create to support mutexes and condition variables.
3. You implementation of mutex is not required to fulfill the starvation freedom requirement.

2.1. Mutex:

Quoting from the pthreads tutorial by Lawrence Livermore National Laboratory:

> "Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur. A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex, as used in pthreads, is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful, and all other threads will get blocked until the mutex becomes available. Mutex differs from a binary semaphore in its unlock semantics - only the thread which locked the mutex is allowed to unlock it, unlike binary semaphores on which any thread can perform up."

Examine pthreads' implementation, and define the type **kthread_mutex_t** inside the xv6 kernel to represent a mutex object. You can use a global static array to hold all the mutex objects in the system. The size of the array should be defined by the macro **MAX_MUTEXES=64**.

The API for mutex functions is as follows:

int kthread_mutex_alloc();

Allocates a mutex object and initializes it; the initial state should be unlocked. The function should return the ID of the initialized mutex, or -1 upon failure.

int kthread_mutex_dealloc(int mutex_id);

De-allocates a mutex object which is no longer needed. The function should return 0 upon success and -1 upon failure (for example, if the given mutex is currently locked).

int kthread_mutex_lock(int mutex_id);

This function is used by a thread to lock the mutex specified by the argument mutex_id. If the mutex is already locked by another thread, this call will block the calling thread (change the thread state to ***BLOCKED***) until the mutex is unlocked.

int kthread_mutex_unlock(int mutex_id);

This function unlocks the mutex specified by the argument mutex_id if called by the owning thread, and if there are any blocked threads, one of the threads will acquire the mutex. An error will be returned if the mutex was already unlocked. The mutex may be owned by one thread and unlocked by another!

2.2. A Synchronization problem:

In this task you will implement a solution to the Firing Squad Synchronization Problem (FSSP) using the synchronization primitives implemented before.

The FSSP problem description (as described by R.M Balzer):

Consider a finite one dimensional array of finite state machines, all of which are alike except the ones at each end. The machines are called soldiers and one of the end machines is called a general. The machines are synchronous, and the state of each machine at time t + 1 depends on the states of itself and of its two neighbors at time t. The problem is to specify the states and transitions of the soldiers in such a way that the general can cause them to go into one particular terminal state (i.e., they fire their guns) all at exactly the same time. At the beginning (i.e., t = 0) all the soldiers are assumed to be in a single state, the quiescent state. When the general undergoes the transition into the state labeled "fire when ready", it does not take any initiative afterwards and the rest is up to the soldiers. The signal can propagate down the line no faster than one soldier per unit of time and their problem is how to get all coordinated and in rhythm. The tricky part of the problem is that the same kind of soldier with a fixed number, k, of states is required to be able to do this regardless of the length, n, of the firing squad. In particular, the soldier with k states should work correctly, even when n is much larger than k. Two of the soldiers, the general and the soldier farthest from the general, are allowed to be slightly different from the other soldiers in being able to act without having soldiers on both sides of them. A convenient way of indicating a solution of this problem is to use a piece of graph paper with the horizontal coordinate representing the spatial position and the vertical coordinate representing the time. Within the (i,j) square of the graph paper a symbol may be written indicating the state of the i'th soldier at time j. Visual examination of the pattern of the propagation of these symbols can indicate what kind of signaling must take place between the soldiers. The figure below shows one solution to the FSSP using 15 states and 3n units of time. Each state is represented by a different color. Time increases from top to bottom.

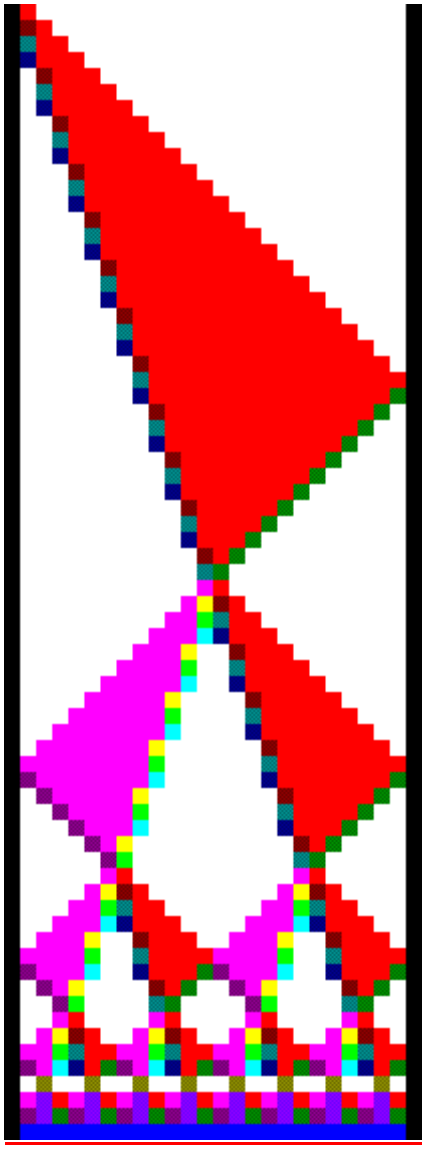
Write a user space program that:

- Receives the number of soldiers by an argument.
- Each soldier is represented by a thread.
- Each state is represented by a number.
- The process's main thread is in charge of synchronizing the threads steps (In the problem description the system of machines is synchronous).

At each time, t, several actions need to be done in a synchronized matter:

- All soldiers change their state according to their transitions function.
- All soldiers write the number corresponding to their new state in a shared array.
- The main thread prints the array (at every iteration).

You have complete freedom to choose the algorithm you choose to implement in order to solve the FSSP. Consider using the algorithm proposed by John McCarthy and Marvin Minsky.

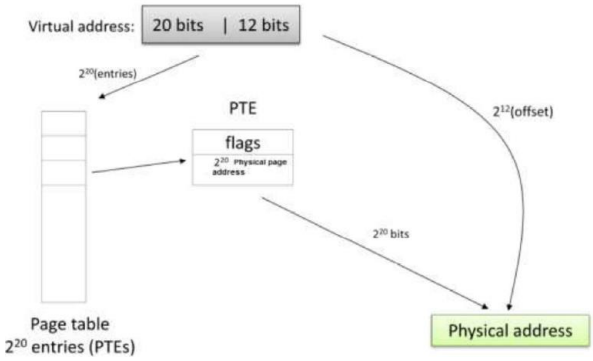


Task 3: Memory management

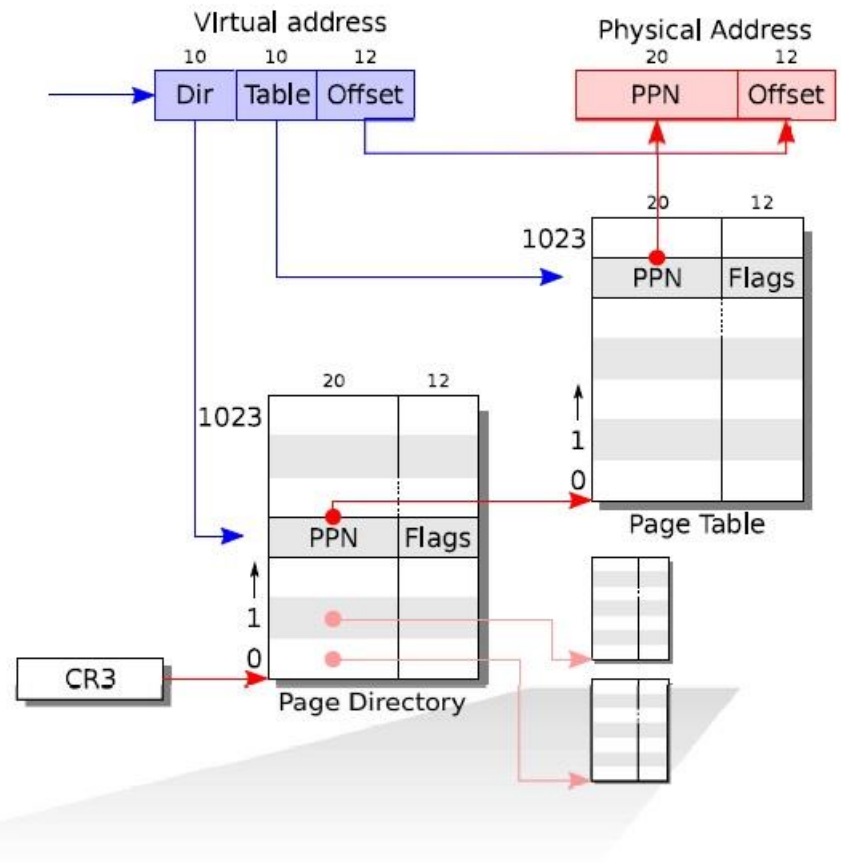
Memory management and memory abstraction is one of the most important features of any operating system. In this assignment we will examine how xv6 handles memory and attempts to extend it. To help get you started we will provide a brief overview of the memory management facilities of xv6. We strongly encourage you to read this section while examining the relevant xv6 files (vm.c, mmu.h, kalloc.c, etc).

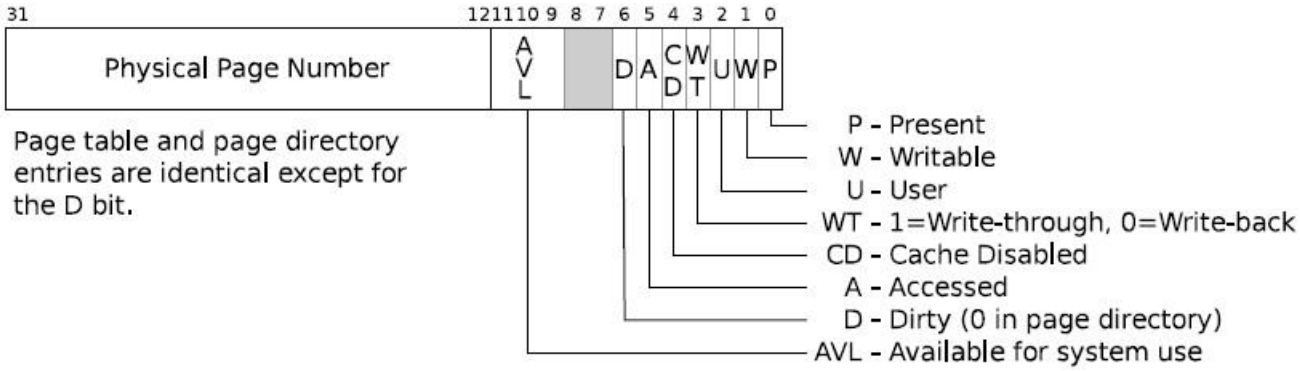
Memory in xv6 is managed in pages (and frames) where each such page is 4096 (=2^{12}) bytes long. Each process has its own page table which is used to translate virtual to physical addresses. The virtual address space can be significantly larger than the physical memory. In xv6, the process virtual address space is $2^{32} = 4$G bytes long while the physical memory is limited to 16MB only.

When a process attempts to access some address in its memory (i.e. provides a 32 bit virtual address) it must translate the address to the relevant page in the physical memory. The *Memory Management Unit (MMU)* uses the first (leftmost) 20 bits to locate the relevant Page Table Entry (PTE) in the page table. The PTE will contain the physical location of the frame - a 20 bits page address (within the physical memory). These 20 bits will point to the relevant page within the physical memory. To locate the exact address within the page, the 12 least significant bits of the virtual address, which represent the in-page offset, are concatenated to the 20 bits retrieved from the PTE. Roughly speaking, this process can be described by the following illustration:



Maintaining a page table may require significant amount of memory as well, so a two level page table is used. The following figure describes the process in which a virtual address translates into a physical one:





Each process has a pointer to its page directory (*line 59, proc.h*). This is a single page sized (4096 bytes) directory which contains the page addresses and flags of the second level table(s). This second level table is spanned across multiple pages which are very much like the page directory.

When seeking an address the first 10 bits will be used to locate the correct entry within the page directory (*see macro PDX(va)*). The physical frame address can be found within the correct index of the second level table (*see macro PTX(va)*). As explained earlier, the exact address may be found with the aid of the 12 LSB (offset).

At this point you should go over the xv6 documentation on this subject: https://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf (chapter 2)

Before proceeding further we strongly recommend that you go over the code again. Now, attempt to answer questions such as:

- How does the kernel know which physical pages are used and unused?
- What data structures are used to answer this question?
- Where do these reside?
- Does xv6 memory mechanism limit the number of user processes?
- What is the lowest number of processes xv6 can 'have' at the same time?

3.1. Enhanced process details viewer:

Prior to updating the memory system of xv6, you will develop a tool to assist in testing the current memory usage of the processes in the system. To do so, enhance the capability of the xv6's process details viewer. Start by running xv6 and then press ***ctrl + P*** (*^P*). You should see a detailed account of the currently running processes. Try running any program (e.g., usertests) and press *^P* again (during and after its execution). The *^P* command provides important information on each of the processes. However it reveals no information regarding the current memory state of each process. Add the required changes to the function handling *^P* (`see procdump() at proc.c`) so that the user space memory state of all the processes in the system will also be printed as written below. The memory state includes the mapping from virtual pages to physical pages (for all the used pages) and the value of the writable flag.

For each process, the following should be displayed (words in italics should be replaced by the appropriate values):

Process information currently printed with ^P
Page mappings:
virtual page number -> physical page number, writable?
…
virtual page number -> physical page number, writable?

For example, in a system with 2 processes, the information should be displayed as follows:

```
1 sleep init 80104907 80104647 8010600a 80105216 801063d9 801061dd
1 -> 300, y
200 -> 500, n
2 sleep sh 80104907 80100966 80101d9e 8010113d 80105425 80105216 801063d9
1 -> 306, y
200 -> 500, n
```

Showing that the first process (*init*) has 2 pages. Virtual page number 1 is mapped to physical page number 300 and its writable bit is set. The virtual page number 200 is mapped to physical page number 500 and its writable bit is not set. The second process (sh) also has 2 pages. Virtual page number 1 is mapped to physical page number 306 and its writable bit is set. The virtual page number 200 is mapped to physical page number 500 and its writable bit is not set.

Note that:

- The values above are made up.
- Don't print anything for pages or page tables that are currently unused.
- Only print pages that are user pages (PTE_U).
- The information obtained from ^P will be used for the frontal grading.

3.2. Copy on write (COW):

Upon execution of a fork command, a process is duplicated together with its memory. The XV6 implementation of fork entails copying all the memory pages to the child process, a scheme which may require a long time to complete due to the many memory accesses needed. In a normal program, many memory pages will have no change at all following fork (e.g., the text segment), furthermore, many times fork is followed by a call to exec, in which copying the memory becomes redundant. In order to account for these cases, modern operating systems employ a scheme of copy-on-write, in which a memory page is copied only when it needs to be modified. This way, the child and parent processes share equal memory pages and unneeded memory accesses are prevented.

In this task you are required to implement the COW optimization in XV6. To accomplish this task, change the behavior of the fork system call so that it will not copy memory pages, and the virtual memory of the parent and child processes will point to the same physical pages. Note that it is ok to copy page tables of a parent process (though you may also share/cow the page tables if you would like to). In order to prevent a change to a shared page (and be able to copy it), render each shared page as read-only for both the parent and the child. Now, when a process would try and write to the page, a page fault will be raised and then the page should be copied to a new place (and the previous page should become writeable once again).

In order to be able to distinguish between a shared read-only page and a non-shared read-only page, add another flag to each virtual page to mark it as a shared page. Notice that a process may have many children, which in turn, also have many children, thus a page may be shared by many processes. For this reason it becomes difficult to decide when a page should become writeable. To facilitate such a decision, add a counter for each physical page to keep track of the number of processes sharing it (it is up to you to decide where to keep these counters and how to handle them properly). Since a limit of 64 processes is defined in the system, a counter of 1 byte per page should suffice. Since the wait system call deallocates all of the process's user space memory, it requires additional attention – remember do not deallocate the shared pages.

Note that:

- When a page fault occurs, the faulty address is written to the cr2 register.
- After copying a page, the virtual to physical mapping for that page has changed, therefore, be sure to refresh the TLB using the following commands:

```
movl %cr3,%eax

movl %eax,%cr3
```

3.3. Sanity test:

To make sure that COW optimization implemented properly write a simple user program (i.e., cowtest) and use ^P to see that everything works properly. In this program use the command malloc to create new values that sit in different pages (in the heap). Use updated fork system call and make sure that the user space pages are shared. Now update a variable in the parent process. Was the proper page copied? Such a scenario, together with the information printed by ^P, can be used to check all the tasks above

Submission Guidelines

Make sure that your Makefile is properly updated and that your code compiles with no warnings whatsoever. We strongly recommend documenting your code changes with comments – these are often handy when discussing your code with the graders.

Due to our constrained resources, assignments are only allowed in pairs. Please note this important point and try to match up with a partner as soon as possible.

Submissions are only allowed through the submission system. To avoid submitting a large number of xv6 builds you are required to submit a patch (i.e. a file which patches the original xv6 and applies all your changes). You may use the following instructions to guide you through the process:

Back-up your work before proceeding!

Before creating the patch review the change list and make sure it contains all the changes that you applied and noting more. Modified files are automatically detected by git but new files must be added explicitly with the 'git add' command:

```
> git add . -Av; git commit -m "commit message"
```

At this point you may examine the differences (the patch):

```
> git diff origin
```

Once you are ready to create a patch simply make sure the output is redirected to the patch file:

```
> git diff origin > ID1_ID2.patch
```

- Tip: Although grades will only apply your latest patch, the submission system supports multiple uploads. Use this feature often and make sure you upload patches of your current work even if you haven't completed the assignment.

Finally, you should note that the graders are instructed to examine your code on lab computers only!

We advise you to test your code on lab computers prior to submission, and in addition after submission to download your assignment, apply the patch, compile it, and make sure everything runs and works.

Enjoy !!!