Assignment 4: Thread the Needle!

CS330: Operating System

1 Introduction

This assignment aims to understand the life cycle of a thread—from creation to its termination. As part of this assignment, you are required to implement user library calls, few system calls along with some specialized handlers for events like page fault etc. in the gemOS.

Thread is a light weight process which can be created by a process and will execute some part of the code segment of the parent process that has been assigned to the thread. When a thread is created, the thread uses the same virtual address space of the parent process. The creating process allocates a part of the virtual address space for the thread to use it as the stack. In gemOS, we represent the threads using `struct exec_context` just like a process. For accounting of threads, the parent process uses a reference to the structure called `ctx_thread_info` (Refer `include/context.h` and `include/clone_threads.h`) which is initialized on the first clone request (part of the template code `clone_threads.c`). Note that, the threads have their own register state (inside the PCB) and stack while they share the same address space of the process. Just like `fork`, when a new thread is created most of the contents of the PCB are copied onto the new thread's PCB.

1.1 Overview

The creation, exit and join of the threads from parent are performed in user space. Let us examine the following diagram which illustrates the flow of a thread from its creation to its termination. Note that, this is similar (but simplified) to POSIX thread (`pthread`) library.

1. First the process P1 calls **gthread_create()** which is to be implemented by the gthread library in the user space.

2. The gthread library invokes the clone system call by passing the function pointer, stack address and the thread parameter address. The thread library may use the the `mmap()` system call (refer to `testcases/clonetc1.c` for example usage) to allocate virtual address space for thread stack.

3. The gthread library may invoke another system call called **make_thread_ready()** to start execution of the thread *before returning* to the caller.

4. At this point a new thread is created with PID 2 and TID 0 and starts executing the function passed as an argument. Note that, just like pthreads, the main process returns to the point of invocation and continues execution. Further, the TID semantic is a user space construct as gemOS does not maintain any thread IDs.

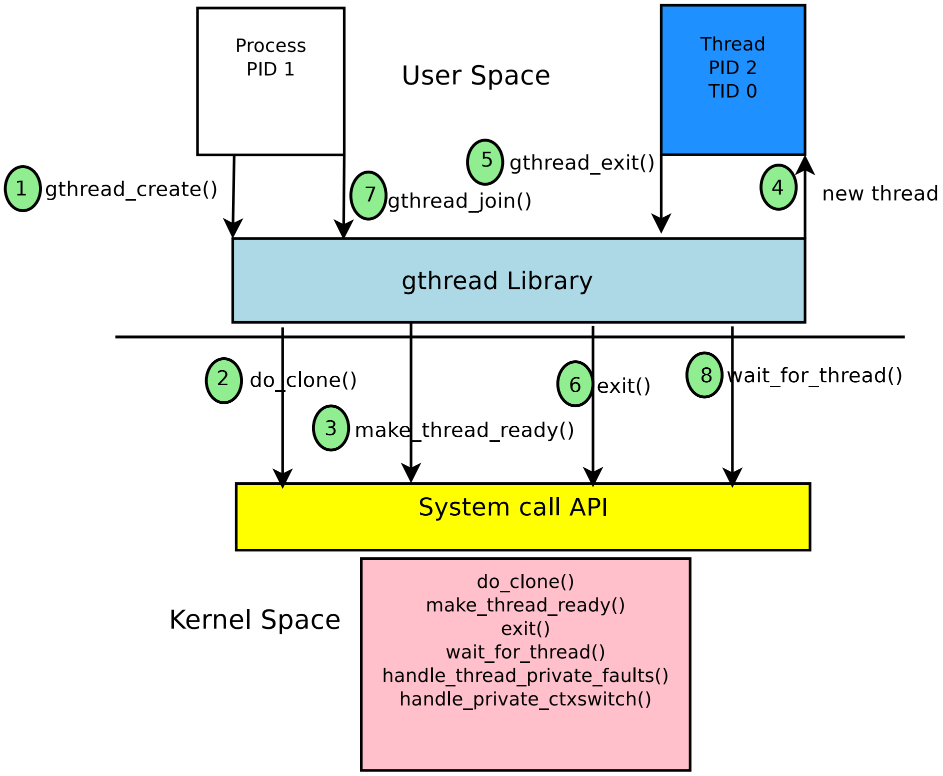


Figure 1: Assignment overview

5. At the end of thread execution, thread either returns from the function or invokes **gthread_exit()**. In both the cases, the return value is a pointer (can be NULL) and the thread library has to pass the return value back to the parent when **gthread_join()** is invoked from the parent process. Further, the **gthread_exit()** must call the exit system call to terminate itself.

6. The **gthread_exit()** library routine invokes the **exit()** system call to terminate the thread.

7. The parent process calls **gthread_join()** (can be long after the thread has exited) to collect the value returned by the thread and perform user-space thread cleanup activities. Note that, we will assume that all threads are joinable (not detached) in this design.

8. The parent process may use a specialized system call **wait_for_thread(tid)** to implement the **gthread_join()** functionality to wait for alive threads. The system call **wait_for_thread(tid)** is optional, you may or may not use it.

1.2 Structure of the template

In the released template, the relevant files are as follows,

clone_threads.c: This file provides function templates (for the OS side design) which are to be completed as part of the assignment. The relevant structures are defined in `include/clone_threads.h`. This file (`clone_theads.c`) *is to be submitted*.

clone_threads.h: This file contains declaration of different structures necessary to implement OS functionalities required to complete different parts of the assignment. As it is already mentioned, the parent process uses a pointer to the structure `ctx_thread_info` which contains further information related to threads. You are not allowed to modify anything in this file and *will not be* part of the submission.

clone_threads_helper.c: This file contains definition of some utility functions to manipulate and manage the OS-level meta-data for thread management. You may make use of these functions for implementing different tasks. If you require any extensions to the functionalities, please implement them in `clone_threads.c` as you *are not allowed to modify* anything in this file and it *will not be* part of the submission.

user/gthread.h: This file contains declaration of different structures necessary to implement user level `gthread` features. The structures and definitions provided are sufficient for the implementation except for the thread status macros. You may define new macros to capture state of threads as per the requirement. Note that, in the current template there is a marker which says not to modify anything above. Please adhere to that requirement. This file *is to be submitted*.

user/gthread.c: This file contains templates for different functions necessary to implement user level `gthread` features. You may define new functions (declare it with `static` qualifier). *Do not define any global/static* variables in this file (and other user space files including `init.c`). A block of code in the beginning of the file should not be modified (marked and commented). This file *is part of the submission*.

user/init.c: This file contains the `main` function to test various features. You may copy one of the test cases from the `user/testcases/` directory to `user/init.c` depending on the functionality being tested. This file *is not part of the submission*.

1.3 Assumptions

For this assignment we will assume the following,

- There is a single process and maximum four threads. Therefore, you can assume there is no invocation of `fork` system call in this assignment.
- If a thread exits (normally or because of a fault), only the thread will be terminated assuming your implementation has not corrupted the kernel itself.
- All threads are joinable i.e., the parent will call join to catch the return value from the threads. If the parent exits (normally or because of a fault) all threads are also terminated.
- As we are not using features like files and pipes, we will not test the implementation for the correctness of file inheritance.

2 Relevant OS Helper Functions

The following kernel methods are already implemented in `clone_threads_helper.c` and you may use them while solving the assignment. Note that, some of the function may not be required for implementing the tasks in the assignment. Moreover, you may have to define custom functions (do that in `clone_threads.c`) as per the requirement. The file `clone_threads_helper.c` *must not be modified* as several functions implement dependencies across other subsystems.

int do_make_thread_ready(long pid)

This kernel function is present in `clone_threads_helper.c` which implements the system call handler for `MAKE_THREAD_READY` invoked from user space using system call wrapper `make_thread_ready()` (defined in `user/lib.c`).

Description: This is an optional system call which can be used to control the thread scheduling. This system call sets the thread state to READY (see `include/context.h`) such that the gemOS scheduler can schedule it depending on its policy. This system call may be invoked from `gthread_create` in the user space thread library.

int do_wait_for_thread(long pid)

This kernel function is present in `clone_threads_helper.c` which implements the system call handler for `WAIT_FOR_THREAD` invoked from user space using system call wrapper `wait_for_thread` (defined in `user/lib.c`).

Description: This is an optional system call which you use to make the parent process wait till the thread corresponding to the pid parameter has exited. Note that, returning 0 from this system call does not mean the thread has finished execution. You need to perform additional checks in the user space to know if a thread has finished or not. Return of a negative value implies that the thread has died or an invalid thread ID is passed in the system call argument. This system call should be called from the parent process and could be used to implement `gthread_join` functionality.

void segfault_exit(int pid, u64 rip, u64 addr)

This kernel function is present in `clone_threads_helper.c` which should be invoked when any thread accesses an illegal address (see part three for more details). The parameters `rip` and `addr` correspond to the user space instruction pointer and the operand address for the illegal access, respectively.

Description: This should be called when there is a page fault that could not be fixed. The parameters are pid of the process/thread that has caused the page fault, instruction pointer (register value can be found in a process's context) and the address to which the access was requested. This function is useful to implement thread private functionality (part three of the assignment).

struct thread* find_unused_thread(struct exec_context *ctx)

This kernel function is present in `clone_threads_helper.c` which finds an unused thread (type `struct thread`) for the calling process.

Description: This function returns a thread whose status is TH_UNUSED if available other wise it returns NULL. The parent (thread creator) process execution context (`struct exec_context`) pointer is passed as a parameter. This should not be invoked with a thread execution context as the parameter.

struct thread* find_thread_from_pid(struct exec_context *ctx, int thread_pid)

This kernel function is present in `clone_threads_helper.c` which returns the thread meta-data (`struct thread` type) for the `thread_pid` parameter.

Description: This function can be used to find a thread from its pid. The `ctx` parameter must be the parent execution context.

Accounting call backs for thread private mapping

The `handle_thread_private_mmap` and `handle_thread_private_unmap` functions implement the callbacks from the `mmap` and `munmap` system call handlers, respectively. These functions are invoked for address ranges allocated using `mmap` system call if appropriate flags (MAP_TH_PRIVATE) and protection values (PROT_SIB_*) are passed from the user space. These callbacks manage and manipulate the `private_mappings` field of the `thread` structure.

static struct thread_private_map *find_thmap(struct thread *th, u64 address, u32 length)

This kernel function is present in `clone_threads_helper.c` which returns the thread private mapping meta-data (`struct thread_private_map` type) for the `thread_pid` parameter matching the address and the length.

Description: This function can be used to find the thread private information for a thread by matching the address with the start address of the private mapping recorded in the `private_mappings` structure. The length is matched only if it is passes as non-zero. Please refer to the comments in the template file.

struct thread* find_thread_from_address(struct exec_context *ctx, u64 addr, u32 length, struct thread_private_map **thmap)

This kernel function is present in `clone_threads_helper.c` which returns the thread structure corresponding to the address and fills the `thmap` result argument by matching the thread private mapping against the address. This function invokes `find_thmap` internally and therefore matching logic is same as the above function.

Page(PFN) management routines

Helper function like os_pfn_alloc and os_pfn_free can be used to allocate and free PFNs in gemOS. The prototype of these functions are present in include/memory.h. Sample allocation and free operations are shown below.

```
u32 os_pfn_alloc(u32 region) // Allocates and returns PFN in the OS ‘region’
// Sample code
u64 new_pfn = os_pfn_alloc(USER_REG);   // new_pfn is allocated from
                                           the ‘USER’ physical region
void *new_pfn_vaddr = osmap(new_pfn);   // new_pfn_vaddr is the OS virtual
                                        // address to access the new_pfn
/* Do something */

os_pfn_free(USER_REG, new_pfn);         // This will free the new_pfn.
                                        // The ‘region’ should be same as
                                           os_pfn_alloc
```

Note that, the two region flags required for this assignment are USER_REG and OS_PT_REG used for allocating PFNs for user and page table, respectively. The PFN should be multiplied with page size (4 kilobytes) to obtain the physical address. The osmap function provides the OS virtual address for any PFN and is generally used to access/modify page table PFNs. For example, after void *base = osmap(ctx->pgd), the base points to the virtual address of the first level page table.

3 Task 1: Clone system call [20 marks]

In this task, you will be implementing the following system call:

3.1 long do_clone(void *th_func, void *user_stack, void *user_arg);

The template for this system call handler can be found in clone_threads.c. Returns the pid of newly created thread on success and -1 on error.

th_func is the function pointer (address in the code segment) that will be executed once the thread returns to the user space.

user_stack is the starting address of the stack used by the newly created thread. Please note that the stack grows from higher address to the lower address. Refer user/testcases/clone to see how the stack address is passed.

user_arg is a pointer to the argument passed to the thread function. Note that, the thread function signature is fixed and only one argument is passed to the thread function.

Description: This system call should create a new child thread where most of the things are copied from the parent's context as most of them are pointers and are shared between parent process and child thread. As we use the struct exec_context to represent both process and threads, there is no distinction between a process and thread representation in the OS. However, the `type` field of the `struct exec_context` is used to distinguish them explicitly. Therefore, the type for the new context *must be set* to EXEC_CTX_USER_TH. The thread being created has to be initialized with its own stack which has been passed as an argument. Applicable registers in the `new_ctx->regs` must be updated such that the thread executes the function with the required argument. The thread state which is set to "UNUSED" in the template code should be removed and a proper state has to be set. If you are setting the thread state as "WAITING", it should be made ready by the explicit invocation of `make_thread_ready` system call. You can assume that the contiguous address space information represented by `struct mm_segment mms` in the execution context is not changed after the `clone` system call. However, the discontinuous address space represented by `struct vm_area * vm_area` in the execution context can change because of `mmap` and `munmap` system calls.

Return: On success it should return the pid of newly created thread to the caller. In case of error, return -1.

Assumptions:

- Sanity checks on the function address and stack address arguments are not required to be performed.
- The prototype of the function that is being passed is fixed i.e., `void * function_name (void *args)`.
- Maximum number of threads alive at any point will be four (MAX_THREADS).
- Maximum stack size will be 16KB. No test case using the stack will cross this limit.

4 Task 2: Thread Library (gthreads) [30 marks]

Now, that we have implemented **do_clone()** we can use the system call to implement few pthread like library calls. For this task, some additional meta-data maintenance is required both in the user space and the gemOS (`clone` system call handler). The structure of the OS side meta-data starts with a pointer `ctx_threads` which points to a structure containing "MAX_THREAD" number of `thread` structures. In the `clone_threads_helper.c` file, example manipulation and access to these structures are provided which you may use for the OS side thread related meta-data management (See Section 2). The user space library definitions of meta-data structures are present in `user/gthread.h`. A global variable `tinfo` of type `struct process_thread_info` is defined in `user/gthread.c` which can be used to access and manipulate subsequent members to implement the following library routines. Please refer to `user/gthread.h` for more details regarding the meta-data. In this task, you will be implementing the following library calls.

4.1 int gthread_create(int *tid, void *th_func, void *user_arg)

The template of this user library call is present in `user/gthreads.c`

tid is the pointer to an integer which holds the thread id (tid) when the function returns. Note that, tid is a user space construct and does not have any gemOS counter part. Therefore, it is up to the implementation logic to maintain the correlation.

th_func is the function pointer for which the thread is being created. The prototype of the function is fixed as shown in the example test cases (`user/testcases/gthread/`).

user_arg is the argument that has to be passed as an argument to the `th_func` when the thread starts execution.

Description: This library call should allocate a stack for the thread and invoke the `clone` system call with appropriate arguments. The stack size used for the thread is `TH_STACK_SIZE`. All maintenance of thread related structures should be performed before returning from the call. As part of the maintenance, the function should check the feasibility of creating new thread by checking the MAX_THREADS limit. The newly created thread can terminate in two ways—(i) by invoking `gthread_exit` and, (ii) by returning from the thread function. Your implementation should ensure correct working of both the cases.

Return: On success it should return the 0. In case of error, return -1.

Assumptions/Notes:

- Number of threads at any point of time cannot exceed `MAX_THREADS` which is set to 4 and will not be changed during testing.
- On successful return from `gthread_create`, the new thread *can not be* in `WAITING` state. Your implementation must ensure this behavior.
- The raw implementation of `clone` system call may not be sufficient and require some additional accounting in the OS.
- This function is invoked only from the parent process.

4.2 int gthread_exit(void *retval)

The template of this user library call is present in `user/gthreads.c`.

retval is return value (of type void *) to be consumed by the parent process when `gthread_join` is called. This value can be NULL.

Description: This library function when called from a thread should perform the necessary cleanup and must terminate the calling thread by calling `exit` system call. If a thread doesn't call `gthread_exit()` (because it can also return from thread function), the gthread library must ensure that the returning thread is terminated by calling the `exit` system call.

Return: This function never returns to the calling function as the corresponding execution context is terminated.

Assumptions/Notes:

- This function is invoked only from the thread, never from the parent process.
- Before returning from this function, `exit` system call *must be* invoked.

4.3 void* gthread_join(int tid);

The template of this user library call is present in `user/gthreads.c`

tid is the thread ID passed by the parent process.

Description: If the thread corresponding to the `tid` has not finished, the parent (caller) has to wait till the thread is terminated. You may use the `wait_for_thread` system call (explained in Section 2) by passing the pid of the thread. Further, the accounting related to the thread has to be appropriately updated during this call.

Return: It should return the value returned by the thread either through the invocation of `gthread_exit()` or return from the thread function. This is a blocking call as mentioned earlier.

Assumptions/Notes:

- This function is invoked only from the parent, never from any thread.
- Return from this function implies that all reference (OS and user) to the thread are destroyed.
- During testing, join will be called for all the threads.

5 Task 3: Thread private memory [50 marks]

Typically, threads share the address space of the parent process and therefore they access the whole user address space without any issues. The objective of this task is to implement an augmented address space where a thread can decide how a range of address is accessed by its siblings. In this task, you will be implementing the following library calls and gemOS handlers.

5.1 void* gmalloc(u32 size, u8 alloc_flag);

The template of this user library call is present in `user/gthreads.c`

size is the size of the private memory allocation request. This will be always a multiple of page size (4KB).

alloc_flag determines the access rights to the private memory for the other threads.

Description: This library call allocates a region of virtual address space (with read and write permission for the allocating thread) based on the `size` argument. The access rights (`alloc_flag`) along with the size is used to determine the arguments to the `mmap` system call to allocate a range of virtual address. The first and last arguments to the `mmap` system call must be NULL and `MAP_TH_PRIVATE`, respectively. The `prot` parameter should be a bit-wise OR of `PROT_READ`, `PROT_WRITE` and one of the following depending on the `alloc_flag` value (defined in `src/user/gthread.h`).

- If the `alloc_flag` is equal to `GALLOC_OWNONLY`, the `prot` parameter must be bit-wise ORed with `TP_SIBLINGS_NOACCESS`.

- If the `alloc_flag` is equal to `GALLOC_OTRDONLY`, the `prot` parameter must be bit-wise ORed with `TP_SIBLINGS_RDONLY`.
- If the `alloc_flag` is equal to `GALLOC_OTWR`, the `prot` parameter must be bit-wise ORed with `TP_SIBLINGS_RDWR`.
- If none of the above, the allocation should return NULL without calling `mmap`.

While the thread private information at the OS layer is already implemented (if the flags are passed correctly), the user space private mapping information is required to be maintained as part of the assignment.

Return: On success, the function should return the starting address of the newly created private mapping. In case of error, return NULL.

Assumptions/Notes:

- A thread can have up to MAP_TH_PRIVATE number of private mappings. This macro will not be changed during testing.
- Only a thread can request for a private mapping using this library call.
- The mapping information in the user space should be cleaned up when a thread is terminated. The cleaning up of the OS part is already implemented.
- The maximum size of allocation during testing will be `GALLOC_MAX`.
- The parent process can access the thread private mapping addresses.
- The parent process will not call `munmap` to deallocate the allocated memory in a out-of-band manner.

5.2 int gfree(void *ptr);

This function deallocates a previously allocated private mapping. The template of this user library call is present in `user/gthreads.c`

ptr is the start address of the private mapping that has to be freed.

Description: This library function when called should free the private mapping with start address as `ptr`. When a thread calls `gfree` on an invalid address, it should return -1. If a thread calls `gfree` for some other thread's private mapping or any other virtual address it should return -1. You should update the private mapping meta-data in user space as part of this function.

Assumptions/Notes:

- Only a thread can request to free the private mapping using this library call.
- Only the owner thread (allocator) of a private mapping should be able to free the area.

- Assume that the address is always the beginning of the private mapping and the entire private mapping will be freed through this call.
- The mapping information in the user space should be updated when a thread invokes this function.
- Any access to the area after a successful completion of this call from any thread/parent will result in a segmentation fault message. Your implementation should ensure that.

Return: On success it should return 0. In case of error, return -1.

5.3 int handle_thread_private_fault(struct exec_context *current, u64 addr, int error_code)

This function is invoked from the generic page fault handler of the gemOS for a private mapped address. The template of this user library call is present in `clone_threads.c`

current is the pointer to the current execution context (can be a thread or process).

addr is the fault location in the virtual address space which was accessed by the current execution context and resulted in a fault.

error_code is the X86 error code used to determine the nature of the fault. Please refer to the appendix for exact semantics of the error codes.

Description: This kernel function will be called when a page fault occurs in a thread private mapping area of the address space. You should find the owner of the faulting address by examining the OS meta-data and apply the fault handling logic according to the access flags for the area. Note that, for parent process, the sibling access logic will not be applicable. To fix the page fault for a legitimate access, you are required to walk the page table and fix the mappings such that the access will be success after returning from the page fault. You can access the Level-1 page table using the `pgd` member of the execution context. Please refer to the Appendix for details of page table layout and structure of page table entries. If the access is not legal, the function should call `segfault_exit()` with the PID of the responsible thread i.e., the thread which caused the fault along with the faulting instruction pointer and address.

Return: If the page fault is fixed, it should return 1. If the fault can not be fixed, `segfault_exit()` with appropriate parameters should be invoked.

Assumptions/Notes:

- There can be faults because of lazy allocation. Your implementation should handle that.

5.4 int handle_private_ctxswitch(struct exec_context *current, struct exec_context *next)

This function is invoked from the gemOS scheduler when there is a context switch between two entities sharing the address space (one of them can be a parent process). The template of this callback is present in `clone_threads.c`.

current is a pointer to the current execution context which in this case is going to be scheduled-out.

next is the execution context which going to be scheduled-in on to the CPU.

Description: This kernel function will be called from the gemOS scheduler. Depending on the `current` and `next` parameters, this function should implement the access logic for the private areas. The access logic is implemented by manipulating the virtual to physical address mapping in the page table. The OS meta-data related to private mappings and the related OS helper routines may be used to determine the owner and apply the policies according to the access permission (set during `mmap` called from `gmalloc`). After this manipulation, the accesses of `next` execution entity (process or thread) should be as per the access policy.

Return: Return 0 on success and -1 in case of any failure.

Assumptions/Notes:

- The page table updation process is similar to fault handling (See Appendix).

Submission guidelines

You have to submit a zip file containing the following source files. First create a directory with your roll number as the name and structure the files as shown below within the directory. We will not evaluate submissions not following the structure shown below.

(A) `gemOS/src/clone_threads.c`

(B) `gemOS/src/user/gthread.c`

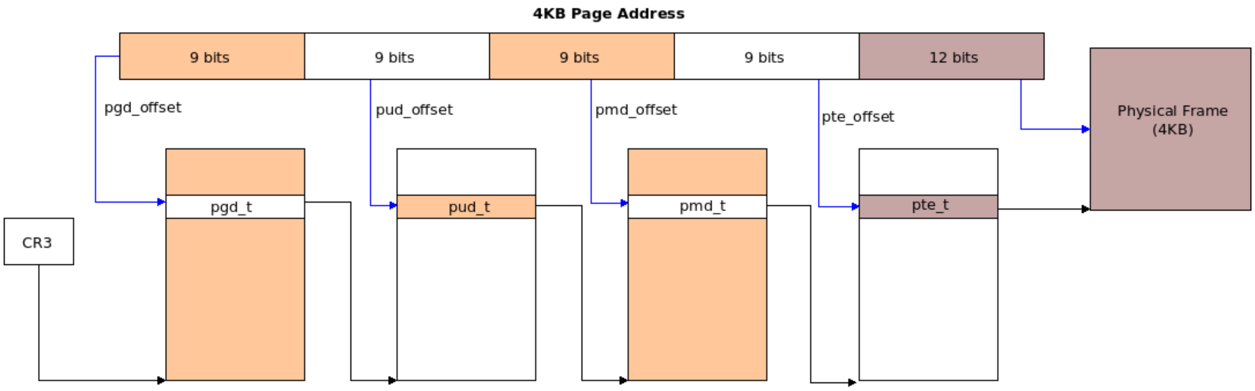
(C) `gemOS/src/user/gthread.h`

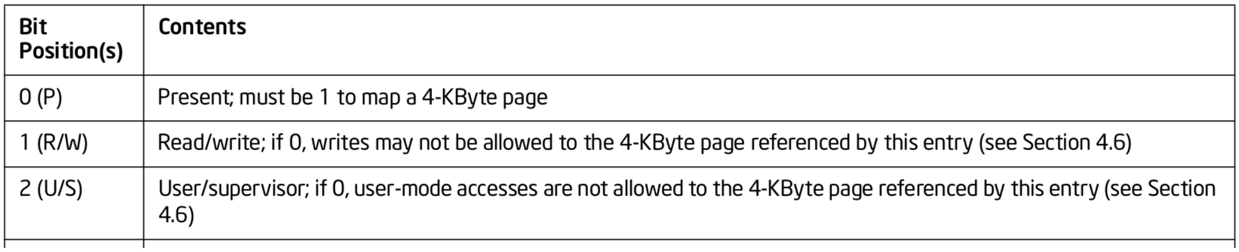
Name of the zip file must be <your rollno>.zip.

Appendix

4-Level Page Table

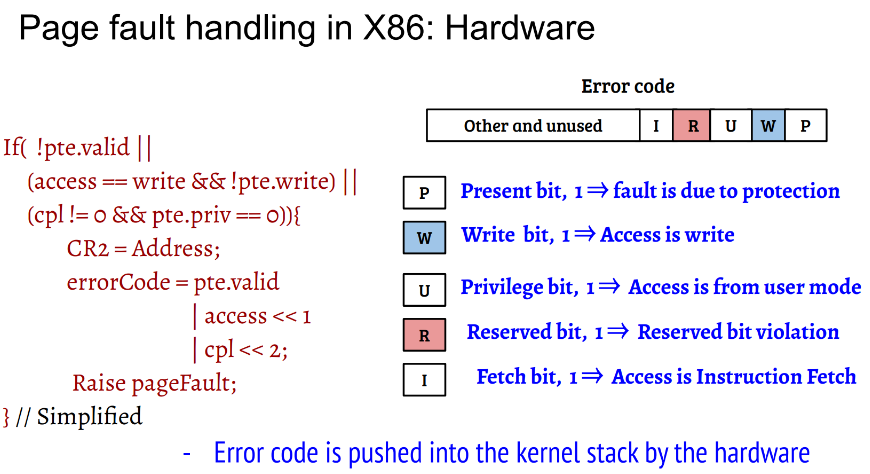
The following figure shows page table layout for 48 bit virtual to physical address translation for a 4KB page. i The `pgd` field of execution context (`struct exec_context` in gemOS) has the base address of `pgd` table. The least significant three bits of the page table entry (relevant for this assignment) signifies the present, read/write and user/supervisor details of mapped 4KB page as shown below. You may refer to Intel Software Developers Manual Volume 3 for more details on other fields in pte entry.





Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)

Page fault error codes



Page fault handling in X86: Hardware

```
If( !pte.valid ||
   (access == write && !pte.write) ||
   (cpl != 0 && pte.priv == 0)){
        CR2 = Address;
        errorCode = pte.valid
                    | access << 1
                    | cpl << 2;
        Raise pageFault;
} // Simplified
```

Error code

Other and unused	I	R	U	W	P

P Present bit, 1 ⇒ fault is due to protection

W Write bit, 1 ⇒ Access is write

U Privilege bit, 1 ⇒ Access is from user mode

R Reserved bit, 1 ⇒ Reserved bit violation

I Fetch bit, 1 ⇒ Access is Instruction Fetch

- Error code is pushed into the kernel stack by the hardware

Figure 2: Page-Fault Error Codes

The error codes generated by the hardware and passed as an argument to the thread private fault handler can be interpretted using the above figure. The value of some of the error codes and their semantics are as follows,

`0x4` User-mode read access to an unmapped page

`0x6` User-mode write access to an unmapped page

`0x7` User mode write access to read-only page

Only the least three significant bits are useful for this assignment.