EEE3535-01 | Fall 2023

Assignment 6: Thread

Due: Thursday, Dec. 21, 2023, 11:59PM

1 Introduction

- The objective of this assignment is to implement multi-threading features in xv6-riscv.
- The baseline xv6-riscv does not have multi-threading features for user processes such as thread creation and join, mutual exclusion, etc. This assignment will implement those missing features in xv6-riscv.

2 Implementation

- To start the assignment, go to the `xv6-riscv/` directory, download `thread.sh`, and run the script.

```
$ cd xv6-riscv/
$ wget https://icsl.yonsei.ac.kr/wp-content/uploads/thread.sh
$ chmod +x thread.sh
$ ./thread.sh
```

- If the update is successful, you can find the new user program `threadtests`. However, the program will fail because xv6-riscv misses the essential multi-threading features. Terminate xv6-riscv by pressing `Ctrl+a`, `x`.

```
$ threadtests
Test #1: creating as many threads as possible           (fail)
Test #2: creating threads and passing arguments         panic: thread_create failed
$
```

2.1 User-side Interfaces:

- In the updated xv6-riscv, `user/uthread.h` defines user-side interfaces for thread creation, join, exit, and mutual exclusion.

```
// Thread and mutex
typedef int thread_t;
typedef struct _thread_mutex_t_ { int locked; } thread_mutex_t;

// Thread interfaces
int thread_create(thread_t *t, void*(*func)(void*), void *arg);
int thread_join(thread_t t, void **ret);
void thread_exit(void *ret) __attribute__((noreturn));

// Mutex interfaces
void thread_mutex_init(thread_mutex_t *mtx);
void thread_mutex_destroy(thread_mutex_t *mtx);
void thread_mutex_lock(thread_mutex_t *mtx);
void thread_mutex_unlock(thread_mutex_t *mtx);
```

- The thread type is defined as `thread_t`, which is, in fact, `int`.
- The mutex type is defined as `thread_mutex_t`. It contains only one variable `locked` that indicates whether the lock is held.
- `thread_create()`, `thread_join()`, and `thread_exit()` are used for creating a thread, waiting for a specified thread to complete, and exiting a thread function, respectively. Their formats are similar to pthread interfaces, except for having no thread attributes, so the functions should be straightforward to understand.

- Similarly, `thread_mutex_init()`, `thread_mutex_destroy()`, `thread_mutex_lock()`, and `thread_mutex_unlock()` are used for initializing, destroying, locking, and unlocking a mutex variable, respectively.

- The thread and mutex functions are in `user/uthread.c`, but they are empty or return arbitrary values (e.g., `return -1`) to avoid compile errors. Fill in the functions to make them work as intended.

2.2 Thread Creation

- Threads are created and scheduled similarly to regular processes.

- The only difference between threads and regular processes is that threads share the same virtual address space, file descriptors, process ID, and process name as their parent process.

- Since threads are treated almost the same as regular processes, we will repurpose `struct proc` in `kernel/proc.h` to manage the threads. Such an implementation will help reuse most of the kernel functions in xv6-riscv.

- To get a hint on how a thread can be created, take a look at the `fork()` syscall in `kernel/proc.c`, which creates a new process.

- `thread_create()` is supposed to do a similar task as `fork()`. However, thread and process creation procedures are not exactly the same, so xv6-riscv will need a new syscall, namely `tfork()`. `thread_create()` is a simple wrapper around `tfork()`.

- Design the `tfork()` syscall as necessary. Syscalls in `kernel/sysproc.c` can retrieve integer arguments using `argint()` and user-space pointers via `argaddr()`.

- Take a look at the existing syscalls to figure out how the input arguments of `thread_create()` can be passed to `tfork()` on the kernel side.

- `tfork()` should do almost all things `fork()` does but differs in a few parts. The following walks through the existing `fork()` function and explains what will be necessary for `tfork()` to create a thread.

- `fork()` first calls `allocproc()` to grab and initialize a process entry in `struct proc proc[]`. Since a thread will be created reusing `struct proc`, `tfork()` should do a similar job as `allocproc()`.

- `allocproc()` searches the `proc[]` array to find a slot whose state is `UNUSED`, which indicates that no active process is in the entry. `tfork()` should do the same thing to find a free slot in `proc[]`.

- Similar to `allocporc()` assigning a process ID (PID) to a new process, `tfork()` also needs to assign a PID to the new thread. However, the PID must come from the parent's PID, not from `allocpid()`.

- Since all threads of the same process share the same PID, the PID alone is insufficient to identify the threads.

- Thus, every thread needs a thread ID (TID) in addition to a process ID. You can imitate how PIDs are generated and assigned to new processes for TIDs. This will require adding a TID field (e.g., `int tid`) to `struct proc`.

- The TID will be returned to a user program in the first argument of `thread_create()` (i.e., `thread_t *t`). The user program will later use the TID for `thread_join()`.

- After working on the PID, `allocproc()` allocates a new physical frame using `kalloc()`, where the trap frame of the new process will be stored. `kalloc()` returns the physical address of the new frame.

- `struct trapframe` is defined in `kernel/proc.h`, which is simply a list of RISC-V CPU registers. `trapframe` is where the user context of the process will be stored on context switching.

- Since every thread needs a `trapframe` to maintain its execution state, `tfork()` has to allocate a new physical frame for the thread.

- Then, `allocproc()` calls `proc_pagetable()`. This function allocates a new physical frame where the root page table (or page directory) of the process will be stored. And it calls `mappages()` twice, one for recording the virtual-to-physical address translation of `trampoline` in the page table and another for `trapframe`.

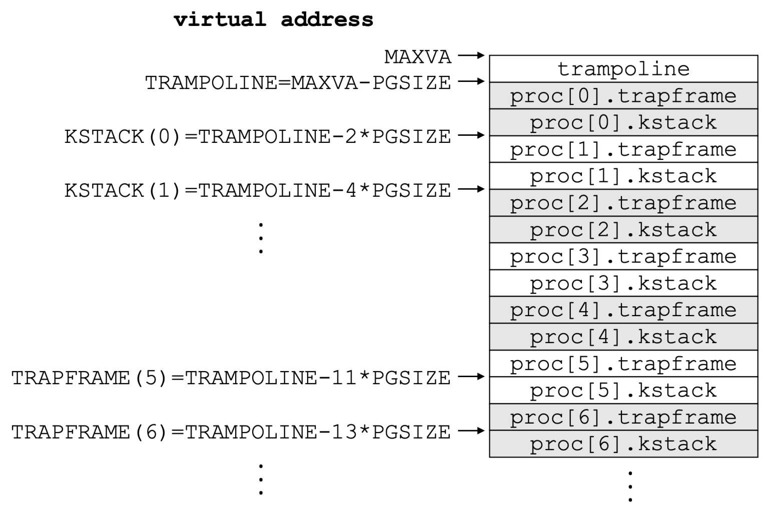

Figure 1: The virtual memory layout of a process near the top of the address space. The topmost page stores `trampoline`, and the `trapframe` and `kstack` of `NPROC` processes are below the `trampoline` page.

- `trampoline` is defined in `kernel/trampoline.S`. This function is executed whenever user-to-kernel mode switching occurs, such as system calls, timer interrupts, exceptions, etc.
- In `proc_pagetable()`, uppercase `TRAMPOLINE` and `TRAPFRAME` are virtual addresses, and lowercase `trampoline` and `trapframe` have physical addresses.
- You do not have to worry about the physical addresses but need to understand the virtual memory layout of a process and where `TRAMPOLINE` and `TRAPFRAME` are.
- Fig. 1 shows a part of the virtual memory layout near the top of the address space.
- The topmost page is reserved for `trampoline`, defined as `#define TRAMPOLINE (MAXVA - PGSIZE)` in `kernel/memlayout.h`.
- Pages below `trampoline` are where the `trapframe` and `kstack` of user processes are located. These pages are accessible only in the kernel mode.
- The `trapframe` of a process stores its user context as explained earlier, and `kstack` is the process's kernel stack. Every process needs a kernel stack because they can be in all different kernel functions during runtime.
- For instance, one process may be paused in `sleep()` of `kernel/proc.c` while another is timer-interrupted in `yield()`. Thus, kernel stacks are used for maintaining the execution state of different processes in the kernel.
- All previous assignments in this class so far used a single CPU, where acquiring and releasing process locks (e.g., `acquire(&p->lock)` and `release(&p->lock)`) had no effects.
- However, this assignment configures xv6-riscv to use two CPUs (or two cores), so process locks must be carefully manipulated to avoid race conditions.
- The multi-cores are defined as `struct cpu cpus[NCPU]`, and each CPU runs an independent process scheduler. The context of a scheduler is stored in `struct context context` of `struct cpu` in `kernel/proc.h`.
- For a regular process, the `context` stores the kernel context of the process on context switching (i.e., `swtch()` for scheduling).
- Fig. 2 shows the timeline of an execution flow in a CPU. A dual-core model simply has another execution flow asynchronous to the other core's execution flow.

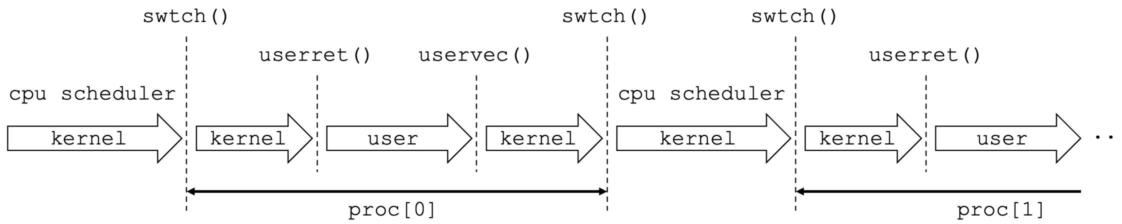

Figure 2: Timeline of the CPU execution flow. The CPU scheduler selects the next process and calls `swtch()` to context-switch to the selected process. The process runs in the kernel mode using its `kstack`. After kernel-side operations are done, `userret()` restores the context of the user program from the process's `trapframe`. Then, the user program runs. When the user program is interrupted, the `trampoline` routine executes `uservec()` to save the user context and jumps to `usertrap()`. Necessary kernel functions are executed for the process using its `kstack`. If the process calls `sched()`, the `swtch()` routine swaps registers (i.e., `context`) between the process and CPU scheduler. The scheduling algorithm selects the next process to run. The rest of the timeline repeats the same steps.

- From the left of the timeline, a CPU scheduler selects a process in `scheduler()` and calls `swtch()` to switch from the CPU scheduler to the selected process.
- The process resumes (or initiates) its execution in the kernel, and this is when the process's `kstack` is used.
- After kernel-side operations are done, `userret()` is called to restore the user context of the process saved at the process's `trapframe`. The user program runs until it has a syscall, causes an exception, or is interrupted.
- When the user program is paused, the `trampoline` routine is executed by default. `uservec()` saves the context of the user program at the process's `trapframe` and jumps to `usertrap()` in `kernel/trap.c`. This is again when the `kstack` of the process is used.
- If the process calls `sched()`, the `swtch()` routine swaps registers (i.e., `context`) between the process and the CPU scheduler. This is when a scheduling algorithm kicks in and selects the next process to run. The rest of the timeline repeats the same procedure.
- With the understanding of `trampoline` and `trapframe`, go back to `proc_pagetable()` in `fork()`. It creates a page table for the new process and maps `trampoline` and `trapframe` in the newly created page table.
- In a multi-threaded program, child threads share the same virtual address space with their parent. It means that the children use the same `pagetable` as their parent.
- Since a new thread should use the existing page table of its parent, it must not map `trampoline` again in the `pagetable`. Otherwise, it will throw a `panic("remap")` error.
- However, `tfork()` still needs to map the new thread's `trapframe` to a proper virtual page based on its relative position in the `proc[]` array.
- `kernel/memlayout.h` provides `KSTACK()` and `TRAPFRAME()` macros to the calculate the virtual address of the `kstack` and `trapframe` of a process (or thread) illustrated in Fig. 1.
- Returning from `proc_pagetable()` goes back to `allocproc()`. The last part of `allocproc()` initializes the kernel context of the new process.
- The return address (i.e., `ra`) of the process's context is set to `forkret` so that it calls `usertrapret()` and starts a user program. The thread creation function should do the same job.
- The stack pointer (i.e., `sp`) of `context` is set to the top of the `kstack` page by adding `PGSIZE` since the stack should grow downward.
- Kernel stacks for process entries in `proc[]` are created only once and for all in `procinit()` when xv6-riscv starts. The kernel stacks are never allocated and deallocated when a process (or thread) is created or killed.

- Returning from `allocproc()` goes to `fork()`. It copies user memory from the parent's virtual address space to the child's virtual memory via `uvmcopy()`. This part is not necessary for `tfork()` because the parent and child threads share the same virtual memory space.
- The next code line copies the value of the virtual memory size (i.e., `sz`) of the parent process to the child.
- `tfork()` also needs to do a similar thing, but it has a pitfall. If the child thread simply copies the value of `sz`, then the parent and child threads end up using two independent `sz` variables for the same virtual address space, which will mess up controlling the heap.
- To resolve the problem, the `sz` of the child thread must be linked to that of the parent process such that they reference the identical `sz` value. An easy solution is to introduce a pointer that points to the parent's `sz` and make the child thread refer to the pointer instead of using a separate `sz` variable.
- Then, `fork()` copies the `trapframe` of the parent to that of the child. It is unnecessary for `tfork()` since the new thread does not return to the caller function, whereas `fork()` makes the parent and its child resume their executions in the same function.
- Instead, the new thread starts a thread function `func` with a function argument `arg`, which are the second and third arguments of `thread_create()`, respectively. How to make the thread execute the thread function `func` will be explained later.
- Continuing the walk through `fork()`, it sets `trapframe->a0 = 0` because the `fork()` syscall should return 0 to the child process.
- However, the new thread will not return to the caller function in a user program, meaning that it has nothing to return. Notably, the `a0` register of RISC-V is used for both input argument and return value of a function, so `a0` in this case should be used to pass the thread function argument, such as `trapframe->a0 = arg`.
- The next several lines of `fork()` copy the file descriptors of the parent process to the child. In a multi-threaded program, the parent and child threads belong to the same process, so they must share the file descriptors, similar to the `sz` problem.
- An easy fix is to introduce pointers that point to the parent's `ofile[]` and `cwd` and make child threads refer to the pointers instead of using separate `ofile[]` and `cwd`.
- Then, `safestrcpy()` copies the characters of the parent process's name to the child. `tfork()` does the same.
- Lastly, the process calling `fork()` becomes the parent of the new process, and the child's state is set to `RUNNABLE`. `tfork()` should do the same with process locks.
- `thread_create()` has technically two roles, i) creating a new thread and ii) making the thread execute a thread function. Thus, `thread_create()` can be regarded as a combination of `fork()` and `exec()` syscalls.
- What is remaining for `tfork()` is the `exec()` part. The `exec()` syscall is defined in `kernel/exec.c`.
- The first part of `exec()` works on creating a new page table and loading a program code at the bottom of the virtual address space. These operations are unnecessary in `tfork()` since the new thread uses the same page table as its parent and executes the same code that is already in the parent's virtual memory.
- Then, `exec()` allocates `NPROC+1` pages. The first page at the lowest address is reserved as a guard page, and the next `NPROC` pages are used for user stacks as shown in Fig. 3. This part was modified from the baseline xv6-riscv, where a process only uses one `PGSIZE`-sized user stack.
- In this assignment, a process reserves `NPROC` pages below the heap and uses one of them as its user stack based on its relative position in the `proc[]` array. In the case of a multi-threaded program, it will use as many user-stack pages, which needs to carefully identify which user stack to use for each thread.
- Reserving (`NPROC+1`) pages with `uvmalloc()` occurs only in `exec()`. `tfork()` must not perform this operation since the parent process has already allocated user-stack pages in the shared virtual memory space.

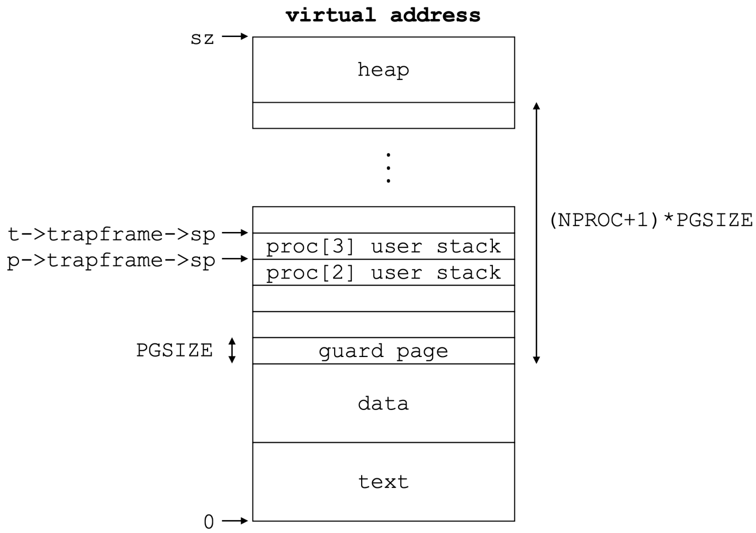

Figure 3: A process reserves (NPROC+1) pages below the heap. The first page at the bottom is a guard page, and the next NPROC pages are for user stacks. A regular process uses one of them as its user stack based on its relative position in the proc[] array. A multi-threaded program uses respective user stacks for its threads.

- Thus, most of the exec()'s operations are unnecessary in tfork(). The only required operation is to set up the sp and epc of the new thread's trapframe.
- tfork() should make trapframe->sp point to the top of the thread's user stack. For example, suppose the new thread (t) is allocated in the fourth entry of proc[] (i.e., proc[3]), and its parent process (p) is in proc[2], such as Fig. 3.
- The parent's stack pointer must be pointing to somewhere in the page labeled as proc[2] user stack. Rounding up the parent's stack pointer (i.e., PGROUNDUP(p->trapframe->sp)) gets the memory address where p->trapframe->sp is pointing to in the figure, and adding the positional difference between t and p (i.e., (t-p)*PGSIZE) will get where t->trapframe->sp should point to.
- t->trapframe->epc is the program counter that the new thread will execute, which must point to the thread function func. Doing so will make the thread start its execution from func.
- On the successful creation of the new thread, tfork() returns the new thread's TID, which will be passed in the first argument of thread_create() back to the user program.
- On receiving a non-zero TID from tfork(), thread_create() returns 0 for success.

2.3 Thread Join

- thread_join() waits for a thread to complete, whose functionality is similar to wait().
- However, the waiting procedures are not exactly the same between regular processes and threads, so thread_join() will need a new syscall, namely twait().
- The following walks through the existing wait() function in kernel/proc.c and explains what will be necessary for twait().
- wait() first searches proc[] to find an entry whose parent is the caller process and if its state is ZOMBIE.
- twait() does the same search but should additionally check if the entry's TID matches the first argument of thread_join() since it waits for only one particular thread at a time.
- If a matching entry is found but its state is not ZOMBIE, the process goes to sleep by sleep().

- If the child is `ZOMBIE`, `wait()` uses `copyout()` to copy the `xstate` value of the finished process to `addr` if `addr` is not a null pointer.
- This operation lets the parent process read the exit state value of the child, such as `exit(0)`, `exit(1)`, etc. A non-zero number passed to `exit()` conventionally means that the process finished with an error. A good use case can be found at line #2162 of `user/usertests.c`, which is `wait(&xstatus)`.
- `twait()` also needs to do a similar work. In this case, `addr` comes from the second argument of `thread_join()`, and what needs to be copied to `addr` is the return value passed to `thread_exit()`.
- Then, `wait()` calls `freeproc()` to release the resources of the finished process and reset its entry in `proc[]` for next use. This operation completely removes the process from the system.
- In `freeproc()`, it first deallocates the physical frame of the exited process's `trapframe`. Then, it calls `proc_freepagetable()` to unmap `TRAMPOLINE` and `TRAPFRAME` from the page table. `proc_freepagetable()` lastly calls `uvmfree()` in `kernel/vm.c` to deallocate all the pages and remove the page table.
- In the case of `twait()`, it should also deallocate the physical frame of the exited thread's `trapframe` and unmap its `TRAPFRAME` from the page table. However, all others (i.e., `TRAMPOLINE` and heap) should remain since the parent and other threads still need them.
- The remainder of `freeproc()` does the tedious work of resetting all the variables of `struct proc` so that the entry can be reused. `twait()` should do the same.

2.4 Thread Exit

- A thread function ends with `thread_exit()`. It takes a `void` pointer as an argument, which will be returned to the parent process.
- Similar to `thread_create()` and `thread_join()`, it needs a new syscall, namely `texit()`.
- The following walks through the existing `exit()` function in `kernel/proc.c` and explains what will be necessary for `texit()`.
- The first part of `exit()` closes file descriptors. These operations are unnecessary in `texit()` because the file descriptors are shared with the parent and other threads. The parent process will close the file descriptors later.
- Then, `exit()` calls `reparent()` to check if it has unfinished child processes, which will become orphans after the process exits. The child processes are adopted to the `init` process. `texit()` should do the same to check if it has unfinished child threads.
- Then, the exit state of the process (e.g., `exit(0)`) is stored in `xstate` so that its parent process can later read it via `copyout()`. `texit()` also needs a similar thing, but `thread_exit()` passes a pointer as its argument, not a plain integer.
- A simple solution is to introduce a new field (e.g., `uint64 xret`) to `struct proc` to save the return pointer of `thread_exit()`. Then, `texit()` should set `xret`, not `xstate`.
- Lastly, `exit()` makes the process's state `ZOMBIE` and calls `sched()` for context switching. `texit()` should do the same.

2.5 Mutual Exclusion

- Unlike `thread_create()`, `thread_join()`, and `thread_exit()` that involve complex operations in the kernel, mutex interfaces are all implemented in the user space using atomic instructions.
- As a reference, take a look at how `kernel/spinlock.c` implements a spin lock using builtin compiler primitives, such as `__sync_lock_test_and_set()` and `__sync_lock_release()`. You may bring the example to your lock implementation and let the compiler generate the necessary RISC-V instructions.
- Alternatively, you may add a new `yield()` syscall and make the caller thread give up the CPU if it fails to acquire the lock. The `yield()` is already in the kernel, but it is not exposed to user programs as a syscall.

3 Validation

- Your assignment will be graded based on the following test cases.

  1. **`threadtests`:**
     - `user/threadtests.c` has seven test cases that validate various aspects of the multi-threading features. Each test must finish cleanly.

       ```
       $ threadtests
       Test #1: creating as many threads as possible            (pass)
       Test #2: creating threads and passing arguments          (pass)
       Test #3: reclaiming return values from thread functions  (pass)
       Test #4: recursively creating threads and checking PIDs  (pass)
       Test #5: sharing virtual memory space between threads    (pass)
       Test #6: sharing file descriptors between threads        (pass)
       Test #7: mutex lock on critical sections                 (pass)
       ALL TESTS PASSED
       $
       ```

     - If you see an error message after `(pass)`, there is a good chance that the previous test actually failed; it luckily produced correct user values but messed up thread states in the kernel.
     - In such a case, you may specify the test number(s) to check if the error is due to the previous test or the current test.

       ```
       $ threadtests 4 5
       Test #4: recursively creating threads and checking PIDs  (pass)
       Test #5: sharing virtual memory space between threads    (pass)
       ALL TESTS PASSED
       $
       ```

  2. **`usertests`:**
     - If you passed `threadtests`, run `usertests` to confirm your thread implementation does not mess up regular processes.

       ```
       $ usertests -q
       usertests starting
       test copyin: OK
       test copyout: OK

       ...

       test sbrklast: OK
       test sbrk8000: OK
       test badarg: OK
       ALL TESTS PASSED
       $
       ```

4 Submission

- In the `xv6-riscv/` directory, execute the `tar.sh` script to create a tar file named after your student ID (e.g., `2023143535`). Upload the tar file (e.g., `2023143535.tar`) on LearnUs. Do not rename the file.

```
$ ./tar.sh
$ ls
2023143535.tar  kernel  LICENSE  Makefile  mkfs  README  tar.sh  user
```

5 Grading Rules

- The following is the general guideline for grading. A 40-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change; a grader may add a few extra rules without notice for a fair evaluation of students' efforts.

-5 points: The submitted tar file includes redundant tags such as a student name, `hw6`, etc.

-5 points: The code has insufficient comments. Comments in the skeleton code do not count. You must clearly explain what each part of your code does.

-5 points: Do not print unasked debugging messages.

-5 points each: `threadtests` has seven test cases. Each failed test loses 5 points. Additionally, failing `usertests` will lose 5 points.

-30 points: No or late submission.

Final grade = F: The submitted tar file is copied from someone else. All students involved in the incidents will get Fs for the final grade.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect, discuss your concerns with the TA. Always be courteous when contacting the TA. If no agreements are made between you and the TA, elevate the case to the instructor to review your assignment. Refer to the course website for the contact information of the TA and instructor: `https://icsl.yonsei.ac.kr/eee3535`
- Arguing for partial credits for no valid reasons will be regarded as a cheating attempt; such a student will lose the assignment scores.