

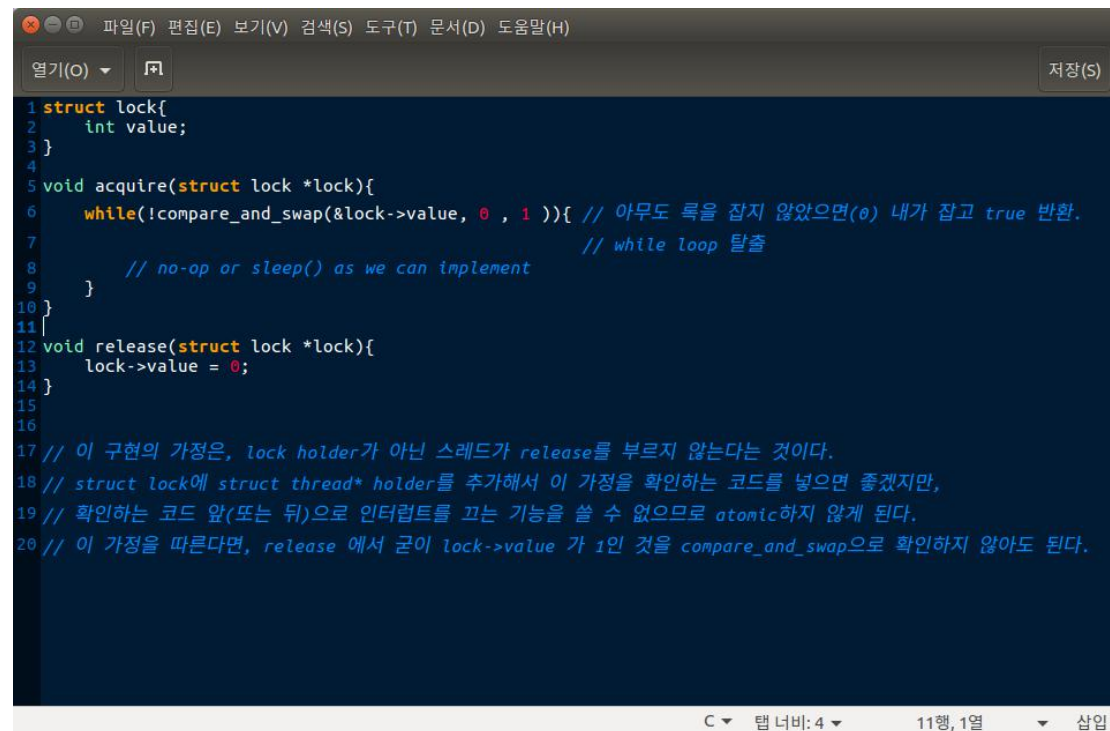
HW1 - Q1

```
파일(F) 편집(E) 보기(V) 검색(S) 도구(T) 문서(D) 도움말(H)
열기(O) 저장(S)

1 #
2 #
3 function customer():
4     if(sema_down(chairs)):      # 남은 자리가 있을 때만 케밥을 받는 루틴 수행가능
5                                 # 남은 자리를 볼 수 있는 고객은 항상 한 명 뿐이다.
6         signal(custReady)      # 고객이 들어오면 요리사는 깨어난다. 이미 깨어나 있으면 무시된다.
7         wait(cookReady)        # 케밥이 준비될 때 까지 기다린다.
8         sema_down(kebab)       # 만들어진 케밥 하나를 산다.
9         sema_up(chairs)
10
11     else:
12         leave_without_kebab()  # 남은 자리가 없으면 그냥 떠난다.
13
14
15
16 function cook():
17     while(True):
18         wait(custReady)        # 고객이 준비될 때 까지 잔다.
19         making_kebab()
20         sema_up(kebab)
21         signal(cookReady)
22
23
24
25
26 semaphore kebab:
27     int kebab
28
29 lock wakeup
30
31 semaphore chairs:  # number of remaining chair
32     int num_chairs # initialized to n
33
34 function sema_down(sema):  # atomic
35     if(sema.elem == 0):    # 남은 자리가 없으면 자리에 앉지 못한다.
36         return false
37     else:
38         sema.elem--        # 남은 자리가 있으면 자리에 앉는다.
39         return true
40
41 function sema_up(sema):    # atomic
42     sema.elem++            # 접근할 수 있다 치자.
43
44 function get_elem(sema):
45     return sema.elem

맞는 대괄호가 다음 줄에 있습니다: 19
파이썬 3 ▼ 탭 너비: 4 ▼ 19행, 23열 ▼ 삼입
```

HW1-Q2



```
1 struct lock{
2     int value;
3 }
4
5 void acquire(struct lock *lock){
6     while(!compare_and_swap(&lock->value, 0, 1)){ // 아무도 록을 잡지 않았으면(0) 내가 잡고 true 반환.
7                                                     // while loop 탈출
8         // no-op or sleep() as we can implement
9     }
10 }
11
12 void release(struct lock *lock){
13     lock->value = 0;
14 }
15
16
17 // 이 구현의 가정은, lock holder가 아닌 스레드가 release를 부르지 않는다는 것이다.
18 // struct lock에 struct thread* holder를 추가해서 이 가정을 확인하는 코드를 넣으면 좋겠지만,
19 // 확인하는 코드 앞(또는 뒤)으로 인터럽트를 끄는 기능을 쓸 수 없으므로 atomic하지 않게 된다.
20 // 이 가정을 따른다면, release 에서 굳이 lock->value 가 1인 것을 compare_and_swap으로 확인하지 않아도 된다.
```

HW2

*Assume an uniprocessor system

- Design a process system
- Show your major data structure
 - thread states: RUNNING, READY, BLOCKED, DYING
 - current_running_thread
 - ready_queue
 - contains all threads of all processes
 - init_thread
 - struct process_control_block{
 - pid
 - address space pointer
 - exit_code
 - parent
 - childs
 - TCBs (pointer list)}
- Define your functions
 - process_init()
 - initialize the process system. & initialize the thread system.
 - create a struct process for initial process & create a struct thread

loader puts the initial thread's stack at the top of a page
 process_start()
 create the first (idle) thread
 enable interrupts, which makes the scheduler enable
 process_create(name, priority, thread_func, aux)
 creates and starts a new thread(and process), and called init_thread
 init_thread must wait its child
 process_wait()
 wait its child which is created by process_create
 process_exit()
 free the current process's all resources
 timer_interrupt()
 round-robin style scheduler
 for every timer interrupt, current_running_thread will be placed at the back
 of the ready queue
 schedule()
 schedule a thread

□ **Show your major system calls**

halt()
 wait()
 exec()
 call process_create(), make the other new process
 file_create()
 file_remove()
 file_open()
 file_close()
 file_read()
 file_write()

• **Justification your design**

일단 스레드는 커널 레벨에서 제어한다.
 프로세스는 단순히 메모리와 리소스(파일 등) 할당의 단위이다.
 따라서 priority 등 실제 실행에 필요한 정보들은 TCB 에 넣었다.
 ready_queue 가 모든 준비 상태 스레드를 관리하며, 커널이 ready_queue 에서 스케줄한다.
 어떤 프로세스에 소속된 스레드인지는 신경쓰지 않는다.
 OS 가 시작되고 process_init()을 부른다. 함수의 설명은 위와 같다.
 프로세스 시스템이 초기화되고 process_start()를 통해 첫 프로세스와 그 프로세스에서 idle 스레드를 만든다.
 process_create()는 프로세스를 새로 생성하고 스레드를 만들어 thread_func 에 해당하는 함수를 실행한다.
 지정된 횟수 만큼(10ms~100ms) timer interrupt 가 일어나면 ready_queue 에서 round robin 방식으로 스케줄한다.

- **Design a thread system : kernel-level threads**

- **Show your major data structure**

```
struct thread_control_block {  
    PCB pointer  
    tid  
    status (RUNNING, READY, BLOCKED, DYING 중 하나)  
    name  
    SP (stack pointer)  
    exit_code  
    init_thread  
    parent  
    childs  
}
```

- **Define your functions**

```
thread_init()  
    thread system 을 초기화하고 첫 메인 스레드를 만든다. process_init() 에서  
    불려진다.  
thread_current()  
    return current thread  
thread_fork()  
    create a copy of current thread  
thread_wait()  
    wait its child thread created by thread_fork()  
thread_exit()  
    if there is the init_thread in this process, just make current thread's  
    status DYING  
    if there is no init_thread, means current thread is the init_thread, call  
    process_exit()  
thread_yield()  
    give up the CPU  
thread_block()  
thread_unblock()
```

- **Show your major system call**

각 시스템 콜들은 이름이 같은 thread_함수를 호출

```
thread_fork()  
thread_wait()  
thread_exit()
```

- **Justification your design**

init_thread 는 각 프로세스를 만들때 가장 처음 만들어진 프로세스이다.

이 프로세스에서 fork 와 wait 을 통해 init_thread 와 비슷한 일을 하는 스레드들
을 만든다.

thread_exit()에서는 init_thread 만 남았을 때 process 의 할당된 자원들을 free
하게 만든다.

시스템 콜로는 fork, wait, exit 이 있고 각각은 thread_fork, _wait, _exit 을 부

른다.

PCB 와 TCB 가 서로를 가리키는 포인터를 저장하고 있다.

스레드에서 자신의 프로세스의 address space 에 접근하게 할 수 있다.

PCB 는 SP 를 따로 가지지 않는다. 스레드들은 각자의 execution flow 를 가지므로, SP 를 TCB 에 저장한다.