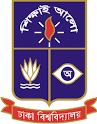
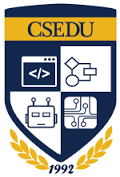
**University of Dhaka**

**Department of Computer Science and Engineering**

CSE-3211, Operating System Lab

3rd Year, 2nd Semester

**Session:** 2018-19

**Assignment No.** 1

**Title of the Assignment:** Practice Thread and Synchronisation(Question Answer)

**Lab Group**: A

**Submitted by:**

1. Meheraj Hossain (Roll - 24)

2. Rahat Rizvi Rahman (Roll - 37)

3. Md. Fahim (Roll - 26)

**Date:** 29th October, 2019

**Submitted to:**

1. Dr. Mosaddek Hossain Kamal Tushar, Professor, Dept. of CSE, DU
2. Dr. Md Mamun-or-Rashid, Professor, Dept. of CSE, DU

Questions & Answers

## **Thread Questions:**

**1. What happens to a thread when it exits (i.e., calls thread\_exit() )? What about when it sleeps?**

thread\_exit() - thread.c

* Decreases the counter of whatever vnode it may be pointing at.
* Virtual file system for the thread is cleared (VFS).
* Set priority level high - disables all interrupts. (splhigh())
* Puts itself into zombie state and and preps itself to panic if it ever runs again before it dies.

wchan\_sleep(struct wchan \*wc) -thread.c

* In sleep mode, it makes sure it’s not in an interrupt handler, yields control to the next thread, enters the S\_SLEEP state.
* Yield the cpu to another process, and go to sleep, on the specified wait channel WC.
* Only starts taking control once more when wakeup() is called on its address.

**2. What function(s) handle(s) a context switch?**

* thread\_switch() - thread.c

**3. How many thread states are there? What are they?**

thread.h

* S\_RUN - running
* S\_READY - ready to run
* S\_SLEEP - sleeping
* S\_ZOMBIE - zombie(exited but not yet deleted)

**4. What does it mean to turn interrupts off? How is this accomplished? Why is it important to turn off interrupts in the thread subsystem code?**

* Current working thread will not be interrupted until interrupts are re-enabled, meaning the code section between disabling and re-enabling interrupts will be executed as if atomically.
* Interrupts are turned off using the function splhigh (set priority level high) and back on again using spl0 (set priority level zero). The priority level can also be set to intermediate levels using the splx function.
* Turning off interrupts for thread operations is necessary to ensure that these operations complete successfully and aren’t broken mid-execution.

**5. What happens when a thread wakes up another thread? How does a sleeping thread get to run again?**

* Call to wchan\_wakeone (thread.c) removes a thread from the sleeping queue and wakes up the thread by putting it in CPU's runqueue. When thread\_switch happens, the woken up thread may run again.
* Calling thread\_make\_runnable (thread.c) will make a thread runnable again.
* A sleeping thread get to run again when wchan\_wakeone() function called on it.

## **Scheduler Questions:**

**6. What function is responsible for choosing the next thread to run?**

* thread\_switch() swaps in the next thread on the CPU's run queue.
* schedule() reshuffles the threads around on the run queue.
* thread\_consider\_migration() migrates threads across CPU's run queues in case of imbalance.

**7. How does that function pick the next thread?**

=> thread\_switch() simply picks the next thread on the run queue. schedule() uses a round-robin run queue that schedules each thread in the queue in equal time-slice without priorities.

**8. What role does the hardware timer play in scheduling? What hardware independent function is called on a timer interrupt?**

* The interrupt handler for the hardware timer calls hardclock, defined in src/kern/thread/clock.c. The method hardclock finishes by calling thread\_yield every time it is run, forcing a context switch.
* hardclock() calls both schedule() and thread\_consider\_migration() periodically.

## **Synchronisation Questions:**

**9. Describe how thread\_sleep() and thread\_wakeup() are used to implement semaphores. What is the purpose of the argument passed to thread\_sleep() ?**

* thread\_sleep is used in the P function of the semaphore. This function suspends the current thread until the semaphore count is greater than zero.
* thread\_wakeup() is used in the V function of the semaphore. This function wakes up all the suspended threads waiting on the current semaphore.
* The argument that is passed in is the address of the object (in this case, semaphore) the sleeping thread is associated with.This is required so that when thread\_wakeup is called on the same semaphore, it can selectively wake up only the threads associated with that particular semaphore.

**10. Why does the lock API in OS/161 provide lock\_do\_i\_hold() , but not lock\_get\_holder() ?**

* Because locks have to be released by the same thread that acquires them ( and thereby prevent malicious actions).

**Here are code samples for two threads that use binary semaphores:**

semaphore \*mutex, \*data;

void me() {

P(mutex);

/\* do something \*/

P(data);

/\* do something else \*/

V(mutex);

/\* clean up \*/

V(data);

}

void you() {

P(data)

P(mutex);

/\* do something \*/

V(data);

V(mutex);

}

**11. Give a sequence of execution and**

**context switches in which these two threads can deadlock.**

* me();

you();

* you();

me();

Both sequence of execution in which the given two threads can deadlock.

**12. Propose a change to one or both of them that makes deadlock impossible. What general principle do the**

**original threads violate that causes them to deadlock?**

* Changed methods that makes deadlock impossible**:**

void me() {

P(mutex);

/\* do something \*/

P(data);

/\* do something else \*/

V(data);

/\* clean up \*/

V(mutex);

}

void you() {

P(data)

P(mutex);

/\* do something \*/

V(mutex);

V(data);

}

The order of calling the P() and V() function for each semaphore must be maintained very carefully.

**Here are two more threads:**

lock \*file1, \*file2, \*mutex;

void laurel() {

lock\_acquire(mutex);

/\* do something \*/

lock\_acquire(file1);

/\* write to file 1 \*/

lock\_acquire(file2);

/\* write to file 2 \*/

lock\_release(file1);

lock\_release(mutex);

/\* do something \*/

lock\_acquire(file1);

/\* read from file 1 \*/

/\* write to file 2 \*/

lock\_release(file2);

lock\_release(file1);

}

void hardy() {

/\* do stuff \*/

lock\_acquire(file1);

/\* read from file 1 \*/

lock\_acquire(file2);

/\* write to file 2 \*/

lock\_release(file1);

lock\_release(file2);

lock\_acquire(mutex);

/\* do something \*/

lock\_acquire(file1);

/\* write to file 1 \*/

lock\_release(file1);

lock\_release(mutex);

}

**13. Can they deadlock? If so, give a concurrent execution in which they do and propose a change to one or both that makes them deadlock free.**

* Yes. They can deadlock.

A concurrent execution in which they dodeadlock:

laurel();

hardy();

**Proposed change to one or both that makes them deadlock free:**

lock \*file1, \*file2, \*mutex;

void laurel() {

lock\_acquire(mutex);

/\* do something \*/

lock\_acquire(file1);

/\* write to file 1 \*/

lock\_acquire(file2);

/\* write to file 2 \*/

lock\_release(file1);

lock\_release(mutex);

/\* do something \*/

lock\_acquire(file1);

/\* read from file 1 \*/

/\* write to file 2 \*/

lock\_release(file2);

lock\_release(file1);

}

void hardy() {

/\* do stuff \*/

lock\_acquire(file1);

/\* read from file 1 \*/

lock\_release(file1); -> These two lines are swapped

lock\_acquire(file2); -> These two lines are swapped

/\* write to file 2 \*/

lock\_release(file2);

lock\_acquire(mutex);

/\* do something \*/

lock\_acquire(file1);

/\* write to file 1 \*/

lock\_release(file1);

lock\_release(mutex);

}