UNIVERSITY OF TORONTO

FACULTY OF APPLIED SCIENCE AND ENGINEERING

MIDTERM EXAMINATION, Feb, 2016
Third Year – Materials
ECE344H1 - Operating Systems
Calculator Type: 2
Exam Type: A
Examiner – D. Yuan

Please Read All Questions Carefully! Please keep your answers as concise as possible. You do not need to fill the whole space provided for answers.

*There are **12** total numbered pages, **4** Questions.*
You have 2 hours. Budget your time carefully!

Please put your FULL NAME, UTORid, Student ID on THIS page only.

Name: _____

UTORid: _____

Student ID: _____

Question 1 (60 marks, 3 marks each): Multiple choices. There are one or more correct choices for each question. Mark all of them. For each question, you will get 3 marks only if your answer is 100% correct, otherwise you will get 0.

(1) Which of the followings are descriptions of an OS?
- (a) It is a piece of software
- (b) The Shell program is typically part of the OS
- (c) It has direct control over the hardware
- (d) None of the above

A,C

(2) What are the benefits of an OS to the *application* programmers?
- (a) Programmers do not need to worry about interference between different processes
- (b) It abstracts complex hardware into a set of clean interfaces
- (c) Programmers do not need to worry about interference between different threads
- (d) Running a program on top of an OS is typically faster than directly on the hardware

A,B

(3) Which of the following lists is ordered from the fastest to the slowest?
- (a) CPU cache, registers, main memory (RAM), magnetic disk
- (b) Registers, main memory (RAM), CPU cache, magnetic disk
- (c) Registers, CPU cache, main memory (RAM), magnetic disk
- (d) None of the above

C

(4) Why separate user mode and kernel mode?
- (a) So the OS cannot modify application's data
- (b) To prevent user applications from executing protected instructions
- (c) It is a performance optimization
- (d) None of the above

B,C

(5) Which of the followings are protected instructions?
- (a) test-and-set
- (b) splhigh()
- (c) modification to sp (stack pointer)
- (d) None of the above

B

(6) When a user application executes a divide-by-zero instruction, it causes a fault and the application is subsequently terminated. Which of the followings are true:
- (a) Right before this fault occurs, the CPU has to be in kernel mode
- (b) CPU, instead of the OS, is the first to detect that a divide-by-zero fault has occurred

(c) The CPU directly terminates the faulting application
(d) None of the above

B

(7) Which of the followings are true about timer interrupts?
(a) timer interrupt is mainly used to prevent the OS from using the CPU forever
(b) timer interrupt cannot be disabled by a user application
(c) timer interrupt cannot be disabled by the OS
(d) timer interrupt is used by the OS to show the time

B

(8) Consider the following code snippet as the implementation of lock_acquire. The semantic of test-and-set is the same as we discussed in the lecture.

```
0: void lock_acquire_new(lock) {
1:   while (1) {
2:     while (lock->held > 0)
3:       ; // spin
4:     if (test-and-set(&lock->held) == 0)
5:       return;
6:   }
7: }
```

Which of the followings are correct?
(a) This code correctly implements lock acquire
(b) This code is incorrect
(c) Compared to the spin lock implementation we discussed in lecture, a system running the same programs using "`lock_acquire_new`" will execute less test-and-set instructions
(d) Compared to the spin lock implementation we discussed in lecture, a system running the same programs using "`lock_acquire_new`" will execute more test-and-set instructions

A,C

(9) Which of the followings correctly describe the relationship between program and process?
(a) Process is a program in execution
(b) There is no difference between a process and a program
(c) One program might correspond to multiple process
(d) None of the above

A,C

(10) When you print the address of a variable in a C program with the following statement:

```
printf ("address: 0x%lx\n", (long) &var);
```

assume you execute your program in a process, which of the followings are true:
(a) If var is on the stack, then the printed address is physical address

(b) If var is a global static variable, then the printed address is physical address
(c) If var is on the heap, then the printed address is physical address
(d) None of the above

D

(11) Consider two PCBs, each corresponds to a process that are in ready state. Each PCB stores the value of PC (program counter), SP (stack pointer), and RAX (general purpose register). Which of the followings are true:

(a) It is possible that the two PCs have the same value
(b) It is possible that the two SPs have the same value
(c) It is possible that the two RAXs have the same value
(d) None of the above

ABC

(12) Consider the "`open (path, flags, mode)`" system call in FreeBSD as we discussed in the lecture:

```
1: open: ; FreeBSD convention:
2:    ; parameters via stacks.
3:   push dword mode
4:   push dword flags
5:   push dword path
6:   mov eax, 5
7:   push dword eax ; syscall number
8:   int 80h
9:   add esp, byte 16
```

Which of the followings are true?

(a) Line 3-8 will execute in kernel mode
(b) Line 9 is used to pop 4 entries off the stack
(c) It is possible that the total execution time, measured as wall clock time, for these 7 instructions (line 3 - 9) to be 7 cycles
(d) None of the above

B

(13) For this program:

```
void main() {
  fork();
  int pid = fork();
  if (pid == 0) {
    fork();
  }
  exit();
```

```
}
```
How many processes are created upon the execution of this program in a Shell (including the first process created by the Shell)?

(a) 2
(b) 4
(c) 6
(d) 8
(e) 10
(f) None of the above

C

(14) Which of the followings are true about exec()?

(a) exec() will create a new process
(b) exec() will never return under normal conditions
(c) exec() is a system call
(d) None of the above

BC

(15) Consider the following implementation of a shell terminal:

```
while (1) {
  char *cmd = read_command();
  int child_pid = fork();
  if (child_pid != 0) {
    Manipulate STDIN/OUT/ERR file descriptors for pipes, redirection, etc.
    exec(cmd);
    panic("exec failed");
  } else {
    waitpid(child_pid);
  }
}
```

Assume that waitpid() will do nothing and return immediately on a child_pid that does not exist or is not a child of the calling process. Now a user has opened one instance of this shell terminal, which of the followings are true?

(a) After the user enters "ls" once, the shell terminal will disappear entirely
(b) After the user enters "ls" once, there will be two shell terminals
(c) After the user enters "ls" once, there will be still exactly one shell terminal
(d) After the user enters "ls" once, there will be infinite number of shell terminals created

C

(16) Which of the followings are true about threads?

(a) Different threads share the same address space
(b) Different threads share the same heap

(c) Different threads share the same stack
(d) Different threads share the same PC

AB

(17) Which of the followings are true about the relationship between kernel- and user-level threads?

(a) Both of them are faster to create than creating a process
(b) When a kernel-level thread execute an I/O system call, the other kernel-threads within the same process are blocked as well
(c) Lock/unlock synchronizations for kernel-level threads are system calls (assume the lock/unlock are implemented using test-and-set)
(d) None of the above

A

(18) Which of the followings are true?

(a) OS161 uses disabling interrupts to implement synchronization primitives
(b) OS161 uses test-and-set to implement synchronization primitives
(c) test-and-set does not work on multi-core systems
(d) None of the above

A

(19) Consider the bank account withdraw example we discussed in lecture. Now suppose that you want to add a feature to prevent the account from going into negative. Your code looks like this:

```
// returns ‘amount’ of cash to the user
withdraw (account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance = balance – amount;
  if (balance < 0) {
    return 0;
  }
  put_balance(account, balance);
  release(lock);
  return amount;
}
```

Which of the followings are true about this code:

(a) This code works correctly
(b) This code breaks when the original balance < amount
(c) This code breaks when the original balance >= amount
(d) None of the above

B

(20) Which of the followings are true about thread_yield()?

(a) thread_yield() never returns
(b) thread_yield() puts the calling thread to sleep
(c) thread_yield() terminates the calling thread
(d) None of the above

D

Question 2 (16 marks, 4 marks each): OS161

Consider the following program snippet from OS161:

```
 1 /* High level, machine-independent context switch code.
 2  * curthread is a pointer to the thread data structure of
 3  *   of the currently executing thread */
 4 static void mi_switch(threadstate_t nextstate) {
 5   int spl = splhigh();
 6   if (curthread != NULL && curthread->t_stack != NULL) {
 7     assert(curthread->t_stack[0] == (char)0xae);
 8     assert(curthread->t_stack[1] == (char)0x11);
 9     assert(curthread->t_stack[2] == (char)0xda);
10     assert(curthread->t_stack[3] == (char)0x33);
11   }
12
13   if (nextstate==S_READY) {
14     result = q_addtail(runqueue, cur);
15   } else if (nextstate==S_SLEEP) {
16     result = array_add(sleepers, cur);
17   } else {
18     result = array_add(zombies, cur);
19   }
20
21   next = scheduler();
22   curthread = next;
23
24   /*
25    * Call the machine-dependent code that actually does the
26    * context switch.
27    */
28   md_switch(&cur->t_pcb, &next->t_pcb);
29   splx(spl);
30 }
```

Answer the following questions.

(a) What are line 5 and line 29 for? What if they are removed?
Disable/enable interrupts. No.

(b) Can we move line 13-19 to the location after line 28 (but before line 29)? Why?
No. Otherwise the current thread will be lost.

(c) What do you think is the purpose of line 6-11?
Sanity check. Prevent a stack overflow from further propagating.

(d) Why certain part of the context switch operation, i.e., md_switch() at line 28, has to be machine dependent? (Hint: mips_switch() is implemented in assembly instead of C. We walked through the code of mips_switch() in class. Think about why it has to be implemented in assembly.)
B/c it has to save the registers, and different CPUs have different registers. (You have to provide concrete evidences such as registers.)

Question 3 (12 marks): Atomic instruction

Besides test-and-set, another atomic instruction that is supported by some CPUs is fetch-and-increment. The pseudocode below illustrates how it works:

```
int fetch-and-increment (int* addr) {
  int old = *addr;
  *addr = old + 1;
  return old;
}
```

Basically it always increment the memory content's value by 1, and return the original value.

Similar to test-and-set, the underlying hardware will guarantee that these steps are performed atomically.

Now, given fetch-and-increment, how do you implement lock_acquire and lock_release? *Note the only hardware support you can only use is fetch-and-increment(). Your lock_acquire and release should only take one argument: the lock (but you can define your own lock data-structure). You can assume that the total number of threads is less than 1 million, and each thread won't call lock/unlock more than 100 times. Other than this, you cannot make any assumption on the number of threads. Both spin-lock and sleep-based semantics are acceptable.* Write your code below:

```
Solution:
struct lock {
  int ticket;
  int turn;
}

void lock_initialize(struct lock* l) {
  l->ticket = 0;
  l->turn = 0;
}

lock(struck lock* l) {
  myturn = f-and-i (l->ticket);
  while (myturn != l->turn) ;
}

unlock(struct lock* l) {
  f-and-i(l->turn);
}
```

Grading policy:

- If the solution causes integer overflow: -1
- Penalize unnecessarily complex solution

Question 4 (12 marks): Synchronization:

Consider how taxi hailing work at airport #344. Taxi drivers arrive at the airport in order. If there are customers waiting, the driver immediately picks up the one waiting customer who arrived the earliest. Otherwise the driver parks the car in a queue and cleans the car. Similarly, when each customer arrives, she checks if there are taxis waiting. If so, she gets into the taxi that arrived the earliest. Otherwise she waits in line and plays with the cell phone. Note that each taxi can only pick up one customer.

Your job is to write two functions, namely driver() and customer(). Each taxi driver and customer will be a separate thread executing driver() and customer() respectively. driver() should call thread_exit() when you picked up a customer, and customer() should call thread_exit() when you boarded a taxi. Using the synchronization primitives you learnt, your program should correctly meet all of the requirements above, and you cannot make any assumptions on the number of drivers, customers, CPU cores, or the behaviors of the scheduler.

Write your code below:

```
char driver_array[MAXLEN][MAXNAMELEN], customer_array[MAXLEN][MAXNAMELEN];
  // these two are circular arrays. ASSUME there won't be more than MAXLEN
```

```
  // waiting customers or drivers
lock clock, dlock; // protect the shared arrays
int driver_ticket = customer_ticket = -1;
// 2 condition variables
CV driver_cv, customer_cv;
```

```
void *driver(void *arg) {
  // Assume arg is the driver name
  char* dname = (char *) arg;
  char cname[MAXNAMELEN];
  int myticket;

  printf("driver %s arrived\n", dname);

  lock_acquire(dlock);
  driver_ticket++;
  myticket = driver_ticket;
  strcpy(driver_array[myticket % MAXLEN],
       dname);
  lock_release(dlock);

  cv_broadcast(driver_cv); // wake up all
    // waiting customers!

  lock_acquire(clock);
  while (customer_ticket < myticket)
    cv_wait(customer_cv, clock);
  // now, customer_ticket >= myticket,
  // and I am holding clock
  strcpy(cname,customer_array[myticket % MAXLEN]);
  lock_release(clock);

  printf("driver %s picked up %s\n",
         dname, cname);

  thread_exit();
}
```

```
void *customer(void *arg) {
  // Assume arg is the customer name
  char* cname = (char *) arg;
  char dname[MAXNAMELEN];
  int myticket;

  printf("customer %s arrived\n", cname);

  lock_acquire(clock);
  customer_ticket++;
  myticket = customer_ticket;
  strcpy(customer_array[myticket % MAXLEN],
       cname);
  lock_release(clock);

  cv_broadcast(customer_cv); // wake up all
    // waiting drivers!

  lock_acquire(dlock);
  while (driver_ticket < myticket)
    cv_wait(driver_cv, dlock);
  // now, driver_ticket >= myticket, and I am
  // the only thread holding dlock
  strcpy(dname, driver_array[myticket % MAXLEN]);
  lock_release(dlock);

  printf("customer %s is picked up by %s\n",
         cname, dname);

  thread_exit();
}
```

The above solution will get full mark.

Now a problem with this solution is that the Nth driver might be picking up the Nth customer before the (N-1)th driver picks up the (N-1)th customer if the Nth driver arrives between the (N-1)th driver released dlock and acquiring c-lock. To further fix this issue, we can have the improvement below. The idea is to use another counter to indicate the ticket number of the first driver/customer in line.

```
char driver_array[MAXLEN][MAXNAMELEN], customer_array[MAXLEN][MAXNAMELEN];
  // these two are circular arrays. ASSUME there won't be more than MAXLEN
  // waiting customers or drivers
lock clock, dlock; // protect the shared arrays
```

```
int driver_ticket = customer_ticket = -1;
int first_driver = first_customer = 0; // the ticket number of the
 // first waiting driver and customer in line. Note, we use clock to
 // protect driver_first, and dlock to protect customer_first.
// 2 condition variables
CV driver_cv, customer_cv;
```

```
void *driver(void *arg) {
  // Assume arg is the driver name
  char* dname = (char *) arg;
  char cname[MAXNAMELEN];
  int myticket;

  printf("driver %s arrived\n", dname);

  lock_acquire(dlock);
  driver_ticket++;
  myticket = driver_ticket;
  strcpy(driver_array[myticket % MAXLEN],
       dname);
  lock_release(dlock);

  cv_broadcast(driver_cv); // wake up all
    // waiting customers!

  lock_acquire(clock);
  while (customer_ticket < myticket
           || myticket != first_driver)
    cv_wait(customer_cv, clock);
  /*
   * now, customer_ticket >= myticket,
   * and myticket == first_driver (I'm the
   * first in line, and I am
   * the only thread holding clock
   */
  strcpy(cname,customer_array[myticket % MAXLEN]);
  first_driver++; // leaving queue, next up
  lock_release(clock);

  printf("driver %s picked up %s\n",
         dname, cname);

  thread_exit();
}
```

```
void *customer(void *arg) {
  // Assume arg is the customer name
  char* cname = (char *) arg;
  char dname[MAXNAMELEN];
  int myticket;

  printf("customer %s arrived\n", cname);

  lock_acquire(clock);
  customer_ticket++;
  myticket = customer_ticket;
  strcpy(customer_array[myticket % MAXLEN],
       cname);
  lock_release(clock);

  cv_broadcast(customer_cv); // wake up all
    // waiting drivers!

  lock_acquire(dlock);
  while (driver_ticket < myticket
           || myticket != first_customer)
    cv_wait(driver_cv, dlock);
  /*
   * now, driver_ticket >= myticket,
   * and myticket == first_customer (I'm the
   * first in line, and I am
   * the only thread holding dlock
   */
  strcpy(dname, driver_array[myticket % MAXLEN]);
  first_customer++; // leaving queue, next up
  lock_release(dlock);

  printf("customer %s is picked up by %s\n",
         cname, dname);

  thread_exit();
}
```

Grading policy:

- If the waiting semantic is correct: 8 points
- If the ordering between drivers/customers is correct, i.e., the Nth driver can only pick up the Nth customer, award another 4 points
- Using big global lock: -1
- Penalize unnecessarily complex solutions