CS2106 Operating Systems
2017/18 Semester 2
Mid-Term Test

"Deponite omnem spem, vos qui intratis."
- **Dante**

Commented [TKYC1]: Abandon hope, all ye who enter.

Instructions

1. SHADE AND WRITE your Student Number on your OCR Answer Sheets.
2. Use 2B pencils only. If you use HB pencils you may find difficulty in erasing mistakes.
3. Shade your answers on the OCR Answer Sheet only.
4. Shade your answers FULLY. The OCR reader may miss partial shading, and we will take it that you did not answer the question and you will receive 0 marks for that question.
5. If you wish to speak to an examiner, please raise your hand high and keep it raise till you are attended to.
6. Do not speak to each other. If you do you will be deemed as having cheated in this test and disciplinary action will be taken against you.
7. Do not borrow items from each other. If you do we will assume that you are communicating with each other and will deem that you have cheated in this test.
8. This test has 20 questions printed on 13 pages including this one.
9. This test is worth 20 marks and will last for 60 minutes.

Questions

1. Which ONE of the following statements is true about the onion model?

   a. Device drivers are closer to the hardware than the kernel because they have higher privilege levels.
   b. +Microcode on a CPU can let us write more compact code for the same equivalent program, than on a CPU without microcode.+
   c. One microcode instruction can be broken up into several machine instructions.
   d. The gcc compiler is part of the kernel.
   e. The outer layers of the onion model have higher priority than the inner layers.

2. Which of the following are system processes?

   i. ssh server
   ii. C compiler
   iii. Word processor
   iv. Print spooler
   v. PDF Reader

   a. **+ONLY i. and iv above are system processes**.+
   b. ONLY i, iii and v above are system processes.
   c. ONLY ii, iii and v above are system processes.
   d. ONLY i. and iii. Above are system processes.
   e. None of the options a. to d. above are correct.

3. Which of the following statement(s) is(are) true about monolithic and micro kernels?

   i. In monolithic kernels all components of the kernel (device drivers, file systems, etc.) are included at compile-time, and it is not possible to add/remove components dynamically.
   ii. In microkernels running device drivers in user space improves reliability of the OS.
   iii. A microkernel consists only of a scheduler running in kernel space, with memory management and everything else running in user space.
   iv. It is possible to mount new file systems in some monolithic kernels.
   v. Microkernels always run faster than monolithic kernels.

   a. +ONLY ii. And iv. above are true.+
   b. ONLY i., iv. and v. above are true.
   c. ONLY iii. and vi. above are true.
   d. ONLY ii. and iii. above are true.

Commented [TKYC2]: Not true; the onion diagram shows layers of a computer system from hardware up to application. The layers do not imply privilege levels. To say that microcode has more privileges than assembly code doesn't make any sense.

Commented [TKYC3]: TRUE: Microcode allows a single instruction to do the work of multiple instructions (e.g. in a string compare), making programs that use these instructions more compact.

Commented [TKYC4]: Note true: It's the opposite; one assembly instruction can be made up of multiple micro instructions.

Commented [TKYC5]: Not true: gcc is a system program but not part of the kernel.

Commented [TKYC6]: Again this is meaningless; an application program doesn't have higher priority than hardware.

Commented [TKYC7]: YES: Runs in the background to allow ssh connections

Commented [TKYC8]: NO: This is run by the user and not by the system to compile a program.

Commented [TKYC9]: NO: Run by user to write stuff like this.

Commented [TKYC10]: YES: Run by system to allow files to be printed on a shared printer.

Commented [TKYC11]: NO, run by user to read stimulating technical documents and the like.

Commented [TKYC12]: NO: LINUX is a counter-example where you can insert and remove modules dynamically.

Commented [TKYC13]: YES: Having device drivers run in kernel space means that the driver cannot corrupt OS memory causing crashes. The device driver itself can crash.

Commented [TKYC14]: NO: Key OS components continue to run in kernel space.

Commented [TKYC15]: YES: You can do this in LINUX

Commented [TKYC16]: NO: This ultimately depends on how the kernel is implemented and not whether it is a microkernel or a monolithic kernel.

4. System calls are often accomplished using software interrupts rather than function calls. What is the main advantage of using software interrupts over a standard function call to the OS's entry point?

    a. A software interrupt is inherently faster than a function call.
    b. +Using a software interrupt allows us to relocate the OS's entry point without causing existing software to stop working.+
    c. A software interrupt allows us to resume execution of the calling process more reliably than a function call.
    d. A software interrupt is slower than a function call.
    e. A software interrupt is more secure than a function call.

5. Key OS components run in kernel space while user programs and some parts of the OS run in user spaces. This is so that:

    a. There is a clear and obvious difference between the OS and user processes in terms of memory usage.
    b. We can increase execution time.
    c. We can increase execution speed.
    d. User code can run concurrently with OS code without interfering with each other.
    e. Damage caused by buggy or malicious code can be minimized.

6. An OS maintains a lot of information about processes, but this information DOES NOT INCLUDE (choose ONE):

    a. CPU register contents.
    b. Execution status.
    c. +Contents of variables used by the process.+
    d. Stack pointer information.
    e. Program counter information.

Commented [TKYC17]: NO: An interrupt works similar to a function call and would have similar execution times for the same code.

Commented [TKYC18]: YES: You can do this by changing the vector for the software interrupt to the new entry point.

Commented [TKYC19]: NO: Software interrupts and function calls use the same resumption mechanism: Stack pops. There's nothing to make it more reliable.

Commented [TKYC20]: Possibly, due to the need to consult a vector table, but this cannot possibly be an advantage.

Commented [TKYC21]: NO: A function call to privileged memory is more secure than a software interrupt vector to unprotected user memory. Software interrupts have nothing to do with safety.

Commented [TKYC22]: This is true, but not the reason why we separate kernel and user spaces.

Commented [TKYC23]: Again true: Switching privilege levels consumes many CPU cycles increasing execution time, but nobody sane would intentionally do this.

Commented [TKYC24]: Since b is true, this cannot be true.

Commented [TKYC25]: NO: The OS can certainly interfere with the user process, e.g. in pre-emption.

Commented [TKYC26]: YES and the main reason we have privilege levels; so a rogue program cannot destroy everything.

Commented [TKYC27]: YES in context switching we swap out CPU registers to the stack.

Commented [TKYC28]: YES the OS keeps track if the process is running, ready, etc.

Commented [TKYC29]: NO: Variables are kept in memory and are not tracked by the OS

Commented [TKYC30]: YES, to allow context restore.

Commented [TKYC31]: YES to allow resuming of a process.

7. The pseudocode below is an attempt to perform context switching. ISR(TIMER, ISR_NAKED) is a timer ISR that is called once per millisecond. It is specified as "ISR_NAKED" so that the compiler does not insert any additional code beyond what is given by the programmer. The current_process variable represents the process that is currently running.

   The time_limit field is initially set to the time quantum allocated to the process, and is decremented each millisecond. When it reaches zero a new process is picked for running. PROCESS_QUANTUM is some quantity in milliseconds that specifies how long a process gets to run. The final asm("iret") statement causes a return from interrupt.

```
ISR(TIMER, ISR_NAKED)
{
        if(current_process->time_limit>0)
                current_process->time_limit--;

        if(current_process->time_limit == 0)
        {
                current_process->time_limit = PROCESS_QUANTUM;
                save_context(current_process);
                current_process = pick_new_process();
                setupStackPointer(current_process);
                restore_context(current_process);

        }
        asm("iret");
}
```

   On testing it is found that this ISR does not switch between processes correctly. Why?

   a. Since we are using the same "current_process" variable in restore_context, we are restoring the context for the same process and not for the new process.
   b. Setting current_process->time_limit to PROCESS_QUANTUM causes the current process to re-run again instead of a new process.
   c. +The ISR corrupts the current process's context before saving it.+
   d. the ISR corrupts the next process's context after restoring it.
   e. Specifying "ISR_NAKED" suppresses code generated by the compiler that will guarantee correctness of this ISR.

8. Which ONE of the following statements is true about the process model (Page 8 of Lecture 2)?

   a. If we have a multi-core CPU, we do not need to do context switching.
   b. If we have multiple CPUs, we do not need to do context switching.
   c. The purpose of context switching is to provide for multiple program counters and CPU registers so that we can execute more than one process at a time.
   d. Context switching is relevant only to single-core single CPU systems.
   e. +The purpose of context switching is so that we can share one program counter and one set of CPU registers between processes.+

Commented [TKYC32]: This section of code causes CPU registers to change, damaging the context of the process being swapped out. When the process resumes it will use incorrect register values. Hence answer is c.

Commented [TKYC33]: NO, as stated in lecture you may have 160 processes running on 4 cores. Each core still needs context switching to manage 40 processes.

Commented [TKYC34]: NO: Only if each CPU runs only one process. Not necessarily true.

Commented [TKYC35]: NO: You will still have just one set of PC and CPU registers.

Commented [TKYC36]: NO: See explanatiom for a.

Commented [TKYC37]: YES: By swapping around register and PC contents we can share a single set of registers and PC with multiple processes.

9. In the following C code, the programmer intends for the child to send the parent a message, after-which both parent and child will exit:

```
.. other code ..
int fd[2];
pipe(fd);
if(fork() == 0)
{
      char *message[]="Hello parent!";
      // strlen(str) returns the length of str excluding
      // the end '\0'.
      write(fd[1], message, strlen(message)+1);
      close(fd[1]);
}
else
{
      char buffer[1024];
      int len;
      do
      {
            // read returns 0 on end-of-file
            len = read(fd[0], buffer, 1024);
            printf("Message from child: %s", buffer);
      } while(len>0);
      close(fd[0]);
}

.. other code ..
```

This code fails to execute correctly. Why?

a. +The do..while statement in the parent never exits.+
b. There is no wait(.) statement, causing this program to fail.
c. The parent and child processes do not share memory, and hence the parent and child cannot use fd to access the pipe.
d. The programmer mistakenly sends the message in the parent instead of the child.
e. The write statement should use strlen(message) instead of strlen(message)+1.

Commented [TKYC38]: Since the parent doesn't close the writing end, it cannot detect EOF when the child closes its end, hence len will never be 0. In fact the read statement will just simply block and never return since the child will not send anymore data.

10. Which ONE of the following statements is FALSE?

a. If we repeatedly call fork(.) and never call wait(.), we can potentially run out of space in the process table.
b. If we repeatedly call pthread_create and never call pthread_join, we can potentially run out of space in the thread table.
c. +If we repeatedly calling pthread_detach immediately after pthread_create, we can potentially run out of space in the thread table.+
d. pthread_exit can be used to pass results to the parent thread via pthread_join.
e. exit can be used together with wait for the child process to pass a result to the parent process.

Commented [TKYC39]: YES because of zombie processes.

Commented [TKYC40]: YES, analogous to part a.

Commented [TKYC41]: NO: The whole point of calling pthread_detach is to tell the OS to re-use resources including thread table entry.

Commented [TKYC42]: YES:

Commented [TKYC43]: YES again. The child can use the status argument in the exit call to pass values back to the wait.

11. Which ONE of the following statements about scheduling is FALSE?

    a. The average waiting time for processes in a shortest-job-first (SJF) scheduler is lower than in first-come-first-served (FCFS).
    b. +The total running time for processes in SJF is lower than in FCFS.+
    c. In a fixed priority system we generally choose a priority level of 0 for the highest priority process as a matter of convenience.
    d. Earliest deadline first (EDF) scheduling can potentially allow for more processes to run in a hard-real-time system than rate-monotonic scheduling (RMS).
    e. RMS is computationally less expensive than EDF.

Commented [TKYC44]: Yes. This is the whole point to SJF

Commented [TKYC45]: No, total running time stays the same.

Commented [TKYC46]: Yes, this is because there is no smaller positive integer than 0, so when we see a process with priority 0 we know it has the highest priority.

Commented [TKYC47]: YES and in fact the reason EDF exists.

Commented [TKYC48]: YES because EDF requires priority queues to be re-sorted as processes approach their deadlines.

For questions 12 and 13 we are given the following list of processes:

Task	Pi	Ci
T1	10	1
T2	8	2
T3	6	2
T4	5	3
T5	4	2

We have the following sets of processes:

Set 1: {T1, T2, T3}
Set 2: {T3, T4, T5}
Set 3: {T1, T2, T4}
Set 4: {T2, T3, T4}

Commented [TKYC49]: 1/10+2/8+2/6 = 0.683 < 1. Schedulable.

Commented [TKYC50]: 2/6+3/5+2/4 = 1.43 > 1 Not schedulable

Commented [TKYC51]: 1/10+2/8+3/5 = 0.95 < 1 Schedulable

Commented [TKYC52]: 2/8+2/6+3/5 =1.183 > 1 Not schedulable.

12. Which sets of processes above are schedulable under EDF?

    a. ONLY sets 1 and 4 are schedulable under EDF.
    b. ONLY sets 2, 3 and 4 are schedulable under EDF.
    c. +ONLY sets 1 and 3 are schedulable under EDF.+
    d. ONLY set 4 is schedulable under EDF.
    e. ONLY set 1 is schedulable under EDF.

13. Which of the following sets of processes is schedulable under RMS?

    a. ONLY sets 1 and 4 are schedulable under RMS.
    b. ONLY sets 2, 3 and 4 are schedulable under RMS.
    c. ONLY sets 1 and 3 are schedulable under RMS.
    d. ONLY set 4 is schedulable under RMS.
    e. +ONLY set 1 is schedulable under RMS.+

Commented [TKYC53]: Sets 2 and 4 are not schedulable under EDF and would not be schedulable under RMS. We can use CIA to test sets 1 and 3. Only set 1 is schedulable under RMS.

The following program is running on a round-robin co-operative multitasker. In the correct execution of this program Process_1 computes some value and writes to variable y in SHARED MEMORY, and Process_2 takes the y value and does some computation, printing out the result. Aside from OSYield(), any sort of system call to the OS automatically causes the process to surrender control of the CPU. PROCESS_ID_0 has an id of 0, PROCESS_ID_1 has an id of 1, etc. The round-robin scheduler starts by picking the process with id of 0, then 1, etc.

```
int x, y;

void Process_1()
{
	while(1)
	{
		x = getValue(); // Gets a value from somewhere important.
		y = 2 * x;
		OSYield();
	}
}

void Process_2()
{
	while(1)
	{
		int z = computeValue(y);
		printf("The result is %d\n", z);
	}
}

int main()
{
	// After launching processes, main() runs with
	// PROCESS_ID_2.
	launchProcess(Process_P1, PROCESS_ID_0);
	launchProcess(Process_P2, PROCESS_ID_1);
	while(1)
		OSYield(); // Always yield.
}
```

Refer to this program to answer questions 14 and 15.

14. Which ONE of the following statements is true?

a. This program works correctly as expected because the while(1) loop in main guarantees that the program does not exit, allowing Process_1 and Process_2 to "do their thing".
b. This program doesn't work correctly because the while(1) loop in main prevents Process_A and Process_B from running.
c. +This program works correctly because Process_1 will always run and does not yield control to Process_2 until y is properly computed.+
d. This program doesn't work correctly because Process_2 may run before Process_1 resulting in Process_2 getting a stale value of y.
e. This program works main() runs at a low priority.

Commented [TKYC54]: True, but doesn't guarantee correctness.

Commented [TKYC55]: FALSE: Main is the last to run and always yields control in any case.

Commented [TKYC56]: YES: P1 runs first since it has process ID of 0, and it doesn't yield until it has computed Y. Only then will P2 run.

Commented [TKYC57]: NO process_2 has an ID of 1, and will run after process_1

Commented [TKYC58]: NO This is meaningless and irrelevant.

15. A programmer on your team thinks that the x variable is getting the wrong value and modifies Process_1 as shown below, adding a printf to do some debug printing:

```
void Process_1()
{
      while(1)
      {
            x = getValue(); // Gets a value from somewhere important.
            printf("The x value we got is %d\n", x);
            y = 2 * x;
            OSYield();
      }
}
```

What will now happen to our program?

a. This modification will print out the value of x and continue to work as before.
b. +This modification will result in Process_2 getting the wrong value of y.+
c. This modification will cause main() to be executed immediately.
d. This modification has no effect at all.
e. None of the options a. to d. above are correct.

Commented [TKYC59]: This statement triggers an OS call which will cause the process to yield control, resulting in P2 executing and getting an old value of y. Hence b. is correct.

16. Who is the Dean of the NUS School of Computing?

a. +Professor Mohan Kankanhalli+
b. Colin Tan
c. Donald John Trump
d. Kim Jong Un
e. Jorge Mario Bergoglio (aka Pope Francis)

Commented [TKYC60]: If you get this question wrong, consider a career in cleaning sewers.

17. We have the following 4 processes that have access to 3 shared binary semaphores s1, s2 and s3, initially set to 1. The read() and write() operations are non-blocking asynchronous calls (i.e. they exit immediately after being called regardless of whether there is a corresponding write/read). The "sem_pend" operation decrements the given semaphore while the "sem_post" operation increments it.

P1:
while(1):
sem_pend(&s1);
sem_post(&s2);
write(f1);
read(f2);
send_post(&s1);
sem_pend(&s2);
	P2:
	while(1)
	sem_pend(&s1);
	sem_pend(&s2);
	read(f2);
	write(f1);
	sem_post(&s2);
	sem_post(&s1);

P3:
while(1)
sem_pend(&s2);
sem_pend(&s1);
read(f1);
write(f2);
sem_post(&s1);
sem_post(&s2);
	P4:
	while(1):
	sem_pend(&s1);
	sem_pend(&s2);
	read(f1);
	write(f2);
	sem_post(&s2);
	sem_post(&s1);

Which of the following pairs of processes may cause deadlock?

a. P1 and P2
b. P1 and P4
c. +P1 and P3+
d. P2 and P4
e. None of the above pairs in a. to d. may cause deadlocks.

Commented [TKYC61]: NO: The pend on s1 at the start of P1 and P2 will cause one of them to be blocked, and the other to execute to completion, where it releases s1.

Commented [TKYC62]: No. Same as a.

Commented [TKYC63]: YES. P1 takes s1 and P3 takes s2. When P1 tries to take s2 it will block and when P3 tries to take s2 it will block. Deadlock.

Commented [TKYC64]: NO: See a.

Commented [TKYC65]: Since c is true, this cannot be true.

18. This is Peterson's Solution, as discussed in the tutorial:

```
#define FALSE  0
#define TRUE   1
#define N      2                    /* number of processes */

int turn;                           /* whose turn is it? */
int interested[N];                  /* all values initially 0 (FALSE) */

void enter_region(int process);     /* process is 0 or 1 */
{
    int other;                      /* number of the other process */

    other = 1 – process;            /* the opposite of process */
    interested[process] = TRUE;     /* show that you are interested */
    turn = process;                 /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)      /* process: who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region */
}
```

What is the purpose of the "turn" variable?

a. It enforces atomicity when both interested[0] and interested[1] are FALSE.
b. It enforces atomicity when both interested[0] and interested[1] are TRUE.
c. It prevents deadlock when both interested[0] and interested[1] are FALSE.
d. +It prevents deadlock when both interested[0] and interested[1] are TRUE.+
e. It speeds up execution of Peterson's Solution.

Commented [TKYC66]: Atomicity is enforced by the check on interested[] and not on turn.

Commented [TKYC67]: They cannot both be false if any process is trying to enter the CS. Hence this option cannot occur.

Commented [TKYC68]: Covered in tutorial 5

Commented [TKYC69]: Not unless you think inserting extra code to test an extra variable somehow makes your program faster.

19. The following code shows an implementation of barriers using semaphores.

```
#define MAX_PROCESSES         20

typedef struct
{
      int numProcesses; // Number of processes we are expecting
                        //to call the barrier.
      int procCount; // # of processes who have called the barrier
      mutex_t mutex;  // A mutex
      sema_t sema[MAX_PROCESSES];
} TBarrier;

// Initializes barrier to expect numProcesses processes to call it.
void initBarrier(TBarrier *barrier, int numProcesses)
{
      barrier->numProcesses = numProcesses;
      barrier->procCount = 0;
      initialize_mutex(&barrier->mutex);

      int i;

      // Initialize all semaphores to 0.
      for(i=0; i<numProcesses; i++)
            initialize_sema(&barrier->sema[i], 0);
}

// A process will call this with its process number when
// it reaches the barrier.
void reachBarrer(TBarrier *barrier, int procNum)
{
      mutex_lock(&barrier->mutex);
      barrier->procCount++;
      mutex_unlock(&barrier->mutex);

      if(barrier->procCount == barrier->numProcesses)
            sem_post(&barrier->sema[procNum]; // STATEMENT A
      else
            sem_pend(&barrier->sema[procNum]; // STATEMENT B

      if(procNum > 0)
            sem_post(&barrier->sema[procNum-1]; // STATEMENT C

}
```

Which ONE of the following statements is correct about this implementation of barriers using semaphores?

a. This implementation is correct.
b. This implementation is incorrect because of race conditions on barrier->procCount.

Commented [TKYC70]: This causes the current process to unblock. It was never blocked in the first place so this doesn't actually have any effect.

Commented [TKYC71]: This part causes processes from 0 to procNum-1 to unblock. Note that the unblocking cascades backwards. I.e. procNum unblocks procNum-1, which unblocks procNum-2, etc to 0.

However there's nothing to unblock procNum+1, procNum+2 … numProcesses-1. Hence e. is correct.

Commented [TKYC72]: No, see explanation above.

Commented [TKYC73]: Protected by mutex, so no race condition.

c. This implementation is incorrect because STATEMENT A and STATEMENT B should be swapped.

d. This implementation is incorrect because Statement C should use sem_pend instead of sem_post

e. +This implementation is incorrect because it will only release processes 0 to procNum, while processes procNum to numProcesses-1 remain blocked.+

Commented [TKYC74]: This will cause processes to pass through the barrier UNLESS every process is there. Opposite of what we want.

Commented [TKYC75]: This will cause each process to block instead of passing every process through.

20. The following is an attempt to solve the sleep/wake issue in the producer/consumer problem by using mutexes (hey if it works for monitors, it must work here right? Right??). The producer and consumer functions are called by producer and consumer processes that loop continuously. The consumer function's code is omitted below but uses mutexes in the same way as in producer, but for the consumer code instead.

```
#define QUEUE_LENGTH    16

… Declaration of queue and other shared variables …

void producer()
{
      lock_mutex(&mutex);
      int x = produce();
      if(count == QUEUE_LENGTH)
      {
            unlock_mutex(&mutex);
            sleep();
      }
      else
      {
            insert_item();
            count=count+1;
            if(count == 1)
            {
                  unlock_mutex();
                  wake();
            }
      }
}

void consumer()
{
      … Code similar in principle to producer, omitted for brevity ..
}

… Other code to initialize count, mutex, etc …
```

Commented [TKYC76]: This code fails if the producer gets pre-empted between the unlock and the sleep, and the consumer sends a wake(producer) which is lost. When producer resumes it will go to sleep. Hence b is true.

Which ONE of the following statements is true?

a. This implementation works and solves the sleep/wake issue in the producer consumer problem because the mutex is released only after a wake has been sent.
b. +This implementation does not work because releasing the mutex can still cause wakes to be lost.+
c. This implementation works because the mutex prevents wakes from being lost.
d. This implementation does not work because releasing the mutex causes incorrect updating of count.
e. I have no idea whether or not this code works. Like, seriously. It's the last question and I no longer care.

Commented [TKYC77]: No, if so the mutex will never be released because the process has gone to sleep.

Commented [TKYC78]: NO: See explanation above.

Commented [TKYC79]: NO: The mutex release is after count is updated.

Commented [TKYC80]: Maybe for some of you, but not the best answer.

~~ END OF PAPER ~~