University of Waterloo
CS350 Midterm Examination Model Solution

Winter 2016

1. (26 total marks)

a. (2 marks)
Is it possible for a thread's userspace stack to contain a switch frame? If yes, write "YES" and identify - clearly and briefly - a situation in which this could occur. If not, write "NO" and explain why this cannot happen.

NO. A switch frame is saved as part of a context switch. Context switches between threads are performed inside the kernel. Switch frames are therefore saved on a thread's kernel stack. See slides 40 and 41 in the Processes and the Kernel section of the notes.

b. (4 marks)
In OS161's `fork` system call, the kernel must return a ENOMEM error code to the user process if there is insufficient memory to create a new process. Explain in detail the steps required to return ENOMEM to the user process. Include in your answer the steps required for the user process to determine that an error has occured and to retrieve the error code value.

1. sys_fork() returns ENOMEM to syscall()
2. syscall sets tf_a3 to 1, and tf_v0 to ENOMEM. These values are loaded into the a3 and v0 registers when the process goes back to userspace.
3. The fork userspace library checks the value of a3. If it is not 0, then it copies v0 to errno (a global variable), and sets v0 to -1 before returning.
4. The fork call returns a -1 (because v0 was set to -1). The user program checks the return value. If it is -1, it retrieves the error code by reading from errno.

c. (4 marks)
Briefly describe two of the methods (discussed in class) an operating system could use to prevent deadlocks. Describe the main disadvantage of each method.

See slide 49 in the Synchronization section of the notes.

1. No Hold and Wait: Can lead to livelocks.
2. Preemption: May require killing/restarting other threads.
3. Resource Ordering: Threads may have to acquire locks long before they need to access the shared variables that the locks are protecting. This can reduce concurrency.

d. (4 marks)

There are two types of internal events (events generated by the user process itself) that may cause control to transfer from a running user program to the kernel. Name these types of events and briefly describe each one.

1. Exception: initiated when a process generates some sort of error (if the error is fatal, process is terminated)
2. System call: process requests a service from OS

e. (2 marks)

There is one type of external event that may cause control to transfer from a running user program to the kernel. Name and describe that type of event.

Interrupt: hardware generated event requiring an attention of OS (timer interrupt, I/O interrupt,)

f. (2 marks)

Which of the following is shared between threads of the same process? Circle the correct answer(s).

(a) Register values

(b) Local variables

(c) Global variables

(d) Heap memory

(c) and (d)

g. (2 marks)

What is the consequence of having the same frame number stored in more than one page table?

Violates isolation between processes. Two processes can now read/write to the same frame.

h. (2 marks)

In a system with paging, what are the advantages and disadvantages of using a large page size compared to a small page size?

- adv.: Smaller page tables, large TLB footprint, more efficient I/O
- dis.: Greater internal fragmentation, increased chance of paging in unnecessary data or code

i. (2 marks)

What are the advantages and disadvantages of software handled TLB misses compared to hardware handled TLB misses.

- adv.: simpler MMU; flexibility of Page Table implementation (can choose structure, format, lookup strategy without constraints); flexibility of TLB entries replacement (can choose replacement policy)
- dis.: Handling of TLB miss is much slower in this case (requires 2 context switches and cycles used by exception handler to handle the miss)

j. (2 marks)

Suppose a TLB contains the following mappings (in hexadecimal representation):

page num	frame num	valid bit
0x0302	0x0007	1
0x1440	0x0031	0
0x208F	0x0014	1
0x20EE	0xFF01	1

Assuming a page size of 4096 bytes, which of the following virtual address will be translated using the TLB, and not require an access to the page table? Indicate answers with HIT or MISS.

```
0x14400A2
0x0302F70
0x0401128
0x208FEE4
```

```
miss  (valid bit not set)
hit   0x0007F70
miss  (no entry in the TLB)
hit   0x0014EE4
```

2. (6 marks)

Consider the application below, which uses the `fork`, `_exit`and `waitpid` system calls.

```
1  int main() {
2      int i, status, rc, total = 0;
3      for (i = 0; i < 10; ++i) {
4         rc = fork();
5         if (rc) {
6             waitpid(rc, &status, 0);
7             total += WEXITSTATUS(status);
8         } else {
9             total = -1;
10            rc = fork();
11            if (rc == 0) {
12                _exit(2);
13            } else {
14                _exit(1);
15            }
16        }
17     }
18     printf("%d\n", total);
19     return 0;
20 }
```

a. (2 marks)
How many processes, in total, will be created when this program is run, including the original parent process?

21 (2 process are created in each iteration. 1 original parent process)

b. (2 marks)
Show the output that will be produced when this program runs. Note that the statement:

```
printf("%d", total);
```

will print the value of integer variable `total`.

10

c. (2 marks)
Suppose that line 10 is removed from the program. Show the output that will be produced when this modified program runs.

20

3. (8 total marks)

Barrier synchronization can be used to prevent threads from proceeding until a specified number of threads have reached the barrier. Threads reaching the barrier block until the last of the specified number of threads has reached the barrier, at which point all threads can proceed. Barriers are commonly used in parallel applications where computation consists of a series of stages, and the work in each stage is divided among N threads. In these applications, each thread completes its work independently, waits until all N threads have reached the barrier indicating that they have all completed their work, exchanges results with the other threads, before starting the next computation stage. Below is a partial pseudocode example of how barrier synchronization might be used.

```
/* Used to wait for all compute threads to finish their computation for
   the current stage. */
struct barrier *compute_barrier;
/* Used to notify the main thread that all of the compute threads have
   finished their computation. */
struct barrier *finish_barrier;

main() {
  unsigned int i;
  compute_barrier = barrier_create(NUM_COMPUTE);
  finish_barrier = barrier_create(NUM_COMPUTE+1);

  for (i=0; i < NUM_COMPUTE; i++) {
    thread_fork("Compute Thread", compute, NULL, i);
  }

  /* Wait here until all threads have finished their computation. The main
     thread can then aggregate their results. */
  barrier_wait(finish_barrier);
  aggregate_results();
}

void compute(void *unused, unsigned long compute_num) {
   unsigned int i;
   /* Perform computation one stage at a time on a portion of the data. */
   for (i=0; i < COMPUTE_STAGES; i++) {
     perform_computation(compute_num);
     /* Wait until all compute threads have completed the current stage */
     barrier_wait(compute_barrier);
     exchange_results();
   }
   /* Wait here until all threads have finished their computation. */
   barrier_wait(finish_barrier);
}
```

Explain at the bottom of the page why the following naive barrier implementation based on semaphores is incorrect. Use the line numbers in `barrier_wait` to identify the point in the code where a context-switch can lead to errors in a program that uses this barrier implementation. Describe a scenario (list of steps) that illustrates your explanation of the problem.

```
/* To simplify the code, assume all calls to kmalloc succeed */
struct barrier {
    volatile unsigned int b_threads_reached;
    unsigned int b_threads_expected;
    struct semaphore* mutex;
    struct semaphore* reached_barrier;
};

struct barrier *barrier_create(unsigned int thread_count) {
    struct barrier *b = (struct barrier *) kmalloc(sizeof(struct barrier));
    b->b_threads_expected = thread_count;
    b->b_threads_reached = 0;
    b->mutex = sem_create("mutex", 1);
    b->reached_barrier = sem_create("reached", 0);
}

1  void barrier_wait(struct barrier *b) {
2     unsigned int i;
3     P(b->mutex);
4     if (++(b->b_threads_reached) < b->b_threads_expected) {
5         V(b->mutex);
6         P(b->reached_barrier);
7     } else {
8         b->b_threads_reached = 0;
9         V(b->mutex);
10        for (i = 1; i < b->b_threads_expected; ++i) {
11            V(b->reached_barrier);
12        }
13    }
14 }
```

Consider the case where N = 2. Both T_1 and T_2 (representing threads) start at the first computation stage. T_1 calls barrier_wait, increments b_threads_reached, and context switches just before line 6. T_2 arrives, enters the else block in barrier_wait, resets the b_threads_reached to 0, and then calls `V(b->reached_barrier)`, which increments the semaphore to 1. It then continues running the second computation stage, and may eventually reach barrier_wait again before T_1 gets to run. On this call to barrier_wait, T_2 increments b_threads_reached to 1, enters the if block, but instead of blocking on `P(b->reached_barrier)`, it passes right through because the semaphore has a value of 1. T_2 can therefore enter the third computation stage before T_1 has even entered the second computation stage. This example illustrates the problem with the provided barrier implementation.

4. (10 total marks)

a. (3 marks)

Suppose a system uses a single-level page table. Suppose further, that the system uses a 46-bit virtual address and a 32-bit physical address. If the page size is 8KB (2^{13} bytes), how many entries will the largest page table have? How many frames will this page table occupy if each page table entry size is 4 bytes? You may express the answers as a power of 2. Is the decision to have a single level page table a reasonable one? Explain.

Number of entries: 2^{33}
Size of the page table: 2^{35} bytes or 2^{22} frames.
Single level page is unreasonable because a maximum size page table is not even going to fit into the RAM (physical address is 32 bit wide, page table requires 2^{35} bytes).

b. (3 marks)

If you are to use a multi-level page table for the above system, how many levels would you introduce and why? Describe the format of a virtual address showing the size of each field under assumption that every sub-table of multi-level page table fits into one frame. You can assume each page table entry is 4 bytes in size.

3-level page table. 11 bits for each level. Each table has at most 2^{11} entries, and since each entry is 4 bytes, the table can be at most 2^{13} bytes in size. This fits within a single frame.

c. (4 marks)

Given a process that requires 32KB for code segment, 10KB for data segment and 128KB for stack segment, what is the minimum and maximum number of frames that will be occupied by your multi-level page table? Assume each page table entry is 4 bytes in size, and segments must start at the beginning of a page but can otherwise be placed anywhere in the virtual address space.

If all 3 segments are next to each other, then the process would only need 3 page tables (one for each level), and therefore only requires 3 frames. If the 3 segments are separated, and each segment spans two entries in the first level page table, then the process would need 6 second level page tables, and 6 third level pages tables for a total of 13 page tables that occupy 13 frames.

5. (12 total marks)

You have been hired by the city of Waterloo to help solve a modified version of the "traffic intersection" problem. In this problem, vehicles are trying to pass through an intersection of two **one-way roads**, one north-to-south, the other east-to-west, without colliding. Each vehicle arrives at the intersection from one of two directions (north or east), called its origin. It is trying to pass through the intersection and exit in the opposite direction of its origin, called its destination. Turns are not allowed in this modified version of the problem.

In the original traffic intersection problem, more than one vehicle can be inside the intersection at the same time as long as their paths will not result in a collision. However, because of on-going construction causing structual weaknesses in the intersection, **only a maximum of 3 cars can be inside the intersection at the same time** in this modified version of the problem. Implement the following four functions. Global variables can be defined in the provided space. Your solution should be both fair and efficient.

```
// Define your global variables here.
#define MAX_CARS_IN_INTERSECTION 3
enum Directions {
    north = 0, east = 1
};
typedef enum Directions Direction;

// Volatile is not necessary because of the use of locks and condition variables
// to provide mutual exclusion and synchronization.
struct lock* l;
struct cv* direction_cv[2];
int num_waiting[2];
int num_inside;
Direction inter_direction; // Direction inside the intersection.


Direction opposing_direction(Direction d) {
    return (d == north) ? east : north;
}

// Called only once, before any vehicles try to enter the intersection.
void intersection_sync_init(void)  {
    l = lock_create("lock");
    direction_cv[north] = cv_create("North/South CV");
    direction_cv[east] = cv_create("East/West CV");
    num_waiting[north] = 0;
    num_waiting[east] = 0;
    num_inside = 0;
    inter_direction = north; // Arbitrary starting direction.
}


// Called only once, at the end of the simulation.
void intersection_sync_cleanup(void) {
    cv_destroy(direction_cv[north]);
    cv_destroy(direction_cv[east]);
    direction_cv[north] = NULL;
    direction_cv[east] = NULL;
    lock_destroy(l);
    l = NULL;
}


// Called by a vehicle simulation before a vehicle enters the intersection.
void intersection_before_entry(Direction origin) {
    lock_acquire(l);
    // Car is waiting (at least momentarily) at the intersection entrance.
    num_waiting[origin]++;
    // Check if there is at least one car waiting in the opposing direction. If
    // there is, then this car, which is just arriving at the intersection,
    // should wait in order to provide temporal ordering. Note that a while
    // loop is not necessary for this cv_wait since it is only used to modify
    // the ordering of cars, and does not affect the safety of the
    // intersection.
    if (num_waiting[opposing_direction(origin)] > 0) {
        cv_wait(direction_cv[origin], l);
    } else if (num_inside == 0) {
        // Intersection is empty, and no one is waiting in the opposing
        // direction. Change the direction of the intersection to allow
        // this car to (possibly) go inside the intersection.
        inter_direction = origin;
    }
    // Wait if the number of cars inside the intersection >= the
    // max number of cars. Also wait if the direction inside the
    // intersection is opposing this car's direction. Note that
    // barging is possible with this solution, but it is unlikely
    // to have a large effect on fairness in practice.
    while (num_inside >= MAX_CARS_IN_INTERSECTION || inter_direction != origin) {
       cv_wait(direction_cv[origin], l);
    }
    num_waiting[origin]--;
    num_inside++;
    lock_release(l);
}

// Called by a vehicle simulation after a vehicle leaves the intersection.
void intersection_after_exit(Direction origin) {
   lock_acquire(l);
   num_inside--;
   // Determining the opposing direction.
   Direction od = opposing_direction(origin);
   if (num_inside == 0) {
      // If the intersection is empty, and there are cars waiting
      // in the opposing direction, change the direction of traffic.
      if (num_waiting[od] > 0)
          inter_direction = od;
      }
      // Broadcast to wakeup the selected direction. A more efficient
      // solution would be to just call cv_signal
      // min(num_waiting, MAX_CARS_IN_INTERSECTION) number of times.
      cv_broadcast(direction_cv[inter_direction], l);
   } else if (num_waiting[od] == 0 && num_waiting[origin] > 0) {
      // No cars are waiting in the opposing direction. Let another
      // car in the same direction go if there is one waiting.
      cv_signal(direction_cv[origin], l);
   }
   lock_release(l);
}
```