ECE 353: Systems Software

Final Exam Winter ’23

Instructor: Jon Eyolfson

Exam Type: C

Calculator Type: 3

April 24, 2023

Duration: 2 hours 30 minutes

Name

Student ID

Questions
Short Answer (20 points).
Page Replacement (20 points).
Virtual Memory (10 points).
Processes (10 points).
Processes and Threads (10 points).
Locking (10 points).
Semaphores (15 points).
Threads (15 points).
Filesystems (20 points).

This is a “closed book” exam, you may have a single double-sided aid sheet.
This sheet must be handwritten and not mechanically reproduced.
You are only permitted a pencil or pen to write your answers.
Answer the questions directly on the exam.
Do not write on the back of pages.

If in doubt, write your assumptions and answer the question as best you can.
There are 16 numbered pages (page 16 is blank if you need extra room).
The pace of the final exam is approximately one point a minute.
There are 130 total points. Good luck!

Short Answer (20 points).

Q1 (4 points). Describe a situation where a buddy allocator might perform poorly in terms of memory utilization and fragmentation.

> If the allocations are 1 byte more than a power of 2 then the buddy allocator will have to allocate the next highest power of 2. This results in almost half of the memory wasted due to internal fragmentation.

Q2 (4 points). Briefly explain the purpose of condition variables and how they are used in conjunction with mutexes to synchronize threads in a multi-threaded program.

> A condition variable allows you to wait for a condition before proceeding in a multi-threaded program. We use a mutex to protect against data races for the condition we check, as well as allowing us to wait atomically (wait safely adds the current thread to a queue and unlocks the mutex). When wait returns it re-acquires the lock so we continue preventing data races.

Q3 (4 points). Briefly explain the concept of RAID5 and how it helps to improve data redundancy and performance in storage systems.

> RAID5 distributes data across multiple disks to improve read and write performance over a single disk. RAID5 uses one disk for parity (although the parity is distributed across all disks), this allows it to reconstruct data and recover in the event that one disk fails.

Q4 (4 points). Briefly explain the concept of virtual machines and how they enable the efficient use of computing resources.

> Virtual machines virtualize a physical machine and allow you to run multiple operating systems simultaneously. Instead of having under utilized machines you can put multiple VMs on the same machine to share the resources and use them more efficiently. If a machine becomes overloaded, you may move the virtual machine to a different physical machine (like a process).

Q5 (4 points). Discuss one advantage and one disadvantage of having all user processes run in kernel mode, instead of using a separate user mode and kernel mode.

> One advantage is that processes don't have to perform system calls. System calls are slow and require a context switch from user mode to kernel mode. One disadvantage is that any process can modify anything, including other processes or the hardware. You would have no security or any guarantees of fairness.

Page Replacement (20 points).

Assume the following accesses to physical page numbers:

1, 2, 3, 4, 3, 5, 2, 5, 3, 1, 2

You have 3 physical pages in memory. Assume that all pages are initially on disk.

Q6 (10 points). Use the clock algorithm for page replacement. Recall on a page hit, you'll set the reference bit to 1. For each access write all the pages in memory *after the access* in the boxes below. State the number of page faults after all the accesses.

1	2	3	4	3	5	2	5	3	1	2
1	1	1	4	4	4	4	4	3	3	3
	2	2	2	2	5	5	5	5	1	1
		3	3	3	3	2	2	2	2	2

8 page faults.

Q7 (2 points). After swapping in a new page, why is the reference bit set to 1 and the clock hand advanced?

The clock algorithm approximates least recently used. If we did not advance the hand and set the reference bit, the most recently used page would be more likely to be replaced. That would defeat the purpose of the algorithm.

Q8 (8 points). Now, use the optimal algorithm for page replacement. All the other constraints are the same as the previous clock algorithm question. For each access write all the pages in memory *after the access* in the boxes below. State the number of page faults after all the accesses.

1	2	3	4	3	5	2	5	3	1	2
1	1	1	4	4	5	5	5	5	1	1
	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3

6 page faults.

Note: the last page replacement could also be 3.

Virtual Memory (10 points).

We created an experimental machine that has 12 bit virtual addresses, 8 bit physical addresses, page size of 16 bytes, and a page table entry (PTE) of 1 byte.

Our PTE has the following layout where the physical page number (PPN or PFN) is the most significant 4 bits, and the permissions are the least significant 4 bits:

PPN	Execute? (X)	Write? (W)	Read? (R)	Valid? (V)
4 bits	1 bit	1 bit	1 bit	1 bit

Page Table Entry (PTE) Layout

Consider the following physical memory dump (add the row and the column for the physical address):

```
        0x00    0x01    0x02    0x03    0x04    0x05    0x06    0x07
0x00    0x00    0x80    0xCF    0xAF    0x00    0x00    0x00    0x00
0x08    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x10    0x00    0x90    0xFF    0xEF    0x00    0x00    0x00    0x00
0x18    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x20    0x0F    0x1F    0x4F    0x5F    0x3F    0x00    0x00    0x00
0x28    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x30    0x00    0x70    0xB0    0xDF    0x00    0x00    0x00    0x00
0x38    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Example: the byte at address `0x24` is `0x3F`.

Q9 (2 points). How many bytes would we need to store a single-level page table, if we used it?

$2^8 = 256$

For the remaining questions, you must use multi-level page tables. Assume that each page table fits in a physical page (or frame).

Q10 (2 points). What is the minimum number of levels of page tables that we need in our system?

$\lceil \frac{12-4}{4} \rceil = \lceil \frac{8}{4} \rceil = 2$

We're going to translate the virtual address (VA) `0x123`. Our highest level page table is currently at PPN `0x2`. Answer the following questions:

Q11 (1 points). What is the value of page offset in the VA?

> `0x3` or 3.

Q12 (1 points). What is the index (in decimal) of the highest level page table in the VA?

> `0x1`, or 1. (`0x21`,the address of the PTE, is okay.)

Q13 (1 points). What is the index (in decimal) of the lowest level page table in the VA?

> `0x2`, or 2. (`0x12`,the address of the PTE, is okay.)

Q14 (2 points). For the VA write either: 1) the value of the final PTE if it is valid, or 2) the value of the PTE that causes the page fault.

> `0x2` is the PPN of the L1 table, so the address of the L1 PTE is `0x21`. The value of the L1 PTE is `0x1F`. This means `0x1` is the PPN of the L0 table, and it's valid, so the address of the L0 PTE is `0x12`. The value of the L0 PTE is `0xFF`.
>
> `0xFF` is the answer we're looking for.

Q15 (1 points). What is the resulting physical address? If there is a page fault write N/A.

> `0xF3`.

Processes (10 points).

Consider the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    int pipefd1[2];
    int pipefd2[2];
    pipe(pipefd1);
    pipe(pipefd2);

    pid_t pid1 = fork();
    if (pid1 == 0) {
        close(pipefd1[0]);
        int value = /* Very expensive computation */;
        write(pipefd1[1], &value, sizeof(int));
        close(pipefd1[1]);
        exit(0);
    }

    pid_t pid2 = fork();
    if (pid2 == 0) {
        close(pipefd2[0]);
        int value = /* Very expensive computation */;
        write(pipefd2[1], &value, sizeof(int));
        close(pipefd2[1]);
        exit(0);
    }

    close(pipefd1[1]);
    close(pipefd2[1]);
    int sum = 0;
    int temp;
    read(pipefd1[0], &temp, sizeof(int));
    sum += temp;
    read(pipefd2[0], &temp, sizeof(int));
    sum += temp;
    close(pipefd1[0]);
    close(pipefd2[0]);

    return 0;
}
```

This code uses two pipes to collect the results from an expensive independent calculation using two processes.

Q16 (3 points). Would removing the calls to `close()` in the child processes still allow the parent process to behave normally? Explain your answer and discuss any potential consequences.

Removing the calls to `close()` in the child process would still allow the parent process to behave normally. The child process only does a `write` system call to add data to the pipe, afterwards it exits the process. When the process exits all of its file descriptors are closed, so we don't need to explicitly `close` them in this case.

Q17 (4 points). Is it possible to modify the given code so that only one pipe is used between the processes? If so, explain how you would do this, and modify the code listing. If not, explain why it's not possible.

Yes, we could modify the code so that only one pipe is used between the processes. We could remove `pipefd2` and any references to it in the code. We would modify the second child process to write to `pipefd1` instead of `pipefd2`. The order of the writes to the pipe does not matter, and the parent process just adds the results together. The parent would just read from `pipefd1` twice.

Q18 (3 points). Is it necessary to call `waitpid()` for the child processes? Explain your answer and discuss any potential consequences if there are no `waitpid()` calls in the code.

The default answer is to say you should always `waitpid()` on your child processes (unless you want them to last longer than your process), so you don't create zombie processes.

In terms of the order, the parent process does wait for the results from the child processes because `read` blocks until they `write` data to the pipe. Since we know the parent will `exit` shortly after both child processes finish. It wouldn't be a very big problem not to call `waitpid()` in this case, since the child processes would only be zombie processes for a short while. After the parent process finishes the zombie child processes would be cleaned up by `init`. If you said that it doesn't matter, you need to justify it.

Processes and Threads (10 points).

Consider the following code:

```
#include <pthread.h>
#include <stdint.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

void* run(void *arg) {
    uintptr_t thread_id = (uintptr_t) arg;

    pid_t pid = fork();

    printf("Thread %ld running in process %d\n", thread_id, getpid());

    if (pid > 0) {
        int wstatus;
        waitpid(pid, &wstatus, 0);
    }

    return NULL;
}

int main() {
    pid_t pid = fork();

    pthread_t thread1;
    pthread_t thread2;
    pthread_create(&thread1, NULL, run, (void *)1);
    pthread_create(&thread2, NULL, run, (void *)2);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    if (pid > 0) {
        int wstatus;
        waitpid(pid, &wstatus, 0);
    }

    return 0;
}
```

Assume that all system calls execute without encountering any issues and return successfully without any errors.

Q19 (4 points). Analyze the given C program and determine the total number of processes and threads created during its execution.

If you interpret the question as saying explicitly created, the answer is 5 processes and 4 threads. You may include that each process has a thread in it as well, and we would accept 9 threads.

Q20 (2 points). Draw a process tree representing the hierarchy of processes created for a complete execution, using hypothetical process IDs of your choice.

```
        100
      /  |  \
    101 102 103
   /   \
104    105
```

Q21 (4 points). Provide a sample output that could be generated when executing this program (using process IDs from the previous answer).

```
Thread 1 running in process 100
Thread 2 running in process 100
Thread 1 running in process 101
Thread 2 running in process 101
Thread 1 running in process 102
Thread 2 running in process 103
Thread 1 running in process 104
Thread 2 running in process 105
```

Locking (10 points).

Consider the following code snippet:

```
void *thread1(void *arg) {
  pthread_mutex_lock(&mutex1);
  pthread_mutex_lock(&mutex2);
  /* Critical section */
  pthread_mutex_unlock(&mutex2);
  pthread_mutex_unlock(&mutex1);
  return NULL;
}

void *thread2(void *arg) {
  pthread_mutex_lock(&mutex2);
  pthread_mutex_lock(&mutex3);
  /* Critical section */
  pthread_mutex_unlock(&mutex3);
  pthread_mutex_unlock(&mutex2);
  return NULL;
}

void *thread3(void *arg) {
  pthread_mutex_lock(&mutex3);
  pthread_mutex_lock(&mutex1);
  /* Critical section */
  pthread_mutex_unlock(&mutex1);
  pthread_mutex_unlock(&mutex3);
  return NULL;
}
```

Assume that we only have 3 threads, we create 3 independent mutexes, and each thread executes a different function above.

Q22 (4 points). Identify and explain the deadlock condition that can occur in this program.

`thread1` acquires `mutex1`, then we context switch to `thread2`. `thread2` acquires `mutex2`, then we context switch to `thread3`. `thread3` acquires `mutex3`. At this point all threads are holding a lock, and trying to acquire a lock another thread has, therefore there's a deadlock.

Q23 (6 points). Propose a solution to avoid deadlock in this code. Modify the code or describe your proposed solution and briefly explain how it addresses the identified deadlock condition.

In `thread3` we could always acquire `mutex1` before `mutex3`. This would eliminate any circular waits since every thread acquires the mutexes in the same order. For the example above, `thread3` could not acquire `mutex3` until `thread1` is done.

Semaphores (15 points).

Consider the following code:

```
#define TOTAL_THREADS 8

static sem_t sem;

void initialize_sema() {

  sem_init(&sem, 0, 0);

}


void run(int thread_id) {

  if (thread_id == 0) {
    /* We only want thread id 0 to execute this */
    initialize_everything();
    sem_post(&sem)
  }
  else {
    sem_wait(&sem)
    sem_post(&sem)
  }





  /* We want all cores to execute this, only AFTER
     initialize_everything() finishes */
  initialize_thread(thread_id);
}
```

Assume we have *N* thread all set up to start executing `run` and each thread receives a unique `thread_id`. In addition, `initialize_sema` executes before any threads get created, so it's safe to initialize a semaphore.

Q24 (9 points). Using a **single** semaphore, `sem`, enforce the synchronization constraints specified in the comments. You may add code directly above in C or pseudocode (be sure to give the semaphore and initial value).
Try to make a solution that supports **any number** of threads (assume that thread 0 always exists).
For a maximum of 7 points, you may use `TOTAL_THREADS` and assume it's the number of threads executing the `run` function.

Consider the following solution without semaphores:

```
void run(int thread_id) {
  if (thread_id == 0) {
    /* We only want thread id 0 to execute this */
    initialize_everything();
  }
  else {
    thread_yield();
  }

  /* We want all cores to execute this, only AFTER
     initialize_everything() finishes */
  initialize_thread(thread_id);
}
```

Q25 (3 points). Assume that we have cooperative **user** threads, explain why or why not this solution is correct.

Assuming we have an implementation similar to Lab 3, where the run queue is FIFO then this would work. Since there's cooperative `user` threads there's no parallelism, and every thread except thread 0 would give up its first turn. As long as no other thread would get re-queued ahead of thread 0 then `initialize_everything()` would run first.

Q26 (3 points). Assume that we have cooperative **kernel** threads running on multiple cores, explain why or why not this solution is correct.

No, this would not work. Since these are kernel threads, then the kernel may run them in parallel if the hardware allows, so you could not ensure any kind of order by yielding once.

Threads (15 points).

Consider the following code:

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 8
#define ARRAY_SIZE 16000

static int array[ARRAY_SIZE];
static int global_sum = 0;

void *run(void *arg) {
    int tid = *((int *)arg);
    int start = tid * (ARRAY_SIZE / NUM_THREADS);
    int end = start + (ARRAY_SIZE / NUM_THREADS);
    int local_sum = 0;

    for (int i = start; i < end; i++) {
        local_sum += array[i];
    }

    global_sum += local_sum;

    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    /* Initialize array with values */

    /* Create worker threads */
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, run, &thread_ids[i]);
    }

    /* Join worker threads */
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Global sum: %d\n", global_sum);

    return 0;
}
```

The code on the previous page calculates the sum of an array, where each thread calculates a portion of the array.

Q27 (2 points). Is it necessary to protect access to the `array` variable during the execution of the threads? Explain why or why not.

> No, `array` is only read within the `run` function.

Q28 (2 points). Is it necessary to protect access to the `local_sum` variable during the execution of the threads? Explain why or why not.

> No, `local_sum` is only accessed by the thread currently executing `run` so there's no concurrent accesses.

Q29 (2 points). Is it necessary to protect access to the `global_sum` variable during the execution of the threads? Explain why or why not.

> Yes, `global_sum` is accessed concurrently by more than one thread and at least one of the accesses is a write.

Q30 (5 points). Propose a synchronization mechanism to prevent all data races between threads executing `run` while ensuring the correct calculation of `global_sum`. You may propose changes directly in the code itself, use pseudocode, describe your changes, or a mixture. Briefly describe how your changes address the identified issue(s).

> There are several solutions, you could create a mutex and put a `lock` and `unlock` around `global_sum += local_sum;`. You could also make `global_sum` atomic. Finally, you could return the `local_sum` from each thread and only have the main thread calculate the `global_sum`.

Q31 (4 points). If the threads were changed from joinable threads to detached threads, would this lead to any issues? Explain your answer and discuss any potential consequences.

> Yes, for this we need to ensure that all threads are finished to know we have the final `global_sum` value. Detached threads release their resources and don't inform any other thread it's complete, so the main thread may see an incomplete answer. We may also exit the process before all threads finish as well.

Filesystems (20 points).

For the following questions assume your filesystem has a block size of 4096 bytes, each pointer to a block is 4 bytes, and inodes are 128 bytes each.

Q32 (3 points). What's the minimum amount of space **(in bytes)** you need in a filesystem (like ext2) to store 1 000 files? Assume that each file is 96 bytes in size. Include the size of the inodes in your calculation. You may skip the final calculation if you don't have a calculator.

Even though each file is 96 bytes, it would need a single block, which is 4096 bytes. Each file also needs one inode which is 128 bytes. Therefore we need:

$(128 + 4\ 096) \times 1\ 000 = 4\ 224\ 000$ bytes

Q33 (2 points). For the previous question, how many bytes are lost due to internal fragmentation? You may say 0 bytes if that's the case.

Each file loses 4 000 bytes due to internal fragmentation. Therefore we lose 4 000 000 bytes in total.

Q34 (4 points). Why might copying a directory containing a single large file be faster than copying a directory with many small files, even if the total size of the small files is less than the size of the large file?

A single large file is only a single inode and as many blocks as it needs to store all the data. A large file would have near zero internal fragmentation. If we have many small files, each one takes an inode, and each one would consume a block even if they're only 1 byte long. This would lead to the small files taking up more blocks than a large file due to fragmentation. The copying would also need to copy the inodes as well which wouldn't be included in the file transfer speed.

Q35 (3 points). Assuming an inode has 12 direct blocks and 1 indirect block, what is the maximum file size **(in KiB)** that can be supported? You may skip the final calculation if you don't have a calculator.

We can fit 1024 pointers on an indirect block. This would allow us to point to 1036 blocks in total (12 + 1024). Since each block is 4 KiB, our maximum file size is:

$1036 \times 4 = 4144$ KiB

Q36 (2 points). How many directories need to be accessed when attempting to read the file located at the following path: `/home/user/utoronto/ece353-labs/grade.txt`? Assume that `grade.txt` is a regular file.

5 in total. The root directory, `home`, `user`, `utoronto`, and `ece353-labs`

Q37 (6 points). You think you should create a copy-on-write filesystem, but someone (who hasn't taken this class) says you don't need to because you can already do the same thing with hard links. Explain the differences between `file1` and `file2` being hard links to the same inode and the concept of copy-on-write. Can copy-on-write be achieved using hard links?

If `file1` and `file2` are hardlinks, if you modify one it will modify the other. Copy-on-write maximizes sharing read-only blocks or pages while still ensuring that two files remain independent. Copy-on-write could not be achieved using hard links, because both hard links would point to the same blocks and we could not separate out shared blocks with blocks that are modified between the files.