Exam 3

601.229 Computer Systems Fundamentals

December 18, 2025

Complete all questions.

Time: 90 minutes.

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device.

Signed: Solution _____

Print name: _____

Date: _____

Reference

Powers of 2 ($y = 2^x$):

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
y	1	2	4	8	16	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,768	65,536

`uint8_t, uint16_t, uint32_t` are 8/16/32 bit unsigned integer types
`int8_t, int16_t, int32_t` are 8/16/32 bit signed two's complement integer types

Pthreads function reference:

```
int pthread_create(pthread_t *thr_id, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
int pthread_join(pthread_t thr_id, void **retval);
```

Mutex function reference:

```
int pthread_mutex_init(pthread_mutex_t *, const pthread_mutexattr_t *);
int pthread_mutex_destroy(pthread_mutex_t *);
int pthread_mutex_lock(pthread_mutex_t *);
int pthread_mutex_unlock(pthread_mutex_t *);
```

Semaphore function reference:

```
int sem_init(sem_t *, int, unsigned int); // pass 0 as 2nd arg, init. count as 3rd arg
int sem_destroy(sem_t *);
int sem_wait(sem_t *);
int sem_post(sem_t *);
```

Condition variable function reference:

```
int pthread_cond_init(pthread_cond_t *, pthread_condattr_t *);
int pthread_cond_destroy(pthread_cond_t *);
int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *);
int pthread_cond_broadcast(pthread_cond_t *);
```

Processes:

```
pid_t fork(void); // returns 0 in child process
int waitpid(pid_t child, int *wstatus, int options); // pass 0 for options
void exit(int status);
```

Robust I/O:

```
void rio_readinitb(rio_t *rp, int fd);
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, const void *usrbuf, size_t n);
```

Question 1. [20 points] The virtual memory organization for one possible implementation of the DEC Alpha architecture has the following characteristics:

- Pages (virtual and physical) and page tables are 8 KB (2^{13} bytes) in size
- Page table entries (PTEs) are 8 (2^3) bytes in size
- There are three levels of page tables
- Virtual addresses are 64 bits in size

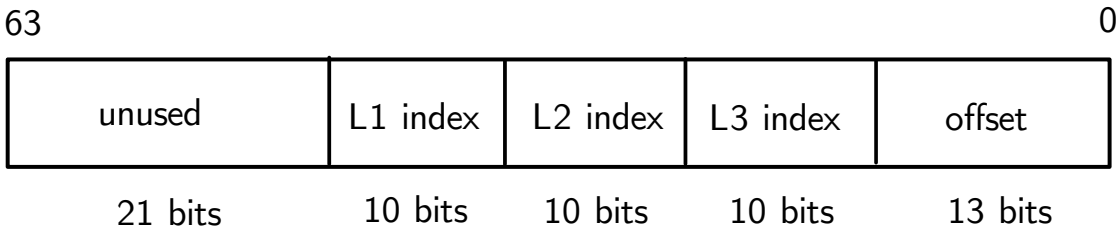
(a) How many bits are in the page offset? (Recall that the page offset is the part of a virtual address indicating the offset of a specific byte in the referenced virtual page.) Explain briefly to justify your answer.

The page size is 2^13 bytes, so 13 bits are needed in the page offset to indicate the specific location in the page being accessed by the virtual address.

(b) How many bits are in the virtual page number? (VPN)? (Recall that the VPN is the part of the virtual address indicating which virtual page is being referenced by the address.) Explain briefly to justify your answer.

Since there are 13 bits in the page offset and 64 bits in the overall virtual address, there are 64-13 = 51 bits in the virtual page number. However, since each page table has 2^13/2^3=2^10 PTEs, only 10 bits are translated by each of the three levels of the hierarchy, meaning that only 3x10=30 bits of the VPN are actually significant in translating the address of a virtual page.

(c) Sketch the format of a virtual address, showing which bits are the page offset and the index values for each level of the page table hierarchy. (Keep in mind that there are three levels!) Assume that if any virtual address bits are unused, that they are in the most signficant bits of the address.



Question 2. [5 points] State one advantage of using threads for concurrency compared to using processes for concurrency.

Some possible answers:

1. Threads are more lightweight (don't require a virtual address space, table of open files, etc.)
2. Threads in a process run in the same address space, so they can cooperate using data structures in shared memory (e.g., queues)

Question 3. [5 points] State one advantage of using processes for concurrency compared to using threads for concurrency.

Some possible answers:

1. Each process has its own address space, providing enforced isolation between concurrent tasks, which is helpful for security

Question 4. [10 points] Consider the following implementation of the *quicksort* sorting algorithm intended to use pthreads for parallel execution of recursive calls:

```
 1:    struct Work { data_t *arr; unsigned start, end; bool success; };
 2:
 3:    void *work( void *arg ) {
 4:      struct Work *w = (struct Work *) arg;
 5:      w->success = quicksort( w->arr, w->start, w->end );
 6:      return NULL;
 7:    }
 8:
 9:    // Sort elements from start (inclusive) to end (exclusive)
10:    bool quicksort( data_t *arr, unsigned start, unsigned end ) {
11:      assert( end >= start );
12:      unsigned len = end - start;
13:      if ( len <= THRESHOLD ) {
14:        // base case: sort sequentially
15:        qsort( arr + start, len, sizeof(data_t), compare_values );
16:        return true;
17:      }
18:      unsigned mid = partition( arr, start, end );
19:      pthread_t left_thread, right_thread;
20:      struct Work left_work  = { arr, start, mid, false },
21:                  right_work = { arr, mid+1, end, false };
22:      if ( pthread_create( &left_thread, NULL, work, &left_work ) != 0 )
23:        return false;
24:      pthread_join( left_thread, NULL );
25:      if ( pthread_create( &right_thread, NULL, work, &right_work ) != 0 )
26:        return false;
27:      pthread_join( right_thread, NULL );
28:      return left_work.success && right_work.success;
29:    }
```

(1) State the most important problem with this implementation, and (2) explain how to fix it. Note: the `quicksort` function *does* correctly sort the array passed as an argument as long as thread creation does not fail. Assume the `qsort`, `compare_values` and `partition` functions work correctly.

The problem is that the quicksort function waits for the thread executing the first recursive call to complete before starting the thread executing the second recursive call, so the recursive calls do not execute in parallel.

To fix the problem, make sure both calls to pthread_create execute before either call to pthread_join.

Question 5. [10 points] *Run-length encoding* is a technique for encoding variable-length network messages where the encoded message consists of two parts: first, a fixed-size *length* value, followed by *content* where the number of bytes in the content is indicated by the length value. For example, the encoded bytes `0012Hello,␣world` might be the encoded form of the string "`Hello,␣world`", since the string's content consists of 12 bytes. A length value consisting of 4 base-10 digits would allow for strings between 0 and 9,999 bytes in length to be encoded.

(a) Briefly explain why, when variable-length messages are encoded using a terminator such as a newline character, it is advantageous to use buffered input. (For example, in a line-based message format such as the one used in Assignment 5, why was it important to use a `rio_t` instance when reading an encoded message?) Hint: when reading a message, it's important to read only the bytes that are part of one message.

If the end of an encoded message is specified by a terminator character, then to avoid reading past the end of the current message, the reading function must read one byte at a time. I.e., we can't know how many more characters remain, so reading multiple characters at once could read past the end of the message. Using buffered input, a relatively large amount of data can be read from the channel into the buffer, but the reading function can consume one byte at a time. If data from a subsequent message remains in the buffer, it can be dealt with later (i.e., in a subsequent call to the message-reading function.) Reading one byte at a time using the read system call would incur significant system call overhead per byte read.

(b) Briefly explain why buffered input is *not* needed when reading messages using run-length encoding.

With run-length encoding, the received can read the (fixed-size) message length information, and then know exactly how many of bytes of data remain in the rest of the message, allowing them to be read with a single invocation of the read system call. This allows the entire message to be read precisely with two system calls (assuming that no short reads happen), without requiring an input buffer.

Question 6. [10 points] Here are two ways of implementing a critical section.

```
// Critical section implemented
// with a guard object
{
  Guard g( lock );
  // ...access shared data...
}
```

```
// Critical section implemented
// using explicit lock/unlock

pthread_mutex_lock( &lock );
// ...access shared data...
pthread_mutex_unlock( &lock );
```

Briefly explain the advantage(s) of using a guard object to implement a critical section using a mutex compared to using explicit lock and unlock operations.

Implementing a critical section using a guard object ensures that the mutex lock is released regardless of how control leaves the critical section. For example, if the "...access shared data..." code throws an exception, contains a return or break statement, etc., the compiler will ensure the guard object's destructor is called, which will release the mutex.
This helps avoid deadlocks caused by a critical section being exited without actually unlocking the mutex lock.

Question 7. [20 points] The `Barrier` data type is intended to work as follows.

A `Barrier` object can be in one of two states, *blocked* and *unblocked*. When a `Barrier` object is initialized (by the constructor), it is in the *blocked* state. When the `unblock()` member function is called, the `Barrier` is set to the *unblocked* state. When the `wait()` member function is called, one of two things happen. If the object is in the *unblocked* state, `wait()` returns immediately. If the object is in the *blocked* state, the calling thread waits until the object changes to the *unblocked* state, and then returns.

You should assume that, for one `Barrier` object, at most one thread will call `unblock()`, but any number of threads might call `wait()`.

Note that `Barrier` is not intended to have value semantics, so ignore the copy constructor and assignment operator.

In parts (a)–(e), write C++ code to implement the `Barrier` class. Hints:

- One valid approach would use a mutex, condition variable, and possibly other member variable(s)
- Another valid approach would use a semaphore and possibly other member variable(s)
- The constructor should initialize the member variables so that the object is in the *blocked* state
- The destructor should do any cleanup required
- You may assume that a `Guard` class exists (like the one in Assignment 5)

blue: mutex/cond var implementation green: semaphore implementation

(a) Complete the class definition by adding member variables:

```
class Barrier {
private:
  // TODO: add member variables
```

```
  bool unblocked;
  pthread_mutex_t lock;
  pthread_cond_t cond;
```

```
  sem_t sem;
```

```
public:
  Barrier();
  ~Barrier();
  void unblock();
  void wait();
};
```

[Question 7 continues on next page.]

[Question 7 continues.]

(b) Complete the constructor:

```
Barrier::Barrier()
  // member initialization if needed
  : unblocked(false)


{
  // body of constructor
  pthread_mutex_init(&lock, NULL);
  pthread_cond_init(&cond, NULL);
```

```
  sem_init(&sem, 0, 0);
```

(c) Complete the destructor:

```
Barrier::~Barrier() {
  pthread_mutex_destroy(&lock);
  pthread_cond_destroy(&cond);
```

```
  sem_destroy(&sem);
```

(d) Complete the `unblock()` member function:

```
void Barrier::unblock() {
    Guard g(lock);
    assert(!unblocked);
    unblocked = true;
    pthread_cond_broadcast(&cond);
```

```
    sem_post(&sem);
```

[Question 7 continues on next page.]

[Question 7 continues.]

(e) Complete the `wait()` member function:

```
void Barrier::wait() {
    Guard g(lock);                          sem_wait(&sem);
    while (!unblocked)                      // allow the next waiter to proceed
      pthread_cond_wait(&cond, &lock);      sem_post(&sem);
```

Question 8. [10 points] Consider the following code:

```
char buf[256];
ssize_t rc = read( fd, buf, 256 );
```

Assume that `fd` is a valid file descriptor open for reading. State a specific reason why after the code above executes, `rc` might contain a positive value less than 256, but that a subsequent call to `read` on the same file descriptor `fd` could return a positive value.

If fd refers to a TCP socket, the OS kernel will prioritize allowing the application to read the data that is available to read right away, even if amount of data available is less than the amount that the caller requested. This type of short read doesn't mean that more data won't be available to read in the future.

A similar issue could occur when reading input from a terminal which implements line buffering: read will return when a complete line is available, even if the line has fewer characters than the read requested.

Question 9. [10 points] Recall that in a *synchronous* application protocol. the communicating peer applications (e.g., client and server) take turns "talking", so that at any given time, one is sending and one is receiving.

In an *asynchronous* application protocol, either peer application can choose to send a message at any time. For example, both peers might send a message at the same time.

Let's say that we want to implement an asynchronous application protocol, such that the following properties hold:

- If a message needs to be sent, it is sent in a timely manner
- If a message is available to read (i.e., its data has arrived and is ready to read from the TCP buffer), it is read in a timely manner

Assume that "in a timely manner" means "without indefinite delay." As an example of an indefinite delay, if the application attempts to read a message, but no message data is available, the read will block until data is available.

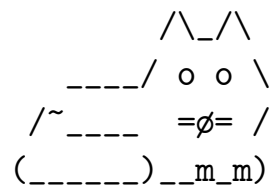
Briefly describe a design that would allow for an implementation of an asynchonous application protocol meeting the properties described above. In particular, think about how you would avoid the problem of indefintely delaying an attempt to send an outgoing message if no incoming message data is available to read. Be specific about the mechanism(s) you would use to solve this problem.

The main problem to be solved is that if reading a message data blocks, we don't want to indefinitely delay writing a message, and vice versa.
One way of solving the problem is to have two threads, one for receiving messages and one for sending messages. Queues could be helpful: for example
- the thread responsible for sending messages could repeatedly wait to dequeue a message object, and then write it to the connection
- the thread responsible for receiving messages could repeatedly wait to read a message from the connection, and then enqueue the received message object

This design would allow the application code to schedule a message to be sent by putting it in the queue used by the sending thread, and wait to receive a message by dequeuing it from the queue used by the receiving thread.

Another possible approach would be to set the connection file descriptor to nonblocking. This would avoid reads from the connection and writes to the connection from blocking indefinitely.

Bonus question. [5 points] The purpose of the internet is pictures of cats. Draw a picture of a cat.



We hope you have enjoyed CSF! Have a great winter break!

[Extra page for answers and/or scratch work.]

[Extra page for answers and/or scratch work.]