

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: _____

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.

(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.

(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 significant bits of the address.

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

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

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.

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.

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

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

```
// Critical section implemented          // Critical section implemented
// with a guard object                  // using explicit lock/unlock
{
    Guard g( lock );                    pthread_mutex_lock( &lock );
    // ...access shared data...         // ...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.

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)

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

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

```
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
```

```
{  
    // body of constructor
```

(c) Complete the destructor:

```
Barrier::~Barrier() {
```

(d) Complete the unblock() member function:

```
void Barrier::unblock() {
```

[Question 7 continues on next page.]

[Question 7 continues.]

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

```
void Barrier::wait() {
```

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.

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 asynchronous application protocol meeting the properties described above. In particular, think about how you would avoid the problem of indefinitely 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.

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.]