ECE 252 Practice Questions

These practice questions can be used for your understanding of the concepts of ECE 252. Past exams from ECE 252 appear separately in Learn. The questions are in no particular order, so don't assume increasing difficulty. These practice questions do not represent all examinable topics or kinds of questions that could appear as exam questions.

1 System Calls and Systems Programming

1.1 Signal Handling in UNIX

You are working on software that makes use of multiple threads (pthreads), interacts with files on the local file system, and communicates over the network. Your supervisor asks you to look into modifying the program so it can handle the `SIGINT` signal (typically generated by a user at the console using the keyboard command `Ctrl-C`).

Describe what happens currently, when the signal is not handled:

What is the (likely) motivation for wanting to handle this signal?

Write a brief description (in point form) of what steps you will likely take when handling the signal.

You are also asked if you can handle `SIGKILL` in the same way. You know the answer is no; explain why.

A team member observes the program sometimes terminates with a segmentation fault. A segmentation fault is really a signal, `SIGSEGV`. He asks if you can tell the program to ignore `SIGSEGV`. Ignoring a signal stops the program from crashing and restarts the instruction that caused the signal. Explain why this is a bad idea.

1.2 Accreditation

Professor Derek Wright needs your help. This year, the Department of Electrical & Computer Engineering needs to assemble a report on all of the ECE courses to submit for accreditation. On disk there is a binary and not-very-human-readable file called `courses.dat`. What Prof. Wright needs is a tool that can interpret this file. For a given course number (e.g., `254`), the tool should print the number of accreditation units (AU) for that course. The first (and only) argument is the course number and the output is the number of AUs for that course. So a sample invocation would look like:

```
./ece_course_au 254
AUs: 5.5
```

The file `courses.dat` contains, in random order, the data for all ECE courses stored in the following format:

```
typedef struct {
  int number; /* integer, e.g. 254 */
  double lecture_hours;
  double tutorial_hours;
  double lab_hours;
  double credits;
  double au; /* Accreditation units (AUs) for this course */
  char mandatory; /* '0' for no, '1' for yes */
} ece_course;
```

Your program should open the file and read one `ece_course`-sized chunk at a time, interpret it as the `ece_course` structure, examine if the `number` field matches the provided value, and if so print the associated AUs to the console. If there are no matches for the provided course number in the file, then a message indicating an error should be printed as in the example and -1 should be returned:

```
./ece_course_au 999
No data found for course ECE 999.
```

Complete the code below to implement the desired behaviour. Be sure to close anything opened and deallocate anything that was allocated. Check the provided input for validity as well.

```
int main( int argc, char** argv ) {









}
```

2 Inter-Process Communication

2.1 IPC

Give two reasons why the operating system needs to be involved when two processes want to communicate via shared memory.

2.2 Interprocess Communication with a Pipe

In the example in the lecture notes of the `pipe( int fd[] )` system call, the call to `pipe` takes place before the call to `fork()`. Explain why the call to `pipe` needs to be before `fork`.

2.3 Fork and Pipe

In the in-class code example demonstrating `pipe()`, both the parent and child process call `close(fd[0])` and `close(fd[1])`, for a total of four calls to `close()`. Explain why both parent and child need to close both ends of the pipe.

2.4 UNIX Pipes

Recall that in UNIX, one method for inter-process communication is the idea of a *pipe*, a unidirectional method of communication. In this question, one process will spawn another. The child process will read the data from a file, send it back to its parent using a pipe, and the parent will send it to a printer. For the child to do this, it will need

to read the file data into a buffer, and then write that into the pipe. The parent will receive the data from the pipe into a buffer, and then call a function to print the data of that buffer.

To create a pipe: call `pipe( int fileDescriptors[] )` This call returns an integer. File descriptors are just integers (so the file descriptors variable is an integer array). The pipe call will fill in the correct values for `fileDescriptors[0]` (the read-end) and `fileDescriptors[1]` (the write-end) so you do not need to initialize them and this will automatically "open" these "files" as well.

To spawn a new process: `fork()`, which takes no parameters and returns a `pid_t` (an integer value). Remember, it creates an exact clone of the parent process.

To open a file: `open ( int fileDescriptor )`. A file must be opened before it can be read.

To close a file: `close( int fileDescriptor )`. It is good programming practice to close all files and pipe ends when finished with them.

To read from a file: `read( int fileDescriptor, void *destination, int numBytes )`. This is a blocking read: the calling function will be blocked until `numBytes` bytes have been read from the source.

To write to a file: `write( int fileDescriptor, void *source, int numBytes )`.

Assume there exists a function `void print( void* data )` that sends the data pointed to by the parameter to the printer. The implementation of this function is not shown.

Assume also there is a function `int fileLength( int fileDescriptor )` that takes the file descriptor of a file and returns its correct length, in bytes. Assume the file to print's descriptor is defined as a constant `PRINT_FILE`. Do not forget to open this file before reading it, and close it when done.

Complete the code below to get the desired behaviour. Remember that system calls like `fork` and `read` and `pipe` may fail and therefore your code should do error checking on their results. If an error is encountered, your `main` function should return `-1` immediately; otherwise it should return `0` to indicate normal exit. Remember to `free()` any memory you allocated with `malloc( int numBytes )`.

```
int fd[2];
pid_t pid;


int main( void ) {









}
```

3 Processes and Threads

3.1 Fork and Find

There is a linear array of integers and you wish to search the array to find the index of a specific value (any location of it will do). Linear searches are slow, but they are a parallelizable task. You would therefore like to break up this job into three (3) processes.

Assume there is a globally defined array of integers called `array` that is filled with appropriate values. Its length is defined in the global variable `array_length`. It may or may not contain the desired search value (defined as the global variable `search_value`). If either thread does find it, print a message to the console with `printf` indicating the index. Example: `printf("Found at %d\n", index);`

Each process should search its (approximately) one third of the array. If one process has a slightly larger section because the array does not divide evenly by 3, that is expected.

Complete the code below to perform a linear search of the `array` using two processes.

```
int main( int argc, char** argv ) {









  return 0;
}
```

3.2 Parallelization with Threads

Consider the following code that computes the value of pi (π) using a monte carlo method: n random points are generated and we count the number of those points x that are inside the unit circle. As we generate only positive random numbers, x / n, the ratio of points, represents only one quadrant and must be multiplied by 4 to give an estimate for pi. The C `rand()` function does not parallelize well; instead assume there is a generator function `unsigned int random( unsigned int i )` that generates a random number between 0 and the unsigned integer maximum (4294967296). The implementation of `random` is not shown, but is safe for use in multithreaded programs.

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

#define INT_MAX 4294967296

int compute( int i ) {
  double x = (double)random( i ) / INT_MAX;
  double y = (double)random( i ^ random( i )) / INT_MAX;
  if ( (x*x + y*y) <= 1) {
    return 1;
  }
  return 0;
}
```

In this question, you will use this code to calculate pi. You must be sure that your program deallocates any resources that are allocated and is free of race conditions or other concurrency problems. Remember that what you write in `main` needs to match up with the specification for the `run` function and vice-versa.

Part 1: Thread wrapper. Assume that the `run` function is provided with a pointer to an integer containing the number of iterations to run. It should return a pointer to the number of points x that were within the first quadrant of the circle for its iterations. Complete the `run` function below.

Part 2. Write `main`. In this part of the question, you will write `main` which needs to (1) create the appropriate number of threads, (2) collect the return value from each thread, and (3) compute the value of pi and print it out. Complete `main` below to implement the desired behaviour.

```
void* run( void* argument ) {

}
```

```
int main( int argc, char** argv ) {

  if (argc < 3) {
    printf("Too few arguments provided.\n");
    return -1;
  }

  int num_threads = atoi( argv[1] );
  if (num_threads < 1) {
    printf("Invalid number of threads.\n");
    return -1;
  }

  int num_iterations = atoi( argv[2] );
  if ( num_iterations < 1 ) {
    printf("Invalid number of iterations.\n");
    return -1;
  }

}
```

3.3 Accreditation Again

Professor Derek Wright still needs your help; the department is still working on the accreditation report. Recently, Professor Aagaard has assembled a database that contains accreditation data. This data includes information about how many AUs a course has in a given term. These can change over time, as the curriculum is updated based on student feedback (and accreditation needs), so the term information is important. Professor Wright has a program that lets him find out the total number of AUs for a given set of courses by querying the database Professor Aagaard has set up. A sequence of course-term pairs is provided as input, and the total AUs is printed out.

Input is parsed using a function `course_term* get_next( )` which returns a pointer to a structure of type `course_term` which is comprised of the course number and the term (see the definition of the structure below). If we've reached the end of input, this function returns `NULL`. The implementation of this function is not shown, but assume it works correctly. This function cannot be run in parallel.

```
typedef struct {
  char course[8]; /* string representation, e.g. ECE254 */
  int term; /* integer, e.g., 1189 for Fall 2018 */
} course_term;
```

To find the number of AUs, there is the function `double query_db( course_term * ct )` . This queries the database for the number of AUs for the specified course in the specified term which is returned as a `double`. Again, the implementation is not shown, but assume it works correctly. The database query is read-only, and the database server is powerful enough, so it can be run in parallel.

Here's the simple implementation of the program:

```
course_term* get_next( );
double query_db( course_term* ct );
double total = 0.0;

int main( int argc, char** argv ) {
  while( 1 ) {
    course_term* next = get_next( );
    if ( next == NULL ) {
      break;
    }
    total += query_db( next );
    free( next );
  }
  printf( "Total AUs: %g\n.", total );
  return 0;
}
```

This program is too slow. You have agreed to rewrite the program to use pthreads to do work in parallel. Rewrite the program to use threads such that each thread carries out one query on the database. Your modified program should be free of race conditions, not have any memory leaks, and produce the same output as the serial program. For your convenience, a structure called `node` has been defined for you, so that you can keep track of the threads in the program. Complete the code below to implement the new behaviour.

```
typedef struct node {
  pthread_t t;
  struct node * next;
} node;

course_term* get_next( );
double query_db( course_term* ct );

node * head;
double total = 0.0;
/* Any other global variables you need go here */
```

```
void* query( void* arg ) {
```

```
}
```

```
int main( int argc, char** argv ) {
```

```
}
```

3.4 Not If I Cancel You First!

We might assume that something has become stuck if a thread takes too long to complete its task. One of the ways that we can deal with this is to have a "watchdog" thread. The watchdog thread will cancel the worker thread if it has taken too long. If the worker finishes in time, however, it cancels the watchdog thread.

Complete the code below so that the watchdog timer will wait for 30 seconds using the standard sleep function: `unsigned int sleep(unsigned int seconds)`. If the watchdog has not yet been cancelled, it should cancel the worker thread. If the worker thread completes `long_running_task` before being cancelled, it should cancel the watchdog instead. Assume that the worker is set up for asynchronous cancellation and the watchdog is set up for deferred cancellation.

```
void* worker( void* arg ) {
  pthread_t * watchdog_thread = (pthread_t*) arg;

  /* Might get stuck! */
  result_t * res = long_running_task( );

  pthread_exit( res );
}
```

```
void* watchdog( void* arg ) {
  pthread_t * worker_thread = (pthread_t*) arg;

  pthread_exit( NULL );
}
```

4 Synchronization and Concurrency

4.1

A fellow student suggests that general semaphores are really just integers that track the current state and therefore it is possible to replace a semaphore type `sem_t` in the program with a simple `int` that can be incremented and decremented using normal C statements. Give two (2) reasons why using an integer is not equivalent to using a semaphore. Explain.

4.2 Synchronization and Concurrency Short Answer

Answer each of the following questions in max. 3 sentences, or complete the code as directed.

1. Suppose there existed a function `int sem_check( sem_t * sem )` that told you the current value of the semaphore. Your co-worker proposes using this function to find out whether your thread would get blocked trying to enter a critical section. Is this a good idea? Explain why or why not.

2. Recall from lecture the *signalling* synchronization pattern that used a semaphore. The UNIX `signal` can be used as a method of interprocess communication. Can it be used to replace a semaphore in this pattern? Explain.

3. Complete the following code fragment to clean up allocated resources in case the task is cancelled before it gets to the end:

```
void foo( void* arg ) {



}

void* bar( void * argument ) {
  task* t = malloc( sizeof( task ) );



  /* Do something useful with task t - code not shown.
     A cancellation point exists in this block. */




  pthread_exit( t );
}
```

4. In the basic Readers-Writers solution, the one in which writers could starve, we discussed modifying the solution to prevent the possibility that a writer starves. Is it possible for readers to starve? Explain your answer.

4.3 The Producer-Consumer Problem

More Threads. A simple way to increase performance of a system is to add more threads. Suppose the buffer size is fixed. For each part below, your answer should include a one sentence explanation.

1. Will increasing only the number of producers help?

2. Will increasing only the number of consumers help?

3. Can we increase both producers and consumers always?

Partial Consumers. The producer/consumer problem in Lab 3 had fixed roles where a thread/process was only a producer or only a consumer. In modern frameworks, a thread/process does one step of the work and then passes it on to another thread/process that will do the next step. Having multiple consumer/producers allows for plug-and-play combinations that are easier to read and more portable across applications.

For this question, you will implement one type of consumer/producer thread that takes an item from its input queue, does some work on it with `do_work`, and then places the modified item in the output queue. Other threads are responsible for putting items into the input queue as well as taking items out of the output queue.

You should use the datatype `queue`, which is created with a maximum size (capacity) and can perform the actions `push()` and `pop()`. These three queue functions are not thread-safe. Also, you should not call `push()` when there is not enough space, and not call `pop()` when there are no items in the queue. Assume any other consumers, producers, and consumer/producer threads in the system will correctly use all concurrency constructs you have created. Hint: each of your queues needs two semaphores and a mutex.

```
typedef struct {
  /* Implementation not shown */
} item;
tyewdef struct {
  /* Implementation not shown */
} queue;

int quit = 0; /* Will changed to 1 by Ctrl-C Handler */
/* Initialize the queue with a maximum size */
void queue_init( queue *q, int capacity );
/* pushes an item onto a queue */
void push( queue* q, item* i );
/* Returns the oldest item in the queue */
item* pop( queue* q );
/* Perform some work on an item */
item* do_work( item* i );
```

```
/* Input items queue */
queue in_queue;
int in_queue_size = 32;
/* Output items queue */
queue out_queue;
int out_queue_size = 64;

/* Put any other global variables you need here */
```

```
int main( int argc, char** argv ) {
   /* Initialize global variables here */
```

```
void* proconsumer( void* arg ) {
  while( !quit ) {
```

```
  /* main() creates threads of different kinds */
  /* main() joins all created threads */
  /* This area runs after all threads have been joined
     Clean up Global Variables */
```

```
  }
  return 0;
}
```

```
  pthread_exit( 0 );
}
```

4.4 Synchronization Patterns

Give a real-life example of when:

1. The *Signalling* synchronization pattern is used:

2. The *Rendezvous* synchronization pattern is used:

3. The *Mutual Exclusion* synchronization pattern is used:

4.5 Semaphore Implementation

You are writing code for a very simple embedded system where the OS does not provide semaphores, but you can use the pthread library and therefore the `pthread_mutex_t` as a mutex. Your task will be to implement the semaphore for this system. In this question, however, you will just write pseudo-code.

The internal definition of the semaphore you are planning to implement is:

```
struct threadinfo {
  int thread_id;
  struct threadinfo * next;
};
```

```
typedef struct {
  int value;
  pthread_mutex_t mutex;
  struct thread_info * queue;
} semaphore;
```

Write pseudocode to show step-by-step what each function named below will do:

```
void sem_init( semaphore * s, int initial_value )
```

```
void sem_destroy( semaphore * s )
```

```
void sem_wait( semaphore * s )
```

```
void sem_signal( semaphore * s )
```

4.6 Swap Meet!

The structure `container_t` is defined as follows:

```
typedef struct container {
   pthread_mutex_lock lock;
   double data;
} container_t;
```

A swap function is defined below:

```
1  void swap( container_t *x, container_t  *y ) {
2    pthread_mutex_lock( &x->lock );
3    double temp = x->data;
4    pthread_mutex_lock( &y->lock );
5    x->data = y->data;
6    pthread_mutex_unlock( &x->lock );
7    y->data = temp;
8    pthread_mutex_unlock( &y->lock );
9  }
```

At lease one concurrency problem exists in this code. Pick one, explain what it is, and describe a scenario that could cause it.

5 Deadlock

5.1 Dealing with Deadlock

During your co-op term at Cyberdyne Systems, you are working on a the control software for the T-1 (forerunner of the T-800, obviously) that occasionally gets deadlocked. A coworker has decided to solve this by replacing all `pthread_mutex_t` structures with one single global variable mutex. Assume there are no other concurrency or synchronization constructs in the program.

Part 1. Will this solve the problem? Explain your reasoning.

Part 2. Name one major downside to this plan.

Part 3. Is there an alternative approach to solve the problem that might be better?

5.2 Rewriting the Program

Although we can't make deadlock categorically impossible, when we have control of the source code of a program, we can make some modifications to our program to make it so that deadlock cannot happen in that program. For each of the strategies below, name which of the four essential elements of deadlock is eliminated, and explain why it works.

1. Use one mutex for everything (no other synchronization constructs).

2. Use more than one mutex in the program, but a thread may hold only one at a time.

3. Use more than one mutex in the program, but threads are required to acquire the mutexes in a specified order.

4. Make each thread work on its own local values (such as the `pthread5.c` example from lecture) instead of shared data.

5. Use trylock functionality for mutex locking.

5.3 Do Or Do Not

You encounter the following code, below on the left, in a program. While it does avoid deadlock, it obviously does not work as expected. Rewrite the code to use `pthread_mutex_trylock()` properly in the space on the right.

```
1  void do_something( ) {
2    /* Some Code */
3    pthread_mutex_trylock( lock1 );
4    pthread_mutex_trylock( lock2 );
5    /* Critical Section */
6    pthread_mutex_unlock( lock2 );
7    pthread_mutex_unlock( lock1 );
8    /* Other code */
9  }
```

```
1  void do_something( ) {
2    /* Some Code */
3
4
5
6
7
8
9
10
11
12
13   /* Critical Section */
14
15
16
17
18
19
20   /* Other code */
21 }
```

5.4 Get the DeLorean up to 88 mph...

When the OS does not provide automatic rollback, you have to do it yourself. We have some structures `struct thing`, `struct widget` and `struct user`. There exists a function:

```
1  int foo( struct user * u, struct thing * t, struct widget * w );
```

This function modifies all of the data structures it receives as parameters. It is possible that function `foo` is involved in a deadlock. A watchdog thread is set up; if the thread running `foo()` takes too long, then it will be terminated. This will leave the structures provided as arguments in an inconsistent state. Explain, using point form, what you need to add to this logic to implement "time travel" (rollback + retry n times):

5.5 Change the Rules

Recall the dining philosophers problem from the lectures: there are five philosophers and five chopsticks. Sometimes deadlock will not occur if we change the rules. For each of the changes below: can deadlock still occur? Explain your answer.

1. Remove a chair (maximum 4 philosophers can sit at the table at a time).

2. 80% of the time, philosophers take the chopstick to their left first; 20% of the time, they take the chopstick to their right first.

5.6 Fix It

You are writing a function called `find_and_resolve_deadlock()`, to resolve a deadlock, once one has been detected. The provided helper function `struct proc * find_deadlock()` returns a pointer to a linked list of type `struct proc` (see definition below). If the pointer is `NULL` then there is no deadlock at the current time. Otherwise, the pointer points to the head of an unsorted linked list containing all the processes involved in the deadlock. Deallocation of the linked list is the responsibility of the caller of `find_deadlock()`.

```
struct proc {
  pid_t pid; /* pid_t is really an int and can be treated as int */
  struct proc* next; /* Is NULL if this is the last item of the list */
};
```

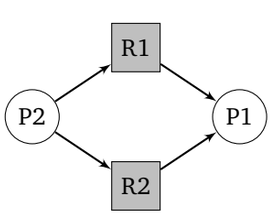
Your system's chosen deadlock strategy is to terminate the youngest process (that is, the process with the highest process id, `pid`). If a deadlock is detected, terminate the chosen process (using signal 9) and check if the deadlock is resolved by running the `find_deadlock()` function again. It may be necessary to kill more than one process to fully resolve the deadlock. Once the deadlock is resolved, the `find_and_resolve_deadlock()` function is done.

Complete the `find_and_resolve_deadlock()` function below to implement the desired behaviour. Remember to deallocate all allocated resources.

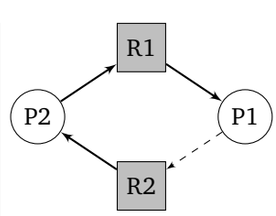
```
struct proc * find_deadlock() {
  /* Implementation not shown (but described above) */
}

void find_and_resolve_deadlock() {









}
```

5.7 Deadlock Detection

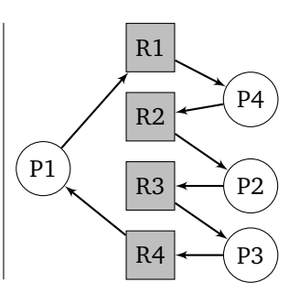
Consider the four (4) resource allocation graph diagrams below. In each of them evaluate: Is there a deadlock? If there is, list the processes involved the deadlock; If not, indicate what single resource request, if any, would cause a deadlock.



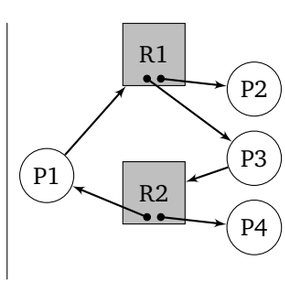
(a)



(b)



(c)



(d)

(a)

(b)

(c)

(d)

6 Files and File Systems

6.1 Opening and Closing Files

Some systems automatically open a file when it is referenced for the first time, and automatically close the file when the process terminates. Give two (2) advantages and two (2) disadvantages of this approach compared to the UNIX approach with the open and close system calls.

6.2 File Allocation

A very small USB drive is formatted such that it has 32 blocks numbered 0 through 31. The table below lists the files currently allocated on that drive and their respective block(s).

File ID	Allocated Block(s)
A	0
B	2, 3, 4
C	28, 29, 30
D	16, 17, 18, 19, 20, 21, 22
E	11, 12, 13

Then two more files are allocated: first file F with a size of 4 blocks, then file G with a size of 2 blocks. Complete the table below to show where the blocks of files F and G would be allocated, according to the algorithm listed in the leftmost column.

File Allocation Method	Allocated Blocks for File F	Allocated Blocks for File G
Contiguous Allocation		
Contiguous Allocation, Best Fit		
Linked Allocation		

6.3 NTFS Journalling

In NTFS (the Windows NT File System) there is journalling which is used to stop metadata from being in an inconsistent state, but it is possible user data ends up in an inconsistent state. What would have to be changed to prevent user data from being in an inconsistent state? Give three reasons why (you think) Microsoft has not implemented this.

7 Valgrind

In this question you take the job of Heimdall: he guards the rainbow bridge to Valhalla. Only the worthy may pass. The following program has been submitted to you for your judgement and you will do what the tools Valgrind and Helgrind do, only better. In addition to reporting memory leaks and concurrency problems you will also explain how to fix the problems you identify. Consider the code below:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <sys/types.h>
4  #include <pthread.h>
5
6  #define NUM_CPUS 8
7
8  int index = 0;
9  int* results;
10 int max_val;
11
12 int compare (const void * a, const void * b) {
13    return ( *(int*)b - *(int*)a );
14 }
15
16 int is_prime( int num ) {
17     if (num <= 1) {
18         return 0;
19     } else if (num <= 3) {
20         return 1;
21     } else if (num % 2 == 0 || num % 3 == 0) {
22         return 0;
23     }
24     for (int i = 5; i*i <= num; i+=6 ) {
25         if (num % i == 0 || num % (i + 2) == 0) {
26             return 0;
27         }
28     }
29     return 1;
30 }
31
32 void *find_primes( void *void_arg ) {
33     int *start = (int *) void_arg;
34     int *count = malloc( sizeof( int ) );
35     *count = 0;
36
37     for ( int i = *start; i < max_val; i += NUM_CPUS) {
38         if ( 1 == is_prime( i ) ) {
39             results[index] = i;
40             ++index;
41             ++(*count);
42         }
43     pthread_exit( count );
44 }
```

```
46 int main( int argc, char** argv ) {
47     if (argc < 2) {
48         printf("A maximum value is required.\n");
49         return -1;
50     }
51     max_val = atoi( argv[1] ); /* Read input as max val
           */
52     results = malloc( max_val * sizeof( int ) );
53     if (results == NULL) {
54         return -1;
55     }
56
57     pthread_t threads[NUM_CPUS];
58     int global_sum = 0;
59
60     for ( int i = 0; i < NUM_CPUS; ++i ) {
61         int* start = malloc( sizeof( int ) );
62         *start = i;
63         pthread_create(&threads[i], NULL, find_primes,
               start);
64     }
65
66     void* temp_sum;
67     for ( int i = 0; i < NUM_CPUS; ++i ) {
68         pthread_join( threads[i], &temp_sum );
69         int *thread_sum = (int *) temp_sum;
70         global_sum += *thread_sum;
71     }
72     printf("Found %d total primes less than %d.\n",
73         global_sum, max_val);
74
75     qsort(results, global_sum, sizeof(int), compare);
76
77     for (int i = 0; i < global_sum; ++i ) {
78         printf( "%d\n", results[i]);
79     }
80
81     pthread_exit(0);
82 }
```

(1) Memory Leaks: Identify any memory leaks below (anything Valgrind would complain about with the options `--leak-check=full` and `--show-leak-kinds=all`). For each memory leak: Indicate whether they are "definitely lost", "indirectly lost", or "still reachable" (you may ignore possibly lost). State the line on which the memory is allocated and explain where the appropriate deallocation statement(s) should go.

(2) Concurrency Problems: Identify any concurrency problems with this code (anything Helgrind would complain about on its own; the `--read-var-info=yes` option just helps you track things down). For each concurrency problem: state the nature of the concurrency problem, the potential effects, and explain how you would solve the problem (no code is necessary but it can be used to illustrate).

8 Asynchronous I/O

8.1

Note that this question was intended to be done with an Asynchronous I/O example available.

If you think you are tired of accreditation-themed questions, imagine how Professors Wright and Aagaard feel! This time, we are in a parallel universe where Spock has a goatee and the threads/OpenMP versions of the accreditation tool have not been built. What we are going to create instead is a non-blocking I/O version of the solution. The simple implementation of the program is:

```
course_term* get_next( );
double query_db( course_term* ct );
double total = 0.0;

int main( int argc, char** argv ) {
  while( 1 ) {
    course_term* next = get_next( );
    if ( next == NULL ) {
      break;
    }
    total += query_db( next );
    free( next );
  }
  printf( "Total AUs: %g\n.", total );
  return 0;
}
```

To make this work using non-blocking I/O, you will use curl multi, as in assignment 1. You should get all the `course_term` structures to be queried and turn them into easy handles (using the `convert` function as specified below), and put them in the curl multi handle. Once they are all ready, you can start the requests. Once all requests have been started, use a loop to check the number that are still running, using the `wait_for_curl` function (also specified below), in the body of the loop. Once all requests are finished, the provided part of `main()` to check the results and clean up the easy handle should run. After that, you can print out the total AUs, clean up, and exit.

Complete the code for `callback` and `main` below to implement the desired behaviour using non-blocking I/O. Your code should not have any race conditions or memory leaks.

```
/* Global variables here */
CURLM* cm;
double total = 0.0;









/* This function takes a pointer to a cURL multi handle
     and waits until something happens. Use this
    function in the body of a loop that is waiting for
     handles to complete. Implementation not shown for
     space reasons. */
void wait_for_curl( CURLM* cm );

/* This function takes a pointer to a course_term
    structure and creates and initializes a cURL easy
    handle with the data it needs to complete the
    request. The easy handle will be set up to call
    the callback function callback() when data is
    received. After this function has run, course_term
     structure is no longer needed. */
CURL* convert( course_term* ct );


/* This function will be run as the callback on a curl
    easy handle */
size_t callback(char *d, size_t n, size_t l, void *p) {
   /* Parses cURL response to extract answer as double
       */
   double aus = parse_response( d, n*l );









   return n*l;
}
```

```
int main( int argc, char** argv ) {
    curl_global_init( CURL_GLOBAL_ALL );









    /* All requests finished, now check they are OK and
         clean up easy handles */
    int left = 0;
    CURLMsg *m = NULL;
    while (( m = curl_multi_info_read( cm, &left ))) {
        if ( m->msg == CURLMSG_DONE ) {
            CURL* eh = m->easy_handle;

            CURLcode return_code = m->data.result;
            if ( return_code != CURLE_OK ) {
                printf("Error %d\n", m->data.result );
                continue;
            }
            curl_multi_remove_handle( cm, eh );
            curl_easy_cleanup( eh );
        } else {
            printf("Error %d\n", m->msg );
        }
    }
    /* All requests finished/callbacks executed */









    curl_global_cleanup();
    return 0;
}
```

8.2 File AIO

You have been asked to design a program that processes a group of files. You can use asynchronous I/O to partially parallelize this: start the read for file $n+1$ and process file n in the meantime. This doesn't work for the first file, so a blocking read takes place below in the starter code. The maximum size of any file we will read is `MAX_SIZE`, so always use this size as the length of a read (even though you may read less, that's okay). You need two buffers: one for the file being processed and one where the next read is taking place.

A list of files to read will be provided as arguments on the commandline to the program. Remember, `argc` tells you the total count of arguments (including the file name of the executable) and `argv` is an array of `char*` ("strings") of the file names. `argv[0]` is the name of the executable, but every subsequent entry in the array is a file name.

After a file is read into memory, you should create and enqueue an asynchronous I/O request using POSIX `aio` to read the next one. Then you can process the current file's data using `void process( char* buffer )`. If processing is finished but the asynchronous read is not, use `sleep(1)` as many times as necessary. Recall that `EINPROGRESS` is returned from `aio_error( aiocb * cb )` if the operation is still in progress.

For your convenience, the AIO structure (of which you only need to set the first 4 fields):

```
struct aiocb {
  int aio_fildes;               /* File descriptor */
  off_t aio_offset;             /* Offset for I/O */
  volatile void* aio_buf;       /* Buffer */
  size_t aio_nbytes;            /* Number of bytes to transfer */
  int aio_reqprio;              /* Request priority */
  struct sigevent aio_sigevent; /* Signal Info */
  int aio_lio_opcode;           /* Operation for List I/O */
};
```

Complete the code below to implement the desired behaviour. Your code should not leak any memory or have any race conditions. Assume that errors won't occur and therefore you do not need to check for them (i.e., memory allocation, enqueuing an asynchronous read, actually reading the data, et cetera, will always succeed). Be sure to close files when reading is finished.

```
void process( char* buffer ); /* Implementation not shown */

int main( int argc, char** argv ) {
  char* buffer1 = malloc( MAX_SIZE * sizeof( char ));
  char* buffer2 = malloc( MAX_SIZE * sizeof( char ));

  int fd = open( argv[1], O_RDONLY );
  memset( buffer1, 0, MAX_SIZE * sizeof( char ));
  read( fd, buffer1, MAX_SIZE );
  close( fd );

  for ( int i = 2; i < argc; i++ ) {









  }



  return 0;
}
```