ECE 252 F19 Final Exam Solutions

(1.1)

```
void* serve_file( void* arg ) {
  char* buf = malloc( BUF_SIZE );

  int* sock = (int*) arg;
  uint32 fileno;
  recv( *sock, &fileno, sizeof( uint32 ), 0 );
  fileno = ntohl( fileno );

  int fd = open_file( fileno );
  if (fd != -1) {
  while( 1 ) {
    /* memset optional as long as you send br
        bytes and not BUF_SIZE */
    int br = read( fd, buf, BUF_SIZE );
    if ( br == 0 ) {
      break;
     }
   send( *sock, buf, br, 0 );
  }
   close( fd );
 }

  close( *sock );
  free( buf );
  free( sock );
  return NULL;
}
```

Serve file component: 12 marks total

- `recv` (okay to specify size of 4 directly rather than sizeof) - 2
- `nthohl` - 1
- Retrieve descriptor from `open_file` - 0.5
- Check for file descriptor being -1 and close without sending data - 0.5
- Loop with exit condition when EOF (read 0 bytes) - 2
- Read file into buffer - 2
- Send - 2
- Close file - 1
- Close socket - 1

```
int main( int argc, char** argv ) {
  /* Create Socket, IPv4 / Stream */
  int sfd = socket( AF_INET, SOCK_STREAM, 0 );

  struct sockaddr_in addr;
  addr.sin_family = AF_INET;
  addr.sin_port = htons( 2520 );
  addr.sin_addr.s_addr = htonl( INADDR_ANY );

  bind( sfd, &addr, sizeof( addr ));
  listen( sfd, 10 );

  while( true ) {
    int newsockfd = accept( sfd, NULL, NULL );
    pthread_t thread;
    int* s = malloc( sizeof( int ) );
    *s = newsockfd;
    pthread_create( &thread, NULL, serve_file,
        s );
    pthread_detach( thread );
  }

  return 0;
}
```

Main thread component: 8 marks total

- `bind` - 2
- `listen` - 1
- `accept` - 2
- Pass arg to thread correctly - 1 (0.5 if abused pointer for its value rather than allocate memory); must be the NEW socket created by `accept`
- Create thread - 1
- Detach thread - 1

General notes: -1 if memory allocated but not deallocated (e.g., integer pointer or exited early)

(1.2)

```
int main(int argc, char **argv) {
  uint8_t childsum;
  uint8_t checksum;

  int shmid = shmget(IPC_PRIVATE, sizeof(size_t) +
      MAX_FILE_BYTES, IPC_CREAT | 0600);

  int pid = fork();
  if (pid < 0) { // fork error
    printf("Error occurred: pid = %d.\n", pid);
    return -1;
  } else if (pid == 0) { // child process
    // read file into shared mem
    FILE *file = fopen(argv[1], "r");
    void *mem = shmat(shmid, NULL, 0);
    size_t bytes = fread(mem+sizeof(size_t), 1, MAX_FILE_BYTES
        , file);
    fclose( file );

    *(size_t *)mem = bytes;
    checksum = calculate(mem + sizeof(size_t), bytes);
    shmdt(mem);

    return checksum;

  } else { // parent process
    // get child checksum
    int child_status;
    wait(&child_status);
    childsum = (uint8_t)WEXITSTATUS(child_status);

    // calculate checksum
    void *mem = shmat(shmid, NULL, 0);
    size_t bytes = *(size_t *)mem;
    checksum = calculate(mem + sizeof(size_t), bytes);

    // detach and destroy
    shmdt(mem);
    shmctl(shmid, IPC_RMID, NULL);

    // compare and return
    return childsum == checksum ? 0 : -1;
  }
}
```

- Open file - 1
- Attach shared mem segment - 1
- Read file into shared memory - 1
- Write size into shared memory - 1
- Detach shared memory (2x) - 1
- Child calculates and returns checksum - 1
- Parent waits for child and collects return value before proceeding - 2
- Parent retrieves size - 0.5
- Parent uses WEXITSTATUS - 1
- Parent calculates checksum - 1
- Parent destroyed shared mem - 1
- Close file - 0.5

(2.1)

```
void* executor( void* arg ) {
    test_fn f = (test_fn) arg;
    bool success = f();
    pthread_mutex_lock( &lock );
    if ( !success ) {
      failed_tests++;
    }
    completed++;
    if ( completed == total_tests ) {
      sem_post( &done );
    }
    pthread_mutex_unlock( &lock );
    sem_post( &next );
}

int run_tests_parallel( test_fn * tests, int num_tests, int threads ) {
    total_tests = num_tests;
    sem_init( &next, 0, threads );
    sem_init( &done, 0, 0 );
    pthread_mutex_init( &lock, NULL );

    for ( int i = 0; i < num_tests; i++ ) {
        pthread_t t;
        sem_wait( &next )
        pthread_create( &t, NULL, executor, tests[i] );
        pthread_detach( t );
    }
    sem_wait( &done );
    sem_destroy( &next );
    sem_destroy( &done );
    pthread_mutex_destroy( &lock );
    return failed_tests;
}
```

In the executor function: 5

- Run test outside of lock - 1
- Use lock to protect global vars - 1
- Increment fail count if needed - 0.5
- Increment completed - 0.5
- post on `done` if all finished - 1
- post on `next` when completed - 1

In the run tests function: 10

- Init the `done` semaphore to 0 - 1
- Init the `next` semaphore to `threads` - 1
- Init the `lock` - 0.5
- Loop for number of tests - 1
- `wait` on `next` in the loop to proceed - 1
- Create thread - 2
- Detach - 1
- `wait` for `done` - 1
- cleanup of semaphores and mutex 1.5

(2.2)

```
void *contestantA(void *arg) {
  while(!show_over) {
    pthread_mutex_lock(&mutex);
    if(!sauce_present() || !cheese_present()) {
      pthread_cond_wait(&cond, &mutex);
    }
    if(sauce_present() && cheese_present()) {
      get_sauce();
      get_cheese();
      sem_post(&sem);
    }
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit( NULL );
}

void *host(void *arg) {
  while(!show_over) {
    sem_wait(&sem);
    pthread_mutex_lock(&mutex);
    for(int i = 0; i < 2; i++) {
      place_ingredient();
    }
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
  }
  pthread_cond_broadcast(&cond);
  pthread_exit( NULL );
}
```

- Contestant (8)
  - Lock mutex - 1
  - Check ingredients - 1
  - If ingredients not present, wait condition variable - 2
  - Check again if ingredients present - 1
  - Get ingredients - 1
  - Post semaphore - 1
  - Unlock mutex - 1
- Host (6)
  - Wait on semaphore - 1
  - Lock mutex - 1
  - Place 2 ingredients - 1
  - Broadcast - 1
  - Unlock mutex - 1
  - Final broadcast when all is over - 1

(2.3)

Initial values: `mutex = 1` and `fighter_queue = 0` and `transport_queue = 0`.

Fighters

```
wait( mutex )
if transports > 0
  transports--
  post( transport_queue )
else
  fighters++
  post( mutex )
  wait( fighter_queue )
end if
launch()
```

Fighter code: 5

- Use mutex to protect modification of variables - 0.5
- If transport is ready, decrement (0.5) and signal it (1)
- If not, increment count (0.5), post mutex (1) and wait (1)
- Launch - 0.5

Transport

```
wait( mutex )
if fighters > 0
  fighters--
  signal( fighter_queue )
else
  transports++
  post( mutex )
  wait( transport_queue )
end if
launch()
post( mutex )
```

Transport code: 5.5

- Use mutex to protect modification of variables - 0.5
- If fighters are ready, decrement (0.5) and signal them (1)
- If not, increment count (0.5), post mutex (1) and wait (1)
- Launch - 0.5
- Last post on mutex - 0.5

(2.4)

```
flights* display( schedule* s ) {
  pthread_rwlock_rdlock( s->lock );
  int size = s->max_flights * sizeof( flight );
  flights* copy = malloc( size );
  memcpy( copy, s->flights, size );
  pthread_rwlock_unlock( s->lock );
  return copy;
}

bool update( schedule* s, flight* f ) {
  bool success = false;
  pthread_rwlock_wrlock( s->lock );
  for ( int i = 0; i < s->num_flights; i++ ) {
    if ( s->flights[i].flight_no == f->flight_no
    && strncmp( s->flights[i].airline, f->airline, 21 )
        == 0 ) {
      s->flights[i] = *f;
      success = true;
    }
  }
  pthread_rwlock_unlock( s->lock );
  return success;
}

void add(schedule* s, flight *f ) {
  pthread_rwlock_wrlock( s->lock );
  if ( s->num_flights == s->max_flights ) {
    flights = realloc( s->flights, s->max_flights * 2 );
    s->max_flights *= 2;
  }
  s->flights[num_flights] = *f;
  s->num_flights++;
  pthread_rwlock_unlock( s->lock );
}
```

- `display` (3.5 total)
  - Readlock - 0.5
  - Calculate correct memory amount - 1
  - Allocate memory - 0.5
  - memcpy - 1
  - return - 0.5
- `update` (5 total)
  - Writelock - 0.5
  - for loop to search - 1
  - compare flight number and airline code - 2
  - assign update - 1
  - return - 0.5
- `add` (4.5 total)
  - Writelock - 0.5
  - Increase size if necessary - 2
  - Assign flight (can also memcpy) - 1.5
  - Increment count - 0.5

- Forget to unlock is -1
- No locking at all -2
- Use read where writelock is correct -1
- Use write where readlock is correct -0.5

(3)

1. Livelock is observed when processes are still executing (not blocked) but also that they are not able to make any forward progress in execution.

2. Livelock can be dealt with using any of the deadlock recovery techniques we talked about except for rollback. Any such strategy would be a valid answer, as well as some other things like suspending one of the processes for a while...

3. No, it is not a good strategy. This would produce unexpected and undesired results. For example, if threads A and B need semaphores X and Y to get into the critical section and a deadlock exists where A holds one semaphore and B holds the other, then signalling on both will cause both threads A and B to enter the critical section.

(4.1)

- Callback
    1. Callback copies data in the wrong direction.
    2. Data to be received is overwritten with garbage
    3. Change order of `p` and `d` in `memcpy` call on line 2.
- Easy Init
    1. Same easy handle is reused repeatedly
    2. Only the last one actually happens / undefined behaviour
    3. Creating the easy handle should be inside the loop (after line 16).
- Multi Perform
    1. `curl_multi_perform` called only once.
    2. Program does not wait for all requests to be done.
    3. Should not proceed to the loop at line 29 before all requests are done (`still_running` is 0). Write a loop where you wait (or at least repeatedly call `curl_multi_perform`).
- Checking for OK
    1. The check if the return code is `CURLE_OK` is incorrect.
    2. Program processes curl handles with errors and ignores those without errors.
    3. Change comparison on line 34 to `!=`
- Cleanup
    1. Missing `curl_global_cleanup` call.
    2. Resources not properly deallocated.
    3. Place before the return statement at the end of main (after 53).

(4.2)

```
char lieutenant(char command, int sockfd1, int sockfd2) {
  char l1 = 'r', l2 = 'r';

  write(sockfd1, &command, sizeof(char));
  write(sockfd2, &command, sizeof(char));

  struct pollfd pfd[2];
  pfd[0].fd = sockfd1;
  pfd[0].events = POLLIN;
  pfd[1].fd = sockfd2;
  pfd[1].events = POLLIN;

  while(pfd[0].fd != -1 || pfd[1].fd != -1) {
    if(poll(pfd, 2, 1000) == 0) {
      break;
    }
    if(pfd[0].revents & POLLIN) {
      read(sockfd1, &l1, sizeof(char));
      pfd[0].fd = -1;
    }
    if(pfd[1].revents & POLLIN) {
      read(sockfd2, &l2, sizeof(char));
      pfd[1].fd = -1;
    }
  }

  return (l1 == l2) ? l1 : command;
}
```

- Write the message to the other lieutenants - 2 (1.5 for the first, 0.5 for the second)
- Set up `pollfd` structures - 2
- Call poll correctly - 1
- Check for timeout - 1
- Check the events for `POLLIN` - 1
- Read the data from the remote systems - 2
- Remove the `pollfd` that we don't need anymore - 1
- Decide at the end what command to return - 1