ECE 254 S18 Final Exam Solutions

(1.1)

```
int main( int argc, char** argv ) {
    pthread_t b_thread;
    /* Initialize global Variables Here */
    sem_init( &items, 0, 0 );
    sem_init( &spaces, 0, BUFFER_SIZE );
    pthread_mutex_init( &mutex, NULL );
    memset( &buffer, 0, BUFFER_SIZE * sizeof(
        item* ));

    consumers = NULL;

  /* Creation of producer threads not shown */

  pthread_create( &b_thread, NULL, balancer,
      NULL );
  pthread_exit( NULL );
}

double buffer_usage( ) {
  int count = 0;
  pthread_mutex_lock( &mutex );
  for ( int i = 0; i < BUFFER_SIZE; i++) {
    if ( buffer[i] != NULL ) {
      count++;
    }
  pthread_mutex_unlock( &mutex );
  return (double) count / BUFFER_SIZE;
}
```

```
void* balancer ( void* arg ) {
  while( 1 ) {
    double usage = buffer_usage( );
    if ( usage >= 0.75 ) {
      for ( int i = 0; i < 3; i++ ) {
        node *n = malloc( sizeof( node ) );
        n->next = consumers;
        consumers = n;
        pthread_create( &(n->thread), NULL,
            consumer, NULL );
      }
    } else if ( usage <= 0.25 ) {
     for ( int j = 0; j < 3; j++ ) {
       if ( consumers->next == NULL ) {
         break; /* 1 consumer remains */
       }
       node *n = consumers;
       consumers = consumers->next;
       pthread_cancel( n->thread );
       free( n );
     }
   }
   sleep( 30 );
}
```

Grading notes:

- Initialize head of consumers list 1
- Buffer usage calculation 2 marks
- Lock and unlock correctly in the balancer 1
- Call the usage function 1
- If usage more than .75 then loop x3 1
- Create new node 1
- pthread_create 1
- Add to linked list 1
- If usage less than .25 then loop x3 1
- Remove from linked list 1
- Don't cancel the last consumer 1
- pthread_cancel 1
- Destroy allocated memory 1

Consumer Add `pthread_testcancel()` as the first or last statement of the while-loop. 2 marks.

Techical note: `sem_wait` is a cancellation point, so if you change nothing then the thread automatically checks its cancellation at that point meaning no changes are necessary. But you have to indicate, in some way, that you know this, and not just say change nothing.

(1.2)

```
typedef struct {
  sem_t sem;
  pthread_mutex_t mutex;
} sem_queue;

void sem_queue_init( sem_queue * q, int
    init_value ) {
 sem_init( q->sem, 0, init_value );
 pthread_mutex_init( q->mutex , NULL );
}

void sem_queue_destroy( sem_queue * q ) {
 sem_destroy( q->sem );
 pthread_mutex_destroy( q->mutex ) );
}
```

```
void sem_queue_wait( sem_queue * q ) {
  pthread_mutex_lock( q->mutex );
  sem_wait( q->sem );
  pthread_mutex_unlock( q->mutex );

}

void sem_queue_signal( sem_queue * q ) {
  sem_post( q->sem );
}
```

(1.3)

1. Yes: if there is only one mutex then deadlock cannot occur. The hold-and-wait condition cannot be fulfilled, because when holding the mutex then there is nothing to wait for. Without this condition, deadlock is not possible.

2. It lowers the performance of the program because there will be a lot more waiting for the mutex.

3. Yes, lock ordering could solve the problem without reducing the performance of the program (as much).

(2.1)

```
int main( int argc, char** argv ) {
  int fourK = 4 * 1024; /* You can also make it 4096, or pow(2, 12), whatever */
  int provided = atoi( argv[1] );
  int page = provided / fourK; /* Integer division cuts off the fractional part */
  int offset = provided % fourK; /* Modulus operator provides offset easily */

  printf( "The page number is %d and the offset is %d.", page, offset );

  return 0;
}
```

(2.2)

- Record a run of the program including memory references
- Note down the actual number of cache misses that occurred
- Simulate a run of the program (by replaying the recording) using the optimal algorithm for page replacement
- Note down how many cache misses occurred during the simulation
- Compare the actual cache misses against the optimal cache misses

(2.3)

1. When a memory access occurs, trying to find the physical address requires consulting the page table. If an access is invalid, we will not find a page for that given address.

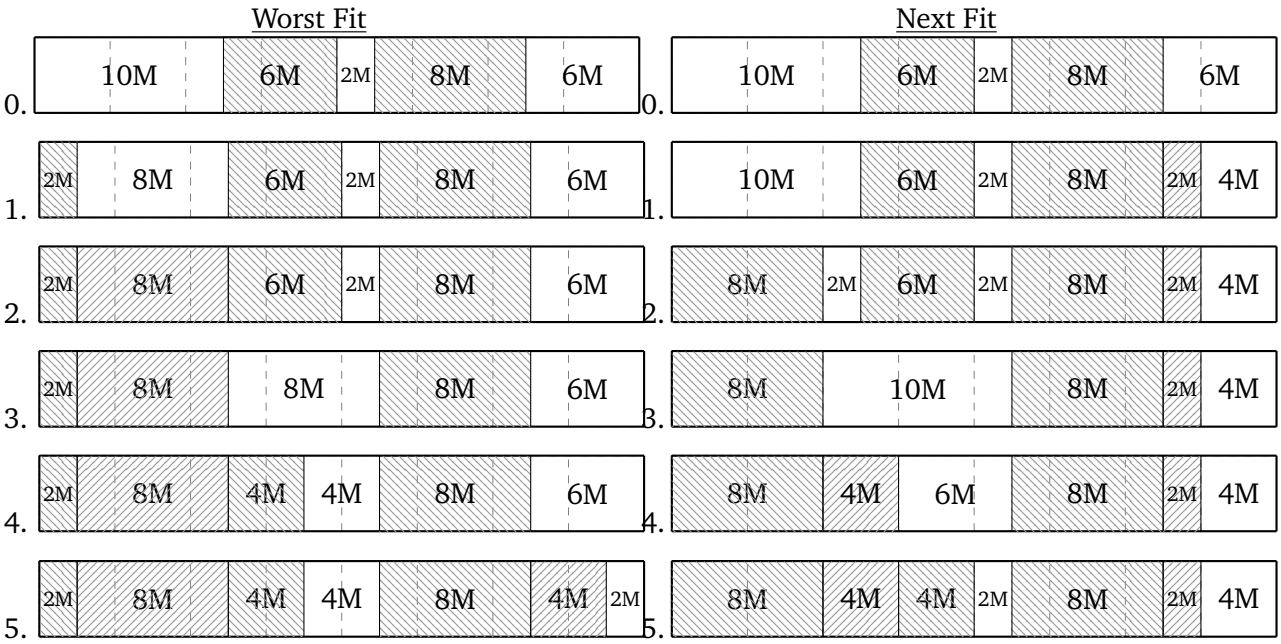
2. If an invalid reference is detected, a segmentation fault signal is sent to the program.

3. The logical address will be split into the page and offset. The TLB is checked to find the frame number, and if it is not found the page table must be consulted to find the frame number. However the frame number is found, it is combined with the offset to produce the physical address.

4. Speed vs cost: a bigger TLB would have a higher hit rate but it would also cost a lot more money.

5. No, a hierarchical page table would require an additional memory access for the 2nd level of the page table whenever the value is not found in the TLB. For the size of memory available this is not worth it.

(2.4)



(3.1)

A	A	A	A	B	B	B	A	–	C	D	D	D	D	C	E	E	F	F	–
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

(3.2) There are 4 operations, although only one is interesting (really) in terms of proving that it is constant time.

1. Add something to a queue: this is a constant time operation since we can just put it in its priority queue by going to the tail and adding it there.
2. Find the next task: here we DO have to look at the priority bitmap but the interesting part is that although we have to search the bitmap and the amount of time it takes to find the next item can vary, **it is constant with respect to the number of tasks in the queues**. And because there are only 140 priority levels the time to find the next item has a hard upper limit.
3. Remove something from a queue: also constant time because we can just dequeue from the head of the selected queue.
4. Swapping active and expired queues: just changing two pointers; also constant time.

One mark each, but two for the finding the next task,

(3.3)

1. If time slices are too long, the system appears unresponsive to users; if they are too short, too much time is spent handling timer interrupts.
2. If the amount of time to choose the next thread is too long, system performance is low because the OS is taking up a lot of time. If it's too short, we probably don't make good decisions about what threads should run.)
3. If no weight is given to priority, priority is meaningless; if too much is given then low priority threads probably do not get to run.
4. If the priority increase given to I/O bound threads is too low it does nothing and CPU-bound processes dominate; if it is too high then I/O processes squeeze out CPU-bound ones.
5. If the priority given to I/O bound threads is too low CPU-bound processes dominate; if it is too high then I/O processes squeeze out CPU-bound ones.

(3.4)

1. In HRRN, a thread's priority rises the longer it waits so it will eventually get to the top of the list.
2. In round robin, all threads do eventually get a turn.
3. SJN does **not** avoid starvation!
4. The constant time scheduler moves all threads from the active to expired queues so a thread will eventually get a chance to run.
5. The completely fair schedule makes sure that each thread gets to run within the "target latency".

(3.5)

Advantages: making decisions about how to assign resources are easy, the ability for programs to get some say in how scheduling is done

Disadvantages: random number generation is difficult, no way to be sure that a process gets its tickets back after giving them away...

It might not be possible to give your tickets to another process if, for example, you don't know who the other process is (ie when the client communicates indirectly with the server).

(4.1)

To save the user data, changes to all files would have to be recorded as a transaction, before actually being written to disk. Possible reasons include (pick 3): (1) lower performance, (2) easier to not implement on their side, (3) no market need for it, (4) not sensible because when writing the transaction to disk one could just as easily have completed the write for the same effort...

(4.2)

1. 2026.66 ns
2. 1786.66 ns
3. 20% is the break-even point.

(4.3)

1. **Block**: networks operate on packets of data, and packets are of some significant size.
2. **Sequential**: network data is sent/received in a sequence and you cannot seek back and forth through it.
3. **Asynchronous**: networks have unpredictable response times.
4. **Shared (not dedicated)**: many processes can be using the network interface concurrently.
5. **Both input and output**: the network can be used to both send and receive data (as expected).