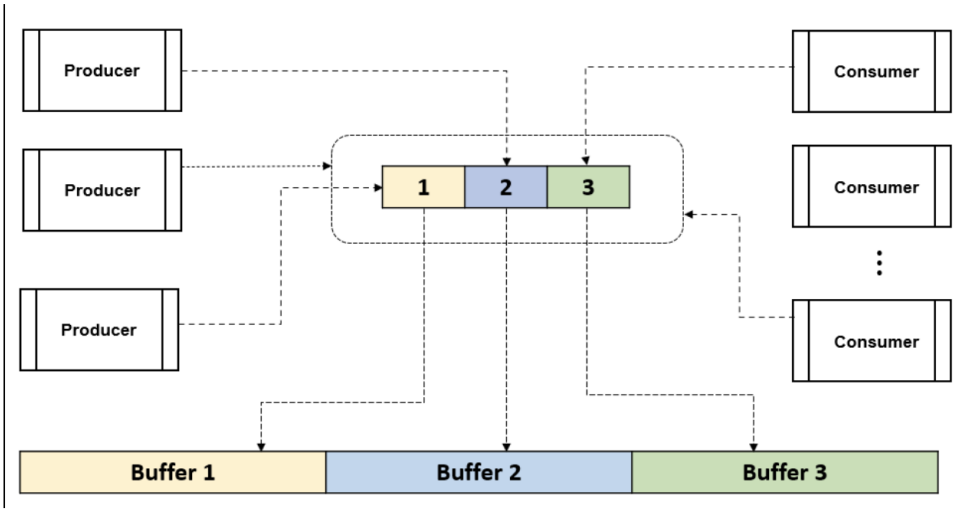
ECE 252 F22 Final Exam Solutions

(1)

Since the producers all have access to the shared buffer, it is a waste of memory to have a separate local buffer for each producer. This memory space can be saved. Another inefficiency with this local buffer is that copying data from the local buffer and the shared buffer is another extra cost. A better approach is to let the producer download each image strip directly into one of the slots in the shared buffer by specifying the shared buffer address as the argument of the curl call. When a producer is using a slot of the buffer, we could lock the slot and keep the rest of slots accessible to other producers/consumers. To implement this mechanism, we introduce another level of indirection. We index each buffer slot and use an array of integers to represent indices. The shared resource becomes the array of indices that points to the actual shared buffers. The producer needs to find an index number that points to an empty buffer slot and the consumer needs to find an index number that points to a full buffer slot. This makes adding items to the buffer and taking out items from the buffer parallelized.

One implementation could be we have two stacks. Stack P contains all the empty buffer indices (for the producers) and a second stack C contains all the full buffer indices (for the consumers). We initialize the stack P with all the indices. When a producer gets an item from stack P, it starts to write data to the corresponding buffer. When it is done, it pushes the index to the stack C and the consumer will start to consume the data. The pseudo code is as follows.



Producer	Consumer
Wait (stack P)	Wait (stack C)
Download data	Remove image data
Post(stack C)	Post(stack P)
	Process image data

(2.1)

Other Requester

```
wait( mutex )
if requesters+priority >= n+1
    signal( mutex )
    return
end if
requesters++
signal( mutex )

signal( worker )
wait( requester )
request_help()

wait( mutex )
requesters--
signal( mutex )
```

Priority Requester

```
wait( mutex )
priority++
signal( mutex )

signal( worker )
wait( priority )
request_help()

wait( mutex )
priority--;
signal( mutex )
```

Worker

```
wait( worker )
wait( mutex )
if priority > 0
  signal( priority )
else
  signal( requester )
end if
signal(mutex)
help_requester()
```

(2.2)

Identify a problem (2 marks): It's possible that the file is deleted by another process after the check on line 4, but before we actually set up the watching of the event at line 8. If that happens, we will "miss" the event and get stuck forever.

Solution (4 marks): After we've set up the watch on line 8, but before the blocking call to `read()` on line 14, we should do another check to see if the file exists. If it's already been deleted, then we should should close the inotify (and deallocate the buffer if we did allocate it) and do the next iteration of the loop. If we check immediately at line 9 that saves us from needing to deallocate.

Notes: it has to be after the inotify watch is set up, otherwise this does not work. -2 marks if the loop continuing logic forgets deallocation of the memory and/or not closing the inotify.

(2.3)

```
pthread_cond_t cv;
pthread_mutex_t lock;
int tasks_done = 0;
int total_tasks = 10;


int main( int argc, char** argv ) {
  init( ); /* Initialize cv, lock */
  start_workers( ); /* Start N worker() threads
       */

  pthread_mutex_lock( &lock );
  if ( tasks_done != 10 ) {
    pthread_cond_wait( &cv, &lock );
  }
  pthread_mutex_unlock( &lock );

  clean_up();
  return 0;
}
```

```
void* worker( void* task ) {
  do_task( task );
  pthread_mutex_lock( &lock );
  tasks_done++;
  if (tasks_done == total_tasks) {
    pthread_cond_signal( &cv );
  }
  pthread_mutex_unlock( &lock );
  pthread_exit( NULL );
}
```

(3)

```
/* Returns the current price, in 1000ths of a cent, that we are willing to pay  to the service
    identified by the service_id parameter */
int get_current_price( int service_id );
void* handle_request( void* a ); /* Function Prototype */


/* Assume that the socket is the socket we have already done bind() and listen() on and it is
    ready */
void loop( int socket ) {
  while( !quit ) {
    int new_sock = accept( socket, NULL, NULL );
    int* a = malloc( sizeof( int ) );
    *a = new_sock;
    pthread_t t;
    pthread_create( &t, handle_request, NULL, a );
    pthread_detach( t );
  }
}
```

3 for accept, 1 for allocating a variable to pass to the new thread, 1 for passing the variable correctly, 2 for creating the thread, 1 for detach. (8 for this part)

```
void* handle_request( void* a ) {
  int * s = (int*) a;
  int sid;
  recv( *a, &sid, 4, 0 ); /* or sizeof(int) */
  sid = ntohl( sid );
  int price = get_current_price( sid );
  price = htonl( price );
  send( *a, &price, 4, 0 );
  close( *a );

  free ( a );
}
```

2 for recv, 1 for converting network to host order, 1 for calling the get-price function, 1 for converting order of the price, 2 for send, 1 for deallocating memory, 1 for closing (9 for this part)

(4.1)

3 marks for the definition:

```
typedef struct {
  sem_t spaces;
  sem_t items;
  log_msg messages[ARRAY_SIZE];
} shared;
```

7 marks for creating and attaching correctly (1 for malloc, 1 for return, 3 for creating, 2 for attaching):

```
shmem* create_and_attach_shmem( ) {
  shmem* m = malloc( sizeof(shmem) );
  m->shmid = shmget( IPC_PRIVATE, sizeof(shared), IPC_CREATE | 0666 )
  m->mem = shmat( m->shmid, NULL, 0 );
  return m;
}
```

3 marks for initializing each semaphore correctly (note 1 for shared!), 1 mark for memset the memory.

```
void initialize_shared( shared* s ) {
  memset( s, 0, sizeof(shared) );
  sem_init( s->spaces, 1, ARRAY_SIZE );
  sem_init( s->items, 1, 0 );
}
```

(4.2)

6 for producer (2 for memcopy) 9 for consumer (2 memcpy, 3 for call to write_to_log)

```
void* producer( void* arg ) {
  shared* s = (shared*) arg;
  int current = 0;

  while( !quit ) {
    log_msg message = get_next_event( );
    sem_wait( s->spaces );
        memcpy( message, p->messages[current], sizeof( log_msg ) );
        sem_post( s->items );
        free( message );
        current = (current + 1) % ARRAY_SIZE;
  }
  pthread_exit(NULL);
}

void* consumer ( void* arg ) {
  shared* s = (shared*) arg;
  int output_file = get_output_file_descriptor();
  int current = 0;

  while( !quit ) {
    char *m;
    int length;
    sem_wait( s->items );
    length = s->messages[current].message_length;
    m = malloc( length );
```

```
        memcpy( s->messages[current].message, m, length);
        sem_post( s->spaces );
        current = (current + 1) % ARRAY_SIZE;
        write_to_log_file( output_file, m, length );
  }
  pthread_exit(NULL);
}
```

(4.3)

2 marks for the cleanup function.

2 marks for allocating the aio control block, 5 for configuring it, 1 for enqueuing the write.

-2 for stack allocating the AIO control block

```
void cleanup( union sigval arg ) {
  struct aiocb* cb = (struct aiocb*) arg.sival_ptr;
  free( cb->aio_buf );
  aio_return( cb );
  free( cb );
}


void write_to_log_file( int log_file_descriptor, char* log_line, int log_line_length ) {

  struct aiocb* cb = calloc( 1, sizeof( struct aiocb ) );
  cb->aio_fildes = log_file_descriptor;
  cb->aio_nbytes = log_line_length;
  cb->aio_buf = log_line;
  cb->aio_sigevent.sigev_notify = SIGEV_THREAD;
  cb->aio_sigevent.sigev_notify_function = cleanup;
  cb->aio_sigevent.sigev_value.sival_ptr = cb;

  aio_write( cb );

}
```