3. [12 marks; 6 each] Consider the following program to demonstrate the operation of pthread_create and pthread_join. The pthread_join function blocks the calling thread until the specified thread has exited. If the thread has already exited at the time pthread_join is called, the caller continues execution without blocking. Assume the existence of a "pthread_attr_setpriority" function which allows the priority for a new thread to be specified. Assume also that higher numbers correspond to higher priorities, and that all printf statements are executed without blocking the caller

```
int main() {
   pthread_t t;
   pthread_attr_t attr;

   pthread_attr_init(&attr);
   pthread_attr_setpriority(&attr,10);
   printf("bam ");
   pthread_create(&t, &attr, A, 0);
   printf("baf ");
   pthread_join(t, 0);
   printf("blam ");
   pthread_exit(0);
}
```

```
void *A(){
   pthread_t t2;
   pthread_attr_t attr2;

   pthread_attr_init(&attr);
   pthread_attr_setpriority(&attr, 20);
   printf("umph ");
   pthread_create(&t2, &attr, B, 0);
   printf("ugh ");
   pthread_join(t2, 0);
   printf("urk ");
   pthread_exit(0);
}

void *B() {
   printf("krak ");
   pthread_exit(0);
}
```

(a) Assume that the scheduler runs threads using a FCFS policy, ignoring priorities. What will the program above print out?

bam baf umph ugh krak urk blam

(b) Assume that the scheduler runs threads using a preemptive priority policy, and that the priority of the "main" thread is 0. What will the program above print out now?

bam umph krak ugh urk baf blam

../continued

5. [11 marks; breakdown given below]

You have been hired as an OS consultant to solve a problem with a client's system. Their OS has a set of queues, each of which is protected by a lock (implemented as a `pthread_mutex_t`). To enqueue or dequeue an item, a thread must hold the lock associated with the queue.

They have implemented an *atomic transfer* routine that dequeues an item from one queue and enqueues it on another. The client requires that the transfer appear to occur atomically. The transfer routine is being used extensively throughout their multithreaded system, and may be called by many different threads with any of the system queues as inputs.

Unfortunately, their first attempt, shown below, causes the system to deadlock.

```
void transfer(Queue *queue1, Queue *queue2)
{
      Item *thing; /* the thing being transferred */
      pthread_mutex_lock(&queue1->lock);
      thing = Dequeue(queue1);
      if (thing != NULL) {
            pthread_mutex_lock(&queue2->lock);
            Enqueue(queue2, thing);
            pthread_mutex_unlock(&queue2->lock);
      }
      pthread_mutex_unlock(&queue1->lock);
}
```

You have already verified that the Enqueue and Dequeue functions are correct, that the synchronization variables have all been initialized properly, and that the transfer function is never called with the same queue for both inputs (that is, the client has guaranteed that transfer(queue*A*, queue*A*) will never occur).

(a) **[5 marks]** Demonstrate how the use of the transfer function can lead to deadlock.

Suppose thread T1 calls transfer(queueA, queueB) and T2 calls transfer(queueB, queueA). We could have the following sequence of events:

T1:	T2:
pthread_mutex_lock(&queueA->lock);	
	pthread_mutex_lock(&queueB->lock);
	thing = Dequeue(queueB);
thing = Dequeue(queueA);	
if (thing != NULL) {	
pthread_mutex_lock(&queueB->lock);	
/* blocks waiting for queueB to be free */	if (thing != NULL) {
	pthread_mutex_lock(&queueA->lock);
	/* blocks waiting for queueA to be free */

../continued

(b) **[2 marks]** Which of the 4 conditions for deadlock could be easily removed to fix this problem?

Either hold-and-wait, or circular-wait could be broken. (Breaking mutual exclusion would violate the atomicity of the transfer function, and could cause the queues to become corrupted; breaking "no preemption" would require a way to roll back a partially complete transfer, which wouldn't be easy)

(c) **[3 marks]** Briefly describe (in words) how you would solve the problem.

(i) Break the hold-and-wait condition on queue locks: Introduce a new, global lock (e.g. pthread_mutex_t transfer_lock). In the transfer function, acquire the new lock before locking either queue involved in the transfer. Release the new lock only when the transfer is complete. Note that all other operations on the queues would have to hold the new transfer_lock as well to really break the hold-and-wait condition.

(ii) Break the circular wait condition: Use the addresses of the queue variables to create a total order of queues in the system. The transfer function could be re-written to always lock the queue with the lower address first. This is preferable to a single global lock as it allows greater concurrency.

(d) **[1 mark]** Which of the 3 strategies for dealing with deadlock does your solution use?

Both of the strategies above use deadlock **prevention** because they prevent one of the four conditions for deadlock from occurring.

Total marks = (50)

End of test

../continued