1 The Key

1. Interprocess Communication/Systems Programming (total 20%):

(a) (12%) Your company’s Unix based software would benefit if it could modify the size of an already allocated piece of shared memory. Unfortunately the Unix shared memory facility does not allow application programs to increase the size of a shared memory segment.

Your current implementation uses direct addressing of a shared memory segment with all access via pointers stored in their local memory (see Figure 1a). Over lunch, a coworker suggests using indirect addressing to get around this, by putting a pointer to the shared memory segment in shared memory (see Figure 1b). To protect the integrity of the pointer she suggests using a semaphore to guard the allocation of the shared memory. She has written the pseudocode in Figure 2 on a napkin over lunch. Discuss the advantages and disadvantages of this scheme.

The problem statement asks for a discussion of the scheme, while the pseudocode has several bugs, these could be potentially fixed if the scheme as a whole were sound. Unfortunately, as was discussed in class the scheme is not sound. The problem lies in the fact that the pointer to shared memory obtained by system call to shmget is a logical address (recall that a critical feature of operating systems memory management is to provide support for relocation). The address translation on the pointer is a function of the process (each process has its own logical address space), so when another process attempts to access the pointer, it will not be able to correctly compute the physical address of the shared memory segment.

If the scheme were not fundamentally flawed, then issues such additional pointer dereferencing operations might cause problems.

(b) (8%) Your company sells this software on many platforms, and is concerned about portability. Would the previously suggested approach be likely to work better or worse under another operating system?

The scheme might be difficult for any operating system supporting relocation, especially if virtual memory is used. The addressing schemes of most operating systems are likely to have the same problem. Even if we could get the physical address of the shared memory region, if swapping occurred due to virtual memory management, the physical address might change, so that is unlikely to solve the problem.

2. Synchronization (25%)
Figure 1: A Shared Memory Design in Problem 1a

```c
semaphore mutex := 1;    /* global mutex */
int shm_key = ftok(...); /* global key for shared memory management */

void *shmrealloc(int *shm_id_ptr, void *shm_ptr, int new_size)
{
    int old_size;
    void *old_shm_ptr;
    sem_wait(mutex);    /* limit access to one process at a time */
    old_size = shm_getsize(*shm_id_ptr); /* get shared memory segment’s size */
    if (old_size != new_size){
        /* change the segment’s size */
        /* create a new shared memory segment using same key */
        old_shm_ptr = *shm_ptr;
        shm_id_ptr = shmget(shm_key, new_size,
            SHM_RW | SHM_W | IPC_CREAT);
        /* ‘attach’ a pointer to the new shared memory’s segment */
        *shm_ptr = shmat(*shm_id_ptr,
            (void *NULL), /* use default location */
            0);        /* use default flags */
        shm_delete(*shm_id_ptr); /* deallocate old segment */
    }
    sem_signal(mutex);    /* release resource for user access */
}
```

Figure 2: Proposed Pseudocode for Problem 1a
(a) (15%) A fraternity has $N$ brothers who are about the same size and share an expensive suit, three couches and five operating systems text books. Each brother sleeps for a while, and then goes out on a job interview wearing the suit, removing the suit when they are done. The suit comes in two parts, the pants and the jacket, each must be put on (and removed) separately. In order to fall asleep each brother needs both an operating systems text book and a couch. A brother that wants to sleep or prepare for a job interview must wait until he can acquire the resources he needs to do so (although if he is kept waiting indefinitely he will get angry). Design a semaphore based solution. Be careful to avoid deadlock and starvation.

Note that we need to assume strong semaphores for a simple solution (recall that strong semaphores have FCFS queuing, while weak semaphores have some other queuing). The code might look like:

```c
/* shared memory variables */
sem bsem = 5, /* book synchronization variable */
csem = 3, /* couch synchronization variable */
psem = 1, /* pants synchronization variable */
jsem = 1; /* jacket synchronization variable */

process brother( int pid ) /* assume there are N brother processes */
{
  loop
    sem_wait(psem);   /* wait for pants */
    put_on_pants();
    sem_wait(jsem);   /* wait for jacket */
    put_on_jacket();
    go_to_interview();
    remove_jacket();
    sem_signal(jsem);
    remove_pants();
    sem_signal(psem);
    sem_wait(bsem);   /* wait for book */
    get_a_book();
    sem_wait(csem);   /* wait for couch */
    get_a_couch();
    rest(rest_duration);
    release_a_couch();
    sem_signal(csem);
    release_a_book();
    sem_signal(bsem);
  end loop;
}
```

Putting on the pants could be done after acquiring the jacket semaphore, but it is a bit less efficient. A similar argument could be for the book and couch. This can be a significant issue if the time it takes to prepare a resource for use after acquisition is large. It is also possible to merge the semaphores pairs for both the couch and book pair and the jacket pants pair but again this might be less efficient (and could cause maintenance problems if the number of instances a resource changes). I have developed a C program to solution corresponding to the pseudocode here in

"manhatt.public/oe421/examples/csrc/fratboys.c"

(b) (10%) Design a monitor based solution for the previous problem.

The solution to this problem depends on the monitor queuing discipline associated with
the condition variables. If it is FCFS, then it is easier, otherwise avoiding starvation is
difficult. The code (assuming FCFS queuing of the condition variables looks like).

```c
monitor Brother( int pid ){
  loop{
    while (num_pants_available > 0){
      wait(pants_available);
    }
    --num_pants_available;
    put_on_pants();
    while(num_jackets_available > 0){
      wait(jacket_available);
    }
    ++num_jackets_available;
    put_on_jacket();
    interview();
    take_off_jacket();
    ++num_jackets_available;
    signal(jacket_available);
    take_off_pants();
    ++num_pants_available;
    signal(pants_available);

    /* non critical section */
    while(num_books_available > 0){
      wait(book_available);
    }
    --num_books_available;
    get_book();
    while(num_couches_available > 0){
      wait(couch_available);
    }
    --num_couches_available;
    get_couch();
    rest(rest_duration);
    release_couch();
    ++num_couches_available;
    signal(couch_available);
    release_book();
    ++num_books_available;
    signal(book_available);
  } end loop;
}
```

3. Concurrency (20 %)

(a) (10 %) Use Bernstein's conditions to transform the following sequential code segment into a a
maximally parallel code segment using parbegin/parend notation. You can assume that you will
translate the code into a multithreaded environment (i.e. the threads can share data).

```c
x := x + z;
y := y * y;
w := w + x;
```
\[ z := y - 3; \]
\[ v := y + w; \]

Assuming that additions and subtractions take one unit of time and multiplications take 4 units of time, what is the minimum execution time.

Recall that Bernstein's conditions are used to detect whether or not two statements are mutually interfering. If the statements are not mutually interfering we will get results equivalent to a correct sequential evaluation when the statements are run in parallel. Consider a statement \( x \) let the set of addresses read by statement \( x \) be denoted \( R_x \) and the set of addresses written to by statement \( x \) be \( W_x \). Then two statements \( i \) and \( j \) are mutually non-interfering if they satisfy:

\[ (W_i \cap W_j) \cup (R_i \cap W_j) \cup (W_i \cap R_j) = \emptyset. \]  

where the left hand side of equation (1) is Bernstein's conditions. A sequential block of code with a single entry and exit point is called a basic block. Rewriting the basic block with line numbers attached gives:

1: \( x := x + z; \)
2: \( y := y * y; \)
3: \( w := w + x; \)
4: \( z := y - 3; \)
5: \( v := y + w; \)

And we can use a fictitious line 0 to denote the values held before entering the basic block.

One way to evaluate Bernstein's conditions is to build the directed acyclic dependency graph (DAG) of the basic block of code as seen in Figure 3. We can see that the dependencies clearly in the DAG, and get the solution:

```
parbegin /* stage 1 */
{ x := x + z; w := w + x; } /* 2 units */
y := y * y; /* 4 units */
parend; /* cost = max(2,4) = 4 */
parbegin /* stage 2 */
z := y - 3; /* 1 unit */
v := w + z; /* 1 unit */
parend; /* cost = max(1,1) = 1 */
```

The cost of a parallel section is the cost of the most expensive statement, so adding the costs of each stage together we see that the cost is \( 5 = 4 + 1 \) units.

(b) (10%) Barrier synchronization constructs are used for ensuring that all processes have arrived at a collective synchronization point. Write an efficient message passing barrier synchronization routine for \( N \) processes. If it helps you may assume that \( N \) is a power of 2.

We solve this sort of problem by applying a parallel reduction operator to the nodes for their synchronization. The well known Parallel Prefix (sometimes called a tournament tree) algorithm solves this sort of problem by recursively doing pair wise merges of neighboring synchronized regions until a single root of the tree is formed and releasing each node back down to the leaves. While not stated explicitly it is possible to assume the processes are all to all connected. The leaves of the tree are each processor, as shown in Figure 4. The code looks like:

```
/* Assume \( N \) is a power of 2 */

/* Concatenates synched regions into synched 
regions 2 * region_size long, stopping at \( N 
*/
Figure 3: Dependency DAG for Problem 3a

Figure 4: Example Parallel Prefix Tree
void synch_pair(int pid, mbox m, int region_size, int N)
{
    if (region_size < N){
        if (pid % region_size == 0){
            if (pid % (2 * region_size) != 0){
                send(pid - region_size, m, synch_request);
            } else {
                recv(pid + region_size, m, synch_request);
            }
        }
        synch_pair(pid, m, 2 * region_size, N);
        if (pid % (2 * region_size) != 0){
            recv(pid - region_size, m, synch_grant);
        } else {
            send(pid + region_size, m, synch_grant);
        }
    }
}

void barrier_synch(int pid)
{
    /* Base Case Every Process Self Synched */
    /* Assumes that the mbox m is initialized at startup,
    and released at normal termination */
    synch_pair(pid, m, 1, N);
}

4. (20 %) Deadlock

(a) (10 %) Discuss (briefly) each of the conditions necessary for deadlock and the cost of preventing each from occurring for each of the following resources:

i. A Printer
ii. Main Memory
iii. A message buffer.

Prevention consists of disallowing some of the conditions necessary for deadlock. Recall the four necessary conditions of deadlock are:

i. Mutual Exclusion — This means not sharing an allocated resource between processes. A printer is inherently not sharable but spooling might help, rather than doing an allocation. Main memory may be among readers, so you might partially overcome mutual exclusion.

ii. Hold and Wait (Partial Allocation) — Partial allocation or the hold and wait condition refers to a process stalling while it concurrently needing many resource but having only some of its current requirements allocated. Since a printer tends to be a write only device, spooling of requests might alleviate the blocking aspect of the hold and wait. Hold and wait can be reduced on memory if preemption occurs (i.e., like virtual memory or swapping). Message buffers could be shared in a serial fashion (since messages are reusable resources).

iii. Nonpreemption — Preemption means (i) interrupting an existing allocation, (ii) allocating temporarily to another process action and (iii) restoring the state of the interrupted process. Printers are not suitable for preemption. Memory is highly suited to preemption, since its state can be preserved and restored readily. Message buffers are not
well suited for preemption, however in the case of protocols with retries, a preempted
message might be tolerated (if sufficiently infrequent).

iv. Circular Wait — This means that there is a cycle in the resource request graph
of blocked processes. Prevention is typically accomplished by imposing a consistent
system wide ordering on resource requests. This may be suited to certain applications
and is really not a function of the resource type.

(b) (10 %) Consider a system with a total of 4 identical instances of a resource to be shared among 3
processes, each requiring 2 instances of the resource. Each resource can be requested one unit at
a time. Show whether or not the system can deadlock, i.e., does it have deadlock unsafe states?

This system is free from deadlock. The system would be deadlocked if all 4 units were
allocated and no process could run. Suppose all 4 units were allocated, then by the
pigeon hole principle some process must have more than 1 unit of the resource allocated
to it, and that process could run. (If you assumed that 2 units of the resource could
be simultaneously requested and that grants were done piecemeal you could obtain a
deadlock condition).

5. (20 %) Intro to Networking

(a) (10 %) What compare and contrast the costs of a network model having relatively few layers (say
a 4 layer network model, such as TCP/IP) and a network model having many layers (say a a 7
layer model such as the ISO/OSI model).

Having more layers tends to promote ease of use, since you can apply higher levels of
abstraction during coding. However additional layers induce a performance penalty of
extra system calls which are not required of systems with fewer layers. On the other
hand, a system with fewer layers tends to be more difficult to program and it can be
more difficult to port solutions. Consider an extreme case where your software interacts
directly with the network control (just say at the data link). Moving the software to
another platform may be difficult then since the datalink layer may have many hardware
dependencies (timing, byte ordering, buffer size, ...) embedded in it.

(b) (10 %) Give the Point to Point Protocol (PPP) packet containing the message (note there are
unprintable ASCII characters represented by C style backslash escape sequences, those with a
leading 0x are in hexadecimal, otherwise octal). The following text is the data to send:

Darn \0x7e Keyboard \0x7d\0x7f At Least I have Completed The Final Exam

The generated packet should read:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Address</th>
<th>Control</th>
<th>Protocol</th>
<th>Information</th>
<th>CRC</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7e</td>
<td>0xff</td>
<td>0x00</td>
<td>0x02</td>
<td>Darn \0x7e Keyboard \0x7d\0x7f At Least I have Completed The Final Exam</td>
<td>CRC</td>
<td>0x7e</td>
</tr>
</tbody>
</table>