COMP3911 Suggested Exercises #1a

These suggested exercises are not to be handed in for marking, but should be used as a study aid. Solutions will be posted on the course web site.

1) Enumerate the possible output sequences generated by the following C function when it is executed concurrently by 2 processes.

Assume that each numbered line of C code is implemented as a single indivisible machine operation on the target architecture, in order to minimize the number of output sequences. Note that there is no loop in the code and each process is assumed to call the function once, so that there is a finite number of possible outcomes. (NB: The integer variable "i" is shared by the processes.)

```
        int i = 1;
1         up() {
2                 if (i >= 2)
3                         i--;
4                 i++;
5                 printf("%d ", i);
6         }
```

2) The following two C functions are called repeatedly by several processes concurrently, manipulating the shared variable "chr". The output sequence should always be:

a b c c b a a b c c b . . .

```
        char chr = ’a’;
        int flag = 1;
        semaphore_t crit = 1;

  void
  addc() {
          WAIT_SEM(crit);
          while (flag == 0) {
                  SIGNAL_SEM(crit);
                  WAIT_SEM(crit);
          }
          printf("%c ", chr);
          chr++;
           if (chr > ’c’) {
                   SIGNAL_SEM(crit);
                   flag = 0;
                   chr = ’c’;
           } else
                   SIGNAL_SEM(crit);
    }
    void
    subc() {
            WAIT_SEM(crit);
            while (flag == 1) {
                    SIGNAL_SEM(crit);
                    WAIT_SEM(crit);
            }
            printf("%c ", chr);
            chr--;
            if (chr < ’a’) {
                  SIGNAL_SEM(crit);
                  flag = 1;
                  chr = ’a’;
            } else
                  SIGNAL_SEM(crit);
    }
```

a) Show whether or not the functions work correctly.

b) If they don’t work correctly, fix them by adding/removing semaphore operations (**WAIT_SEM()** and **SIGNAL_SEM()**) on the semaphore "crit", only. (Do not make any other changes to the code, including declaring other semaphores.)

3) The following two C functions are called repeatedly by several processes. They share the variable "i", which is only used by these two functions.

Add semaphore(s) and semaphore operations (**WAIT_SEM()** and **SIGNAL_SEM()**) to the functions to ensure that the output generated is what is listed in a), b) and c). When doing this, you may not make any other changes to the code, except for adding semaphores and semaphore operations.

```
int i = 1;
up() {
        printf("%d ", i);
        i++;
}
down() {
        printf("%d ", i);
        i--;
}
```

a) 1 2 1 2 1 2 1 2 . . .

b) 1 2 3 4 3 4 5 4 5 6 5 6 7 6 7 8 . . .

c) 1 1 1 1 1 1 1 1 . . .

4) The following C functions implement a variation on the Producer/Consumer problem and are both called by several processes concurrently. This variation of the Producer/Consumer problem labels entries in the Queue for a specific process by number. The processes are numbered 0<-->9. Add seaphores and semaphore operations to ensure that the functions operate correctly. You may not make any other changes to the code. Correct operation would imply that no entries are lost or duplicated and that the "Panic..." printf's never occur. (And yes, an array of semaphores is allowable.)

```
#define     QSIZE     4

/*
 * This union holds the value for part a or b.
 */
union dataval {
        int data;       /* Data value for process */
        struct {
                /*
                 * These fields are only used for part b)
                 */
                int read_write;     /* Read vs write flag */
                int disk_blkno;     /* Block# on disk */
                char *memory_addr;  /* Memory address for data buffer */
        } io;
};
struct queue {
        int num;                /* Process number */
        union dataval val;      /* Data being passed through q */
} q[QSIZE];

/*
 * This function adds an entry to the queue for the specified process.
 */
void
add_queue(num, val)
        int num;
        union dataval val;
{
        int i;

        for (i = 0; i < QSIZE; i++)
                if (q[i].num == -1)
                        break;
        if (i == QSIZE)
                printf("Panic: no empty slots in queue\n");
        q[i].num = num;
        q[i].val = val;
}

/*
 * This functions gets an entry from the queue for the specified process.
 */
union dataval
get_queue(num)
        int num;
```

```
{
        int i;
        union dataval retval;

        for (i = 0; i < QSIZE; i++)
                if (q[i].num == num)
                        break;
        if (i == QSIZE)
                printf("Panic: No entry for num=%d\n", num);
        retval = q[i].val;
        q[i].num = -1;
        return (retval);
}
```

Part b) Now, assume instead that the above add_queue() and get_queue() functions are in a BSD kernel, with the get_queue() function being called by the following interrupt service routine for a disk controller, to get a read/write disk operation off the queue for a specific disk. The function doio() is called by the kernel file system code to do an I/O operation on a disk, which it queues for the drive by calling add_queue(). (The "num" now identifies which disk drive the I/O operation request is for, instead of which process.) Use tsleep()/wakeup(), along with spltty()/splx() to control masking off of the disk interrupt, to make the routines (add_queue, get_queue, diskintr, doio, startio) work correctly. (Semaphore operations are not available in a BSD kernel.)

```
/*
 * This function is called by the kernel file system code to do an I/O
 * operation on a disk.
 */
int busy[10];
void
doio(drive, read_write, blkno, mem_addr)
        int drive;
        int read_write;
        int blkno;
        char *mem_addr;
{
        union dataval val;

        val.io.read_write = read_write;
        val.io.disk_blkno = blkno;
        val.io.memory_addr = mem_addr;
        add_queue(drive, val);
        if (busy[drive] == 0)
                startio(drive, val);
        tsleep(mem_addr, PZERO, "diskio", 0);
}

/*
 * This function is called via the IRQ for the disk controller. inb(0x170)
 * returns which drive has just completed an I/O operation and is ready
 * to do the next one.
 */
void diskintr()
{
        int drive;
        static union dataval diskop[10];

        drive = inb(0x170);
        /*
         * Note completion of the I/O operation for the sleeping process
         * that queued it.
         */
        ...
        /*
         * Get next I/O request off queue and start it with appropriate
         * commands on the port.
         */
        diskop[drive] = get_queue(drive);
        if (diskop[drive].io.read_write != -1)
                startio(drive, diskop[drive]);
}

/*
 * This function starts an I/O operation on a drive.
 */
void
startio(drive, op)
        int drive;
        union dataval op;
{
        outb(0x174, op.io.read_write);
        outb(0x176, op.io.disk_blkno);
        outb(0x178, op.io.mem_addr);
        outb(0x17a, drive);
        outb(0x17c, BLKSIZE);
        busy[drive] = 1;
}
```

You may assume the BSD kernel is running on a single CPU system. Note that the C keyword "static" declares a variable that is static instead of automatic. That means that the variable is allocated memory only once and uses that memory location for all calls to the function, instead of being instantiated each function call. In other words, it behaves like a global variable, but is only visible within the scope of the function it is declared in. (Just to make it confusing, the same keyword "static" can be applied to a global variable. In this case it simply restricts the scope of the variable to the C file it is declared in.)

5) For the Dining Philosophers problem (if you don't know the problem, refer to the textbook where it is listed in the index (it moves around a bit, but is present in every edition)), implement a solution for the philosopher process using semaphores and semaphore operations, that avoids deadlock by having the philosophers try and get the forks in different orders. The forks/chopsticks should be implemented as semaphores and you may also use whatever other semaphores that you need for the solution.

Part b) Now, implement a solution that avoids deadlock by enforcing a strict ordering of 0..4 on the order in which the philosophers eat.