COMP3911 Suggested Exercises #1 - Solutions

1) Since each line of C is assumed to be a single indivisible machine instruction, statement line #s can be used to indicate order of execution going down the page:

Process	Line #s	i	Output generated
Case1			
P1	1-3	1	
P2	1-3	1	
P1	4-5	2	2
P2	4-5	3	3
Case2			
P1	1-3	1	
P2	1-3	1	
P1	4	2	
P2	4	3	
P1	5	3	3
P2	5	3	3
Case3			
P1	1-5	2	2
P2	1-5	2	2
Case4			
P1	1-4	2	
P2	1-3	1	
P1	5	1	1
P2	4-5	2	2

I don't believe there are any sequences that will generate output other than the four output sequences listed above:

Case1: 2 3
Case2: 3 3
Case3: 2 2
Case4: 1 2

In general, solving this kind of question requires an exhaustive examination of all possible sequences of execution by the N processes. Since all the processes execute the same code sequence, the processes are not distinct, such that the labelling of P1, P2, ... Pn is just a convenience and the ordering of P1, P2, ... Pn is not relevent.

2) a) The following concurrent execution results in:

```
                    flag     chr       output
                    1        a
P1 calls subc()
loops at #20-23
P2 calls addc()
    doesn't loop
    at #8                          a
    aft #9                  b
    returns
P3 calls addc()
    doesn't loop
    at #8                          b
    aft #9                  c
    returns
P4 calls addc()
    doesn't loop
    at #8                          c
    aft #9                  d
    at #11 SIGNAL_SEM()
    aft #12          0
P1 comes out of loop #20-23
    at #24                          d
```

This sequence of execution generates the output:

a b c d

which is not correct and therefore the functions don't work correctly. There are, of course, many other execution sequences that generate incorrect output. Finding any one of these demonstrates that the code is incorrect.

2) b)
Move the SIGNAL_SEM() call at #11 to after #13 and the SIGNAL_SEM() call at #27 to after #29, so the variables are updated before another process is allowed to leave WAIT_SEM().

3) a) Generate 1 2 1 2 1 2 1 2 . . .

```
semaphore_t add_sem = 1, sub_sem = 0;
int i = 1;

void up() {
        WAIT_SEM(add_sem);
        printf("%d ", i);
        i++;
        SIGNAL_SEM(sub_sem);
}

void down() {
        WAIT_SEM(sub_sem);
        printf("%d ", i);
        i--;
        SIGNAL_SEM(add_sem);
}
```

Note that, since the semaphores add_sem and sub_sem never have a value greater than 1, the WAIT_SEM() calls enforce mutual exclusive over the critical sections as well as enforce an appropriate partial ordering on execution.

3) b) Generate the output 1 2 3 4 3 4 5 4 5 6 5 6 7 6 7 8 . . .

```
semaphore_t crit1 = 1, crit2 = 1, sup = 3, sdown = -1;
int i = 1;

void up() {
        WAIT_SEM(crit1);
        WAIT_SEM(sup);
        printf("%d ", i);
        i++;
        SIGNAL_SEM(sdown);
        SIGNAL_SEM(crit1);
}

void down() {
        WAIT_SEM(crit2);
        WAIT_SEM(sdown);
        WAIT_SEM(sdown);
        printf("%d ", i);
        i--;
        SIGNAL_SEM(sup);
        SIGNAL_SEM(sup);
        SIGNAL_SEM(crit2);
}
```

Note that, for this solution, the number of WAIT_SEM() and SIGNAL_SEM() calls are different in up() vs down() so that each will execute a different number of times and result in the appropriate output sequence. I also suggest that you figure out how the solution breaks if you take away either crit1 or crit2 along with the WAIT_SEM() and SIGNAL_SEM() operations on them. See the Note at the end of the solution for 3)c) for more on this.

3) c) Generate the output 1 1 1 1 1 1 1 1 . . .

```
semaphore_t up = 1, down = 0, add = 0, crit = 1;
int i = 1;

void up() {
        WAIT_SEM(crit);
        WAIT_SEM(up);
        printf("%d ", i);
        SIGNAL_SEM(down);
        WAIT_SEM(add);
        i++;
        SIGNAL_SEM(crit);
}

void down() {
        WAIT_SEM(down);
        printf("%d ", i);
        i--;
        SIGNAL_SEM(add);
        SIGNAL_SEM(up);
}
```

Note that it required a "3 step" handshake in order to enforce the correct ordering of execution through up() and down(). The crit semaphore was also required, to ensure that a process didn't go past WAIT_SEM(up) when another process was midway through up() waiting at WAIT_SEM(add). For correct operation, only one process can be in up() at a time and the WAIT_SEM(up) was not sufficient to enforce this, even though the value of up never exceeds 1. The problem is that the WAIT_SEM(add) is inside the critical section, so a separate semaphore crit was required to enforce mutual exclusion within the critical section in up().

4) Solution with sempahores added.

```
semaphore_t crit = 1, not_full = QSIZE,
        has_entry[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
/*
 * This function adds an entry to the queue for the specified process.
 */
void
add_queue(num, val)
        int num;
        union dataval val;
{
        int i;

        WAIT_SEM(not_full);
        WAIT_SEM(crit);
        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;
        SIGNAL_SEM(has_entry[num]);
        SIGNAL_SEM(crit);
}

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

        WAIT_SEM(has_entry[num]);
        WAIT_SEM(crit);
        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;
        SIGNAL_SEM(not_full);
        SIGNAL_SEM(crit);
        return (retval);
}
```

Note that, since the queue is searched for an entry, that add_queue() and get_queue() share the same elements of q[]. This implies that manipulation of q[] forms a critical section.

I suggest that you see how the solution breaks if the order of the WAIT_SEM() calls is reversed.

Part b), in the BSD kernel, queuing disk I/O ops:

```
/*
 * Must be called with disk interrupts masked off, since it manipulates
 * the queue that is also manipulated by get_queue(), which is called
 * from the interrupt handler diskintr().
 */
void
add_queue(num, val)
	int num;
	union dataval val;
{
	int i;

	/*
	 * Since there might be several processes tsleeping here for a
	 * queue slot and the wakeup occurs when one slot becomes available,
	 * there must be a loop here.
	 * The first process waking up gets the slot, kinda like the
	 * "early bird gets the worm".
	 */
	do {
		for (i = 0; i < QSIZE; i++)
			if (q[i].num == -1)
				break;
		if (i == QSIZE)
			tsleep(q, PZERO, "diskq", 0);
	} while (i == QSIZE);
	q[i].num = num;
	q[i].val = val;
}

/*
 * This functions gets an entry from the queue for the specified process.
 * Since it is called from diskintr(), disk interrupts are always masked
 * off.
 */
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) {
		retval = q[i].val;
		q[i].num = -1;
		wakeup(q);
```

```
        } else
                retval.io.read_write = -1;  /* no entry for this drive */
        return (retval);
}

/*
 * Note what disks are busy doing an I/O operation.
 */
int busy[10];

/*
 * Called from the file system code to do an I/O (read or write) of a block
 * on a disk.
 * The arguments are:
 *      drive - which disk drive on the controller
 *      read_write - read == 0, write == 1
 *      blkno - disk block number
 *      mem_addr - physical memory address of buffer for block
 */
void
doio(drive, read_write, blkno, mem_addr)
        int drive;
        int read_write;
        int blkno;
        char *mem_addr;
{
        union dataval val;
        int s;

        val.io.read_write = read_write;
        val.io.disk_blkno = blkno;
        val.io.memory_addr = mem_addr;
        s = spltty();
        add_queue(drive, val);
        if (busy[drive] == 0)
                startio(drive, val);
        splx(s);
        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.
```

```
         */
        busy[drive] = 0;
        wakeup(diskop[drive].io.memory_addr);
        /*
         * 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.
 * (Since it manipulates the disk controller hardware, it must be called
 *  with disk interrupts masked off.)
 */
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;
}
```

5) Dining Philosophers must acquire the forks/chopsticks in different orders to avoid deadlock. Although there are other orders, simply having even numbered philosophers (real philosophers would probably not like to be referred to by a number, but...) get the left fork first vs odd numbered philosophers that get the right fork first.

```
semaphore_t fork[5] = { 1, 1, 1, 1, 1 };

void philosopher(int i) {
	do {
		if (i % 2) {
			/* odd numbered philosopher */
			WAIT_SEM(fork[(i + 1) % 5]);
			WAIT_SEM(fork[i]);
		} else {
			/* even numbered philosopher */
			WAIT_SEM(fork[i]);
			WAIT_SEM(fork[(i + 1) % 5]);
		}
		- eat
		SIGNAL_SEM(fork[i]);
		SIGNAL_SEM(fork[(i + 1) % 5]);
		- think big thoughts
	} while (1);
}
```

Although this is best seen by drawing a diagram, the trick is that you can only get deadlock if all 5 philosophers have one fork and all need the other fork (ie. all stuck in the second WAIT_SEM()). By using the above ordering of WAIT_SEM()s, philosopher 0 gets fork 0 first, philosophers 1 and 2 compete for fork 2 first, and philosophers 3 and 4 compete for fork 4 first. As such, it is impossible for each of the 5 philosophers to get a different fork first, and therefore no deadlock can occur.

Dining Philosophers eating in order to avoid deadlock.

```
semaphore_t fork[5] = { 1, 1, 1, 1, 1 }, next[5] = { 1, 0, 0, 0, 0 };

void philosopher(int i) {
	do {
		WAIT_SEM(next[i]);
		WAIT_SEM(fork[i]);
		WAIT_SEM(fork[(i + 1) % 5]);
		- eat
		SIGNAL_SEM(fork[i]);
		SIGNAL_SEM(fork[(i + 1) % 5]);
		SIGNAL_SEM(next[(i + 1) % 5]);
		- think big thoughts
	} while (1);
}
```

Note that, although I included the fork[] semaphores so that it would look like the solution in the first part, they are not really required, since this solution only allows one philosopher to eat at a time.

Since only one philosopher is eating at a time, deadlock cannot occur, since all 5 forks

cannot be in use at one time. (All that is really needed is to allow no more than 4 philosophers to eat at one time.)