```
semaphore:
wait_for_update() {
    P(updated)
}
signal_update_occured(){
    V(updated)
}
```

Tutorial Week 3

```
condition variable:
wait_for_update(){
    lock_acquire(1)
    while(flag == 0)
        cv_wait(cv, 1)
    lock_release(1)
}
```

```
signal_update_occured(){
    lock_acquire(1)
        flag = 1
        cvsignal(cv, 1)
    lock_release(1)
}
```

Questions

Synchronisation Problems

The following problems are designed to familiarise you with some of the problems that arise in concurrent programming and help you learn to identify and solve them.

semaphore counter
1: if the resource is accessible immediately
0: if the shared resource needs to be signaled ready first
N > 1: we have N spaces for processess to access the resource

Coordinating activities

1. What synchronisation mechanism or approach might one take to have one thread wait for another thread to update some state? semaphore, condition variable

---

2. A particular abstraction only allows a maximum of 10 threads to enter the "room" at any point in time. Further threads attempting to enter the room have to wait at the door for another thread to exit the room. How could one implement a synchronisation approach to enforce the above restriction?

---

3. Multiple threads are waiting for the same thing to happen (e.g. a disk block to arrive from disk). Write pseudo-code for a synchronising and waking the multiple threads waiting for the same event.

Identify Deadlocks

4. Here are code samples for two threads that use semaphores (count initialised to 1). Give a sequence of execution and context switches in which these two threads can deadlock.

   Propose a change to one or both of them that makes deadlock impossible. What general principle do the original threads violate that causes them to deadlock?

if me() executes P(mutex) while you() executes P(data), then me() will be waiting for P(data) and you() will be waiting for P(mutex). and a deadlock will occur.

```
semaphore *mutex, *data;

void me() {
        P(mutex);
        /* do something */

        P(data);
        /* do something else */

        V(mutex);

        /* clean up */
        V(data);
}

void you() {
        P(data)
        P(mutex);

        /* do something */

        V(data);
        V(mutex);
}
```

More Deadlock Identification

5. Here are two more threads. Can they deadlock? If so, give a concurrent execution in which they do and propose a change to one or both that makes them deadlock free.

```
lock *file1, *file2, *mutex;

void laurel() {
        lock_acquire(mutex);
        /* do something */

        lock_acquire(file1);
        /* write to file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(file1);
        lock_release(mutex);

        /* do something */

        lock_acquire(file1);

        /* read from file 1 */
        /* write to file 2 */

        lock_release(file2);
        lock_release(file1);
}

void hardy() {
        /* do stuff */

        lock_acquire(file1);
        /* read from file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(file1);
        lock_release(file2);

        lock_acquire(mutex);
        /* do something */
        lock_acquire(file1);
        /* write to file 1 */
        lock_release(file1);
        lock_release(mutex);
}
```

Synchronised Lists

6. Describe (and give pseudocode for) a synchronised linked list structure based on thread list code in the OS/161 codebase (`kern/thread/threadlist.c`). You may use semaphores, locks, and condition variables as you see fit. You must describe (a proof is not necessary) why your algorithm will not deadlock.

   In a general sense, the interface to the synchronised list is as follows.

```
    init(list_t *);
    add_head(list_t *list, node_t *node);
    add_tail(list_t *list, node_t *node);
    remove_head(list_t *list, node_t **node);
    remove_tail(list_t *list, node_t **node);
    insert_after(node_t *in_list, node_t *new_node);
    insert_before(node_t *in_list, node_t *new_node);
```

```
remove(node_t *in_list);
```

Make sure you clearly state your assumptions about the constraints on access to such a structure and how you ensure that these constraints are respected.

In addition to a single lock solution, consider a solution involving a lock per node in the list. The instructive cases are `insert_after()` and `insert_before()`, and `remove()`

The thread subsystem in OS/161 uses a linked list of threads to manage some of its state (`kern/thread/threadlist.c`). This structure is not synchronised. Why not?

Concurrency and Deadlock

7. For each of the following scenarios, one or more dining philosophers are going hungry. What is the condition the philosophers are suffering from?

    a. Each philosopher at the table has picked up his left fork, and is waiting for his right fork
    b. Only one philosopher is allowed to eat at a time. When more than one philosophy is hungry, the youngest one goes first. The oldest philosopher never gets to eat.
    c. Each philosopher, after picking up his left fork, puts it back down if he can't immediately pick up the right fork to give others a chance to eat. No philosopher is managing to eat despite lots of left fork activity.

---

8. What is starvation, give an example?

---

9. Two processes are attempting to read independent blocks from a disk, which involves issuing a *seek* command and a *read* command. Each process is interrupted by the other in between its *seek* and *read*. When a process discovers the other process has moved the disk head, it re-issues the original *seek* to re-position the head for itself, which is again interrupted prior to the *read*. This alternate seeking continues indefinitely, with neither process able to read their data from disk. Is this deadlock, starvation, or livelock? How would you change the system to prevent the problem?

---

10. Describe four ways to *prevent* deadlock by attacking the conditions required for deadlock.

---

11. Answer the following questions about the tables.

    a. Compute what each process still might request and display in the columns labeled "still needs".
    b. Is the system in a safe or unsafe state? Why?
    c. Is the system deadlocked? Why or why not?
    d. Which processes, if any, are or may become deadlocked?
    e. Assume a request from p3 arrives for (0,1,0,0)
        1. Can the request be safely granted immediately?
        2. In what state (deadlocked, safe, unsafe) would immediately granting the request leave the system?
        3. Which processes, if any, are or may become deadlocked if the request is granted immediately?

available			
r1	r2	r3	r4
2	1	0	0

	current allocation				**maximum demand**				**still needs**			
process	r1	r2	r3	r4	r1	r2	r3	r4	r1	r2	r3	r4
p1	0	0	1	2	0	0	1	2				

p2	2	0	0	0	2	7	5	0				
p3	0	0	3	4	6	6	5	6				
p4	2	3	5	4	4	3	5	6				
p5	0	3	3	2	0	6	5	2				

12. Solve the Dining Philosopher's problem below using locks and a different strategy to the one shown in lectures.

```
void take_both_forks(unsigned long phil_num)
{

/*
 * Take forks ensures mutually exclusive access to two forks
 * associated with the philosopher.
 *
 * The left fork number = phil_num
 * The right fork number = (phil_num + 1) % NUM_PHILOSPHERS
 */

}



void release_forks(unsigned long phil_num)
{
/*
 * Releases forks releases the mutually exclusive access to the
 * philosophers forks.
 */
```