1 – Tasks and task states (6 pts)

In class, we learned that there are 3 major task states.

a) What are these task states?

b) List all feasible transitions between these states and explain why these transitions occur?

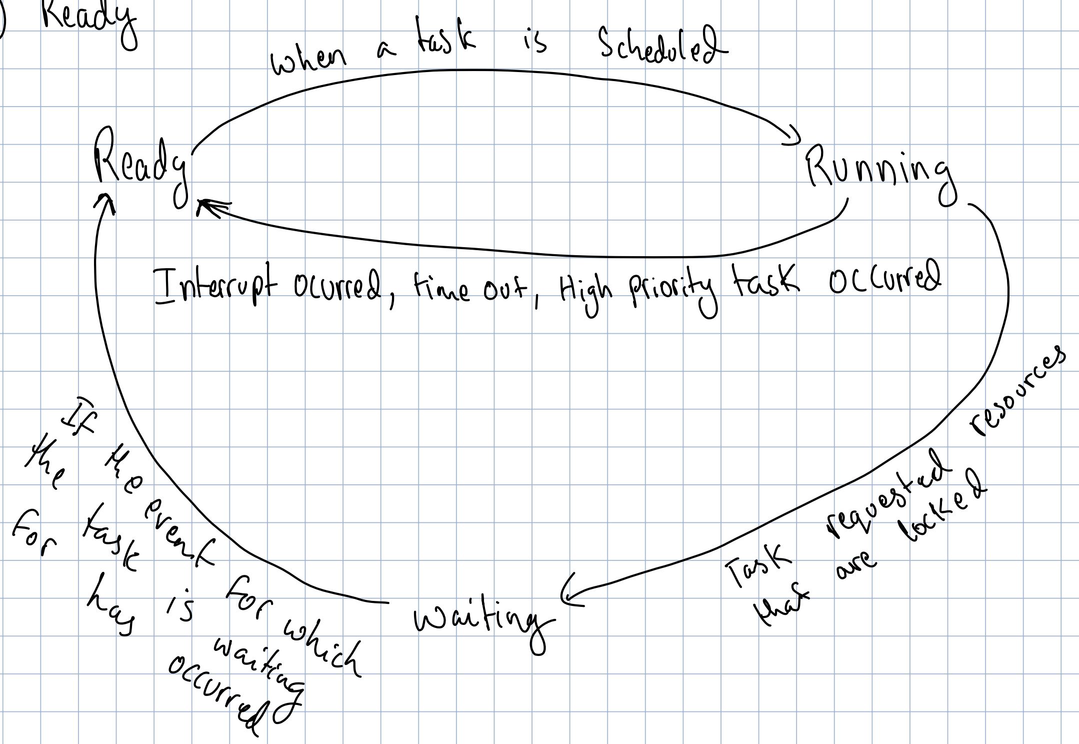
c) What are the possible extensions to these major task states?

A) 1) Running

2) Waiting

3) Ready

B)



c) the Ready and Running states would have the New and terminated States as extensions Respectively. the New State is for when a new process is created and the terminated State is for when the currently running task is terminated

2 – Deadlock (15 pts)

a) (5 pts) In class, we saw that there are two major ways to deal with deadlocks. List these and provide an example for each. Then, talk about the feasibility of their implementations in real systems.

b) (10 pts) Assume we have 4 processes, $\{P_1, P_2, P_3, P_4\}$, and 2 resources $\{R_1, R_2\}$. The current resource allocation matrix is follows:

Resources	R_1	R_2
P_1	1	3
P_2	1	2
P_3	1	2
P_4	2	0

How you should read this table: P_1 currently has 1 R_1 and 3 R_2 resources (similar for other processes). The following matrix shows the additional resource requests by each process:

Resources	R_1	R_2
P_1	1	2
P_2	4	3
P_3	1	7
P_4	5	1

How you should read this table: P_1, in addition to the resources it has, wants 1 R_1 and 2 R_2 resources (similar for other processes). Finally, the below is the availability vector of the resources:

$$\{R_1, R_2\} = \{1, 4\}$$

This vector shows how many of resource the system has in addition to the current resource allocations. Question: Is the system with this current state deadlocked? What changes if we update the availability vector as $\{2, 4\}$.

A) One way is Avoid or prevent deadlocks and other way is to Overcome deadlocks. The first method involves ways of preventing a deadlock from Occurring by Monitoring for them and Seeing if there is a way a deadlock may occur. This creates additional overhead. The second method involves recovering from a deadlock by Killing the deadlocked task. This also increases overhead.

(B) $\{R_1, R_2\} = \{1, 4\}$

P_1 uses 1 R_1 and 2 R_2 So R_1 is used up and R_2 has 2 left $\{0, 2\}$

P_1 is done running $\{2, 7\}$

P_3 uses 1 R_1 and 7 R_2 $\{1, 0\}$

P_3 is done running $\{3, 9\}$ and not enough resources for P_2 and P_4

System is dead locked

$\{R_1, R_2\} = \{2, 4\}$

P_1 uses 1 R_1 and 2 R_2 $\{1, 2\}$
P_1 is done running $\{3, 7\}$

P_3 uses 1 R_1 and 7 R_2 $\{2, 0\}$
P_3 is done running $\{4, 9\}$

P_2 uses 4 R_1 and 3 R_2 $\{0, 6\}$
P_2 is done Running $\{5, 11\}$

P_4 uses 5 R_1 and 1 R_2 $\{0, 10\}$

final $\{7, 11\}$

No dead lock.

3 – Processes (15 pts)

i) (5pts) How many times does the following program print ***hello***?

```
1.    #include <stdio.h>
2.    #include <unistd.h>
3.
4.    main() {
5.          int i;
6.          for (i = 0; i < 3; i++) {
7.                fork();
8.          }
9.          printf("hello\n");
10.         //DO SOME WORK
11.   }
```

ii) (10 pts) How can you modify the above code to differentiate all available processes in the system after line 10?

(i) hello will print 8 times

(ii) We would need to create an array and have each element assigned to each fork() call within a for loop

4 – Concurrency (10 pts)

For each program below, indicate whether or not it could have a deadlock, a race condition, or both. If so, explain why. Assume that the function **thread1** runs in one thread and the function **thread2** runs in another thread, and that the following data are shared between threads, and initialized as indicated prior to execution in each case:

```
semaphore m = 1;
semaphore n = 1;
int x = 0;
int y = 0;
```

a)

```
thread1() {                 thread2() {
  wait(&m);                   wait(&n);
  x ++;                       x ++;
  wait(&n);                   wait(&m);
  y ++;                       y ++;
  signal(&n);                 signal(&m);
  signal(&m);                 signal(&n);
}                           }
```

b)

```
thread1() {                 thread2() {
  wait(&m);                   wait(&m);
  x ++;                       x ++;
  wait(&n);                   wait(&n);
  signal(&m);                 signal(&m);
  y ++;                       y ++;
  signal(&n);                 signal(&n);
}                           }
```

(A) A deadlock and Race Condition can occur since thread1 and thread2 have wait for m and n and this loop can cause a deadlock. There is a circular dependency. A race condition can occur as they can try to Access the Same Variable at the Same time

(B) There is no circular dependency of resources and variable access would be Sequential. there would be no Deadlock or race Conditions.

5 – Semaphores (10 pts)

The semaphore wait and signal operations are defined as indivisible (or atomic) operations.

a) Why do they have to be indivisible?

b) Because they are indivisible does that mean that all other processes running on a multiprocessor system must stop when a wait or signal operation is executed? Explain why or why not.

(A) we cant have multiple wait and Signals operations happening at the Same time So they need to be defined as indivisible.

(B) No, Only the processes that use the semaphore at the Same time would need to wait and other tasks would continue.

6 – Process Synchronization (10 pts)

Show all possible output sequences resulting from running these two processes synchronously. Assume that the processes will not terminate before executing all the instructions. Explain your answer.

```
int x = 0;
int y = 0;
```

"initialization"

Process A	Process B
while(x==0){};	printf("b");
printf("a");	x=1;
y=1;	while(y==0){};
printf("d");	x=0;
while(x==1){};	while(y==0){};
y=0;	printf("c");
printf("e");	
y=1;	

(1) b a d c e

(2) b a d e c

(3) b a c d e   are the possible outputs of the 2 Processes.

7 – Communicating Processes (10 pts)

Given the following communicating process network descriptions and the following input sequence:
(5 pts) i) explain how the network works by drawing a diagram and
(5 pts) ii) provide the output sequence.

Note: 1) Assume that the read operations are blocking (i.e. process cannot continue without getting the message from a buffer) and write operations are non-blocking. 2) You can assume that the channels have enough bandwidth so that there is no overwriting necessary.

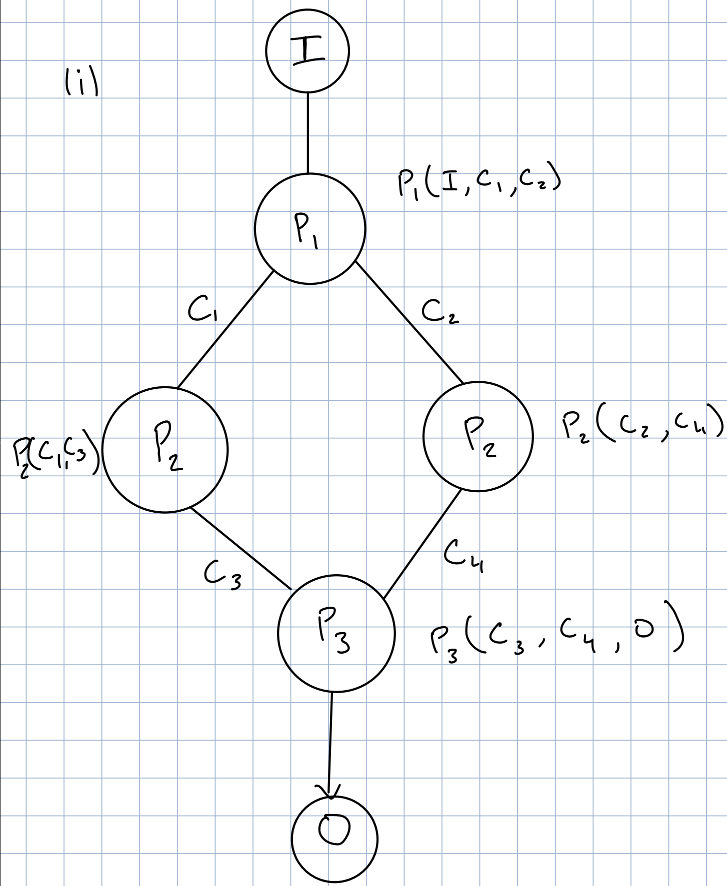
```
Process p1 (input int a, output int x, output int y){
    int k;
    loop
        k = a.receive();
        if k mod 2 == 0 then
            x.send(k);
        else
            y.send(k);
        end if;
    end loop;
}
Process p2 (input int a, output int x){
    int k;
    loop
        k = a.receive();
        x.send(k);

    end loop;
}
Process p3 (input int a, input int b, output int x){
    int k;
    boolean alternate = false;
    loop
        if alternate then
            k = a.receive();
        else
            k = b.receive();
        end if;
        x = send(k);
        alternate = !alternate;
    end loop;
}

channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

Input sequence = [3, 3, 6, 9, 11, 3, 6, 6, 1, 0, 3, 7]

(i)



(ii) [3, 6, 3, 6, 9, 6, 11, 0, 3]

8 - Synchronization with semaphores (12 pts)

The below shows the sequence of semaphore operations at the beginning and at the end of the tasks A, B, C.
Determine for the 4 cases, i, ii, iii, and iv given below, whether, and in which sequence, the tasks are executed, using the initializations of the semaphore variables given in the respective table. Each column under the cases corresponds to a different initialization with semaphore given values. For example, in case i), we have initially SA=2, SB=0, and SC=2. If a task gets stuck, point out where (in which line) it gets stuck.

Task A	**Task B**	**Task C**
wait(SA)	wait(SB)	wait(SC)
wait(SA)	…	wait(SC)
wait(SA)	…	wait(SC)
…	…	…
…	signal(SC)	…
…	signal(SA)	signal(SB)
signal(SB)		signal(SB)
END	**END**	**END**

Cases:

Semaphore	**Cases**			
	i)	**ii)**	**iii)**	**iv)**
SA	2	3	2	0
SB	0	0	1	0
SC	2	2	1	3

(i) Task A: there are 3 wait(SA) so its stuck

Task B: Stuck at wait(SB)

Task C: Stuck at 3rd wait(SB)
Nothing is executed

ii) Task A: Signals SB, A executes

Task B: Signals SC, B executes

Task C: C executes
executeded in order (A→B→C)

iii) Task A: Stuck at 3rd wait(SA)
Task B: executes, Signals SC, SA, Task A executes
Task C: Stuck on 3rd wait(SC)
execution order (B → A) C isnt executed

iv) Task A: Stuck at 1st wait(SA)
Task B: Stuck at 1st wait(SB)
Task C: executes, Signals SB, Task B executes
execution order: (C → B) A isnt executed.

9 – Multitasking (12 pts)

Consider the C program below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
main() {
        if (fork() == 0) {
                if (fork() == 0) {
                        printf("3");
                }
                else {
                        wait();
                        printf("4");
                }
        }
        else {
                if (fork() == 0) {
                        printf("1");
                        exit(0);
                }
                printf("2");
                wait();
        }
        printf("0");
}
```

Out of the 6 outputs listed below, choose only the valid outputs of this program. Assume that all processes run to normal completion. For the invalid outputs, explain why they are invalid. For the valid outputs, demonstrate an execution sequence that makes them possible. Selections with no explanations will not receive any credit.

a. 2030401 **b.** 1234000 **c.** 2300140 **d.** 2034012 **e.** 2130400 **f.** 4030120

output :- 2130400

(1) P_1 Prints 2 → waits for P_2 and P_4 → P_4 Prints 1 and terminates

P_2 waits for P_3 → P_3 Prints 3 → P_3 Prints 0 and terminates

P_2 Prints 4 and then 0 and terminates

P_1 Prints 0