CNG 334 – OPERATING SYSTEMS

ASSIGNMENT 2 REPORT

Fatma Erem Aksoy – 2315075

Task 1:

Q1: Consider the Available refers to the remaining resources after the allocation shown in the table. Provide a safe sequence if possible. Please, provide your answer with detailed steps.

Solution:

Step 1: Create the table showing Maximum, Allocated and Needed resources.

Processes	Maximum				Allocated				Needed			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
A	0	0	1	2	0	0	1	2	0	0	0	0
B	1	7	5	0	1	0	0	0	0	7	5	0
C	2	3	5	6	1	3	5	4	1	0	0	2
D	0	6	5	2	0	6	3	2	0	0	2	0
E	0	6	5	6	0	0	1	4	0	6	4	2

```
↓   ↓   ↓    ↓
2   9   10   12
```

Step 2: Select one process at a time according to currently available resources.

1	**5**	**2**	**0**
R1	**R2**	**R3**	**R4**

Select process D first (enough resources for this process, checking the Needed section in the table for this process):

```
  1 5 2 0
  0 6 3 2  -> allocated resources are released after the process execution
+ ________
  1 11 5 2 -> new available resource table, now we select process B next
```

Select process B:

```
  1 11 5 2
  1  0 0 0
+ _________
  2 11 5 2 -> new available resource table, now we select process C next
```

Select process C:

```
   2 11  5 2
   1 3   5 4
+ __________
   3 14 10 6 -> new available resource table, now we select process E next
```

Select process E:

```
   3 14 10 6
   0 0   1 4
+ __________
   3 14 11 10 -> new available resource table, now we select process A next
```

Select process A:

```
   3 14 11 10
   0 0   1 2
+ __________
   3 14 12 12 -> new available resource table
```

➔ All the resources are released and all processes are executed successfully. As all the processes were able to execute, we can say that a safe sequence is possible.

Task 2:

Q2: Draw the resource allocation graph of this system. Does the graph illustrate a deadlock or not, please explain your answer in detail.

Solution:

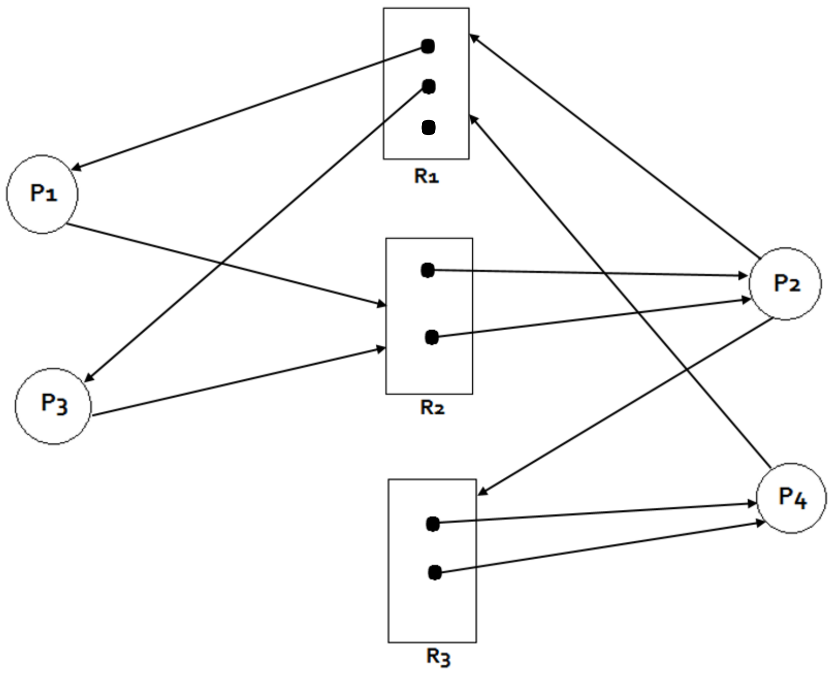
Processes(P) = {P1, P2, P3, P4}

Resources(R) = {R1, R2, R3}

Edges(E) = {R1->P1, P1->R2, R2->P2, R2->P2, P2->R1, P2->R3, R1->P3, P3->R2, R3->P4, R3->P4, P4->R1}

➔ Edges are written according to the given statements in the question.

- The corresponding RAG would look like shown below:



- For a deadlock to occur, there needs to be a cycle that involves a process waiting for a resource that is held by another process in the cycle. In the graph above, it can be seen that there happens a cycle involving P1, P2, P3, R1 and R2 where the order for using resources are: P1, R2, P2, R1, P3, R2. This shows that there is a potential deadlock in the system and it needs to be checked and verified considering the number of available instances for each resource.

- When we check R1, it has 3 instances, and currently 1 is held by P1 and 1 by P3. And it is requested by P2 and P4 as 1 instance per process, so 1 instance of R1 is available. As there is at least one available instances of R1, even though both P2 and P4 wait for it, the deadlock is not certain yet because that instance can be given to one of the processes and only one of them would have to wait, so it needs to be checked before calling it a deadlock.

- R2 has 2 instances, and currently both are held by P2. P1 and P3 request 1 instance (per process) of R2. As there are no available instances of R2, and both P1 and P3 are waiting for it, the situation can potentially lead to a deadlock.

- R3 has 2 instances, and currently both are held by P4. P2 requests 1 instance of R3, so there is at least one available instance of R3. In this situation, we can say that the deadlock is not certain.
    - As a result, there is a potential deadlock involving P1, P2, P3, R1 and R2 due to a cycle (waiting conditions of each other). So to avoid a deadlock, the execution

order for the processes can be as: P4, P2, P1, P3 or P4, P2, P3, P1. So we can say that P4 needs to be executed before P2 so that the other processes can also execute as P2 cannot be executed before P4 (wait condition problem) and other processes will need the instances that P4 and P2 hold.

Task 3: Processes

Solution:

➔ The code given illustrates how to create a shared memory region and unmap it by using the functions **mmap()** and **munmap**(), and displays some values to show how the child and parent processes access the shared elements & updates values in the same memory region. The steps of what the code does can be listed as:

- **mmap()** function creates a shared memory region with the amount of N*sizeof(int) bytes and assigns (maps) it to the pointer **ptr**.
- It checks if the creation is successful or not with an 'if' condition. If failed (done by checking if the **MAP_FAILED** value, which is 0, is equal to the **ptr** value), it prints a message stating that the mapping failed, and if not, then it continues.
- The parameters of **mmap()** function:
    - **NULL**: it represents the starting address of the memory area to be mapped.
    - **N*sizeof(int)**: it represents the length to map in bytes. If it is 0, then it'll give an error which is checked with the 'if' condition afterwards, as mentioned above.
    - **PROT_READ | PROT_WRITE**: it shows that the memory can be read from and written to, and the read/write access is allowed according to their values.
    - **MAP_SHARED | MAP_ANONYMOUS**: it creates a shared mapping and the changes are shared among processes as they have access to the same memory.
    - **0, 0**: The last two parameters are set to 0, and it represents that the open file descriptor is 0 (the first one), and where the map should begin in bytes is also 0 (the second one).
- The 'for' loop initializes the **ptr** elements in the shared memory to the specified value 334.
- Then the other 'for' loop prints these values of **ptr**.
- With **fork()**, a child process called is created which is a copy of the parent process.
- If (**pid**==0)… block is executed by the child process. It prints a message stating that the values are updating and then it updates all the values of **ptr** to 462.
- Else condition block is executed by the parent process. It first waits (with **wait(NULL)**) for the child process to finish execution, then it prints the new updated values of **ptr** in the shared memory region.
- **munmap()** is called to unmap the shared memory region (for **ptr**) and it releases the mapped memory for other processes.

- It then checks if the unmapping operation is successful or not. If not, it displays a message stating that the unmapping is failed and returns 1.

➔ As a shared memory region is created, and both child and parent processes have access to the same memory region, they can make changes in the array residing in the shared memory region. So that's why in the displayed output, the values in the array are all 334 first for the child process, but then all these values are changed to 462 in the parent process for the same array in the shared memory region.

Task 4: Threads

- The code part is provided as a separate file. The differences between the original code (mystery.c) and the new one can be listed as:
    - The **pthread** library is used here for thread related operations (functions) and data types.
    - A new function **threadCall()** is created to be executed by the thread
    - **pthread_create()** is called to create a new thread in the modified code, **fork()** was used to create a new process in the original mystey.c code.
    - A shared memory region was created with the **mmap()** function in the original code, but it is managed by declaring a global array ('**memory_array[]**') in the new version of the code.
    - Instead of **wait()** which is used in the original one to wait for the child process to complete its execution, **pthread_join()** function is used to wait for the thread to be done with its execution.

➔ Other than these differences, both codes have error handling by checking if the initializations are successful and displaying the corresponding messages, then returning to appropriate outputs. And both code files have the same output displayed at the end of their execution.

2) Without using the mutex, the threads were executed concurrently and the **circle_count** value was incremented independently, not protected/monitored by mutex. This situation may lead to a potential race condition resulting incorrect final output. So the mutex is necessary to synchronize access to the **circle_count** variable by allowing only one thread at a time to modify the value to prevent the race condition.
   In the code:
    - Variable called '**mutex'** is created by using **PTHREAD_MUTEX_INITIALIZER.**
    - **pthread_mutex_lock (&mutex)** in **worker()** function is called before updating the **circle_count** value to allow only one thread to proceed so it blocks other threads.
    - After a thread is done in the CS, **pthread_mutex_unlock (&mutex)** is called to release the mutex showing that the thread is completed and so other threads are allowed to access the shared variable now.

➔ So the mutex basically is used to make sure that the shared variable is updated correctly by several threads without any problem, preventing race condition.