1 Deadlock Prevention

(a) Describe a real-life deadlock situation. Explain why it satisfies the four necessary conditions (mutual exclusion, hold-and-wait, non-preemption, circular wait). How do people recover from that situation? Upon recovery, which condition becomes false?

(b) If the four conditions (mutual exclusion, hold-and-wait, non-preemption, circular wait) are true, has the system entered a deadlock state? If yes, why is it the case? If no, give a counter-example.

(c) Give an example, where the system is not in a safe state, but if the processes of the system are allowed to be executed, then they will be successfully completed.

(d) Considering the classic five philosophers' problem, give a specific example that demonstrates that imposing a total order on the philosophers' chopsticks avoids deadlocks. With this solution, how many philosophers are able to eat at the same time? Describe a scenario that leads to the starvation of one of the philosophers.

2 Deadlock Avoidance

Consider the following snapshot of a system:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	3	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

(a) According to this snapshot, the system is in a safe state. Execute the Banker's algorithm to verify this statement. Show the different values of the *work* vector after every iteration. What is the sequence of processes that the algorithm implicitly created?

(b) If a request from process P_0 arrives for (0,3,0,0), can the request be granted immediately? Justify your answer, using only the knowledge of the sequence you fount at sub-question (a).

(c) Give a pair $(process, request)$, such that if the system grands *request* to *process*, the system will be founded in an unsafe state. Explain why this is the case.

3 Memory Management

(a) Why is protection of processes' memory space important? Describe a scenario where absence of memory protection leads to problems.

(b) What is MMU? Where is it found and why is it used?

4 Contiguous Memory Allocation

Suppose we have a system with 1024 Bytes (1KB) memory space. Suppose, also, that we follow a contiguous memory allocation scheme with variable partitions. At time $t = 0$, the *input* queue (queue that keeps track of the processes that want to be executed but they have not been loaded in the memory yet), which is maintained by the Operating System, contains processes $P_1, P_2, P_3, P_4, P_5, P_6$. P_1 is the head of the queue and P_6 is the tail of the queue. The following table shows, for each process, what is its need in memory (in bytes) and for how long it will be executed (all of them need N time units). Every time the OS checks the *ready* queue to decide which process will acquire the CPU next (using FCFS CPU-scheduling algorithm), it first checks the input queue. According to the *allocation strategy* it uses, the OS loads all the processes whose memory need may be satisfied in the memory, it transfers the corresponding PCBs from the input queue to the ready queue, and then it decides which process will acquire the CPU next. Assume, at time $t = 0$, the memory and the ready queue are empty.

Process	Memory Need	CPU-burst
P_1	600	N
P_2	300	N
P_3	400	N
P_4	310	N
P_5	100	N
P_6	600	N

(a) Assume the *allocation strategy* the OS uses is the following:

- Start form the head of the input queue, and iterate until you reach the tail.
- For every element, check if the memory need can be satisfied (if there is a contiguous block that can accommodate the process). If it can be satisfied, pick the first such available block (first fit) and reserve the needed memory space. Also, move the corresponding PCB from the input queue to the ready queue.

Demonstrate for each point of time, how the memory space is covered by different processes. For each process, how much time it should wait in the input queue? What is the sum of those values (*total* waiting time in input queue)? Finally, pinpoint the cases of external fragmentation (where even if the total free space in memory is grater than a waiting process' memory need, this space is not contiguous).

(b) Now, we will try to do better than the previous allocation algorithm, in terms of *total* waiting time in the input queue. For this specific case of memory demands, how would you rearrange the order with which processes become ready, in order to achieve lower total waiting time in the input queue and eliminate all the cases of external fragmentation?

5 Paging

Suppose we have a system with 32-bit logical and 16-bit physical address space. Also, the page size is 512 Bytes and we use hierarchical paging. For this problem, we neglect the valid/invalid bits in the page table.

(a) In how many sections the logical address will be divided and how many bits each section will have?

(b) How many levels the structure of the page table will have?

(c) Give a simple example of converting a logical address into a physical address. Using the structure you proposed at sub-question (b), explain what section of the logical address is used to retrieve what information and how the physical address is formed in the end.

6 Programming

At this problem we will examine three different algorithms for page replacement; FIFO, LRU and OPT. Please, fill the code gaps at the file replace.py and then answer the following questions:

(a) Execute the program changing the variable $frame\_num$ to 3, 4 and 5, for all the three algorithms, and record the number of the page faults for each case. Compare the results and explain briefly whether or not they are expected. Do you notice the Belady's Anomaly at your results?

(b) For $frame\_num = 7$, what is one reference string of pages that produces as many page faults as the number of entries in the reference string, for the FIFO and LRU algorithms? (number of entries in the string >14). Assume that the page numbers in the string are between 0 and 7.