Solver: CHEN BENJAMIN, HOANG VIET, WILLIS TEE TEO KIAN, YONG SHAN JIE

1)

a)

i) For (i+1 mod 2), what it basically means is that if i = 0, (i+1 mod 2) = 1, and vice versa.

P_i	Line	flag[0]	flag[1]	turn	Remarks
P_0	flag[i] = true;	true	?	?	
Context Switch					
P_1	flag[i] = true;	true	true	?	
Context Switch					
P_0	turn = i;	true	true	0	
	while(flag[1] and turn == 1);	true	true	0	Since turn == 0, hence while loop broken & **P0 enters critical section.**
Context Switch					
P_1	turn = i;	true	true	1	
	while(flag[0] and turn == 0);	true	true	1	Since turn == 1, hence while loop broken & **P1 enters critical section.** Both processes are in critical section, hence **Mutual Exclusion violated.**

ii) There are multiple solutions to this question that will rid violation of mutual exclusion (3 User-Level Synchronisation Algorithms from Lectures). While the first two solutions cause other problems, either solution should still be correct.
*The question is asking for **modification** of what we have, thus only use the 3 algo below. Otherwise any OS mechanism for process synchronization such as Mutex and Semaphore would be an easier answer.*

Solution	Remove flag	Remove turn	Modify 'turn' in the original code

Code	while(1){
	    while(turn != i);
	    Critical section
	    turn = i+1 mod 2;
	    Remainder section
	}
		while(1){
		    flag[i] = true;
		    while(flag[i+1 mod 2]);
		    Critical section
		    flag[i] = false;
		    Remainder section
		}
			while(1){
			    flag[i] = true;
			    turn = i+1 mod 2;
			    while(flag[i+1 mod 2] and turn == i+1 mod 2);
			    Critical Section
			    flag[i] = false;
			    Remainder Section
			}

b)

Q = 1

P1	P2	P1	P2	P3	P1	P2	P4	P1	P2	P3	P4
1	2	3	4	5	6	7	8	9	10	11	12

P2 and P3 have the longest wait time of 6.

Q = 4

P1	P1	P1	P1	P2	P2	P2	P2	P3	P3	P4	P4
1	2	3	4	5	6	7	8	9	10	11	12

P4 has the max waiting time of 5.

Having a **Q = 4** will minimize the maximum waiting time.
Hint: Q size should not be longer than the longest CPU burst length as it would become FCFS => not Round Robin, remember this upper boundary! In this question, one should try to draw out Q = 2 and Q = 3 as well to calculate the max waiting time– Comment by Vetter

c)

i) Note: *Question does not have information on scheduling protocols used (e.g. preemptive or not). A good practice is to provide assumptions so that examiners know the context to mark your question in. It is alright to assume that this scheduling is non-priority based and only P_0 is available in the ready queue at the beginning (P_1 enters the system at D).*

Time Instant	P_0	P_1	Remarks
A	Ready > Running	-	P_0 is only process in system in ready queue
B	Running > Waiting	-	I/O interrupt so go into waiting state for the I/O event to be executed
C	Waiting > Ready > Running	-	Since the I/O event has completed, the waiting process transits to ready state, added to ready queue and is scheduled to run (only process in ready queue)
D	Running > Ready(2nd)	New > Ready(1st)	1) P1 is added to the ready queue 2) Interrupt happens, need to run ISR so process P0 is added to the ready queue.
E	-	Ready > Running	P0 is behind P1 on the queue. Hence P1 get dispatched after timer interrupt finished (when scheduler is then invoked)

F	-	Running > Ready	Interrupt happens, need to run ISR so process is added to ready queue (behind P0)
G	Ready > Running	-	P0 is at the top of the queue so it gets dispatched by scheduler.

ii) OS kernel executes during all time instants A-G as state transitions occurred during these instants. The OS kernel stores and retrieves the PCB of the processes being swapped in and out**. In fact, all context switches require kernel mode as PCB are stored in kernel space**. If a user process can execute context switch without the kernel, it might preempt the processor and starve every other process. OS kernel might be invoked in API calls within interrupt service routines or scheduler as these mechanisms required lower access level to the hardware.

2)

a)

i) **True**. This ensures that only one process is reading or writing the shared variable at any instance.
ii) **False**. It is possible to implement a simple batch system in a multiprocessor.
iii) **True**. Even for DMA, one interrupt still needs to be generated per memory block.
iv) **True**. They can only be executed in kernel mode because all I/O instructions are privileged instructions. “root” user is still a user account in an OS.
v) **False**. Base and limit registers are used for memory protection, to determine the range of legal addresses a program may access.
vi) **False**. System call is only executed in kernel mode and because disabling of interrupts is a privileged instruction.

b) There is no deadlock as there is no cycle in the graph.
P2 can finish executing as it only requested for S and has currently obtained S.
P2 releases S -> P1 acquires S and finishes executing -> P1 releases S and R -> P3 acquires R and finishes executing.

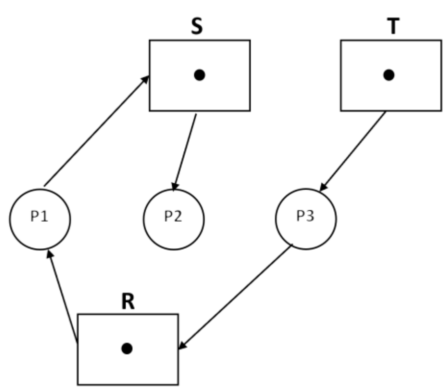

Figure 1: No circular wait, hence deadlock cannot happen.

c) **Mutual Exclusion**: Since Wait() and Signal() are both atomic instructions, only one process can modify the binary semaphore at any time instant before granted access to its critical section, therefore **mutual exclusion is ensured**.

Progress: Counterexample: P_0 is holding onto the semaphore currently (S.value = 0), and P_1 calls Wait(S). Since S.value = 0, the "IF" statement of Wait(S) will be executed (S.value is now -1). Once P_0 is done, it calls Signal(S), and after executing S.value++, S.value becomes 0. Since (S.value<0) is FALSE, wakeup() function will not be called. P_1 will not be able to enter the critical section, despite no process being in the critical section. Therefore, **progress is not ensured**.

Bounded Waiting: The blocked queue follows a FIFO policy, any process queued first will have the request granted first (no starvation), hence **bounded waiting is ensured**, with other processes that are later in the queue not allowed to enter their critical section prior.

3)

a)

i) **False.** Size of stack can increase and decrease dynamically as the program runs as it is used to store variables created and destroyed during runtime.
ii) **True**. The loader will then bind the relocatable address to absolute address.
iii) **False**. Worst-Fit produces the largest leftover hole in memory as the largest hole is allocated for the process.
iv) **True**. The entire inverted page table must be searched for address translation as it is sorted by physical address even though the lookups occur on the logical address.
v) **False**. Thrashing can occur as degree of multiprogramming increases if processes are busy bringing pages in and out due to high page fault frequency. This leads to increased disk I/O instead of doing useful work, resulting in lower CPU utilization.
vi) **False**. Variable frame allocation can imply either global or local replacement.

b)

i) TLB Look-up Time = ε
Memory Access Time = μ
Hit Ratio = α
EAT = $\alpha(\mu + \varepsilon) + (1 - \alpha)(2\mu + \varepsilon)$
EAT = $(2 - \alpha)\mu + \varepsilon$
EAT = (2 - 0.8)(80ns) + 20ns
EAT = 116ns

ii) Without TLB, EAT = 2μ, (μ for page table, another μ for data/instruction)
2μ = 2(80ns) = 160ns.
160ns = $(2 - \alpha)$ (80ns) + 20ns
$\alpha = 0.25$
Minimum hit ratio = 25%

c)

	Paging	Segmentation
Fragmentation	Internal fragmentation occurs if allocated memory is slightly larger than requested memory and is internal to a partition. No external fragmentation.	External fragmentation occurs when processes leave the system. Occupied segments become holes which may be too small for new processes to occupy. No internal fragmentation.
Support for code sharing	Code sharing can also be done but only through shared pages to reduce memory usage. Code within the page MUST NOT be modifiable for this to be possible.	Facilitates code sharing at segment level since code is broken up into logical segments. (e.g. main program/subroutine)

d)

i) Since 4K bytes pages, $2^n = 4096$
n = 12, therefore 12 bits is required to represent the offset in logical address.

ii) Since Local Replacement Policy is assumed, only Frame 1, 4, 8 and 15 is usable. Also, assume that current tick is 276. Since physical and logical address is 16-bit and offset is 12-bit, 4 bits is used for Page/Frame number.

Tick		275		276		277		278		279		280		281	
Reference		-		0x76F4		0xB89F		0x1A86		0x6987		0x7E56		0xD908	
		Page	**TLR**	**Page**	**TLR**	**Page**	**TLR**	**Page**	**TLR**	**Page**	**TLR**	**Page**	**TLR**	**Page**	**TLR**
Frame	**1**	1	268	1	268	1	268	**1**	**278**	1	278	1	278	1	278
	4	10	245	10	245	**11**	**277**	11	277	11	277	11	277	**13**	**281**
	8	7	230	**7**	**276**	7	276	7	276	7	276	**7**	**280**	7	280
	15	12	275	12	275	12	275	12	275	**6**	**279**	6	279	6	279
Hit?		-		Hit		Miss, Evict Page 10		Hit		Miss, Evict Page 12		Hit		Miss, Evict Page 11.	

Total: **3 page-faults** generated.

iii)

Frame	Page
1	1
4	13
8	7
15	6

iv)

Reference	Frame No.	Translated Physical Address
0x76F4	8	0x86F4
0xB89F	4	0x489F
0x1A86	1	0x1A86
0x6987	15	0xF987
0x7E56	8	0x8E56
0x0908	4	0x4908

4)

a)

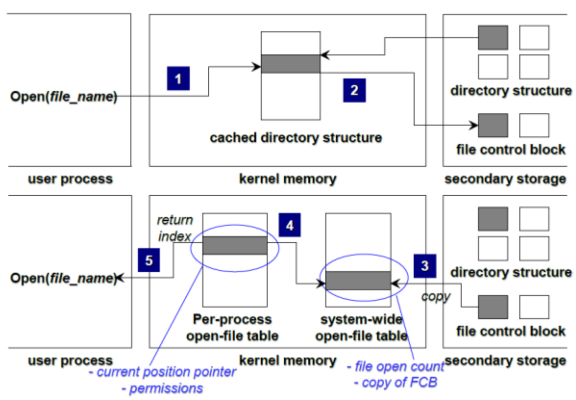

Figure 2: Lecture Slides (File Systems) on File Opening

Per-process open file table is needed for the current process to perform various operations on the file quickly. The entry in the Per-Process File Table points to the System-wide Open File Table entry corresponding to the file. System-wide open file table is needed to keep track of the details of all the files that are being used.

When open() is invoked, the file control block (FCB) is copied from the on-disk storage to the kernel memory (step 3) - into the system-wide open file table. The file open count will increment by 1 (to indicate one more instance of this file being opened). The kernel will check for the permission in the FCB and see if the operation is legal (step 4). If so, it will copy the FCB into the per-process open-file table, and return a pointer to the file as return value of open() call (step 5)

b) Each block size is 1000 bytes, and each pointer size is 10 bytes. This means that any block can contain either some data, or 100 pointers.
To help better illustrate problem, refer to the diagram below:

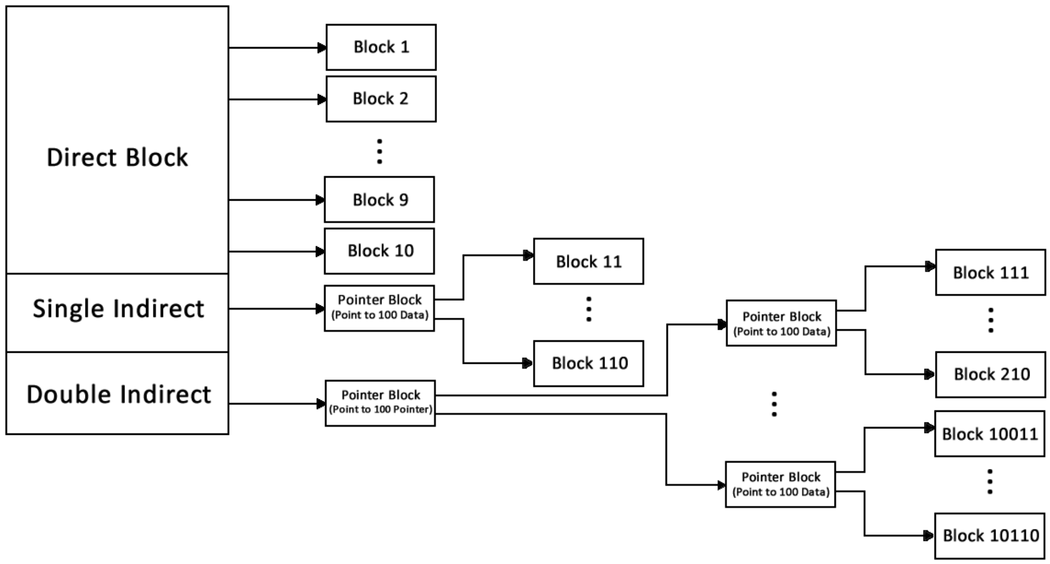

Figure 3: inode

i) Byte 0 - 999 is in block 1,
Byte 1000 - 1999 is in block 2,
Byte 9500 is in block 10.
Since 1000 bytes needs to be read from Byte 9500, Block 10 and 11 needs to be read.

Block Read 1: Direct Block 10
Block Read 2: Single Indirect Pointer Block
Block Read 3: Block 11
Therefore, a total of **3 blocks** must be retrieved.

ii) File opened is 110,000 bytes long, therefore currently it occupies $\frac{110{,}000}{1000} = 110$blocks. If a new data block needs to be appended to the end of file, it must be appended from Block 111 onwards.
Block Read 1: Double Indirect Pointer Block
Block Read 2: First Pointer Block of Double Indirect Pointer Block
Block Read 3: Block 111.
Therefore, **3 blocks** must be accessed.

iii) Direct Blocks: 10

Single Indirect Block: 100
Double Indirect Block: 100 x 100 = 10,000
Total number of blocks = 10,000+100+10 = 10,110
Maximum File Size = 10,110 x 1000 bytes = **10,110,000 bytes**.

c)

i) 100->110->88->74->45->134->150
Total head movement by SSTF:
|100-110| + |110-88| + |88-74| + |74-45| + |45-134| + |134-150|
= **180**

ii) Total head movement by FCFS:
|100-45| + |45-134| + |134-88| + |88-110| + |110-74| + |74-150|
= 324
Improvements = 324-180 = **144**

iii) If many requests are clustered together, SSTF will constantly service the requests that are clustered together, resulting in starvation for the requests further away. FCFS will ensure that any requests that arrive first will get serviced, regardless of the head movement required.

--End of Answers--