Computer Science and Engineering Department
IIT Kharagpur

Operating Systems

3rd year CSE: 6th Semester (*Class Test 2*)

Time limit: 1 hour 30 minutes Date: 7th April, 2026

Max Marks: 60

Roll no: _____ Name: _____

1. (a) Recall the deadlock-free solution of the dining philosopher's problem, taught in the class, which uses a semaphore vector s[], initialized to 0, and an integer vector state[], and implements `take_fork()`, `put_fork()`, and `test()` functions. Now, consider the following two variations (i) and (ii) separately.

(i) Modify the `test()` function as follows. Explain the effect on the solution. **[5]**

```
test(i) {
    if ((state[i] == HUNGRY) && (state[LEFT] != EATING) && (state[RIGHT] != EATING)) {
        signal(s[i]);
        state[i] = EATING;
    }
}
```

There will be no effect on the solution. This function is called with mutex acquired, and the mutex is released after the function returns, so shuffling mutually independent statements inside the function will not hurt.

(ii) Modify the `put_forks()` function as follows. Explain the effect on the solution. **[5]**

```
put_forks(i) {
    wait(mutex);
    state[i] = THINKING;
    signal(s[LEFT]);
    signal(s[RIGHT]);
    signal(mutex);
}
```

Mutual exclusion violation. No checking of neighbors. For example, if LEFT is HUNGRY but LEFT of LEFT is EATING, so LEFT is supposed to wait on s[LEFT]. But the signal on s[LEFT] ends the wait of LEFT, so both LEFT and LEFT of LEFT can eat together.

(b) Each process P_i, $i = 0, 1, 2, 3, \ldots, 9$ is coded as follows.

```
wait(mutex);
        {Critical Section}
signal(mutex);
```

The code for P_{10} is identical except that it uses signal(mutex) instead of wait(mutex). What is the largest number of processes that can be inside the critical section simultaneously at any moment (the semaphore mutex being initialized to 1)? Justify with argument. (No marks will be awarded without justification.) **[4]**

Three. Say, mutex = 1. P_{10} runs, P_{10} goes to the CS with mutex = 2. It allows two other processes. As one process leaves CS, it allows another to enter (so the number never exceeds three). Similarly, take mutex = 0 (one process is already in critical section). P_{10} can still enter setting mutex = 1. This allows yet another process to enter CS. Again, as one process leaves, it allows another to enter (so the number never exceeds three).

2. Consider a computing center with 5 processes (P0 to P4) and 4 resource types (A, B, C, D). The center is equipped with a total of 3 instances of A, 14 instances of B, and 12 instances of resources C and D each. Consider the following snapshot of the execution of these 5 processes.

Allocation Matrix

	A	B	C	D
P0	0	0	1	2
P1	1	0	0	0
P2	1	3	5	4
P3	0	6	3	2
P4	0	0	1	4

Maximum Requirement

	A	B	C	D
P0	0	0	1	2
P1	1	7	5	0
P2	2	3	5	6
P3	0	6	5	2
P4	0	6	5	6

At this moment, if a resource request from process P1 arrives for (0, 4, 2, 0), can the request be granted immediately? Show your detailed calculations. **[8]**

Yes…

If the request is granted, then AVAILABLE = (1, 1, 0, 0), and the NEED matrix becomes:

	A	B	C	D
P0	0	0	0	0
P1	0	3	3	0
P2	1	0	0	2
P3	0	0	2	0
P4	0	6	4	2

Allocation for P1 becomes (1, 0, 0, 0) + (0, 4, 2, 0) = (1, 4, 2, 0)

P0 can finish, making AVAILABLE = (1, 1, 0, 0) + (0, 0, 1, 2) = (1, 1, 1, 2).
P2 can finish, making AVAILABLE = (1, 1, 1, 2) + (1, 3, 5, 4) = (2, 4, 6, 6).
P1 can finish, making AVAILABLE = (2, 4, 6, 6) + (1, 4, 2, 0) = (3, 8, 8, 6).
P3 can finish, making AVAILABLE = (3, 8, 8, 6) + (0, 6, 3, 2) = (3, 14, 11, 8).
P4 can finish, making AVAILABLE = (3, 14, 11, 8) + (0, 0, 1, 4) = (3, 14, 12, 12)

Since <P0, P2, P1, P3, P4> is a safe sequence, the request can be granted.

3. Consider a memory-management system that uses a 3-level hierarchical paging scheme (L1, L2, and L3), where L1 is the outermost page table. The virtual address space is 42 bits, and the page size is 4 KB. The maximum physical memory to be supported is of size 16 TB. Each page of the intermediate page table (L2) and the inner page table (L3) must fit in one frame, whereas the outermost page table (L1) is stored in exactly two frames (assume that the PCB of the process stores these two frame numbers for L1). Each entry of L1 and L2 stores frame numbers only. Each entry of the innermost page table (L3) stores the frame number, and reserves a few additional bits to store valid/invalid bit, dirty bit, protection bits, time of update, and so on.

(a) Compute the total number of additional bits reserved for each page-table entry of L3. **[10]**

Virtual address space = 2^{42} bytes

Page size = 4 KB

Physical memory 16 TB (maximum supported)

Number of frames = 16 TB / 4 KB = 2^{32}. Hence 32 bits are needed for frame number.

Number of pages = $2^{42} / 2^{12} = 2^{30}$

Hence we have 2^{30} entries in the L3 page table. Assume each entry of L3 page table is of size 2^x bytes. Hence the total size of L3 page table is $2^{30} \times 2^x$ bytes. We split this L3 page table in 4KB size pages. So the number of pages in the L3 page table is $2^{30+x} / 2^{12} = 2^{18+x}$

Subsequently, the number of entries in first outer page table L2 is 2^{18+x}. Each entry of L2 only contains only a frame number, hence its size is 4 bytes. So the total size of L2 page table is $2^{18+x} \times 4$. This L2 we split in 4KB size pages. Hence the number of pages of L2 is $2^{18+x} \times 4 / 2^{12} = 2^{8+x}$.

Subsequently, these 2^{8+x} pages of L2 forms the 2^{8+x} entries of outermost page table L1. Each entry of the outermost page table is 4 bytes. So size of the L1 page table is $2^{8+x} \times 4$ bytes. On the other side, the size of the L1 page table is two pages, that is, 2^{13} bytes. Hence $2^{8+x} \times 4 = 2^{13}$. This gives $x = 3$.

So each entry of L3 contains $2^x = 8$ bytes, out of which 4 bytes store the frame number. Therefore 4 bytes will be reserved for special bits.

(b) Clearly show the translation of virtual addresses to physical addresses for the aforesaid multilevel paging scheme. In your solution, clearly show the virtual address (with proper length) generated by the CPU, its reference to L1, L2, and L3 (with the necessary numbers of bits required), and the final construction of the physical address (with proper length). **[7]**

Number of PT entries in L1 = 2^{11} ⇒ 11 bits for p1.

Number of entries in L2 = 2^{21}. Each entry size 4 bytes. We split L2 in pages. Size of each page of L2 is 4KB. So number of PT entries in one page of L2 is 4KB / 4 = 2^{10} ⇒ 10 bits for p2.

Number of entries of L3 is 2^{30}. Each entry is of size 8 bytes. Each page size is 4 KB. So number of PT entries in one page is 2^{12} / 8 = 2^{9} ⇒ 9 bits for p3

So virtual address split is: 11 (p1) + 10 (p2) + 9 (p3) + 12 (offset)

4. (a) Consider a paging system that uses a single-level page table stored in the main memory along with a Translation Lookaside Buffer (TLB) for address translation. Each main-memory access takes 100 ns, and a TLB lookup takes 20 ns. Each page transfer to or from the disk takes 5000 ns, and the time to update the TLB is negligible. The system gives each program a fixed number of frames, and uses swapping with paging. When a page to be accessed is not in one of these frames, one page of the program is swapped out, and the desired page is swapped in. A program repeatedly generates memory accesses in the following pattern: it performs nine consecutive accesses to pages that are already present in the memory, followed by one access to a page that is not present in the memory, after which the same sequence repeats indefinitely. Statistical study of the program reveals that (i) the TLB hit ratio is 95%, and (ii) for 20% of the cases resulting in page swaps, the swapped out page is modified and must be written back to the disk (in the remaining 80% cases, the swapped out page is not needed to be written back to the disk, because it is already there in the latest form). Compute the average time for one memory access during the run of the program. Assume that except the disk I/O, the swapping overhead (like stopping the running process, checking the PCB, updating the page table, and so on) is negligible. **[8]**

Page fault rate: 10%
TLB hit: 95%
TLB hit ⇒ No page fault
TLB miss ⇒ may or may not be a page fault
A page fault is surely a TLB miss.

Effective memory-access time
$= 0.95 \times (20 + 100) + 0.05 \times \left[0.9 \times (20 + 100 + 100) + 0.1 \times \left(0.2 \times (20 + 100 + 5000 + 5000) + 0.8 \times (20 + 100 + 5000)\right)\right]$
$= 154.5$ ns

(b) A contiguous memory-management system has a total memory of 2100 KB, with the lowest 300 KB permanently occupied by the operating system. The remaining memory is allocated to user processes following the Worst-Fit strategy. Assume that each memory access (read or write) takes 200 ns per byte.

A set of processes arrive and terminate in the following sequence: P1 (200 KB) arrives, P2 (500 KB) arrives, P3 (200 KB) arrives, P4 (700 KB) arrives, P2 terminates, P5 (100 KB) arrives, P4 terminates, P6 (300 KB) arrives, P7 (400 KB) arrives, P6 terminates, and P8 (800 KB) arrives. Assume that an arriving process is always allocated the lower-address portion of the chosen hole.

(i) Show the final memory layout after processing all the above events. **[4]**

| OS 300 | P1 200 | P5 100 | Free 400 | P3 200 | Free 330 | P7 400 | Free 200 |
P8 waits. (Details on next page)

(ii) Identify the fragmentation issue here. Briefly explain. **[2]**

P8 cannot be accommodated because of external fragmentation.

(iii) Propose a method with the lowest cost to handle the issue of fragmentation identified in Part (ii). Show the outcome of this lowest-cost solution. Compute the cost, and show how this solution can address the problem of fragmentation in this case. **[5]**

Compaction: Move P3 to end, and get the following partitioning of the memory.

| OS 300 | P1 200 | P5 100 | Free 900 | P7 400 | P3 200 |

Data moved: 200 KB = 204800 bytes

Each byte movement: 1 read + 1 write = 2 × 200 ns = 400 ns

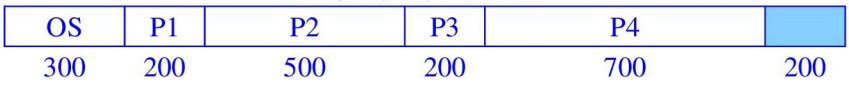
Total cost = 204800 × 400 = 81,920,000 ns = 81.92 ms

P8 can now be accommodated: | OS 300 | P1 200 | P5 100 | P8 800 | Free 100 | P7 400 | P3 200 | (Details on next page)

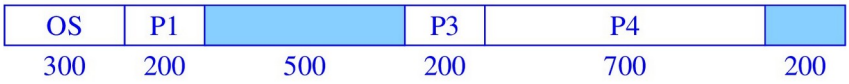
(iv) State which type of address binding is most suitable for your solution, and justify your answer. **[2]**

Run-time, because relocation of a program has less overhead than compile-time or load-time address binding.

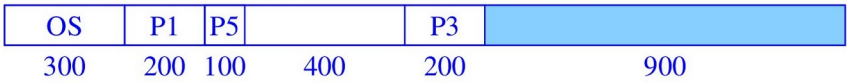
P1, P2, P3, P4 arrive



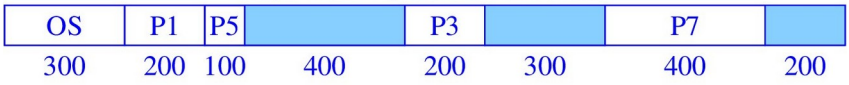
P2 terminates



P5 arrives and P4 terminates

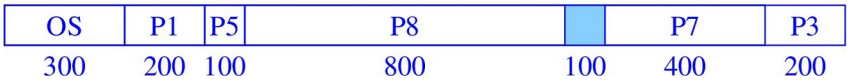


P6, P7 arrive, and P6 terminates



P8 arrives and waits

P3 relocated to end, P8 can now be allocated memory



Rough Work