CLO # 4 - Understand synchronization and deadlocks. Implement synchronization using Mutex/Semaphores.

Q1. [15 marks] *Instruction: Write answers as bullets points.*

a) Consider a ticket booking system where the available number of tickets for each program must be maintained. A person who wishes to book a ticket calls the bookTicket() function, which checks if tickets are available and then decreases the available tickets count by one. This is illustrated in the code shown. Now **explain** how a race condition is possible in this situation. Also, **show a Pthread library-based C program** that avoids race condition using a mutex. [1 + 3]

```
void bookTicket() {
    if (availableTickets > 0)
        availableTickets--;
}
```

Race condition is possible as follows:

- This function code has no mutual exclusion.
- Threads, running on multiple cores, will update the variable "availableTickets" – which is a single memory location – in random order resulting in race condition.
- *** full program needs to be written transforming given bookTicket() function given.
- No marks if semaphores are used. 1.5 marks if syntax is 100% correct + 1.5 marks if 100% logic is correct.

```
#include <stdio.h>
#include <pthread.h>
int availableTickets = 10;
pthread_mutex_t ticketMutex;
void bookTickets() {
    pthread_mutex_lock(&ticketMutex);
    if (availableTickets > 0) {
        availableTickets--;
        printf("Ticket booked successfully. Tickets remaining: %d\n", availableTickets);
    } else {
        printf("No tickets available.\n");
    }
    pthread_mutex_unlock(&ticketMutex);
}
void* threadFunc(void* arg) {
    bookTickets();
    return NULL;
}
int main() {
    pthread_mutex_init(&ticketMutex, NULL);
    pthread_t threads[10];
    for (int i = 0; i < 10; i++) {
        pthread_create(&threads[i], NULL, threadFunc, NULL);
    }
    for (int i = 0; i < 10; i++) {
        pthread_join(threads[i], NULL);
    }
    pthread_mutex_destroy(&ticketMutex);
    return 0;
}
```

b) Explain **synchronization issue(s)** in producer and consumer codes in a producer-consumer problem. Write and explain C code snippets that use two types of semaphores as a solution for this problem. [2+2]

- Both have the same synchronization issue: i.e. ensuring that the producer and consumer access the shared buffer in a mutually exclusive manner, and
- coordinating their activities so that the buffer does not overflow or underflow.
- 1 wt. for correct code and 1wt. for explanation of code.

```c
#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
int in = 0, out = 0;
sem_t emptySlots;
sem_t filledSlots;
pthread_mutex_t bufferMutex;

// Producer function
void* producer(void* arg) {
    int item;
    for (int i = 0; i < 10; i++) { // Produce 10 items
        item = rand() % 100; // Produce an item
        sem_wait(&emptySlots); // Wait for an empty slot
        pthread_mutex_lock(&bufferMutex); // Lock the buffer
        buffer[in] = item;
        printf("Produced: %d at %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&bufferMutex); // Unlock the buffer
        sem_post(&filledSlots); // Signal that there is a filled slot
        sleep(1); // Simulate production time
    }
    return NULL;
}
void* consumer(void* arg) {
    int item;
    for (int i = 0; i < 10; i++) { // Consume 10 items
        sem_wait(&filledSlots); // Wait for a filled slot
        pthread_mutex_lock(&bufferMutex); // Lock the buffer
        item = buffer[out];
        printf("Consumed: %d from %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&bufferMutex); // Unlock the buffer
        sem_post(&emptySlots); // Signal that there is an empty slot
        sleep(1); // Simulate consumption time
    }
    return NULL;
}
```

c) Write answers to the following questions:

i. Why **hardware atomic instructions** are necessary in the implementation of locking primitives? [1]

These instructions ensure mutual exclusion even if threads run on different cores in parallel.

ii. How do two threads become deadlocked while accessing resources? **Write C code snippet using semaphores** and a step-by-step explanation for any scenario. [1.5+1.5]

- 1.5 wt. will be awarded only if out-of-order pairs of waits are used in the code.
- Textual answer will get no awards.
- 1.5 wt. will be awarded if step-by-step explanation given.

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

d) How busy waiting is implemented during synchronization? **Write two C code snippets** showing two different methods. **Also, explain the working of any one snippet using a dry** run with variable values. Assume two processes where one is executing in its critical section and other is waiting [2 + 1].

2 wt.

- The busy term (a.k.a spinning) is used as waiting involves a loop that continuously checks the condition, consuming CPU cycles until the condition is met.
- Two Option A and Option B (or similar) is given with dry run.

1wt.

- Only if explicit dry run done by the student.

```
#1

Option A
...
while (!available); -- busy wait
available = false;
/* critical section */
available = true;
...

Option B

wait(S) {
   while (S <= 0); // busy wait
S--;
}
```

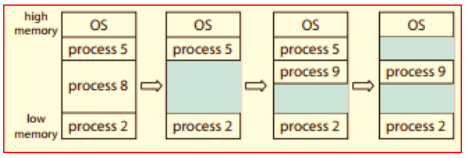
CLO # 5 - Understand virtual memory and its management.

Q2. [15 marks] *Instruction: Write answers as bullets points. Make suitable assumptions, if needed.*

a) Explain **external** fragmentation in main memory **using a labeled diagram**. What causes **internal** fragmentation in a page? Explain with an example. [2 + 1]

2wt.

nal fragmentation. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.



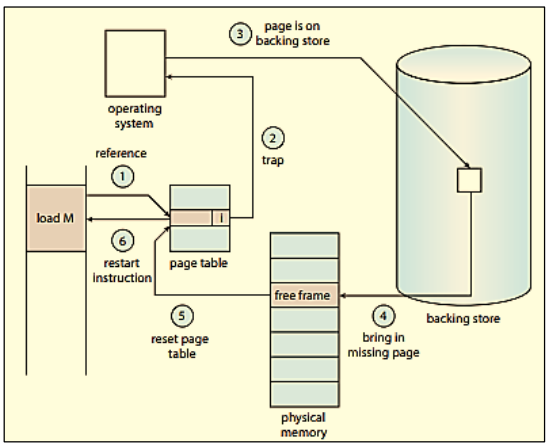
1 wt.

Internal fragmentation occurs when memory allocated for a process is slightly larger than what the process requires. This "extra" allocated memory within a fixed-size block (page) is wasted because it cannot be used by other processes. Suppose total size of executable code is 3.5k and page allocated to text segment of the process will always be 4k in a system where page size is fixed to 4k. Therefore, 0.5k is waster due to internal fragmentation.

b) Page faults are critical in a virtual memory system.

i. **Explain** Page fault processing **using a labelled diagram.** [2]

See diagram.



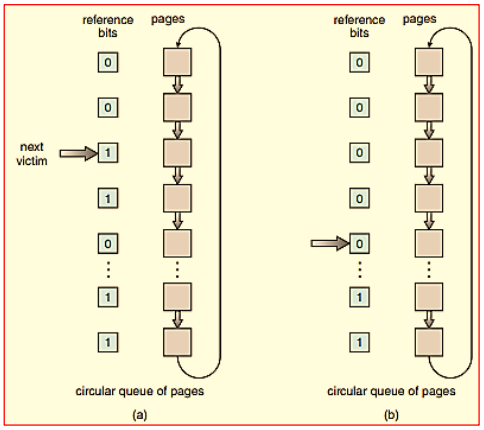
ii. In a computer system, memory access time is 40 nanoseconds and hard disk access time is 1.25 milliseconds. Calculate effective memory access time if page fault probability is 0.000025. [2]

effective access time = (1 – p) × (40) + p (1.25 milliseconds)
= (1 – p) × 40 + p × 1,250,000
= 40 + 1249960 × p
= 40 + 1249960 (0.000025)
= 40 + 31.249 ➔ 71.249

c) LRU is one of the many page replacement algorithms. Explain its two implementations as follows:

i. Use a labelled diagram and stepwise hints to **describe clock (second chance) algorithm**. Show it does approximate LRU functionality. [2]

ii. Now, **enhance** the above algorithm to ensure that dirty victim pages are flushed to memory before reuse. [2]



- The reference bits are used to approximate the LRU policy.
- Pages that are used frequently will have their reference bit set to 1, giving them a second chance and making it less likely for them to be replaced immediately, much like how LRU would prefer recently used pages over those used a long time ago.

10.4.5.2 Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

One way to implement the second-chance algorithm (sometimes referred to as the clock algorithm) is as a circular queue. A pointer (that is, a hand on the clock) indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 10.17). Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

10.4.5.3 Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify bit (described in Section 10.4.1) as an ordered pair. With these two bits, we have the following four possible classes:

1. (0, 0) neither recently used nor modified—best page to replace
2. (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean—probably will be used again soon
4. (1, 1) recently used and modified—probably will be used again soon, and the page will be need to be written out to secondary storage before it can be replaced

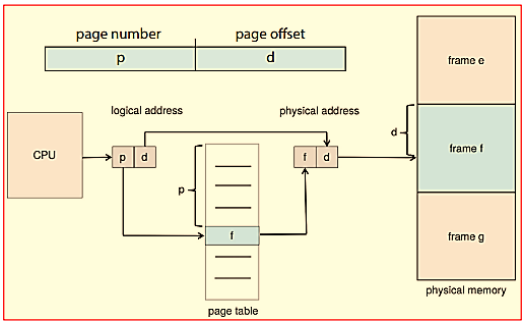
Each page is in one of these four classes. When page replacement is called for, we use the same scheme as in the clock algorithm; but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified in order to reduce the number of I/Os required.

d) Suppose a program is accessing main memory in a virtual memory system having a page size of 2K.

i. State the difference between paged and demand paged memory system? [1]

Paged Memory System	Demand Paged Memory System
All pages that are part of process are loaded into available frames at execution start.	Pages are loaded only when accessed. Pages are loaded on page fault occurrence

ii. Show a labelled diagram which shows how a virtual memory address generated by a program is translated into a physical address. Now, in your diagram, assume suitable values for page number and page offset and show how the processor calculates the physical address. [3]

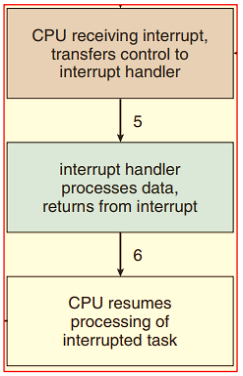


- Virtual address (logical address) is generated by the executing program.
- Page table lookup is done to read the frame number.
- Displacement within page/frame remains same.
- ----
- Suppose page number is 26 and offset is 87 so the logical address is 2687. The page table lookup for page number 26 resulted in a frame number 890. So the physical address becomes 89087.

CLO # 1 - Describe, discuss, and analyze, services provided by the modern Operating Systems.

Q3. [6 marks] *Instruction: Avoid unnecessary details in your answers.*

a) Create a diagram **illustrating the interrupt handling process** in a typical operating system. What is the purpose of timer in operating systems and how it is related to interrupts. [1+2]

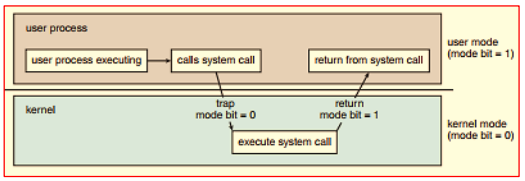


1 wt. for diagram and 2wt. for the answer below.

- OS needs to be reminded to perform periodic tasks.
- This can be done using a time, which when reaches a pre-set value, it generates a hardware interrupt known as a timer interrupt.
- This is used system wide to facilitate process scheduling, time-sharing, and system resource management.
- For example, a timed interrupt (say 5 milliseconds) in process scheduling preempts the currently executing process and brings in new one (known as context switching) thus allowing fair CPU allocation.

b) Describe **execution steps of a (generic) system call using two modes** in a modern operating system with the help of a labelled diagram. [2]

- When a system call is executed, it is treated as a software interruption.
- Control is transferred through the interrupt vector to a service routine in the operating system.
- Switching the mode to kernel mode.
- The kernel identifies the system call type and retrieves necessary parameters, which may be passed via registers, stack, or memory.



- After verifying the parameters, the kernel executes the request and returns control to the user program.

What **problems do you expect if OS executes this system call in user mode only**? Explain. [1]

- Privileged instructions (switching to kernel mode, I/O control, and timer management.) can only be executed in kernel mode. These facilitate managing resources that are kept private from users. The dual mode of operation protects the operating system and users from errant actions by designating certain harmful instructions as privileged.
- If privilege mode instructions are allowed in user mode (i.e. there is only one mode of operation) a bug in the user program will crash the whole OS.
- Multitasking and Multiprogramming will be difficult as in old DISK Operating System (DOS).

CLO # 2 - Understand, design, and implement solutions employing concepts of Processes and Threads

Q4. [6 marks]

a) Consider Google's Chrome browser as a multi-process architecture. What benefit would this implementation provide to the user? [2]

- If each tab is a separate process, then this provides protection and security as each process memory will be sperate and unaccusable by other processes. can be accessed by other processes.
- Each process can be scheduled on a different core of the processes. Thus, response time will be improved.

b) Show how many processors are needed to gain a 3.5 times speedup if the serial portion is 20%. [2]

speedup = 1 / (serial portion + (parallel portion /N) where N is number of processors. Putting values
3.5 = 1 / (0.2 + (0.8/N))
3.5 (0.2 + (0.8/N)) = 1 ➔ 0.7 + (2.8/N) = 1 ➔ 2.8 / N = 0.3 ➔ N = 2.8 / 0.3 ➔ N = 9.33 ~ 10 processors.

c) Explain different types of mapping between user-level and kernel-level threads. [2]

- ***One-to-One***: True parallelism as each user threads gets its own kernel threads which can be scheduled independently.
- ***Many-to-One***: Thread library (like Pthread) can manage user threads and maps then to a single kernel thread.
- Many-to-Many: Flexible mapping where many user threads are mapped equal number and less kernel threads.

CLO # 3 – Understand mechanisms for scheduling of tasks in modern operating systems.

Q5. [8 marks] *Instruction: Write answers as bullets points.*

a) Consider **Linux Completely Fair Scheduler**. Explain: i) the computations done to takes scheduling decisions, and ii) give a diagram to show how CFS efficiently computes the next runnable task. [2+2]

Completely Fair Scheduler (CFS)

Default in 2.6.23 kernel

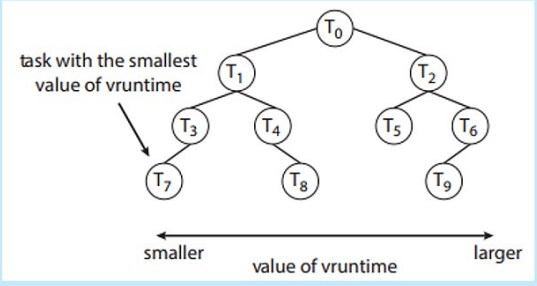
- It does not associate a relative priority value with the length of a time quantum.
- CFS scheduler assigns a proportion of CPU processing time to each task.
  - The proportion is calculated based on the nice value (priority) of each task. Nice values range from −20 to +19, where lower value indicates a higher relative priority.
  - Lower nice values receive a higher proportion of CPU processing time. The default nice value is 0.
- No discrete values of time slices. Instead CFS identifies a targeted latency - an interval of time during which every runnable task should run at least once. It can increase if the number of active tasks in the system grows beyond a certain threshold.
- Proportions of CPU time are allocated from the value of targeted latency.

A process is being nice if it increases its nice value (say, 0 to +10) ...nice processes finish last! 35

Priority Assignment

- CFS records how long each task has run by maintaining the virtual run time of each task using the per-task variable vruntime. vruntime is not the shortest CPU burst time.
  - There is a decay factor based on the priority of a task: lower-priority tasks have higher rates of decay than higher-priority tasks.
    - For nice = 0, if a task runs for 200 msec, its vruntime = 200 msec.
    - For nice > 0, if a task runs for 200 msec, its vruntime > 200 msec.
    - For nice < 0, if a task runs for 200 msec, its vruntime < 200 msec.
- Which task to run next? task with the smallest vruntime. A higher-priority task that becomes available to run can preempt a lower-priority task.
- Task vacating the CPU, adds its execution time to the virtual runtime and is then inserted back into the tree if runnable.

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Rather than using a standard queue data structure, each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below.



b) Assume an Operating System **uses FCFS scheduler**. Suppose there are only two processes: first is running with fifty threads to compute the value of Pi using 5000K iterations, and the second process is an Internet browser where a single tab is shown to the user. **Explain how you change the code of the first process such that the FCFS scheduler run** both processes alternatively i.e. before finishing the first process. *Write precise answer in 2-3 lines only* [2].

Cooperative or non-preemptive scheduling. FCFS is non-preemptive, so once it starts execution of the first process the 2nd process must wait till the completion of the 1st.

2wt. To achieve alternating execution with an FCFS scheduler, the programming has to code either sleep () or other system calls to relinquish the CPU within each thread's computation loop. This allows the second process (the Internet browser) to run.

c) Compare and contrast **pre-emptive** and **non-preemptive** scheduling [2].

Pre-emptive Scheduling	Non-preemptive Scheduling
OS can interrupt processes at any time	Processes run until they voluntarily relinquish or finish
Overhead due to frequent context switches	Lower due to minimal context switches
More complex to implement and manage	Simple (process must relinquish CPU)
Fair CPU utilization	May lead to starvation
Highly responsive	Depend upon