HW 6

Q1.

HW 6 Hayan Al-Machnouk 1

1945954

Q1 :- Since $1KB = 2^{10} B$ $\therefore$ offset 10

a) $3085_{10} \Rightarrow 1100\ 0000\ 1101_2$

3 is page # 13 is offset

b) $42095_{10} \Rightarrow 1010\ 0100\ 0110\ 1111$

41 is page # 111 is offset

c) $215201 \Rightarrow 0011\ 0100\ 1000\ 1010\ 0001$

207 is page # 161 is offset

d) $650060 \Rightarrow 1001\ 1110\ 1011\ 0001\ 0000$

634 is page # 784 offset

e) $2000001 \Rightarrow 0001\ 1110\ 1000\ 0100\ 1000\ 0001$

1953 is page # 129 offset

Q2.

Q2:-

a) $\frac{2^{32}}{2^{12}} = 2^{32-12} = 2^{20} = 1MB$

b) $\frac{2^{27}}{2^{12}} = 2^{27-12} = 2^{17} = 128KB$

Q3.

Internal fragmentation: refers to the scenario where a portion of the memory assigned to a process goes unused, as it is too large for the process' requirements. This results in the inefficient utilization of available memory resources.

External fragmentation: occurs when there is sufficient available memory to satisfy a request or accommodate a process, but it is not contiguous, making it unusable. In this scenario, the memory space is fragmented into smaller, non-contiguous blocks, causing difficulty in allocating the required memory to a process.

The difference in summary, internal fragmentation is caused by assigning too large a memory block to a process, while external fragmentation is caused by fragmentation of the memory space into smaller, non-contiguous blocks.

Q4.

Process Size					
P1	P2	P3	P4	P5	P6
200MB	15MB	185MB	75MB	175MB	80MB

Partition Size					
S1	S2	S3	S4	S5	S6
100MB	170MB	40MB	205MB	300MB	185MB

Running the algorithms "see appendix" we get the following results.

First-Fit Algorithm Result		
Process	**Partition Taken**	**Space Remaining**
P1	**S4**	**205-200 = 5MB**
P2	**S1**	**100-15 = 85MB**
P3	**S5**	**300-185 = 115MB**
P4	**S1**	**85-75 = 10MB**
P5	**S6**	**185-175 = 10MB**
P6	**S2**	**170-80 = 90MB**

Best-Fit Algorithm Result		
Process	**Partition Taken**	**Space Remaining**
P1	**S4**	**205-200 = 5MB**
P2	**S3**	**40-15 = 25MB**
P3	**S6**	**185-185 = Zero**
P4	**S1**	**100-75 = 25MB**
P5	**S5**	**300-175 = 125MB**
P6	**S5**	**125-80 = 45MB**

Worst-Fit Algorithm Result		
Process	**Partition Taken**	**Space Remaining**
P1	**S5**	**300-200 = 100MB**
P2	**S4**	**205-15 = 190MB**
P3	**S4**	**190-185 = 5MB**
P4	**S6**	**185-75 = 110MB**
P5	**-**	**Not Allocated**
P6	**S2**	**170-80 = 90MB**

Clearly Best-Fit algorithm is the most efficient, while Worst-Fit algorithm is the least efficient. In addition, First-Fit gave decent results, but its variance depends on case.

Q5.

The purpose of paging the page tables is to break down large data structures, such as page tables, into smaller units called pages. Each page can be independently managed and stored in physical memory or on disk, allowing for better control over memory access. By breaking down large data structures into smaller pages, the operating system can manage memory more effectively, reducing the amount of physical memory required to hold the page tables in memory and improving the overall performance of the system.

Q6.

LRU Algorithm																			
7	2	3	1	2	5	3	4	6	7	7	1	0	5	4	6	2	3	0	1
7	7	7	1	1	1	3	3	3	7	7	7	7	5	5	5	2	2	2	1
	2	2	2	2	2	2	4	4	4	4	1	1	1	4	4	4	3	3	3
		3	3	3	5	5	5	6	6	6	6	0	0	0	6	6	6	0	0
X	X	X	X		X	X	X	X	X		X	X	X	X	X	X	X	X	X

Total Page Faults is 18

FIFO Algorithm																			
7	2	3	1	2	5	3	4	6	7	7	1	0	5	4	6	2	3	0	1
7	7	7	1	1	1	1	1	6	6	6	6	0	0	0	6	6	6	0	0
	2	2	2	2	5	5	5	5	7	7	7	7	5	5	5	2	2	2	1
		3	3	3	3	3	4	4	4	4	1	1	1	4	4	4	3	3	3
X	X	X	X		X		X	X	X		X	X	X	X	X	X	X	X	X

Total Page Faults is 17

Optimal Algorithm																			
7	2	3	1	2	5	3	4	6	7	7	1	0	5	4	6	2	3	0	1
7	7	7	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	5	5	5	5	5	5	5	5	5	4	6	2	3	3	3
		3	3	3	3	3	4	6	7	7	7	0	0	0	0	0	0	0	0
X	X	X	X		X		X	X	X			X		X	X	X	X		

Total Page Faults is 13

Q7.

- **TLB miss with no page fault:** This scenario occurs when the virtual address is not present in the TLB, but the page corresponding to the virtual address is already loaded into physical memory. In this case, the MMU performs a page table lookup to translate the virtual address to the physical address, but there is no page fault as the page is already present in memory.
- **TLB miss and page fault:** This scenario occurs when the virtual address is not present in the TLB and the page corresponding to the virtual address is not loaded into physical memory. In this case, the MMU performs a page table lookup and realizes that the page is not present in memory, leading to a page fault. The operating system then performs a page-in operation to load the page into physical memory and updates the TLB and page tables to reflect the new mapping.
- **TLB hit and no page fault:** This scenario occurs when the virtual address is present in the TLB and the page corresponding to the virtual address is already loaded into physical memory. In this case, the MMU can immediately translate the virtual address to the physical address without performing a page table lookup, and there is no page fault as the page is already present in memory.
- **TLB hit and page fault:** This scenario is not possible, as if the virtual address is present in the TLB, the corresponding page would have already been loaded into physical memory, thus there would be no need for a page fault to occur. The presence of the virtual address in the TLB indicates that the page has already been loaded and its physical address is known, so there would be no need for the operating system to perform a page-in operation.

Q8.

Q8 :- the address in binary form

0x11123456 ⇒ 0001 0001 0001 0010 0011 $\underbrace{0100\ 0101\ 0110}$

Since page size is 2^{12}, Page # is 0x456

Remaining bits are displacement on table

Q9.

Yes, other user threads belonging to the same process as the user thread that incurred a page fault will also be affected by the page fault.

In a many-to-one mapping of user threads to kernel threads, multiple user threads share the same kernel thread and run concurrently in the same process address space. As a result, all user threads within a process share the same page tables, which are used to map virtual memory addresses to physical memory addresses.

When one user thread incurs a page fault while accessing its stack, the operating system blocks the kernel thread associated with that process, as the page must be brought from disk into physical memory. During this time, all other user threads belonging to the same process are also blocked and unable to run, as they are also using the same kernel thread and share the same page tables.

Therefore, when one user thread incurs a page fault, all other user threads belonging to the same process will also have to wait for the faulting page to be brought into memory. This is because the operating system only allows one kernel thread to access the virtual memory of a process at a time, so all user threads within the process are blocked until the page fault is resolved and the kernel thread is unblocked.

In summary, in a many-to-one mapping of user threads to kernel threads, all user threads within a process share the same page tables and are blocked together when a page fault occurs in any of the user threads.

Q10.

In a one-level indirect addressing scheme, the memory access instructions contain an indirect memory reference, which is a pointer that specifies the address of the data to be accessed. When a program tries to access a memory location using one-level indirect addressing, the operating system first must translate the virtual address stored in the pointer to a physical memory address.

When all of the pages of a program are currently non-resident, this means that none of the pages of the program have been loaded into physical memory. In this scenario, when the first instruction of the program is an indirect memory load operation, a page fault will occur as the operating system must load the page containing the virtual address stored in the pointer into physical memory.

If the operating system is using a per-process frame allocation technique and only two pages are allocated to this process, this means that the operating system can only provide two pages of physical memory to the process. In this scenario, when the first page is loaded into physical memory, there will be only one page of physical memory left. When the next page fault occurs, the operating system must choose which page to evict from physical memory to make room for the new page. This process is known as page replacement and can have a significant impact on the performance of the system.

In conclusion, in a one-level indirect addressing scheme, a sequence of page faults occurs when all of the pages of a program are currently non-resident, and the first instruction of the program is an indirect memory load operation. The number of page faults incurred and the performance of the system can be impacted by the per-process frame allocation technique used by the operating system, as well as the size of the physical memory available to the process.

Appendix:

First Fit Algorithm

```cpp
// Function to allocate memory to blocks as per First fit algorithm
void First_Fit(int block_size[], int total_blocks, int process_size[], int total_process) {
    int allocation[total_process];
    memset(allocation, -1, sizeof(allocation));
    //this for loop wll pick eact process and allocate a first fit block to it
    for (int i = 0; i < total_process; i++) {
        for (int j = 0; j < total_blocks; j++) {
            if (block_size[j] >= process_size[i]) {
                allocation[i] = j;
                block_size[j] -= process_size[i];
                break;
            }
        }
    }
    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < total_process; i++) {
        cout << " " << i+1 << "\t\t" << process_size[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
            cout << endl;
    }
}
```

Figure 1: First Fit Algorithm in C++

Best Fit Algorithm

```
// To allocate the memory to blocks as per Best fit algorithm
void Best_Fit(int bsize[], int m, int psize[], int n) {
   // To store block id of the block allocated to a process
   int alloc[n];
   // Initially no block is assigned to any process
    memset(alloc, -1, sizeof(alloc));
   // find suitable blocks according to its process size
   for (int i=0; i<n; i++) {
      // Find the best fit block for current process
      int bestIdx = -1;
      for (int j=0; j<m; j++) {
         if (bsize[j] >= psize[i]) {
            if (bestIdx == -1) {bestIdx = j;}
            else if (bsize[bestIdx] > bsize[j]) {bestIdx = j;}
         }
      }
      // If we could find a block for current process
      if (bestIdx != -1) {
         // allocate block j to p[i] process
         alloc[i] = bestIdx;
         // Reduce available memory in this block.
         bsize[bestIdx] -= psize[i];
      }
   }
   cout << "\nProcess No.\tProcess Size\tBlock no.\n";
   for (int i = 0; i < n; i++) {
      cout << " " << i+1 << "\t\t\t\t" << psize[i] << "\t\t\t\t";
      if (alloc[i] != -1)
         cout << alloc[i] + 1;
      else
         cout << "Not Allocated";
         cout << endl;
   }
}
```

Figure 2: Best Fit Algorithm in C++

Worst Fit Algorithm

```cpp
// Function to allocate memory to blocks as per worst fit algorithm
void Worst_Fit(int blockSize[], int m, int processSize[], int n)
{
    // Stores block id of the block allocated to a process
    int allocation[n];
    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));
    // find suitable blocks according to its process size
    for (int i=0; i<n; i++)
    {   // Find the best fit block for current process
        int wstIdx = -1;
        for (int j=0; j<m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                if (wstIdx == -1) {wstIdx = j;}
                else if (blockSize[wstIdx] < blockSize[j]) {wstIdx = j;}
            }
        }
        // If we could find a block for current process
        if (wstIdx != -1)
        {   // allocate block j to p[i] process
            allocation[i] = wstIdx;
            // Reduce available memory in this block.
            blockSize[wstIdx] -= processSize[i];
        }
    }
    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++)
    {
        cout << "    " << i+1 << "\t\t" << processSize[i] << "\t\t";
        if (allocation[i] != -1) {cout << allocation[i] + 1;}
        else {cout << "Not Allocated";}
        cout << endl;
    }
```

Figure 3: Worst Fit Algorithm in C++