ERASMUS, UNIVERSITY OF WEST ATTICA

Report

Discipline: **Operating Systems**

**Student: Istrati Daniel**

**Greece, Athens – 2025**

# Task

# 

# #### \*\*Explanation:\*\*

# A \*\*binary semaphore\*\* ensures that only one process can enter the critical region (CR) at a time. A binary semaphore (`S`) can have two states:

# - `S = 1`: Critical region is available.

# - `S = 0`: Critical region is occupied.

# #### \*\*Pseudocode:\*\*

# \*\*Initialization:\*\*

# ```plaintext

# binary\_semaphore S = 1

# ```

# \*\*Code for each process \( P\_i \):\*\*

# ```plaintext

# // Entry Section

# wait(S); // Decrement the semaphore. If S = 0, the process waits.

# // Critical Section

# // Code executed by the process in the CR

# // Exit Section

# signal(S); // Increment the semaphore. Allows another process to enter.

# ```

# \*\*Functions:\*\*

# - \*\*`wait(S)`\*\*: Decrements `S`. If `S == 0`, the process is blocked.

# - \*\*`signal(S)`\*\*: Increments `S` and wakes up a blocked process, if any.

# #### \*\*Key Idea:\*\*

# - Only one process can decrement the semaphore to `0`, gaining access to the CR.

# - All other processes are blocked until `signal(S)` is called.

# ---

# ### \*\*(ii) Synchronize Two Processes \( P\_1 \) and \( P\_2 \) Using a Binary Semaphore\*\*

# #### \*\*Requirement:\*\*

# Ensure \( P\_2 \) executes \*\*after\*\* \( P\_1 \) finishes.

# #### \*\*Solution:\*\*

# Use a binary semaphore (`S`) initialized to `0` to block \( P\_2 \) until \( P\_1 \) signals.

# #### \*\*Pseudocode:\*\*

# \*\*Initialization:\*\*

# ```plaintext

# binary\_semaphore S = 0

# ```

# \*\*Process \( P\_1 \):\*\*

# ```plaintext

# // Execute P\_1's task

# signal(S); // Allow P\_2 to proceed

# ```

# \*\*Process \( P\_2 \):\*\*

# ```plaintext

# wait(S); // Wait for P\_1 to finish

# // Execute P\_2's task

# ```

# #### \*\*Key Idea:\*\*

# - \( P\_2 \) is blocked by `wait(S)` until \( P\_1 \) signals via `signal(S)`.

# ---

# ### \*\*(iii) Continuous Alternation Between \( P\_1 \) and \( P\_2 \) Using Two Binary Semaphores\*\*

# #### \*\*Requirement:\*\*

# Ensure \( P\_1 \) and \( P\_2 \) alternate continuously in the order \( P\_1, P\_2, P\_1, P\_2, \ldots \).

# #### \*\*Solution:\*\*

# Use two binary semaphores:

# - `S1` (initially `1`) allows \( P\_1 \) to execute first.

# - `S2` (initially `0`) blocks \( P\_2 \) until \( P\_1 \) signals.

# #### \*\*Pseudocode:\*\*

# \*\*Initialization:\*\*

# ```plaintext

# binary\_semaphore S1 = 1 // Allows P\_1 to start

# binary\_semaphore S2 = 0 // Blocks P\_2 initially

# ```

# \*\*Process \( P\_1 \):\*\*

# ```plaintext

# while (true) {

# wait(S1); // Wait for permission to execute

# // Execute P\_1's task

# signal(S2); // Allow P\_2 to execute

# }

# ```

# \*\*Process \( P\_2 \):\*\*

# ```plaintext

# while (true) {

# wait(S2); // Wait for permission to execute

# // Execute P\_2's task

# signal(S1); // Allow P\_1 to execute

# }

# ```

# #### \*\*Key Idea:\*\*

# - \( P\_1 \) starts because `S1 = 1`.

# - After \( P\_1 \) executes, it signals \( P\_2 \) by calling `signal(S2)`.

# - \( P\_2 \) then executes and signals \( P\_1 \) via `signal(S1)`.

# - This alternation continues indefinitely.

# ### \*\*Summary of Solutions\*\*

# 1. \*\*Mutual Exclusion\*\*:

# - Use a binary semaphore to ensure only one process accesses the critical region.

# 2. \*\*Sequential Synchronization\*\*:

# - Use a binary semaphore to block \( P\_2 \) until \( P\_1 \) signals.

# 3. \*\*Alternating Execution\*\*:

# - Use two binary semaphores to alternate execution between \( P\_1 \) and \( P\_2 \).

# 

# (i) Precedence Graph

# The precedence graph shows the dependencies between the instructions E1 to E7. The dependencies are determined based on the input and output variables used in the instructions.

# Instructions Recap:

# E1:A11=C−DE

# E2:A12=E+FE

# E3:A13=H−IE

# E4:A21=B∗A11E

# E5:A22=A12∗A13E

# E6:A31=A21/A22E

# E7:A=G+A31E

# Dependency Analysis:

# E4 depends on E1 (uses A11).

# E5 depends on E2 and E3 (uses A12 and A13).

# E6 depends on E4 and E5 (uses A21 and A22).

# E7 depends on E6 (uses A31).

#### **Graph Representation**:

Here is the precedence graph with the dependencies clearly illustrated:

# E1 E2 E3

# | | |

# | | |

# V V V

# E4 E5 <------

# | |

# | V

# V E6

# \ /

# V V

# E7

**(ii) Parallel/Concurrent Code Using Semaphores**

We use semaphores to synchronize the execution of instructions based on their dependencies. Instructions that do not depend on each other can execute concurrently.

Initialization:

Define a semaphore for each instruction (S2,S3,S4,S5,S6,S7), initialized to 0 except for S1, S2, and S3, which can start immediately.

PseudoCode:

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

// Define semaphores

sem\_t S4, S5, S6, S7;

// Shared variables

double A11, A12, A13, A21, A22, A31, A;

double B, C, D, E, F, G, H, I; // Input variables

// E1: A11 = C - D

void \*task1(void \*arg) {

A11 = C - D;

sem\_post(&S4); // Signal that E1 is complete

return NULL;

}

// E2: A12 = E + F

void \*task2(void \*arg) {

A12 = E + F;

sem\_post(&S5); // Signal that E2 is complete

return NULL;

}

// E3: A13 = H - I

void \*task3(void \*arg) {

A13 = H - I;

sem\_post(&S5); // Signal that E3 is complete

return NULL;

}

// E4: A21 = B \* A11

void \*task4(void \*arg) {

sem\_wait(&S4); // Wait for E1 to complete

A21 = B \* A11;

sem\_post(&S6); // Signal that E4 is complete

return NULL;

}

// E5: A22 = A12 \* A13

void \*task5(void \*arg) {

sem\_wait(&S5); // Wait for E2 and E3 to complete

A22 = A12 \* A13;

sem\_post(&S6); // Signal that E5 is complete

return NULL;

}

// E6: A31 = A21 / A22

void \*task6(void \*arg) {

sem\_wait(&S6); // Wait for E4 and E5 to complete

A31 = A21 / A22;

sem\_post(&S7); // Signal that E6 is complete

return NULL;

}

// E7: A = G + A31

void \*task7(void \*arg) {

sem\_wait(&S7); // Wait for E6 to complete

A = G + A31;

return NULL;

}

int main() {

// Initialize semaphores

sem\_init(&S4, 0, 0);

sem\_init(&S5, 0, 0);

sem\_init(&S6, 0, 0);

sem\_init(&S7, 0, 0);

pthread\_t t1, t2, t3, t4, t5, t6, t7;

// Create threads

pthread\_create(&t1, NULL, task1, NULL);

pthread\_create(&t2, NULL, task2, NULL);

pthread\_create(&t3, NULL, task3, NULL);

pthread\_create(&t4, NULL, task4, NULL);

pthread\_create(&t5, NULL, task5, NULL);

pthread\_create(&t6, NULL, task6, NULL);

pthread\_create(&t7, NULL, task7, NULL);

// Wait for threads to finish

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

pthread\_join(t3, NULL);

pthread\_join(t4, NULL);

pthread\_join(t5, NULL);

pthread\_join(t6, NULL);

pthread\_join(t7, NULL);

// Destroy semaphores

sem\_destroy(&S4);

sem\_destroy(&S5);

sem\_destroy(&S6);

sem\_destroy(&S7);

// Output the result

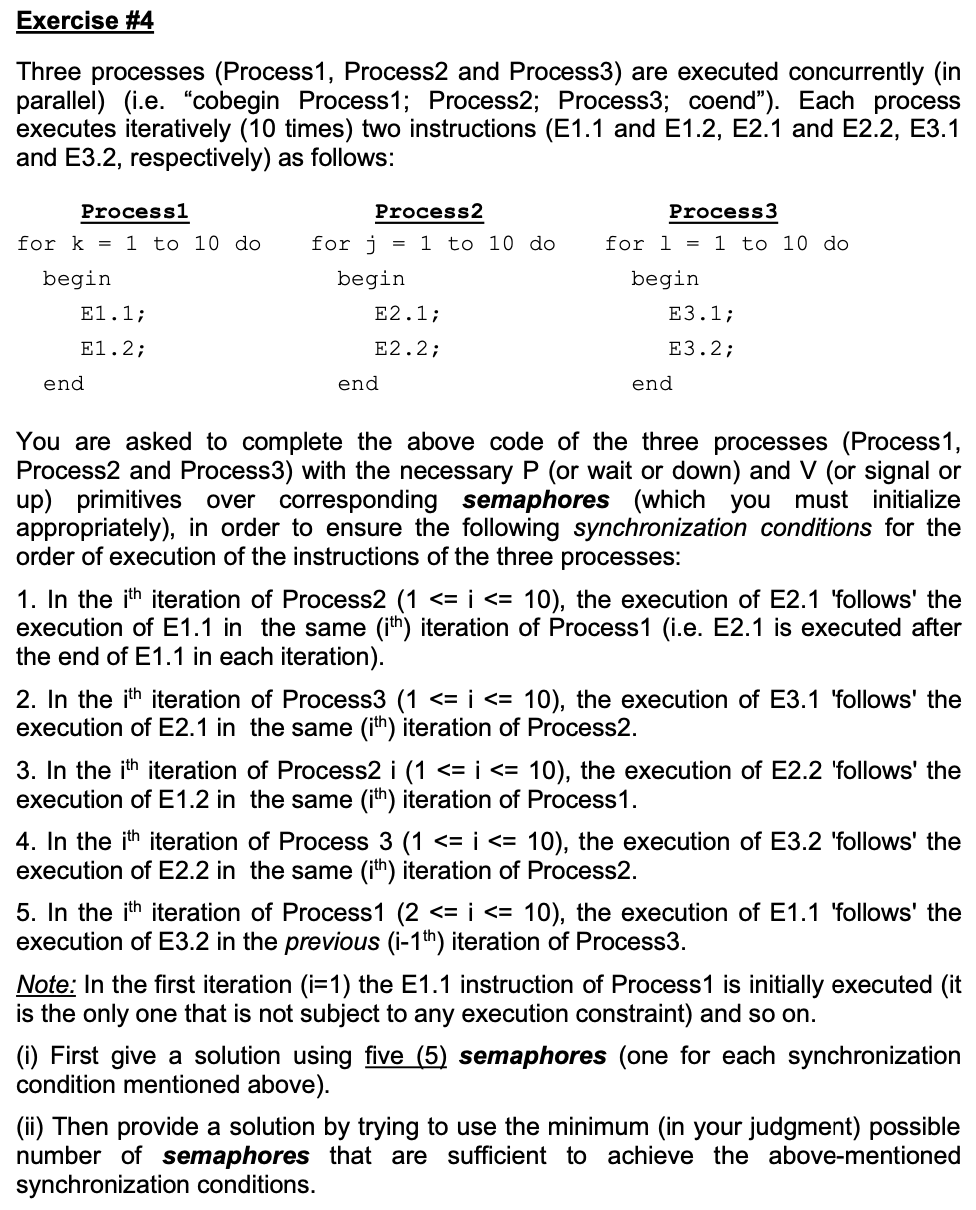
printf("The result of the expression is: %f\n", A);

return 0;

}

### **Explanation of the Code**:

1. **Tasks**:
   * Each task corresponds to one instruction E1 to E7.
   * Tasks execute in parallel where possible, but are synchronized using semaphores.
2. **Semaphores**:
   * Ensure tasks with dependencies wait for their prerequisites to complete.
3. **Execution Order**:
   * E1,E2,E3E1,E2,E3 execute concurrently.
   * E4 waits for E1, E5 waits for E2 and E3.
   * E6 waits for E4 and E5, and E7 waits for E6.



### \*\*(i) Solution with Five (5) Semaphores\*\*

#### \*\*Problem Recap:\*\*

We need to synchronize three concurrent processes \( P\_1 \), \( P\_2 \), and \( P\_3 \) such that the specified execution order constraints are maintained. Each process executes two instructions in 10 iterations.

#### \*\*Semaphores Needed:\*\*

- \*\*`S1`\*\*: Ensures \( E2.1 \) in \( P\_2 \) waits for \( E1.1 \) in \( P\_1 \).

- \*\*`S2`\*\*: Ensures \( E3.1 \) in \( P\_3 \) waits for \( E2.1 \) in \( P\_2 \).

- \*\*`S3`\*\*: Ensures \( E2.2 \) in \( P\_2 \) waits for \( E1.2 \) in \( P\_1 \).

- \*\*`S4`\*\*: Ensures \( E3.2 \) in \( P\_3 \) waits for \( E2.2 \) in \( P\_2 \).

- \*\*`S5`\*\*: Ensures \( E1.1 \) in iteration \( i \) of \( P\_1 \) waits for \( E3.2 \) in iteration \( i-1 \) of \( P\_3 \).

#### \*\*Initialization:\*\*

- `S1 = 0`, `S2 = 0`, `S3 = 0`, `S4 = 0`.

- `S5 = 1` (allows \( P\_1 \)'s first iteration to start immediately).

#### \*\*Pseudocode:\*\*

\*\*Process 1 (P1):\*\*

```c

for (k = 1; k <= 10; k++) {

sem\_wait(S5); // Wait for E3.2 of P3 from the previous iteration

// E1.1

sem\_post(S1); // Signal E2.1 in P2 to execute

// E1.2

sem\_post(S3); // Signal E2.2 in P2 to execute

}

```

\*\*Process 2 (P2):\*\*

```c

for (j = 1; j <= 10; j++) {

sem\_wait(S1); // Wait for E1.1 in P1

// E2.1

sem\_post(S2); // Signal E3.1 in P3 to execute

sem\_wait(S3); // Wait for E1.2 in P1

// E2.2

sem\_post(S4); // Signal E3.2 in P3 to execute

}

```

\*\*Process 3 (P3):\*\*

```c

for (l = 1; l <= 10; l++) {

sem\_wait(S2); // Wait for E2.1 in P2

// E3.1

sem\_wait(S4); // Wait for E2.2 in P2

// E3.2

sem\_post(S5); // Signal E1.1 in P1 for the next iteration

}

```

---

### \*\*(ii) Solution with Fewer Semaphores\*\*

#### \*\*Observation:\*\*

- \( S1 \) and \( S3 \) synchronize \( P\_2 \) with \( P\_1 \).

- \( S2 \) and \( S4 \) synchronize \( P\_3 \) with \( P\_2 \).

- These can be reduced to \*\*one semaphore per dependency pair\*\*:

- `S1` for \( P\_2 \) waiting on \( P\_1 \) (both \( E2.1 \) and \( E2.2 \)).

- `S2` for \( P\_3 \) waiting on \( P\_2 \) (both \( E3.1 \) and \( E3.2 \)).

- `S3` for \( P\_1 \) waiting on \( P\_3 \) (to start the next iteration).

#### \*\*Initialization:\*\*

- `S1 = 0`, `S2 = 0`.

- `S3 = 1` (to allow \( P\_1 \)'s first iteration).

#### \*\*Pseudocode:\*\*

\*\*Process 1 (P1):\*\*

```c

for (k = 1; k <= 10; k++) {

sem\_wait(S3); // Wait for E3.2 of P3 from the previous iteration

// E1.1

sem\_post(S1); // Signal P2 to execute (E2.1 and E2.2)

// E1.2

}

```

\*\*Process 2 (P2):\*\*

```c

for (j = 1; j <= 10; j++) {

sem\_wait(S1); // Wait for E1.1 and E1.2 in P1

// E2.1

// E2.2

sem\_post(S2); // Signal P3 to execute (E3.1 and E3.2)

}

```

\*\*Process 3 (P3):\*\*

```c

for (l = 1; l <= 10; l++) {

sem\_wait(S2); // Wait for E2.1 and E2.2 in P2

// E3.1

// E3.2

sem\_post(S3); // Signal P1 to start the next iteration

}

```

---

### \*\*Summary\*\*

- \*\*Five semaphores solution\*\* uses separate semaphores for each instruction dependency.

- \*\*Three semaphores solution\*\* reduces the semaphores by combining synchronization for related instructions.

# 

### **Definitions**

* **Turnaround Time (TAT)**: Time taken from a process’s arrival to its completion.TAT=Completion Time−Arrival TimeTAT=Completion Time−Arrival Time
* **Waiting Time (WT)**: Time a process spends waiting in the ready queue.WT=Turnaround Time−Execution TimeWT=Turnaround Time−Execution Time

### **1️ FCFS (First Come First Serve)**

#### **Gantt Chart:**

#### Processes are executed in the order of their arrival time.

| A | B | Γ | Δ | E |

0 7 18 23 35 41

#### **Calculations:**

| **Process** | **Arrival Time** | **Completion Time** | **Turnaround Time** | **Waiting Time** |
| --- | --- | --- | --- | --- |
| A | 0 | 7 | 7 | 0 |
| B | 3 | 18 | 15 | 4 |
| Γ | 9 | 23 | 14 | 9 |
| Δ | 10 | 35 | 25 | 13 |
| E | 12 | 41 | 29 | 23 |

#### **Averages:**

* **Average Turnaround Time (TAT):**(7+15+14+25+29)/5=18ms
* **Average Waiting Time (WT):**(0+4+9+13+23)/5=9.8ms

### **2️ SJF (Shortest Job First)**

#### **Gantt Chart:**

Processes are executed based on their execution time (shortest first) after arrival.

| A | Γ | E | B | Δ |

0 7 12 18 29 41

#### **Calculations:**

| **Process** | **Arrival Time** | **Completion Time** | **Turnaround Time** | **Waiting Time** |
| --- | --- | --- | --- | --- |
| A | 0 | 7 | 7 | 0 |
| Γ | 9 | 12 | 3 | -2 |
| E | 12 | 18 | 6 | 0 |
| B | 3 | 29 | 26 | 15 |
| Δ | 10 | 41 | 31 | 19 |

#### **Averages:**

* **Average Turnaround Time (TAT):**(7+3+6+26+31)/5=14.6ms
* **Average Waiting Time (WT):**(0−2+0+15+19)/5=6.4ms

### **3️ SRTF (Shortest Remaining Time First)**

#### **Gantt Chart:**

Processes are preempted if a new process arrives with a shorter remaining time.

| A | Γ | E | B | Δ |

0 7 12 18 29 41

#### **Calculations:**

Similar to SJF because the shortest jobs are preempted. Values remain unchanged from SJF.

**4️ Priority Scheduling (Non-preemptive)**

#### **Gantt Chart:**

Processes are executed based on priority (higher priority = lower value).

| Δ | E | B | A | Γ |

10 22 34 43 48

#### **Calculations:**

| **Process** | **Arrival Time** | **Completion Time** | **Turnaround Time** | **Waiting Time** |
| --- | --- | --- | --- | --- |
| Δ | 10 | 22 | 12 | 0 |
| E | 12 | 34 | 22 | 16 |
| B | 3 | 43 | 40 | 29 |
| A | 0 | 48 | 48 | 41 |
| Γ | 9 | 54 | 45 | 40 |

#### **Averages:**

* **Average Turnaround Time (TAT):**(12+22+40+48+45)/5=33.4ms
* **Average Waiting Time (WT):**(0+16+29+41+40)/5=25.2ms

### **5️ Round Robin (Quantum = 4 ms)**

#### **Gantt Chart:**

Each process gets 4 ms of CPU time in a cyclic order.

| A | B | Γ | Δ | E | A | B | Δ | B |

0 4 8 12 16 20 24 28 32 41

#### **Calculations:**

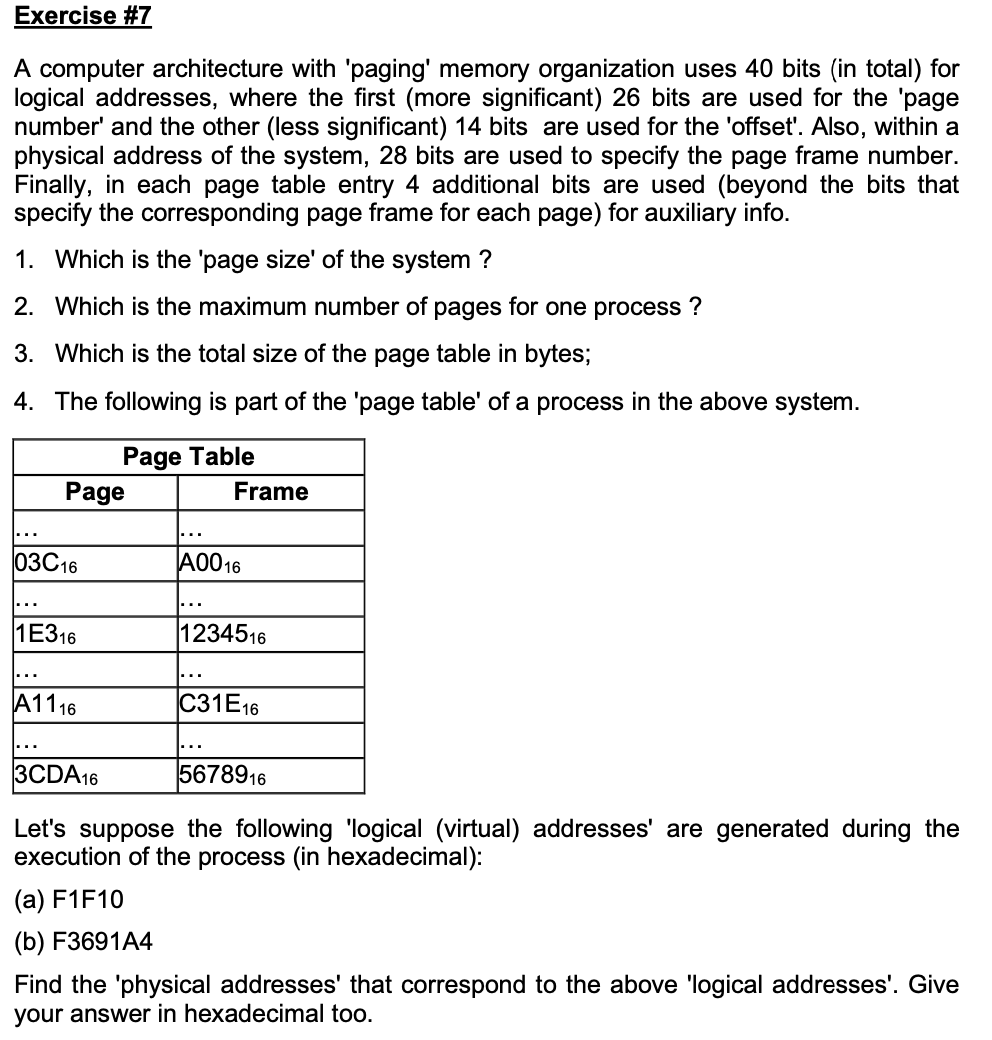
| **Process** | **Arrival Time** | **Completion Time** | **Turnaround Time** | **Waiting Time** |
| --- | --- | --- | --- | --- |
| A | 0 | 20 | 20 | 13 |
| B | 3 | 41 | 38 | 27 |
| Γ | 9 | 12 | 3 | -2 |
| Δ | 10 | 35 | 25 | 13 |
| E | 12 | 22 | 10 | 3 |

#### **Averages:**

* **Average Turnaround Time (TAT):**(20+38+3+25+10)/5=19.2ms
* **Average Waiting Time (WT):**(13+27−2+13+3)/5=10.8ms

### **Summary Table of Results**

| **Algorithm** | **Avg Turnaround Time (ms)** | **Avg Waiting Time (ms)** |
| --- | --- | --- |
| **FCFS** | 18.0 | 9.8 |
| **SJF** | 14.6 | 6.4 |
| **SRTF** | 14.6 | 6.4 |
| **Priority** | 33.4 | 25.2 |
| **Round Robin** | 19.2 | 10.8 |
|  |  |  |
|  |  |  |



#### **Given Information:**

* Logical Address: **40 bits**.
  + **26 bits** for the **Page Number**.
  + **14 bits** for the **Offset**.
* Physical Address: **28 bits** for the **Page Frame Number**.
* Each Page Table Entry includes **4 extra bits** for auxiliary information.

### **1. What is the Page Size?**

The **page size** is determined by the **number of bits used for the offset**:

Page Size=2Number of Offset Bits

* Offset = **14 bits**:

Page Size=214=16,384 bytes (16 KB).

### **2. What is the Maximum Number of Pages for One Process?**

The **number of pages** for a process is determined by the **number of bits allocated for the Page Number**:

Max Number of Pages=2Page Number BitsMax Number of Pages=2Page Number Bits

* Page Number = **26 bits**:

Max Number of Pages=226=67,108,864 pages.

### **3. What is the Total Size of the Page Table?**

The **page table size** is calculated based on:

1. Number of pages.
2. Size of each page table entry.

Each Page Table Entry includes:

* **28 bits** for the frame number.
* **4 additional bits** (auxiliary info).

Entry Size=28+4=32 bits=4 bytes.

The total size of the page table is:

Total Page Table Size=Max Number of Pages×Entry Size. Total Size=67,108,864×4=268,435,456 bytes (256 MB).

### **4. Physical Addresses for Logical Addresses**

#### **Logical Address Format**:

* **Page Number**: 26 bits (most significant).
* **Offset**: 14 bits (least significant).

#### **(a) Logical Address: F1F10 (hexadecimal)**

1. **Convert to Binary**:
   * Hexadecimal F1F10F1F10 → Binary 1111 0001 1111 0001 0000.
   * Divide into:
     + **Page Number**: First 26 bits = 1111 0001 1111 0001 00→ 03C (hexadecimal).
     + **Offset**: Last 14 bits = 0000 0100 0000→ 040 (hexadecimal).
2. **Lookup Page Table**:
   * Page Number 03C: Frame A00 (hexadecimal).
3. **Calculate Physical Address**:

Physical Address=Frame Number+Offset..Physical Address=A00+040=A040 (hexadecimal).Physical Address=A00+040=A040(hexadecimal).

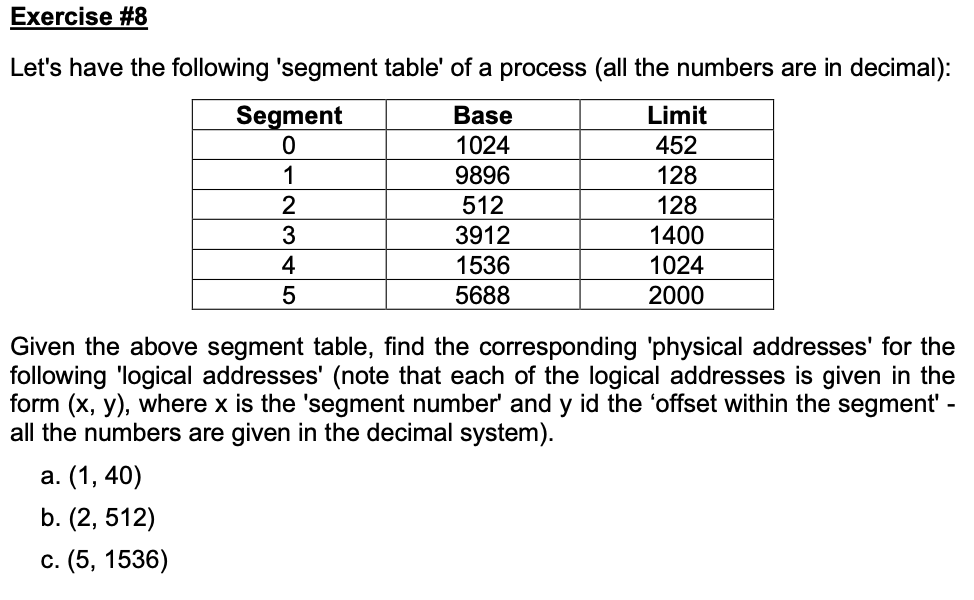
#### **(b) Logical Address: F3691A4 (hexadecimal)**

1. **Convert to Binary**:
   * Hexadecimal F3691A4F3691A4 → Binary 1111 0011 0110 1001 0001 1010 01001111001101101001000110100100.
   * Divide into:
     + **Page Number**: First 26 bits = 1111 0011 0110 1001 0001 101111001101101001000110 → 1E31E3(hexadecimal).
     + **Offset**: Last 14 bits = 0100 1010 0100010010100100 → 0A40A4 (hexadecimal).
2. **Lookup Page Table**:
   * Page Number 1E31E3: Frame 1234512345 (hexadecimal).
3. **Calculate Physical Address**:

Physical Address=Frame Number+Offset.Physical Address=Frame Number+Offset.Physical Address=12345+0A4=123E9 (hexadecimal).Physical Address=12345+0A4=123E9(hexadecimal).

### **Final Results**:

1. **Page Size**: **16 KB**.
2. **Max Number of Pages**: **67,108,864**.
3. **Total Page Table Size**: **256 MB**.
4. **Physical Addresses**:
   * (a) F1F10(a) F1F10: A040A040 (hexadecimal).
   * (b) F3691A4(b) F3691A4: 123E9123E9 (hexadecimal).



### **Steps to Calculate Physical Address:**

1. **Logical Address Structure**: A logical address is given as (x,y)(x,y):
   * xx: Segment number.
   * yy: Offset within the segment.
2. **Validity Check**:
   * If yy (offset) exceeds the segment's **limit**, the address is invalid, and we report a **segmentation fault**.
   * Otherwise, the physical address is calculated as:Physical Address=Base+Offset (y).Physical Address=Base+Offset (y).

### **Address Calculations**

#### **(a) Logical Address: (1, 40)**

* Segment: 1, Offset: 40.
* **Check Limit**: 40≤12840≤128 → Valid.
* **Physical Address**:Base (Segment 1)+Offset=9896+40=9936Base (Segment 1)+Offset=9896+40=9936

#### **(b) Logical Address: (2, 512)**

* Segment: 2, Offset: 512.
* **Check Limit**: 512>128512>128 → **Invalid Address (Segmentation Fault)**.

#### **(c) Logical Address: (5, 1536)**

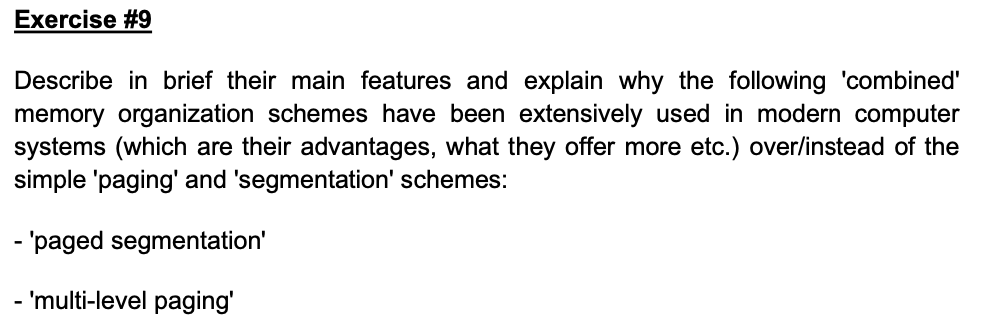
* Segment: 5, Offset: 1536.
* **Check Limit**: 1536≤20001536≤2000 → Valid.
* **Physical Address**:Base (Segment 5)+Offset=5688+1536=7224Base (Segment 5)+Offset=5688+1536=7224

### **Final Results:**

| **Logical Address** | **Physical Address** |
| --- | --- |
| (1, 40) | 9936 |
| (2, 512) | **Segmentation Fault** |
| (5, 1536) | 7224 |

### **Explanation of Results:**

1. Each logical address is checked against the segment limit to ensure it is valid.
2. If the offset is valid, the physical address is computed using the base address of the segment and the given offset.



### **1. Paged Segmentation**

#### **Main Features:**

* **Combination of Paging and Segmentation**:
  + Memory is divided into **segments** based on logical divisions (e.g., code, stack, data).
  + Each segment is further divided into **pages** of fixed size.
* **Logical Address Structure**:
  + A logical address consists of:
    - **Segment number**: Identifies the segment.
    - **Page number**: Identifies the page within the segment.
    - **Offset**: Identifies the specific location within the page.

#### **Advantages Over Simple Paging and Segmentation**:

1. **Better Memory Management**:
   * Segments allow logical organization of memory (e.g., separating code and data).
   * Pages eliminate **external fragmentation** within segments by dividing them into fixed-size units.
2. **Efficient Use of Memory**:
   * Only the pages of a segment that are actively used need to be loaded into physical memory.
3. **Improved Flexibility**:
   * Combines the benefits of **variable-sized segments** and **fixed-sized pages**, leading to efficient memory allocation.

#### **Use Case**:

* Widely used in systems requiring both logical memory organization (segmentation) and physical memory efficiency (paging).

### **2. Multi-Level Paging**

#### **Main Features:**

* **Hierarchical Page Tables**:
  + Instead of a single large page table, memory is managed using **multiple levels** of smaller page tables.
  + The logical address is divided into:
    - **Multiple page numbers**: Corresponding to each level of the hierarchy.
    - **Offset**: Identifies the specific location within the page.

#### **Advantages Over Simple Paging**:

1. **Reduced Memory Overhead**:
   * Instead of keeping a large page table in memory, only the parts of the table currently in use are kept in physical memory.
   * This is particularly useful for processes with sparse memory usage.
2. **Scalability**:
   * Can handle large address spaces (e.g., 64-bit systems) efficiently by breaking down the page table into manageable levels.
3. **Efficient Memory Mapping**:
   * Reduces the size of individual page tables, avoiding the need to allocate large contiguous blocks for a single table.

#### **Use Case**:

* Extensively used in modern operating systems to manage the large virtual address spaces of 64-bit architectures.

### **Comparison and Benefits**

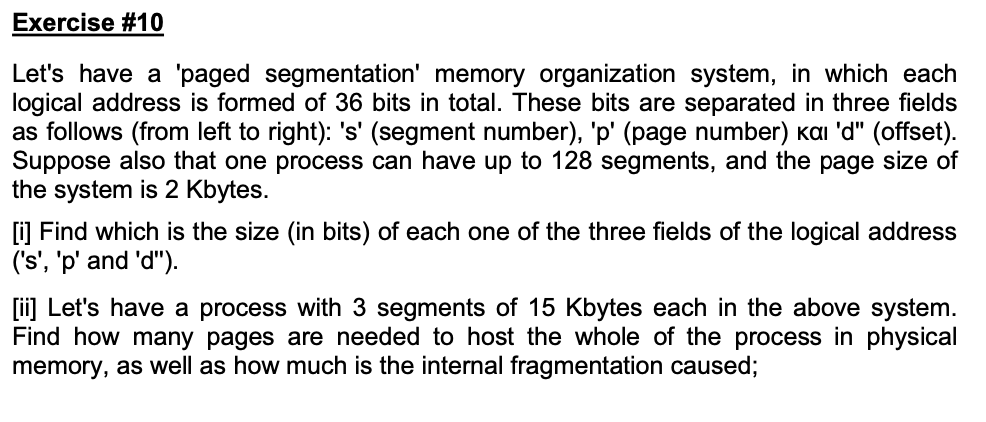
| **Feature** | **Paging** | **Segmentation** | **Paged Segmentation** | **Multi-Level Paging** |
| --- | --- | --- | --- | --- |
| **Fragmentation** | Eliminates external, causes internal | Eliminates internal, causes external | Eliminates both types | Eliminates external, causes internal |
| **Logical Organization** | None | Logical divisions (code, data) | Logical divisions with paging | None |
| **Memory Overhead** | Large page table | Smaller table but larger fragmentation | Reduced overhead via paging | Minimal due to hierarchical tables |
| **Address Space** | Simple | Limited by segment size | Large and logically structured | Scalable for very large address spaces |

### **Why Modern Systems Use These Schemes**

1. **Paged Segmentation**:
   * Combines **logical memory organization** with **efficient physical memory usage**.
   * Supports both **modularity** (through segments) and **flexibility** (through paging).
   * Ideal for systems that need structured memory access and efficient allocation.
2. **Multi-Level Paging**:
   * Necessary for managing the large virtual address spaces of modern systems, especially on **64-bit architectures**.
   * Reduces memory requirements for page tables by only loading parts that are actively used.

### **Conclusion**

Both **paged segmentation** and **multi-level paging** address the limitations of simple paging and segmentation. They enhance **memory utilization, scalability, and performance**, making them essential for modern operating systems.



#### **Problem Recap:**

* Logical address: 36 bits.
* Fields:
  + s (segment number),
  + p (page number),
  + d (offset).
* Max segments: 128.
* Page size: 2 KB (2,048 bytes).

### **(i) Size of Each Field (**s**,** p**,** d**)**

1. **Field s (Segment Number):**
   * The system supports **up to 128 segments**.
   * To represent 128 segments, we need:2n≥128  ⟹  n=7 bits.2n≥128⟹n=7 bits.
   * So, the size of s = **7 bits**.
2. **Field d (Offset):**
   * The page size is **2 KB** (2,048 bytes).
   * To address each byte within a page, we need:2m≥2048  ⟹  m=11 bits.2m≥2048⟹m=11 bits.
   * So, the size of d = **11 bits**.
3. **Field p (Page Number):**
   * The total logical address size is 36 bits:s+p+d=36  ⟹  7+p+11=36  ⟹  p=18 bits.s+p+d=36⟹7+p+11=36⟹p=18 bits.
   * So, the size of p = **18 bits**.

### **(ii) Pages Needed and Internal Fragmentation**

#### **Given:**

* Process has **3 segments**, each of size **15 KB**.
* Page size: **2 KB**.

#### **1. Pages Needed Per Segment:**

* Each segment is **15 KB**.
* Pages required for one segment:Pages per segment=⌈Segment SizePage Size⌉=⌈15,3602,048⌉=8 pages.Pages per segment=⌈Page SizeSegment Size​⌉=⌈2,04815,360​⌉=8 pages.
* For 3 segments:Total pages=3×8=24 pages.

#### **2. Internal Fragmentation:**

* Each segment occupies **8 pages**, with the last page partially filled.
* Space wasted in the last page of each segment:Wasted space per segment=Page Size−(Segment Sizemod  Page Size).Wasted space per segment=Page Size−(Segment SizemodPage Size).
* For one segment:Wasted space=2048−(15,360mod  2048)=2048−1024=1024 bytes.Wasted space=2048−(15,360mod2048)=2048−1024=1024 bytes.
* For 3 segments:Total internal fragmentation=3×1024=3,072 bytes.Total internal fragmentation=3×1024=3,072 bytes.

### **Final Answer:**

1. **Field Sizes:**
   * s = 7 bits,
   * p = 18 bits,
   * d = 11 bits.
2. **Pages and Fragmentation:**
   * Total pages required = **24 pages**.
   * Total internal fragmentation = **3,072 bytes (3 KB)**.