CS 3113 – Intro to Operating Systems – Spring 2025
Project Four: Due April 13, 20205, 11:59 PM

Description: This project extends Project 3 by simulating a segmented memory system. Each process is now allowed to occupy non-contiguous memory segments. A segment table will be used to manage address translation, and logical addresses generated by the CPU will be mapped to physical addresses using this table.

Each process can have up to 6 segments, and the segment table itself, along with its length value, must be stored contiguously in memory.

Enhancements Over Project Three: Key Concepts

Segment Table

- Stored in the **logical memory of the process**.
- Format:
  - segmentTableSize (always even, 2 * number of segments)
  - For each segment: startAddress, size
- Example for 2 segments:
  - segmentTableSize = 4
  - segment 0: start = 100, size = 50
  - segment 1: start = 300, size = 100

PCB Fields (in logical memory after segment table)

Assume N = segmentTableSize + 1

Index (from logical 0)	Field	
0	segmentTableSize	
1 to N-1	segment entries	
N	processID	
N+1	state	
N+2	programCounter	
N+3	instructionBase	
N+4	dataBase	
N+5	memoryLimit	
N+6	cpuCyclesUsed	
N+7	registerValue	
N+8	maxMemoryNeeded	
N+9	mainMemoryBase	<= physical address of segment 0
N+10+	Instructions begin	

Important Assumptions

- Each process can have at most 6 segments
- Segment table + its length must fit in a single memory hole of at least 13 integers
- If no such hole exists, the process remains in the NewJobQueue
- Segment table must be contiguous in memory

Dynamic Memory Allocation Changes (from Project 3)

- Memory is still tracked using a linked list, where each node represents a memory block.
- A process may now be allocated multiple non-contiguous blocks.
- Each allocated block becomes a separate segment in the segment table.
- If the sum of available free block sizes is less than the process's memory requirement, the process cannot be loaded and remains in NewJobQueue.

- Memory coalescing is now performed during the search for free blocks, not only when a process cannot be loaded.
- This ensures better utilization of fragmented memory and helps create larger blocks when possible.

Copying the PCB that needs to be stored across multiple segments:

Since the contents of the PCB are stored across multiple non-contiguous segments, the memory copy mechanism must correctly distribute the PCB fields, instructions, and data into the segments as indicated in the segment table. The following **pseudo-code** demonstrates how this copying mechanism works:

```
// This function copies the logical contents of a process (including its segment table,
// PCB fields, instructions, and data) into the allocated segments as listed in the
segment table.
// The segment table is assumed to begin at logical address 0, and the logical layout is
linear.

void copyProcessToMemory(int* processLogicalMemory, int totalLogicalSize, int* pcb, int*
mainMemory) {
    int segmentTableSize = pcb[0];
    int numSegments = segmentTableSize / 2;

    int logicalIndex = 0;
    for (int i = 0; i < numSegments; i++) {
        int start = pcb[1 + 2 * i];      // physical start of segment i
        int size  = pcb[1 + 2 * i + 1];  // size of segment i

        for (int j = 0; j < size && logicalIndex < totalLogicalSize; j++) {
            mainMemory[start + j] = processLogicalMemory[logicalIndex];
            logicalIndex++;
        }
    }

    if (logicalIndex < totalLogicalSize) {
        cout << "Error: not enough space in allocated segments to hold process." <<
                 endl;
    }
}
```

This ensures that the entire logical memory of the process (including segment table, PCB fields, instructions, and data) is physically distributed as per the allocated segments.

Address Translation (CPU)

When CPU generates a logical address x, the translation steps are:

```
int translateLogicalToPhysical(int logicalAddress, int* pcb) {
    int segmentTableSize = pcb[0];
    int numSegments = segmentTableSize / 2;
    int remaining = logicalAddress;

    for (int i = 0; i < numSegments; i++) {
        int start = pcb[1 + 2*i];
        int length = pcb[1 + 2*i + 1];
        if (remaining < length) {
            return start + remaining;
        } else {
            remaining -= length;
        }
    }
    cout << "Memory violation: address " << logicalAddress << " out of bounds." << endl;
    return -1;
}
```

Let’s walk through it again:

Logical address x = 63

Segment Table (in PCB):

- Segment 0 → Start = 100, Size = 50
- Segment 1 → Start = 300, Size = 100

Steps:

```
remaining = 63

Segment 0: size = 50
→ 63 > 50 → remaining = 13

Segment 1: size = 100
→ 13 < 100 → return 300 + 13 = 313
```

Final result: physical address = 313

Example PCB for Process 1 (2 segments)

Logical Addr	Value	Description
0	4	segmentTableSize (2 segments)
1	100	Segment 0 physical start
2	50	Segment 0 size
3	300	Segment 1 physical start
4	100	Segment 1 size
5	1	processID
6	READY	state
7	0	programCounter
8	15	instructionBase
9	17	dataBase
10	150	memoryLimit
11	20	cpuCyclesUsed
12	42	registerValue
13	150	maxMemoryNeeded
14	100	mainMemoryBase (physical start of seg 0)
15	1	Instruction 1: Compute
16	5	5 iterations

Execution Flow

1. Initialize the memory as a single large free block.
2. For each job in NewJobQueue:
   - Attempt to allocate multiple non-contiguous memory blocks whose total size satisfies the job's memory requirement.
   - During this search, perform memory coalescing to combine adjacent free blocks.
   - If enough memory is available:
     - Allocate a contiguous block of at least 13 integers to hold the segment table.
     - Allocate the necessary segments.
     - Fill the segment table and complete the PCB fields.
     - Copy the contents of the PCB (segment table + metadata + instructions + data) into the allocated segments as per the layout.
     - The PCB, including its segment table and fields, may span across multiple physical memory blocks (segments), as indicated in the segment table.
     - Move the job to the ReadyQueue.
   - If memory is insufficient (either total or segment table space):
     - Leave the job in NewJobQueue.
3. Execute jobs from ReadyQueue according to the scheduling policy from Project Two.
4. For each instruction:
   - Use the segment table to translate logical to physical addresses.

        - Validate bounds and access the correct physical memory location.
5. Upon job termination:
    - Free all memory segments associated with the process.
    - Free the block that holds the segment table.
    - Update the memory linked list accordingly.
6. After a job terminates, recheck NewJobQueue to see if any waiting jobs can now be loaded.
7. Repeat until all jobs have been loaded and executed.

Output Requirements

In addition to the outputs from Project 3, this project must:

- When a process is **loaded successfully**:
  `Process <PID> loaded with segment table stored at physical address <X>`
- If the loader cannot find a memory hole ≥ 13:
  `Process <PID> could not be loaded due to insufficient contiguous space for segment table.`
- When a logical address is translated:
  `Logical address <X> translated to physical address <Y> for Process <PID>`
- If address is invalid:
  `Memory violation: address <X> out of bounds for Process <PID>`
- At process termination:
  `Process <PID> terminated and freed memory blocks.`

Rules and Submission Policy

All projects in this course are **individual assignments** and are not to be completed as group work. The use of **automatic plagiarism detection tools** will be employed to ensure academic integrity. Collaboration with others or the use of outside third parties to complete this project is strictly prohibited. Submissions must be made through **GradeScope**, where automated grading will be conducted. Additionally, manual grading may be performed to ensure correctness and adherence to requirements.

Several input files will be used to evaluate your program. While a subset of these input files will be provided to you for testing, additional files not shared beforehand will also be used during grading. Your score on this project will depend on producing correct results for all input files, including the undisclosed ones used in GradeScope evaluation.

All programs must be written in **C++** and must compile successfully using the **GNU C++ compiler**. It is your responsibility to ensure that your program adheres to these requirements.

The course syllabus provides more details on the late and submission policy. You should also go through that.