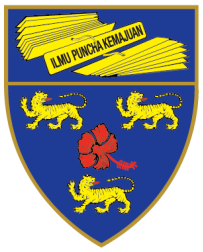


UNIVERSITI MALAYA

WIA2004 Operating Systems

Lab 4: Memory Management Techniques

Group : F5U

Occ: 10

Lecturer: Dr. Fazidah Othman

GROUP MEMBERS	MATRICS NUMBER
ZHANG ZHIYANG	S2193685
MEILIN LI	S2174975
HUMYRA TASMIA	S2176677
HUSNA NADIAH BINTI ROSTHAM	22060027
HUSNA BINTI IHSANUDDIN	U2102305

Table of Content

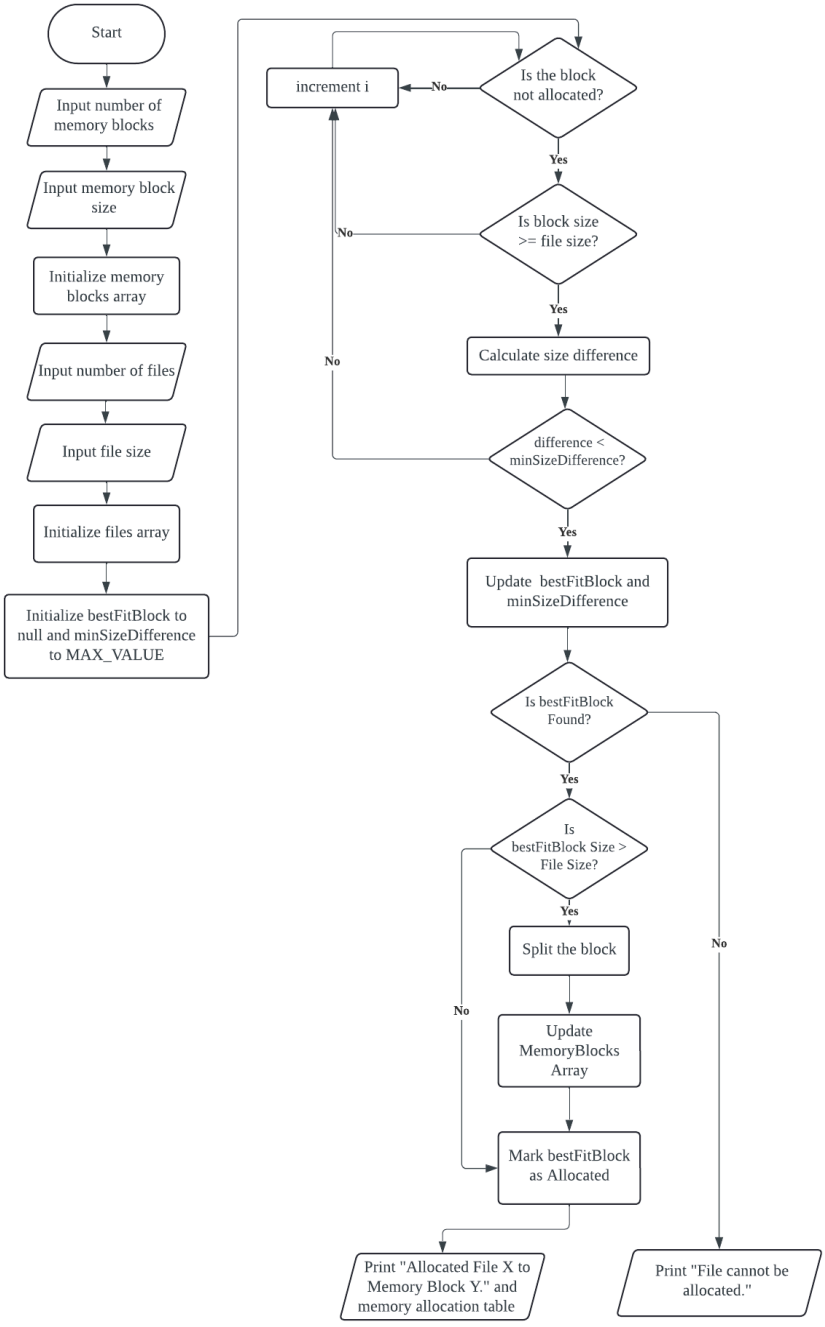
1.0 Introduction

Memory management in computer operating systems involves organizing and allocating memory resources to processes running concurrently in a multiprogramming environment. It ensures efficient division of memory space among processes, optimizing allocation while minimizing wastage and preventing fragmentation.

The best-fit algorithm seeks out the smallest available partition in memory that is sufficiently large for the process. In this approach, the operating system scans through all memory segments to identify the closest size match to the allocated process. While best fit effectively reduces external fragmentation by assigning processes to the smallest possible partitions, yielding highly efficient memory allocation, it does require additional time due to the thorough search involved in finding the suitable partition.

2.0 Methodology

Flowchart



Pseudocode

```
1. Prompt user to enter the number of memory blocks (numBlocks)

2. Initialize an array memoryBlocks to store MemoryBlock objects

3. For each process of numBlocks:
   3.1 Prompt user to enter the size of the block
   3.2 Initialize block number and size, and store it in memoryBlocks

4. Prompt user to enter the number of files (numFiles)

5. Initialize an array files to store File objects

6. For each file i from 1 to numFiles:
   6.1 Prompt user to enter the size of the file
   6.2 Initialize file number and size, and store it in files

7. For each file in files array:
   7.1 Initialize bestFitBlock to null
   7.2 Initialize minSizeDifference to MAX_VALUE
   7.3 For each block in memoryBlocks:
      7.3.1 If block is not allocated and block's size is >= file's size:
         7.3.2 Calculate size difference between block size and file size
         7.3.3 If size difference is less than minSizeDifference:
            7.3.4 Update minSizeDifference
            7.3.5 Update bestFitBlock to current block
   7.4 If bestFitBlock is found:
      7.4.1 Mark bestFitBlock as allocated
      7.4.2 Print message about file allocation to the block
      7.4.3 If bestFitBlock's size > file's size:
         7.4.4 Split the block
         7.4.5 Update memoryBlocks array to include the new split block at the correct
position

8. Print the header for memory allocation table:
   "Block Number    Block Size    File Number    File Size"

9. For each block in memoryBlocks:
   9.1 Print block's details including block number and size, file number and size

10. Check and print for any files that could not be allocated
   10.1 For each file in files:
      10.2 If file is not allocated (size > 0):
         10.3 Print message that the file could not be allocated
```

3.0 Implementation

3.1 Coding

```java
package Lab4;

12 usages
class MemoryBlock {
    6 usages
    int blockNumber;
    7 usages
    int size;
    3 usages
    boolean allocated;
    2 usages
    boolean isNewBlock; // New block flag

    2 usages
    MemoryBlock(int blockNumber, int size) {
        this.blockNumber = blockNumber;
        this.size = size;
        this.allocated = false;
        this.isNewBlock = false; // Default to false
    }

    1 usage
    MemoryBlock split(int allocatedSize) {
        this.size -= allocatedSize;
        MemoryBlock newBlock = new MemoryBlock( blockNumber: this.blockNumber + 1, allocatedSize);
        newBlock.isNewBlock = true; // Set to new block
        return newBlock;
    }
}
```

```
6 usages
class File {
    5 usages
    int fileNumber;
    10 usages
    int size;

    1 usage
    File(int fileNumber, int size) {
        this.fileNumber = fileNumber;
        this.size = size;
    }
}
```

```
package Lab4;

import java.util.Scanner;


public class MemoryAllocationSimulator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Input: Number of memory blocks
        System.out.print("Enter the number of memory blocks: ");
        int numBlocks = scanner.nextInt();
        MemoryBlock[] memoryBlocks = new MemoryBlock[numBlocks];

        // Input: Memory block sizes
        for (int i = 0; i < numBlocks; i++) {
            System.out.print("Enter size of block " + (i + 1) + ": ");
            int blockSize = scanner.nextInt();
            memoryBlocks[i] = new MemoryBlock( blockNumber: i + 1, blockSize);
        }

        // Input: Number of files
        System.out.print("Enter the number of files: ");
        int numFiles = scanner.nextInt();
        File[] files = new File[numFiles];

        // Input: File sizes
        for (int i = 0; i < numFiles; i++) {
            System.out.print("Enter size of file " + (i + 1) + ": ");
            int fileSize = scanner.nextInt();
            files[i] = new File( fileNumber: i + 1, fileSize);
        }
```

```java
// Allocate files to memory blocks (Best-Fit Allocation)
for (File file : files) {
    MemoryBlock bestFitBlock = null;
    int minSizeDifference = Integer.MAX_VALUE;

    for (MemoryBlock block : memoryBlocks) {
        if (!block.allocated && block.size >= file.size) {
            int sizeDifference = block.size - file.size;
            if (sizeDifference < minSizeDifference) {
                minSizeDifference = sizeDifference;
                bestFitBlock = block;
            }
        }
    }

    if (bestFitBlock != null) {
        if (bestFitBlock.size > file.size) {
            MemoryBlock newBlock = bestFitBlock.split(file.size);
            // Expand the memoryBlocks array
            MemoryBlock[] newMemoryBlocks = new MemoryBlock[memoryBlocks.length + 1];
            int newIndex = 0;
            for (int i = 0; i < memoryBlocks.length; i++) {
                if (memoryBlocks[i].blockNumber == bestFitBlock.blockNumber) {
                    newMemoryBlocks[newIndex++] = memoryBlocks[i];
                    newMemoryBlocks[newIndex++] = newBlock;
                } else {
                    newMemoryBlocks[newIndex++] = memoryBlocks[i];
                }
            }
            memoryBlocks = newMemoryBlocks;
        }
        bestFitBlock.allocated = true;
        System.out.println("Allocated File " + file.fileNumber +
                " to Memory Block " + bestFitBlock.blockNumber);
```

```
        } else {
            System.out.println("File " + file.fileNumber + " cannot be allocated.");
        }
    }
    // Display the table
    System.out.println("\nMemory Allocation Table:");
    System.out.println("\tBlock Number\tBlock Size\tFile Number\tFile Size");
    for (MemoryBlock block : memoryBlocks) {
        System.out.print("\t" + block.blockNumber + "\t\t\t" + block.size + "\t\t\t");
        boolean allocated = false;
        for (File file : files) {
            if (file.size > 0 && file.size <= block.size) {
                System.out.println(file.fileNumber + "\t\t\t" + file.size);
                allocated = true;
                file.size = 0; // Mark the file as allocated
                break;
            }
        }
        if (!allocated) {
            System.out.println("-\t\t\t-\t\t\t-");
        }
    }

    // Output files that could not be allocated
    for (File file : files) {
        if (file.size > 0) {
            System.out.println("File " + file.fileNumber + " cannot be allocated.");
        }
    }
    scanner.close();
}
```

3.2 Sample output

```
Enter the number of memory blocks: 2
Enter size of block 1: 20
Enter size of block 2: 30
Enter the number of files: 3
Enter size of file 1: 15
Enter size of file 2: 40
Enter size of file 3: 30
Allocated File 1 to Memory Block 1
File 2 cannot be allocated.
Allocated File 3 to Memory Block 2

Memory Allocation Table:
    Block Number    Block Size  File Number File Size
    1           5           -           -           -
    2           15          1           15
    2           30          3           30
File 2 cannot be allocated.
```

Block 1 needs to be split into 2 parts for fitting the size of file 1. Consequently it would create another usable memory block automatically which size is 5.

The size of file 2 is too big for both of these Memory block so it cannot be allocated.

4.0 Conclusion

In conclusion, the implementation of the Best Fit algorithm for memory allocation provides an efficient method for managing memory resources in computer operating systems. By seeking out the smallest available partition that can accommodate a process, the algorithm minimizes wastage and fragmentation, contributing to optimal memory utilization. However, it is essential to consider the trade-off between efficiency and computational overhead, as the thorough search required by the Best Fit algorithm may result in slightly longer processing times compared to other allocation strategies.

5.0 References

- GeeksforGeeks. (2021, August 18). *Memory Management in Operating System*. GeeksforGeeks. https://www.geeksforgeeks.org/memory-management-in-operating-system/
- Syed, A. A. (2023, April 30). *Contiguous Memory Allocation: First Fit, Best Fit, and Worst Fit. Medium.* https://medium.com/@khanzadaaneeda/contiguous-memory-allocation-first-fit-best-fit-and-worst-fit-734fd6f78ab#:~:text=%F0%9F%A7%90%20Best%20Fit
- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). John Wiley & Sons, Inc. & https://www.youtube.com/watch?v=18CJkT50l0o