Experiment – 9

Partitioning

In operating systems, Memory Management is the function responsible for allocating and managing computer's main memory. Memory Management function keeps track of the status of each memory location, either allocated or free, to ensure effective & efficient use of Primary Memory. There are two Memory management Techniques

- Contiguous
- Non-Contiguous

In contigious Technique, executing process must be loaded entirely in main memory. Contigious Technique can be divided into:

1. Static (or Fixed) Partitioning
2. Dynamic (or Variable) Partitioning

Dynamic Partitioning

It is used to alleviate the problems faced by Static partitioning. Here partitions are not made before the execution. Initially RAM is empty and partitions are made during the run time according to processes need instead of partitioning during system configure. The size of partitioning will be equal to incomming processes. The partition size varies according to the need of processes so as to prevent internal fragmentation and to ensure efficient utilisation of RAM. But this also leads to a problem called External fragmentation.

Static Partitioning

This is the simplest technique used to put more than one process in the main memory. In this partitioning, number of partitions (non-overlapping) in RAM are fixed but size of each partition may or may not be the same. As it is contigious allocation, hence no spanning is allowed. Here partitions are made before execution or during system configure. Processing of Fixed Partitioning require lesser excess & indirect computational power. But this type of partioning faces with both internal fragmentation & external fragmentation.

Algorithm

First Fit

1. Find the first available partition with ample space available for current process
2. Place this process in that partion, if found.
3. Repeat for all process

Best Fit

1. Traverse all the partitions and find a partition which will have minimum space left after current process will be placed in it
2. Place this process in that partition, if found.
3. Repeat for all processes

Worst Fit

1. Traverse all the partitions and find a partition which will have maximum space left after current process will be placed in it
2. Place this process in that partition, if found
3. Repeat for all processes

Numericals

Q1. For the given process and partitions, find out best fit, worst fit and first fit.

Partitions: 200, 400, 600, 500, 300, 250

Processes: 357, 210, 468, 491

Ans

First Fit

	375	210	468		
200	400	600	500	300	250

Best Fit

	357	491	468		210
200	400	600	500	300	250

Worst Fit

		375	210		
200	400	600	500	300	250

Q2 For the given process and partitions, find out first fit, best fit and worst fit

Partitions: 500, 320, 300, 100, 50

Processes: 100, 100, 100, 200

Ans First Fit

100, 100, 100, 200				
500	320	300	100	50

Best Fit

	200	100, 100	100	
500	320	300	100	50

Worst Fit

100	100	100		
500	320	300	100	50

Q3. For the given process and partitions, find out best fit, first fit and worst fit

Partitions: 100, 200, 300, 450, 30, 90

Processes: 250, 95, 21, 310

Ans First Fit

95	21	250	310		
100	200	300	450	30	90

Best Fit

95		250	310	21	
100	200	300	450	30	90

Worst Fit

		95, 21	250		
100	200	300	450	30	90

Code

```
from rich.console import Console
from rich.table import Table
from rich import box

console = Console()

class Block:
    def __init__(self, size, available, usable):
        """
        size -> total size of block
        usable -> remaining usable space in block
        available -> if the block can be used
        """
        self.size = size
        self.available = available
        self.usable = usable

    def __repr__(self) -> str:
        return f"({self.size=}, {self.available=}, {self.usable=})"


class RAM:
    def __init__(self, size, usable, processes):
        self.blocks = [Block(size[x], usable[x], size[x]) for x in range(len(siz
e))]
        self.processes = processes
        self.positioning = [[] for _ in range(len(size))]
        self.fail = []
        self.run()
        self.output()

    def run(self):
        ...

    def output(self):
        table = Table(show_header=False,box=box.SQUARE,show_lines=True)
        table.add_row(
            *[ " | ".join( [ f"P{y[0]+
1}:{y[1]}MB" for y in x] ) if x else " X " for x in self.positioning]
        )
        table.add_row(
            *[str(x.size) for x in self.blocks]
        )
```

```
        console.print(table)
        print("\n")
        # print(self.fail)


class FF(RAM):
    def __init__(self, size, usable, processes):
        super().__init__(size, usable, processes)

    def run(self):
        for ip, p in enumerate(self.processes):
            for ib, b in enumerate(self.blocks):
                if b.available and b.usable >= p:
                    self.positioning[ib].append((ip, p))
                    self.blocks[ib].usable -= p
                    break
            else:
                self.fail.append((ip, p))
        print("\nFirst Fit")


class BF(RAM):
    def __init__(self, size, usable, processes):
        super().__init__(size, usable, processes)

    def run(self):
        for ip, p in enumerate(self.processes):
            minIdx = None
            for ib, b in enumerate(self.blocks):
                if (
                    b.available
                    and b.usable >= p
                    and (
                        minIdx is None
                        or (
                            self.blocks[minIdx].usable - p > b.usable - p
                            and b.usable-p>=0
                        )
                    )
                ):
                    minIdx = ib
            if minIdx:
                self.positioning[minIdx].append((ip, p))
                self.blocks[minIdx].usable -= p
            else:
```

```
                self.fail.append((ip, p))

        print("\nBest Fit")


class WF(RAM):
    def __init__(self, size, usable, processes):
        super().__init__(size, usable, processes)

    def run(self):
        for ip, p in enumerate(self.processes):
            maxidx = None
            for ib, b in enumerate(self.blocks):
                if (
                    b.available
                    and b.usable >= p
                    and (
                        maxidx is None
                        or (
                            self.blocks[maxidx].usable - p < b.usable - p
                            and b.usable-p>=0
                        )
                    )
                ):
                    maxidx = ib
            if maxidx:
                self.positioning[maxidx].append((ip, p))
                self.blocks[maxidx].usable -= p
            else:
                self.fail.append((ip, p))
        print("\nWorst Fit")

size = list(map(int, input("Ram block sizes: ").split()))
usable = list(map(int, input(f"Block usable[{len(size)}] (0/1): ").split()))
processes = list(map(int, input("Processes sizes: ").split()))

print("Deekshant Wadhwa\n01296303118")

FF(size, usable, processes)
BF(size, usable, processes)
WF(size,usable,processes)
```

Output

```
PS D:\Drive\Sem 6\OS\lab> python -u "d:\Drive\Sem 6\OS\lab\partitioning.py"
Ram block sizes: 200 400 600 500 300 250
Block usable[6] (0/1): 1 1 1 1 1 1
Processes sizes: 357 210 468 491
Deekshant Wadhwa
01296303118

First Fit
```

X	P1:357MB	P2:210MB	P3:468MB	X	X
200	400	600	500	300	250

```
Best Fit
```

X	P1:357MB	P4:491MB	P3:468MB	X	P2:210MB
200	400	600	500	300	250

```
Worst Fit
```

X	X	P1:357MB	P2:210MB	X	X
200	400	600	500	300	250