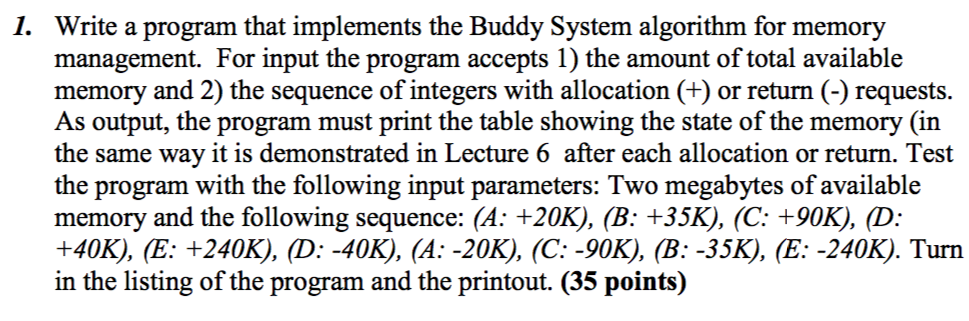
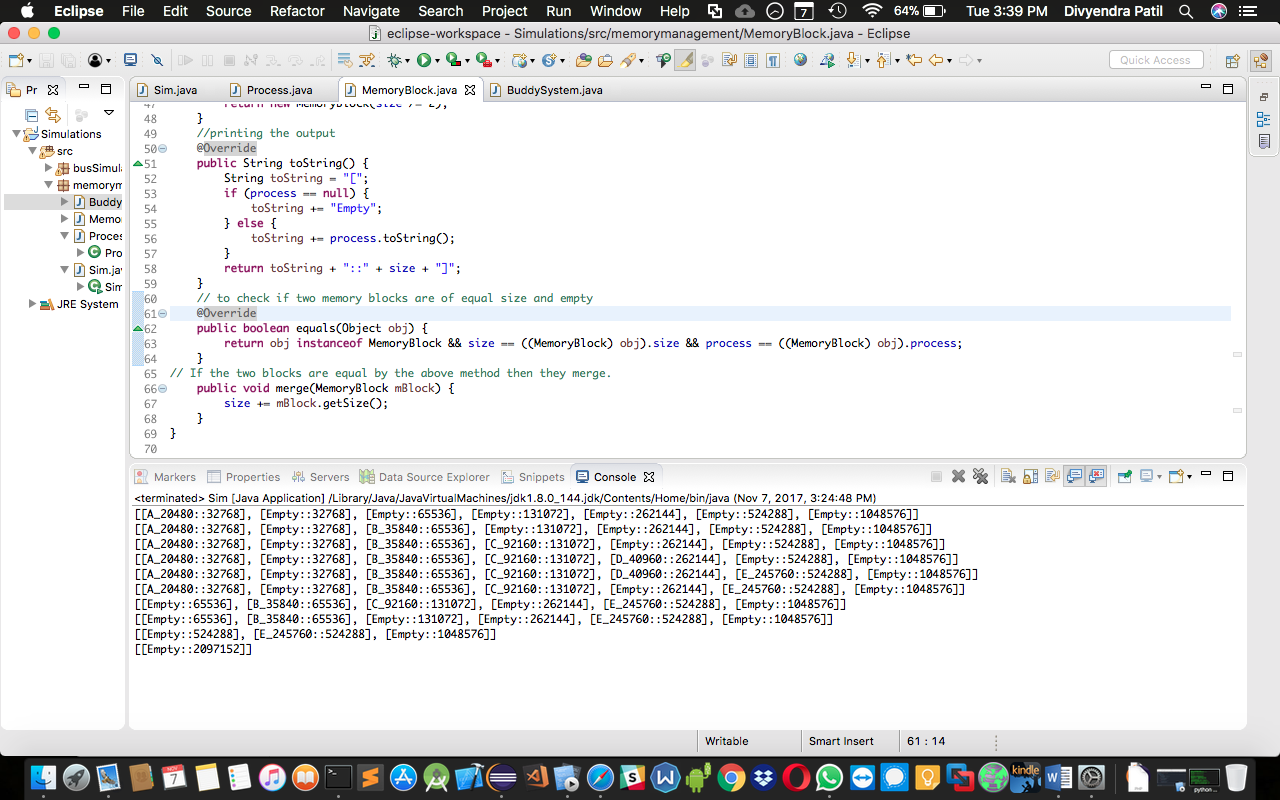
CS 520 Homework 5 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 11/01/2017





Sim.java

package memorymanagement;

public class Sim {

public static long K = 1024;

public static long M = 1024 \* 1024;

public static void main(String[] args) {

BuddySystem bSystem = new BuddySystem(2\*M); //2 Megabytes

//  Process are initialized with respective sizes given in the question

Process processA = new Process("A", 20\*K);

Process processB = new Process("B", 35\*K);

Process processC = new Process("C", 90\*K);

Process processD = new Process("D", 40\*K);

Process processE = new Process("E", 240\*K);

// Order of process being inserted (True for +) in memory and removed out as per Buddy System (False for -)

bSystem.action(processA, true);

bSystem.action(processB, true);

bSystem.action(processC, true);

bSystem.action(processD, true);

bSystem.action(processE, true);

bSystem.action(processD, false);

bSystem.action(processA, false);

bSystem.action(processC, false);

bSystem.action(processB, false);

bSystem.action(processE, false);

}

}

Process.java

package memorymanagement;

public class Process {

private String process\_id; // process ids

private long size; //sizes of process required in memory

private Process() {

size = 0;

}

public Process(String process\_id, long size) {

this();

this.size = size;

this.process\_id = process\_id;

}

public String getProcessId() {

return process\_id;

}

public Long getSize() {

return size;

}

@Override

public String toString() {

return process\_id+"\_"+size;

} }

MemoryBlock.java

package memorymanagement;

public class MemoryBlock {

private long size;

private Process process;

private MemoryBlock() { }

public MemoryBlock(long size) {

this();

this.size = size;

}

// putting a process into the memory block

public boolean load(Process process) {

if (this.process == null) {

this.process = process;

return true;

}

return false;

}

// removing process from memory block

public boolean unload(Process process) {

if (this.process != null && this.process.equals(process)) {

this.process = null;

return true;

}

return false;

}

public Long getSize() {

return size;

}

public Long getUsedSpace() {

return process.getSize();

}

// to know how much space is left in the memory block after the process has occupied the block

public Long getFreeSpace() {

return size - (process == null ? 0 : process.getSize());

}

// checking if the block can be divided into two equal sizes no process gets broken into two block

public boolean canBeSubdivided() {

return getFreeSpace() >= size/2;

}

// dividing the current block into two new blocks

public MemoryBlock getEmptyNewBlock() {

return new MemoryBlock(size /= 2);

}

//printing the output

@Override

public String toString() {

String toString = "[";

if (process == null) {

toString += "Empty";

} else {

toString += process.toString();

}

return toString + "::" + size + "]";

}

// to check if two memory blocks are of equal size and empty

@Override

public boolean equals(Object obj) {

return obj instanceof MemoryBlock && size == ((MemoryBlock) obj).size && process == ((MemoryBlock) obj).process;

}

// If the two blocks are equal by the above method then they merge.

public void merge(MemoryBlock mBlock) {

size += mBlock.getSize();

}

}

BuddySystem.java

package memorymanagement;

import java.util.ArrayList;

public class BuddySystem {

private long memory\_size;

// dividing whole memory into buddy system blocks

private ArrayList<MemoryBlock> memory = new ArrayList<>();

private BuddySystem() {}

// Allocating memory to the entire system

public BuddySystem(long memory\_size) {

this();

this.memory\_size = memory\_size;

}

private void load(Process process) {

// calculating space required in the buddy system

long needSpace = (long) Math.pow(2, Math.ceil(Math.log10(process.getSize())/Math.log10(2)));

//divding the memory in buddy system into blocks

if (memory.size() == 0) {

memory.add(new MemoryBlock(needSpace));

for (long s = needSpace; s < memory\_size; s\*=2) {

memory.add(new MemoryBlock(s));

}

}

// process to be loaded into memory block

MemoryBlock toBeLoadedInto = null;

int index = 0; // to identify the position of the memory block in array list

for (MemoryBlock mBlock : memory) { // to check where the process can fit

index++;

if (mBlock.getFreeSpace() >= needSpace && mBlock.canBeSubdivided()) {

if (mBlock.load(process)) {

break;

}

toBeLoadedInto = mBlock.getEmptyNewBlock();

toBeLoadedInto.load(process);

break;

}

}

if(toBeLoadedInto != null) { // loading process into the block decided by above loop

memory.add(index, toBeLoadedInto);

}

}

// removing the process from the block it has occupied

private void unload(Process process) {

for (MemoryBlock mBlock : memory) {  // to find and remove the process from the block it resides in

if (mBlock.unload(process)){

break;

}

}

// Merging two equal empty blocks

while(memory.size() > 1) {

if (memory.get(0).equals(memory.get(1))) {

memory.get(1).merge(memory.get(0));

memory.remove(0);

} else {

break;

}

}

}

// this method is used to interact witht he buddy system

public void action(Process process, boolean toAddInMemory) {

if (toAddInMemory) {

load(process);

} else {

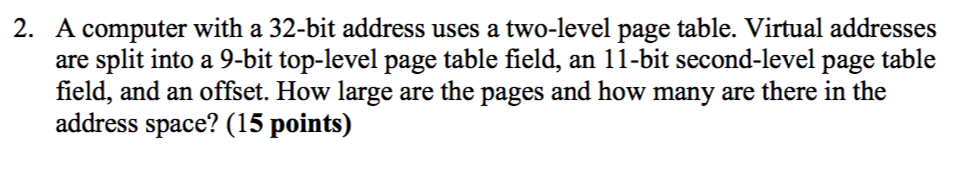
unload(process);

}

System.out.println(memory);

}

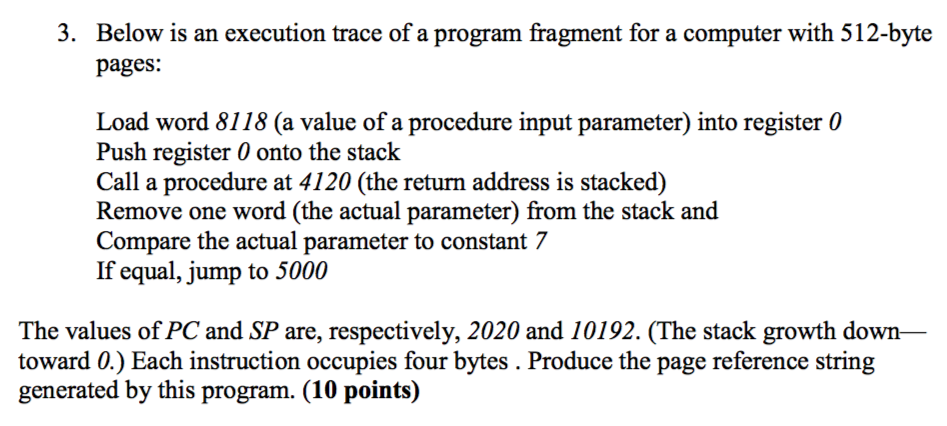
}



Solution:

1] There are 20 (9 + 11) bits which are used for virtual page numbers in the memory.   
2] This means there are 12 (32-20) bits for offset.   
3] Therefore, the page size = 2^12 = 4KB.

4] Since 20 bits are used for virtual pages insinuates that there are 2^20 pages.



Solution:

PC = 2020

SP = 10192

Page Size = 512 B

Starting page no.= (Address by PC / Page Size) = (2020/512) = 3

Reference string is generated by Page no, data reference number

2020: Load word 8118 (a value of a procedure input parameter) into register 0 3,15 (2020 / 512)

2024: Push register 0 onto the stack 3,19 (10192/512)

2028: Call procedure at 4120 (the return address is stacked) 3,8,19 [subroutine string], …

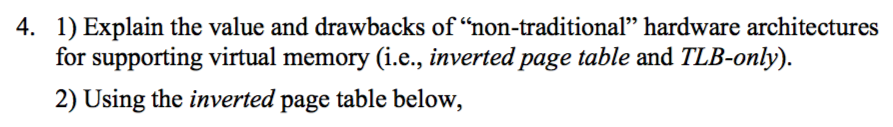
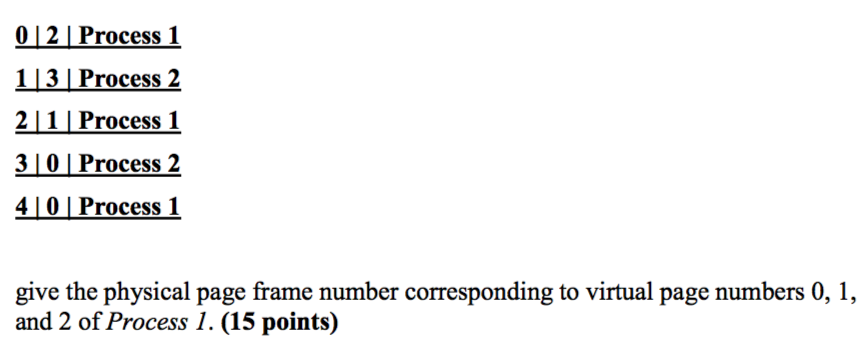
2032: Remove one word (the actual parameter) from the stack and 3,19

2036: Compare the actual parameter to constant 7 3

2040: If equal, jump to 5000

3,9

So, the reference string is 3, 15, 3, 19, 3, 8, 19, …, 3, 19, 3, 3, 9

Solution:

**1]**

**Importance**:

a] Virtual memory makes application based programming effortless by hiding fragmentation of physical memory.

b] If virtual memory can be matched (in TLB), then it can get the physical address directly, otherwise OS checks the inverted page table to find the physical address.

c] Since the logical address will always be greater than physical address, the virtual memory can plot or map more number address of disks.

**Disadvantage:**

a] The use of Inverted table and TLB is limited by the design because the design of virtual memory is very complicated.

b] The physical page cannot have more than one virtual address shared (which becomes a major drawback) because the IPT (Inverted Page Table) is not suitable for shared memory as only one virtual page entry is allowed for every physical page, therefore

2]

**Inverted Page Table (IPT) Format:**

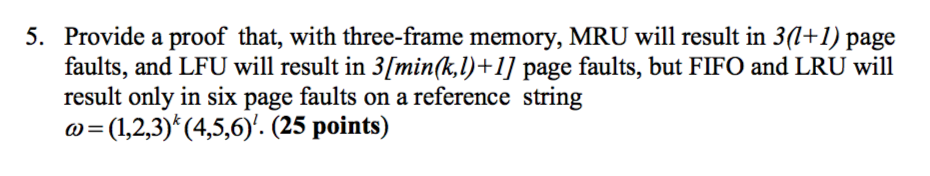
|  |  |  |
| --- | --- | --- |
| Table Index (Physical Page number) | Virtual Page Number | Process Id |

If we apply the format to given above table, we get physical page frame numbers of **0, 1 and 2** to be **4, 2 and 0** respectively.

“Virtual page number 0” Corresponding physical page frame number is (4, offset-value)

“Virtual page number 1” Corresponding physical page frame number is (2, offset-value)

“Virtual page number 2” Corresponding physical page frame number is (0, offset-value)



1] **In MRU algo**, pages (1, 2, 3) will create trap of 3 page faults as they will be accessed first time. The rest of (1, 2, 3) k-l pages, page fault will not occur since they are in the memory and recently being used. But (4, 5, 6) will be accessed first time, again 3 page faults will be created as they will be replacing recently accessed pages in the memory.   
This demonstrates that MRU will result in total 3(l+1) page faults.

2] **In LFU algo,** pages (1, 2, 3) will result **in 3 page faults**. If k < L, then after accessing them for “k” times pages (4, 5, 6) will replace them causing page faults.   
Hence 3(k+1) page faults will occur.   
But if k > L, even if pages (4, 5, 6) have been accessed L times they will still be considered as least used than pages (1, 2, 3).   
Therefore, total page faults will be 3(l+1).   
This demonstrates that LFU will result in 3[min(k,l)+1] page faults.

3] **In FIFO algo,** pages (1, 2, 3) will result in 3 page faults since they are being loaded in memory for the first time.   
For rest of (1, 2, 3) k-1 pages, page fault will not occur since they are already there in the memory.   
When (4, 5, 6) will be accessed first time, 3 page faults will be created as they will be replacing first inserted pages in the memory.   
For rest of (4, 5, 6) l-1 pages, page fault will not occur since they are already present in the memory.   
This indicates that FIFO will result in 6 page faults.

4] **In LRU algo,** the logic of FIFO can be extended. Just that in this case OS will create page faults based on time rather than insertion index. And there will be 6 page faults (again).