**Students go to a store to play video games. If there's no free game machine, student**

**will wait until the supervisor assigns a machine for him/her, otherwise s/he will**

**just take one of the free machines. When any machine becomes free, the supervisor will**

**let the first student on line use the machine. After a student finishes, s/he releases**

**the computer and waits until another one is done. When the group (of two) is formed,**

**they leave.**

**Initially all the game machines are available. The number of machines is numMachines=5.**

**Using semaphores and operations on semaphores, synchronize the 2 types of threads**

**(student and supervisor). Roughly, before synchronization, a possible execution in**

**pseudo-code might be:**

**Student: Supervisor:**

**arrive at store //napping while(true)**

**if(numMachines > 0){**

**P(Mutex)**

**numMachines --;**

**V(Mutex)**

**P(Mutex)**

**P(NumMachines)**

**play //if machine is available { P(Mutex)**

**if(numMachines <1){**

**P(NumMachines)**

**assign game machine**

**}V(Mutex)**

**V(NumMachines)**

**}**

**Else{**

**P(NumMachines)**

**}**

**P(Mutex2)**

**group++**

**If(group %2 ! = 0){P(Group)**

**Else{**

**Form group;**

**Leave**

**}**

**V(Mutex2)**

**}**

**Binary Semaphore Mutex = 1;**

**Binary Semaphore Mutex2 = 1;**

**Binary Semaphore Group = 0;**

**Counting Semaphore NumMachines = 5;**

**Int group = 0;**

**After each step, give the value of the updated semaphore and the content of the updated**

**semaphore queue.**

**COUNTING SEMAPHORES: S1, S3, S6**

**BINARY SEMAPHORES: S2, S4, S5**

**Semaphore queues use Priority scheduling algorithms where low PID means low priority.**

**Semaphore initial values:**

**S1=0, S2=1, S3=3, S4=0, S5=1, S6=0**

**1) P1, P(S3)** S3 = 2

**2) P2, P(S5)** S5 = 0

**3) P3, P(S1)** S1 = -1 S1 Q (P3)

**4) P5, P(S4)** S4 = 0 S4 Q (P5)

**5) P2, V(S5)** S5 = 1

**6) P1, P(S6)** S6 = -1 S6 Q (P1)

**7) P2, P(S4)** S4 = 0 S4 Q (P5,P2)

**8) P6, V(S3)** S3 = 3

**9) P4, V(S4)** S4 = 0 S4 Q(P2) P5 has higher priority

**10) P4, P(S6)** S6 = -2 S6 Q (P1, P4)

**11) P1, V(S5)** S5 = 1

**If it is possible, implement, with a minimum number of semaphores, a complete**

**serialization (for all variables) between READ and WRITE, such that the READ operation**

**is always done after the WRITE operation. If a complete serialization isn't possible**

**for all the variables, give the solution for a partial serialization and specify for**

**which variables the serialization is impossible. For a partial serialization, consider**

**as more important:**

**1) higher concurrency (vs minimum number of semaphores)**

**2) minimum number of semaphores (vs higher concurrency)**

**Specify the number, type, and initial values of the necessary semaphores.**

**Binary Semaphore X = 0;**

**ThreadA ThreadB**

{ {

**write z; read y;**

P(X)

**read x; write x;**

V(X)

**write y; read z;**

**} }**

**(B) Reader-Writer problem (correct code)**

**reader() writer()**

**{ {**

**while(true) while(true)**

**{ {**

**P(mutex); P(OKaccessDB);**

**readerCount++; accessDB;**

**if(readerCount==1) V(OKaccessDB);**

**P(OKaccessDB); }**

**V(mutex); }**

**accessDB;**

**P(mutex);**

**readerCount--;**

**if(readerCount==0)**

**V(OKaccessDB);**

**V(mutex);**

**}**

**}**

**What would be the outcome of replacing (in reader)**

**FROM:**

**if(readerCount==0)**

**V(OKaccessDB);**

**V(mutex);**

**TO:**

**if(readerCount==0)**

**{**

**V(OKaccessDB);**

**V(mutex);**

**}**

DeadLock, Or Mutual Exclusion is violated.

**Memory Management**

**Given the memory partitins of 600K, 300K, 700K, 200K, in order, how would each of the First – Fit, Best – Fit algorithms place processes of 230K, 525K, 175K, and 300K ( in order)?**

First Fit – 230K to 600K, 525K to 700K, 175K to 300K, 300K process will not be loaded.

Best Fit – 230K to 300K, 525K to 600K, 175K to 200K, and 300K to 700K.

**Consider a paging system with the page table stored in memory and registers. If 70 percent of all page table references are found in the TLB, what is the effective access time? Assume that finding a page table entry in the TLB takes 25 NS and a memory access takes 175 NS. Give the formula and solve.**

**Define Internal Fragmentation. Give examples of memory allocation schemes (at least two) that suffer from internal fragmentation.**

Internal Fragmentation is when a process is loaded into main memory and part of the allocated space for that process is not being used. This is a waste of space.

Fixed Sized Partitioning and Non-Equal Size Partitioning.

**Describe the associative mapping. Give its advantages and disadvantages.**

Page tables are in CPU registers. The advantages of this is that it takes practically no time to search through the registers, so there is no cost to time. The disadvantage is that it is expensice. It will need over 200 CPU registers.

**Consider the two dimensional array A:**

**Int A[][] = new int [9][9];**

**Where A[0][0] is at location 89, in a paged system with pages of size 27. A small process is in page 1 for manipulating the matrix; every instruction fetch will be from page 1.**

**For three page frames, how many page faults are generated by the following array initialization loop? All frames were initially empty. A Least Recently Used algorithm is used :**

**For (int j = 0, j < 9; j++)**

**For (int i = 0; i < 9; i++)**

**A[i][j] = 0;**

**In the solution of this problem you should:**

1. **Show the contents of the logical addess space, how the array and instructons are stored**
2. **Give the reference string generated by the CPU during initialization**
3. **On the reference string, show the page faults**
4. **For each reference explain what element is initialized or if the instrution is fectched.**
5. **Explain the final computation of the page fault number.**

|  |
| --- |
| 1. 0-26 |
| 1. 27-53 Instruction 0 |
| 1. 54 – 80 |
| 1. 81 – 107   Location starting from 89: A[0][0] – A[0][8] A[1][0]-A[1][8] |
| 1. 107 – 134   A[2][0] – A[2][8] A[3][0]-A[3][8] A[4][0] – A[4][8] |
| 1. 135 – 161   A[5][0] – A[5][8] A[6][0]-A[6][8] A[7][0] – A[7][8] |
| 1. 162 – 188   A[8][0] – A[8][8] A[9][0]-A[9][8] |

Frames :

|  |
| --- |
| Page 1 |
| Page 3 (changes) |
| Page 4 (changes) |

1 Column of Initialization:

Instr 0, A[0][0] PF, Instr 0, A[1][0] No PF, Instr 0, A[2][0] PF, Instr 0, A[3][0] No PF, Instr 0 , A[4][0] No PF, Instr 0, A[5][0] PF, Instr 0, A[6][0] No PF, Instr 0, A[7][0] No PF, Instr 0, A[8][0] PF, Instr 0, A[9][0] No PF................

4 Page Faults \* 9 Columns = 36 Page Faults.

**Is this the most efficient initialization? Briefly explain your answer.**

No since the array is initializing by column first it will go through many more page faults. A more efficient way is to initialize by row. Since the Array is stored in numerical order, the OS will only have to service a page fault once for every page the Array is stored in. This equates to 4 page faults. For the coloumn to be initialized first, the OS must page fault 36 times to initialize the array.

**What is Thrashing?**

When resources have become too little or exhausted. A process will ask the Operating System for resources which in turn the OS will take resources from another process and give it to the new process. Then the previous process will also ask for resources and the OS will take resources from another process or the new process. This will continue to happen which leaves the computer gain little or no progress. All of the time is spend allocating resources over and over.

**Consider the logical addess spaces and page tables for P1 and P2. The main memory contains pages of P1 and P2. The main memory has 12 frames. There are free frames.**

1. **Draw a picture that will illustrate the content of the Main Memory. ( what variable, from P1 and P2, each frame might contain).**

|  |
| --- |
|  |
| 1. Process 2 (B) |
|  |
| 1. Process 1 (S) |
| 1. Process 1 (T) |
|  |
|  |
| 1. Process 1 (L) |
| 1. Process 2 (U) |
|  |
| 1. Process 1 (K) |
|  |
|  |

1. **Describe the step that will be followed by the OS and MMU in the case that the next instruction to be executed by the CPU for P2 is: load M. A reference variable M is a legal reference.**

If there is no free frame, the OS will save the victim frame to secondary memory.

The page table will be updated to show that the victim frame’s V/I will change from 1to 0 and Frame is in DA.

The OS will then swap in the demanded page and the page table will be updated to show the demanded page is in the frame and V/I to 1.

Then execution will be restarted.

Since there are free frames, M will be loaded into the Main Memory Frame (any free frame will do), the Page Table will be updated, M V/I will change from 0 to 1 and the Frame will change from DA to which ever frame has been chosen by OS. And execution will start.

1. **For the logical or physical address registers of size 16 are used. The offset field takes 6 bits, page or frame number fields take 10 bits. Offset for variable L is 18.**

**What will the logical address generated by the CPU for load L?**

**Page Number |Page Offset 1|5**

**(Frame\*Page Size) + (R%S) = 7 \* 10 + 18 % 10 = 72**

**What will be the physical address generated by the CPU for load L?**

**Frame Number |Frame Offset 7|18**

**This is the binary representation for page/frame number and the displacement.**

**Logical Addess Space for P1 Logical Address Space for P2**

**0 O 0 B**

**1 L 1 M**

**2 S 2 U**

**3 N 3 C**

**4 K**

**5 T**

**Page Table P1 Page Table P2**

**P F V/I P F V/I**

**0 DA 0 0 1 1**

**1 7 1 1 DA 0**

**2 3 1 2 8 1**

**3 DA 0 3 DA 0**

**4 10 1**

**5 4 1**