CODE:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Operating Systems
Ph.D. Qualifying Exam

Spring 2022

1. **PAGING**

Write a method(s) to implement Last In First Out (LIFO) page replacement algorithm using your favorite programming language with the assumption that the LIFO can hold up to 100 entries. You may use any language provided data structures. The add method assigns a frame/process/page element to the LIFO. When the LIFO is full, the algorithm should update the LIFO appropriately and execute an **“evict(process, page, frame)”** for the page being evicted. You can assume frames are initially assigned incrementally with a counter. When you evict an entry, you can reuse the frame number from the entry that you are removing. **Evict( )** is an OS primitive and you do not need to implement this function, you only need to implement **add( )**. Each entry on your LIFO can be a record (or object) of the form:

struct LIFO\_element {int process, page, frame; struct LIFO\_element \* next);

add(int process, int page); // takes parameters and adds to LIFO; evicts eligible candidate if full

Is the LIFO page replacement algorithm and effective page replacement algorithm? Support your response.

**Any implementation that effectively does a LIFO. Stack is the obvious data structure. LIFO is a terrible page replacement algorithm because it evicts the page that we needed just now and probably will need again.**

1. **Memory Management**

An operating system is running on a machine with 4GB of physical memory. The virtual memory is implemented as a pure paging scheme. Assume that each page table entry takes one byte

What is the size of a single level page table if the the virtual address space is 64-bits, page size is 8K?

 What will be the size of an inverted page table for this system?

**Page Table size 2^64/2^13 = 2^51 entries
Inverted Page Table: 2^34/2^13 = 2^21 Inverted table entries**

1. **Mutual Exclusion**

Many processes in an operating system use a shared counting variable (often called a one-up number). Assume there is a single common, shared variable called COUNT. Write a method **Get\_count( )**  returns the current value of COUNT and increments COUNT atomically. You are to use OS provided libraries and tools that allow you to guarantee that COUNT is always updated correctly and avoids race conditions.

 **Semaphore p = 1;**

 **Int count = 0;**

 **Int update\_count(){**

 **Int local**

 **Down(p);**

 **Local = count;**

 **Count = count + 1;**

 **Up(p);**

 **Return count;
}**

1. **OS Performance**

Consider a demand-paging system in which processes are performing sequential data accesses with the following time-measured utilizations:

 CPU utilization 20%
 Paging disk 98%
 Other I/O devices 10%

 For each of the following, indicate yes or no and provide a short supporting sentence on whether the proposed change is likely to ***improve CPU utilization***:

 a. Install a faster CPU

 b. Install a bigger paging disk.

 c. Increase the degree of multiprogramming.

d. Decrease the degree of multiprogramming.

e. Install more main memory.

f. Install a faster hard disk.

g. Install multiple controllers with multiple hard disks and stripe the data across the disks.

h. Add prepaging to the page-fetch algorithms.

 i. Increase the page size.

 j. Increase the I/O bus speed.

**a. Install a faster CPU NO—faster CPU reduces CPU utilization
b. Install a bigger paging disk. NO – more pages wont increase CPU, CPU still waits on pages
c. Increase the degree of multiprogramming. YES – maybe, more processes usually use more CPU
d. Decrease the degree of multiprogramming. YES -- if two processes are causing page waits
e. Install more main memory. YES – less paging would occur
f. Install a faster hard disk. YES – page requests would be handled faster making process wait reduced
g. Install multiple controllers with multiple hard disks and stripe the data across the disks. YES – maybe if the page system can be spread out
h. Add prepaging to the page-fetch algorithms. YES – assuming page fetch is effective
i. Increase the page size. NO --
j. Increase the I/O bus speed. YES – if our IO wait is on network, for example.**

1. **Deadlock Avoidance**

What is Deadlock Avoidance? Give an example algorithm that implements deadlock avoidance. What are the requirements for deadlock avoidance?

What is Deadlock Prevention? How is Deadlock prevention implemented? How effective is deadlock prevention?

**Deadlock Avoidance is a technique where the process indicated the max number of resources it needs (per category). Each process incrementally requests resources as needed. Deadlock avoidance uses a Banker’s Algorithm for committing resources. The algorithm will stall a process request if that request would place the system in an un-safe state. An unsafe state means that future requests could cause a circular wait. Deadlock Avoidance reduces throughput as the OS will still processes until others have relinquished allocated resources.**

**Deadlock Prevention is when we break one of the four conditions of mutual exclusion. The only reasonable method is to break circular wait. To break circular wait a process must request all of its needed resources at the start of running, making throughput slow down. Deadlock prevention has to break**

**a) Hold and Wait – process has to give back resources. This either causes resource invalidaton (printer for example) or starvation of the process**

**b) Mutual Exclusion – process all get to use resource at the same time. Printer example would then have all print intermixed**

**c) No Premption – OS could take resource away from process… Printer example still applies: process has partial print and then another process gets pringt
d) No Circular wait – run one process at a time , or allocate all resources at once, of make every process request resources in a hierarchical manner. This does not cause resource contention, but it does cause resource under-utitlization.**

1. **Concurrency**

main() {

 int a = 0;

 int rc = fork();

 a++;

 if (rc == 0) {

 rc = fork();

 a++;

 } else {

 a++;

 }

 printf(“Hello!\n”);

 printf(“a is %d\n”, a);

 } // or main

**Hello!**

**a is 2**

**Hello!**

**a is 2**

**Hello!**

**a is 2**

**How many times does “Hello” print? 3

What is the largest value of a printed? 2**

1. **Scheduling**

Suppose there are four processes (P1 - P4) with respective arrival times of 0, 10, 20, and 40, priorities 1, 2, 3, and 4(highest) , and job times of 30, 20, 50, and 20 ms. These processes are scheduled from a single run queue to run on a **dual-processor machine**. Assume the context switch has no overhead.

What is the average turnaround time for scheduling the four processes using the following preemptive schedulers: priority ( highest-priority-first), round-robin, or shortest time remaining first? For round-robin, assume a 20 ms time quantum.

**SJF**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **Start** | **Stop** | **Time remaining** | **TT** |
| **Processor 1** | **P1** | **0** | **30** |  | **30** |
|  | **P3** | **30** | **80** |  | **60** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| **Processor 2** | **P2** | **10** | **40** |  | **30** |
|  | **P4** | **40** | **60** |  | **20** |
|  |  |  |  |  |  |
|  |  |  |  |  | **140/3** |

 **Priority Queue**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **START** | **STOP** | **TT** |
| **Processor1** | **P1** | **0** | **20** | **40** |
|  | **P3** | **20** | **70** | **50** |
|  |  |  |  |  |
|  |  |  |  |  |
| **Processor 2** | **P2** | **10** | **30** | **20** |
|  | **P1** | **30** | **40** |  |
|  | **P4** | **40** | **60** | **20** |
|  |  |  |  | **AVG 32.5** |

**Round Robin**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **START** | **STOP** | **TT** |
| **Processor1** | **P1** | **0** | **20** |  |
|  **Proessor 2** | **P2** | **10** | **30** | **20** |
| **Processor1** | **P3** | **20** | **40** |  |
| **Processor 2** | **P1** | **30** | **40** | **40** |
| **Processor 1** | **P4** | **40** | **60** | **20** |
| **Processor 2** | **P3** | **40** | **60** |  |
| **Processor 4** | **P3** | **60** | **70** | **50** |
|  |  |  |  |  |
|  |  |  | **AVG TT** | **32.5** |
|  |  |  |  |  |