CODE:_____

Operating Systems
Ph.D. Qualifying Exam
Fall 2021

1. PAGING

Consider the following piece of code which multiplies two matrices:

```
int a[1024][1024], b[1024][1024], c[1024][1024];

multiply()
{
  unsigned i, j, k;
  for(i = 0; i < 1024; i++)
     for(j = 0; j < 1024; j++)
        for(k = 0; k < 1024; k++)
```

```
            c[i][j] += a[i,k] * b[k,j];
    }
```

Assume that the binary for executing this function fits in one page, and the stack also fits in one page. Assume further that an integer requires 4 bytes for storage. Compute the number of TLB misses if the page size is 4096 bytes and the TLB has 8 entries (4 bytes per entries) with a replacement policy consisting of LRU.

Solution:

1024*(2+1024*1024) = 1073743872 (+2 if text and stack are included)
The binary and the stack each fit in one page each, thus each takes one entry in the TLB. While the function is running, it is accessing the binary page and the stack page all the time. Therefore the two TLB entries for these two pages would reside in the TLB all the time and the data can only take the remaining 6 TLB entries.

We assume the two entries are already in TLB when the function begins to run. Then we need only consider those data pages.

Since an integer requires 4 bytes for storage and the page size is 4096 bytes, each matrix requires 1024 pages. Suppose each row of a matrix is stored in one page. Then these pages can be represented as a[0..1023], b[0..1023], c[0..1023]: Page a[0] contains the elements a[0][0..1023], page a[1] contains the elements a[1][0..1023], etc.

For a fixed value of i, say 0, the function loops over j and k, we have the following reference string:

a[0], b[0], c[0], a[0], b[1], c[0], a[0], b[2], c[0]
…
a[0], b[1021], c[0], a[0], b[1], c[01022], a[0], b[1023], c[0]

1. **For the reference string (1024 rows in total), a[0], c[0] will contribute two TLB misses. Since a[0] and c[0] each will be accessed every four memory references, the two pages will not be replaced by the LRU algorithm. For each page in b[0..1023], it will incur one TLB miss every time it is accessed. Therefore, the number of TLB misses for the second inner loop is 2+1024*1024 = 1048578**

So the total number of TLB misses is 1024*1048578 = 1073743872 (+2 if we count text and stack misses).

2. Memory Management

An operating system is running on a machine with 16M of physical memory. The virtual memory is implemented as a pure paging scheme. What is the size of a single level page table if the the virtual address space is 32-bits and size of a page is 4K. What will be the size of an inverted page table for this system? Assume that each page table entry takes one byte.

Page Table size 2^32/^12 = 2^20 entries
Inverted Page Table: 2^24/2^12 = 2^12 Inverted table entries

3. Mutual Exclusion

Consider the following implementation of a counting semaphore. Answer the following questions:

a) Define a counting semaphore
b) Being specific, when could a counting semaphore be used?
c) Does the following code correctly implement a counting semaphore? If so, then argue why it does. If not, provide a specific reason why it does not

```
void Wait (Semaphore S) {
   while (S.count <= 0) {}
```

```
    S.count = S.count - 1;
}
void Signal (Semaphore S) {
    S.count = S.count + 1;
}
```

A) A counting semaphore is an object provided by the OS to allow programmers to create mutual exclusion to critical sections. The counting semaphore deals with multiple copies of a resource

B) Using multiple printers

C) No it does not. The S.count+1 and S.count-1 can occur asynchronously

4. File Systems

Consider a file system similar to the Second Extended File System (ext2) in which the inodes contain 16 double-indirect addresses, each block is 8KiB, and each block address takes up 4 bytes. What is the largest file size possible using this system (assuming that the inode contains a sufficiently large field to record the file size itself)? What would the largest file size be if using 16KiB blocks?

Each block can hold 8096/4 = 2048 block addresses. Each double-indirect address in the inode then links to $2048^2 = 2^{22}$ (about 4 million) data block addresses. So the largest possible file has $16 * 2^{22}$ blocks, or $16 * 2^{22} * 8192$ bytes = 512GiB.

5. Deadlock Avoidance

Consider a system with three processes and three resource types. The following table gives, for each process, how many of each resource is currently allocated to it and what its maximum total

need is for each resource type. Suppose that the number of available (unallocated) resources for each type is **(1 1 *x*).** What is the lowest value of *x* for which this is a safe state? Explain your answer.

	Current			Maximum		
Process	**1**	**2**	**3**	**1**	**2**	**3**
A	1	1	1	2	1	2
B	1	1	0	3	3	3
C	1	2	1	4	3	3

***x* = 0 is not safe because no maximum request could be satisfied. *x* = 1 is not safe because, although A's maximum request could be satisfied, after that neither B's nor C's could be satisfied. *x* = 2 *is* safe: A's maximum request can be satisfied, then B's, and finally C's.**

6. Concurrency

What is the exact output of the following? Explain.

```
for (int i = 0; i < 2; i++ )
  {
     int pid=fork();

      if (pid ==0) printf("*");

      else { printf ("**");
               wait(NULL);;
           }
  }
```

Nine * .. *******

7. Scheduling

For the below Processes table, calculate ***the average waiting time*** for the algorithms. For arrival purposes, all of the processes arrive at the same time. Lowers process number will be chosen first in FCFS.

First Come First Serve (FCFS)
Shortest Job First (SJF) and
Priority Scheduling

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

First Come First Serve

Process	Start	Stop	Total	Wait(total-burst)
P1	**0**	**10**	**10**	**0**
P2	**0**	**11**	**11**	**10**
P3	**0**	**13**	**13**	**11**
P4	**0**	**14**	**14**	**13**
P5	**0**	**19**	**19**	**14**
FCFS Average				**9.6**

Shortest Job First

Process	Start	Stop	Total	Wait(total-burst)
P1	**0**	**19**	**19**	**9**
P2	**0**	**1**	**1**	**0**
P3	**0**	**2**	**4**	**2**
P4	**0**	**2**	**2**	**1**

P5	**0**	**4**	**9**	**4**
Shortest Job First Average				**3.2**

Priority

Process	Start	Stop	Total	Wait(total-burst)
P1	**0**	**16**	**16**	**6**
P2	**0**	**1**	**1**	**0**
P3	**0**	**18**	**18**	**16**
P4	**0**	**19**	**19**	**18**
P5	**0**	**0**	**6**	**1**
Priority				**8.2**

8. Priority Queue

Operating Systems often implement a priority ready queue. You are to implement Insert() into a basic ready queue based only on the priority value. If the new process has the same priority as the elements in the list, the new Process is put in after all that are the same value (and smaller) .

```
struct PQueue{  int PID; int PRIORITY; struct PQueue
*next;};

struct Pqueue *RQ=NULL;   // global Priority queue

void Insert(Struct PQueue *Process); // insert Process into the
global RQ properly.
                                        // Process should be
inserted in RQ using Process.PRIORITY before all process that
are strictly Larger.
```

Solution:

```
void Insert(struct PQueue *Process) {
      {
        struct PQueue *P, *C;
         P= NULL;
         C= RQ;
         while ( (C!=NULL && (C->PRIORITY <= Process->
PRIORITY))
              {  P = C;
                 C = C->next;
              }
           Process->next = C;
           if (P==NULL) RQ=C
           else  P->next = Process;
       }
```