**Operating Systems**

**Quiz - Memory**

1. Consider the following program:

double A[256];

double B[256];

float D[256][256];

float F[256][256];

int row, col;

for ( row = 0; row < 256; row++ )

{

**// Loop 1**

for (col = 0; col < 256; col++ )

A[row] = D[row][col] \* F[row][col];

**// Loop 2**

for (col = 0; col < 256; col++ )

B[row] = D[row][col] \* F[col][row];

}

Assume this program runs on a system where a **float** takes **4** bytes, and a **double** takes **8** bytes, and the page size is **1024** (1KB) bytes. Assume each array is stored starting at the beginning of a virtual page. Arrays are stored by row; that is, all elements in one row are stored contiguously, first all elements in row 0, then in row 1, etc. Assume that the variables row and col are stored in registers throughout the execution of the program. Assume the size of the program code (instructions) is just **832** bytes, and it is stored starting at the beginning of a virtual page.

* How many pages are required to store?

**The program code – 1 page**

**Array A – 2 pages**

**Array B – 2 pages**

**Array D – 256 pages**

**Array F – 256 pages**

* During the execution of loop 1, what is the size of the program’s working set in pages? Explain.

**1 page of code, 1 page of A, 1 page of D, 1 page of F = 4 pages**

* During the execution of loop 2, what is the size of the program’s working set in pages? Explain.

**1 page of code, 1 page of B, 1 page of D, 256 pages of F= 259 pages**

* Suppose this program is allocated just 4 physical pages frames. What will be the approximate page fault frequency (expressed in faults per loop iteration) if an LRU page replacement algorithm is used. Justify your answer.

For loop 1? **approximately 0 per iteration**

**4 pages are loaded and no more faults in following iterations**

For loop 2? **approximately 1 per iteration**

**4 pages are loaded, then 1 fault per iteration for the rows of F**

* Suppose the execution of 1 loop iteration takes a total of 1 microsecond if all required items are in physical memory: this includes the instruction fetches, the loads, the store and the multiplication. Suppose each page fault requires 10 milliseconds to find the page on disk and load it into physical memory. Consider again that the LRU algorithm is being used. How long will it take to execute each loop? Justify your answer.

**Do the math based on the answer above**

**Loop 1: 40 ms + 256 usec**

**Loop 2: 40 ms + 2550 ms + 256 usec**

1. Assume you have a virtual memory that consists of 16 pages (pages 0 – 15), and a physical memory that consists of 4 pages (pages 0 – 3).

* Suppose your program generates the following stream of virtual memory page accesses. For each of the following page replacement algorithms (FIFO and LRU), show the state of the physical page frames at each step in the tables provided, and give the total number of page faults at the bottom. Assume the physical page frames begin empty.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Optimal** | | | | **FIFO** | | | | **LRU** | | | |
| VM page accessed | **PF0** | **PF1** | **PF2** | **PF3** | **PF0** | **PF1** | **PF2** | **PF3** | **PF0** | **PF1** | **PF2** | **PF3** |
| 0 | **0** |  |  |  | **0** |  |  |  | **0** |  |  |  |
| 4 |  | **4** |  |  |  | **4** |  |  |  | **4** |  |  |
| 5 |  |  | **5** |  |  |  | **5** |  |  |  | **5** |  |
| 6 |  |  |  | **6** |  |  |  | **6** |  |  |  | **6** |
| 7 |  |  |  | **7** | **7** |  |  |  | **7** |  |  |  |
| 2 | **2** |  |  |  |  | **2** |  |  |  | **2** |  |  |
| 5 |  |  |  |  |  |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  | **4** |  |  |  |  | **4** |
| 0 | **0** |  |  |  |  |  |  | **0** |  | **0** |  |  |
| 5 |  |  |  |  | **5** |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  | **7** |  |  |  |  |  |  |

7 faults **10 faults 8 faults**

* Fill in what the page table will look at the end of this set of accesses, assuming the LRU algorithm was used and assuming each page table entry contains just the following 2 items.

|  |  |  |
| --- | --- | --- |
| Virtual Page # | Valid bit | Page frame # |
| 7 | **1** | **0** |
| 6 | **0** |  |
| 5 | **1** | **2** |
| 4 | **1** | **3** |
| 3 | **0** |  |
| 2 | **0** |  |
| 1 | **0** |  |
| 0 | **1** | **1** |

1. Consider a swapping system with the given hole list, in memory order, and the successive set of memory requests: 14K, 20K, 4K. Show the block selected and also **rewrite the new hole list between each request considering using the following algorithms:**

First fit:

Initial hole list: 12K, 20K, 10K, 24K, 7K, 16K, 4K, 22K ( take 14K)

12K, 6K, 10K, 24K, 7K, 16K, 4K, 22K (take 20K)

12K, 6K, 10K, 4K, 7K, 16K, 4K, 22K (take 4K)

8K, 6K, 10K, 4K, 7K, 16K, 4K, 22K

Best fit:

Initial hole list: 12K, 20K, 10K, 24K, 7K, 16K, 4K, 22K (take 14K)

12K, 20K, 10K, 24K, 7K, 2K, 4K, 22K (take 20K)

12K, 10K, 24K, 7K, 2K, 4K, 22K (take 4K)

12K, 10K, 24K, 7K, 2K, 22K

1. Assume we have written the following correct code to test your encryption drivers (all error handling has been removed to make the code shorter):

int main( int argc, char \*argv[] ) {

int device, infile, n, offset;

char buf[BUFSZ];

memset( buf, 0, BUFSZ );

infile = open( argv[1], O\_RDONLY );

device = open( "/dev/testdevice", O\_WRONLY );

printf( "Please enter an 8 character password: " );

fgets( buf, BUFSZ, stdin );

offset = 8;

while ( (n = read( infile, buf+offset, BUFSZ-offset )) != 0 ) {

write( device, buf, n+offset );

offset = 0;

}

close( infile );

close( device );

}

Two assumptions are made about the password — that the user will enter an 8-char password, and that the device expects an 8-char password. Explain what will happen if either of those assumptions is incorrect.

***The first part can easily be seen in the above code. The buffer is cleared to 0s, so if the user enters a password less than 8 chars, the password will be padded with 0s up to 8 chars. If the user enters more than 8 chars, then the password will be truncated to just 8 chars, because the file data is written into the buffer starting after 8 positions.***

***If the device is expecting less than 8 chars, it will treat the extra chars up to 8 as part of the file data to be encrypted. If the device is expecting more than 8 chars as a password, it will take some of the initial chars of the file and treat them as the password, and those will be effectively lost from the file.***

A sneaky assumption is made that BUFSZ is less than or equal to the internal buffer size of the device. Explain what will happen if this assumption is incorrect. Explain how to, or correct the code, to fix this issue.

***The system call* write*, which is implemented by the driver for whatever device some code is writing to, is passed the file, buffer, and number of bytes to be written as parameters, and then it returns the number of bytes successfully written. This number may be less than or equal to the number of bytes attempted to be written. If this caller's buffer is larger than the device's buffer, then the device may freely return a number less than desired, and it is the duty of the program calling write to take care of writing the amount that was missed. This can be managed by checking the return value of* write*, and calling* write *successively until all the desired data is written.***

***You should have seen this behavior exhibited by the Unix program* cat *when you tested your driver. Even if you programmed your driver to remain in the same* device\_write *function until the passed in number of bytes were written, there exists some maximum size buffer that is feasible to allocate and pass to a system call. For example, when I tested with a 600K input file,* cat *selected a 128K length buffer, so severals calls to* write *were needed and made by* cat *to pass the whole file through the device. Therefore you had to write code to handle that possibility in your driver anyway, so writing code to empty each* write *buffer in just one shot was actually an unnecessary extra, since it is the duty of the caller to handle that possibility (otherwise why return the number of bytes written? On error you return an error condition, so less than the total number of bytes is not an error.***