Names:

Rachel Peters - 816003509

Shivana Sookdeo - 816003030

Teneil Rodriguez - 816003952

Akil Hosang - 816003000

David Orr – 816003099

Course Code & Name: COMP 2604 – Operating Systems

Assignment 1

Year 2

Semester 2

**Question 1**

//this program uses the argument,n, entered by the user as the number of elements to be added to the array

//the program then prompts the user for a series of n numbers which is added to the array

//the array is sorted in ascending order using insertion sort

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <pthread.h>

int \*my\_data; //global (pointer to the array of numbers)

int hi; //global(holds size of the array)

struct argStruct{

    int size; //size of the array

    int \*data;//pointer to the arrray

};

void \*insertionSort(void \*arg){

struct argStruct \*my\_args;

int lo=0; //first value in the array

my\_args = (struct argStruct \*)arg; //unmarshalling

hi = my\_args-> size;

my\_data = my\_args->data; //assign pointer to the array.This variable is global so that it can be easily printed by the parent thread

free(my\_args);

int j;

for(j=lo+1;j<hi;j++){

    int key=my\_data[j];

    int k=j-1;

    while(k>=lo && key < my\_data[k]){

        my\_data[k+1]=my\_data[k];

        k=k-1;

     }

        my\_data[k+1]=key;

}

    pthread\_exit(0);

}

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

    int i;

    int \*data; //pointer to array of numbers

    int count; //size of array

int n;

    struct argStruct \* args;

pthread\_t thread\_id;

if(argc==2){ //only 2 agruments allowed .i.e ./a.out 10

count=atoi(argv[1]); //convert argv[1] to integer

if(count > 0){ //if positive number

data = (int \*)malloc(count\*sizeof(int));

for(i=0; i < count; i++) {

printf("%s","Please enter a number :");

scanf("%d",&n);

data[i] =n; //populating array

}

args = (struct argStruct \*)malloc(sizeof(struct argStruct));

args->size = count;

args->data = data;

pthread\_create(&thread\_id, NULL, insertionSort, (void \*)args); // thread created

pthread\_join(thread\_id,NULL); //wait for thread to exit

for(i=0;i<hi;i++){

     printf("%d",my\_data[i]); //print the contents of array

printf("%s"," ");

     }

}

else{

printf("%s","Argument must be a postive integer");

}

}

else{

printf("%s","Invalid number of arguments entered");

}

return 0;

}

**Question 2**

a) 2 page faults occur with this address stream

FIFO

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 0 |  | 0 |  | 0 |  | 3 |  | 3 |  | 3 |  | 3 |  | 3 |  | 3 |  |
|  |  | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  | 0 |  | 0 |  |
|  |  |  |  | 7 |  | 7 |  | 7 |  | 7 |  | 7 |  | 7 |  | 7 |  | 7 |  |
|  |  |  |  |  |  | 2 |  | 2 |  | 2 |  | 2 |  | 2 |  | 2 |  | 2 |  |
|  |  |  |  |  |  |  |  | F |  |  |  |  |  |  |  | F |  |  |  |

LRU

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 0 |  | 0 |  | 0 |  | 3 |  | 3 |  | 3 |  | 3 |  | 0 |  | 0 |  |
|  |  | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  |
|  |  |  |  | 7 |  | 7 |  | 7 |  | 7 |  | 7 |  | 7 |  | 7 |  | 7 |  |
|  |  |  |  |  |  | 2 |  | 2 |  | 2 |  | 2 |  | 2 |  | 2 |  | 3 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | F |  |  |  |

b)

000001 01110111110

Page # Offset

000001 = 000110 (Frame Number from process page table)

000110 01110111110

Frame # Offset

c) Internal fragmentation is the concept where space in main memory is wasted due to the block of data that is loaded being smaller than the partition (occurs when using fixed partitioning). The memory is divided into several static partitions at system generation time.

External fragmentation is the occurrence where in main memory, the process of loading in processes starts out well (using dynamic partitioning), but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines.

**Question 3**

i) First fit(FF)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3MB | 1M |  | 6MB | 3MB |  | 1MB | 0M | 9M |  | 7M |  | 12M |  | 5M |

ii) Best fit(BF)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3MB | 1M |  | 9M |  | 1MB | 0M | 9M |  | 6MB | 1MB |  | 12M |  | 5M |

iii) Worst Fit(WF)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 4M |  | 6MB | 3MB |  | 1M | 1MB | 8MB |  | 7M |  | 3 MB | 9M |  | 5M |

iv) Next fit(NF)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3MB | 1M |  | 6MB | 3MB |  | 1MB | 0M | 9M |  | 7M |  | 12M |  | 5M |

b) i) There would be 20 entries in the page table as 4096 is 2^12 and a virtual address is 32 bit which means that it is 2^32 bits. 2^32/2^12 = 2^20 thus showing that there is 20 entries in the page table.

ii) The memory management unit converts a virtual address into an address in real memory by utilizing the page number (high-order bits) and offset (low-order bits) from the virtual address. The page number is taken and checked against a process page table or pushing it into some algorithm and receiving the page frame number. The page frame number is attached to the high order end of offset and will give you the real/physical address.

iii) NFU (Not Frequently Used) algorithm requires a software counter associated with each page, initially zero. At each clock interrupt, the operating system scans all the pages in memory. For each page, the R bit, which is 0 or 1, is added to the counter. The counters roughly keep track of how often each page has been referenced. When a page fault occurs, the page with the lowest counter is chosen for replacement. The main problem with NFU is that it never forgets anything. For example, in a multipass compiler, pages that were heavily used during pass 1 may still have a high count well into later passes. In fact, if pass 1 happens to have the longest execution time of all the passes, the pages containing the code for subsequent passes

c) Inverted page tables are used in 64 bit computers. This is because address spaces are 264 bytes and pages are 4 KB each, which means that the page table will need 252 entries. If each entry is 8 bytes, then the table will be 30 million gigabytes will is an unreasonably large size for a computer to handle.

In this design, there is one entry per page frame in real memory, rather than one entry per page of virtual address space. For example, with 64-bit virtual addresses, a 4-KB page, and 1 GB of RAM, an inverted page table only requires 262,144 entries. The entry keeps track of which (process, virtual page) is located in the page frame.

**Question 4**

a) The operations that can be performed on a semaphore are:

down – if the semaphore’s value is more than 0, then decrement the semaphore and continue. If the value becomes 0, put the process to sleep.

Up – this increments the value of the semaphore. If one or more processes were sleeping on that semaphore that couldn’t complete an earlier down operation, one is randomly chosen by the system to complete its down

b)

User-level approach: In user mode, the kernel is not aware of the existence of threads so it operates as it usually does. So it would pick a process, say X, and gives X control for its quantum. The thread scheduler in X decides which thread to run, say X1. Since there are no clock interrupts, this thread may continue running as long as it wants. If it uses up the processes’ entire quantum the kernel will select another process to run. Because of the nature of being user mode, it will not affect any other processes.

Eg. If there was another process, say B, the sequence A1, A2, A3, A1, A2, A3 is possible but not A1, B1, A2, B2, A3, B3

Kernel-level approach: The kernel picks a particular thread to run. It doesn’t have to take into account which thread belongs to which process. The thread is given a quantum and is forcibly suspended if it exceeds the quantum.

c)

To enter a critical region:

registerLock = lock

lock = 1

while registerLock is 1:

registerLock = lock

lock = 1

return

To leave a critical region:

lock = 0

return

d)

The main defect of both Peterson’s and the TSL solutions stems from them both requiring busy waiting. This approach wastes CPU time but can also have some undesired effects such as the priority inversion problem. This is where there are say 2 processes, a high priority process H and a low priority process L. The scheduling rules are that H is run whenever it is in ready state. At a certain moment, with L in its critical region, H becomes ready to run. H begins busy waiting but since L is never scheduled while H is running, L never gets the change to leave its critical region so H loops forever.