**Assignment – 1**

**Question 1 (20%): Comparing the Cost of Output with Three I/O Models:**

**The textbook discussed three I/O modes: Programmed I/O, Interrupt-driven I/O, and DMA. Answer the following questions:**

1. **What are the main differences between the three I/O models and what are their strength and weakness?**

### Programmed I/O

Programmed I/O is the most simple technique for exchanging data or communication b/w the CPU and any other external device. Data is sent and received by the processor and the I/O module. The processor gives complete control to the I/O operation i.e. sensing device status, sending READ/WRITE command. Once the processor issues a command to the I/O module, it then waits for the operation to execute and complete. However, the processor confirms the status of the operation till the I/O module is executed completely.

**Strengths:**

* Programmed I/O is simple to implement.
* Requires very little hardware support.

**Weakness:**

* Longer wait time.
* Ties up the CPU for longer time with no

### Interrupt Driven:

Interrupt I/O is a way of controlling I/O jobs where the external device/peripheral/ command line sends a signal about the due work. Hence, it causes an interrupt to be set. So, in interrupts have a priority level in the processor. Therefore, it requires more complex system and software, however, is a lot more efficient than programmed I/O in terms of efficiently using computer’s time and processing capacity.

**Strength:**

* It is fast because the I/O devices only interrupt when in ready state.
* It is efficient because it lets the CPU perform other tasks while it is waiting for an interrupt.

**Weakness:**

* CPU spends most of the time in a loop, waiting fir the device to become ready, called *busy waiting*.
* Data transfer b/w a fast and slow memory unit is essentially, limited by the speed of CPU.
* Requires an interrupt for every character read/write. Interrupting a running process is can be expensive operation.

### Direct Memory Access (DMA)

DMA is a technique of transferring data within main memory and external device w/o passing it through the CPU. DMA improves the processing capacity of the operating system because it takes automates the task of transferring data from the processor and let it perform other tasks.

**Strength:**

* Allows the peripheral devices to READ/WRITE without going through the CPU.
* Frees the processor from transferring data, hence, allows faster processing since, the processor can be doing something else.

**Weakness:**

* When peripheral device wants to READ/WRITE it makes the CPU wait. This is called the *cycle stealing*.

### Key Difference:

|  |  |  |
| --- | --- | --- |
| **Programmed I/O** | **Interrupt-driven I/O** | **Direct Memory Access** |
| Each data item transfer is initiated by an instruction in the program. Requires constant monitoring of the CPU and sometimes unnecessary. | This approach avoids keeping the processor busy unnecessarily, however, the processor only transfers data when interrupted. Otherwise, continues performing its own tasks. | Direct Memory Access (DMA) allows the peripherals directly communicate with one another by using the memory bus. Hence, completely removes the intervention of the CPU. |

1. **Assuming that an application needs to output 1000 words from the internal memory to the hard disk, calculate the following values for each I/O model:**
   * **How many times the processor is interrupted?**
   * **How many times the internal memory is read by the processor for those 1000 words?**
   * **How many times the disk controller is read by the processor?**
   * **How many times the disk controller is written to by the processor?**

**Draw a table to contain your answers.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Programmed I/O** | **Interrupt-Driven I/O** | **Direct Memory Access** |
| **Processor is interrupted** | 0 | >1000 | 1 |
| **Memory is read** | >1000 | >1000 | 0 |
| **Disk control is read** | >1000 | 1000 | 1 |
| **Disk control is written** | 1000 | 1000 | 1000 |

**Justification:**

Under programmed I/O a processor will never be interrupted. However, it’s difficult to be specific, the number of times the memory is read or the disk control is read.

1. For programmed I/O the memory will be read at least 1000 times, at least one time for each word. The register reads the data bit by bit. Hence, the disk control is read and written 1000 times.
2. An Interrupt-Driven I/O will not allow for any real certainty, other than each operation will be attempted at least 1000 times, once for each word. This is because an Interrupt-Driven I/O would never be implemented on a single-user/single-task OS, it wouldn’t make sense as the need for Interrupt-Driven I/O only arises on multi-user and/or multi-tasking OS’s.
   1. The processor will be interrupted at the end of each transferred word, but may also be interrupted by other processes or users.
   2. The memory will be read at least 1000 times but like Programmed I/O, we don’t know how many times the OS will need to read the memory for its own use.
   3. If we assume that no other user or process requires use of the I/O device during the 1000 word transfer, then the CPU will query the disk controller then wait for the interrupt then as long as no errors occur will write to it once for each word.
3. Direct memory access is the least CPU intensive. If it is assumed that no other process on the OS requires the system bus and each transfer is completed within the time slice allocated to that process. The CPU will assign disk controller with the starting address in the memory and the length of the data in words.

**2. (10 marks) Based on the information generated from the above program, produce a memory map table showing the layout of literals, initialised global variables, uninitialized global variables, formal parameters of each function, local variables, dynamically allocated variables, functions, environment and command line arguments in the memory when the program runs at label L in function f2.**

**The memory map table must show the addresses of each variable, literal, and function and their sizes. It should also show the start**

**and end addresses of the environment and the command line arguments and their sizes.**

**The memory map table must contain at least the following columns:**

**1. The start address of an entity such as a variable or a function**

**2. The length of the storage space of the entity in bytes**

**3. The name of the entity, such as global\_pointer1 or Hello, world!**

**4. The nature of the entity, such as *function*, or *uninitialized global variable***

**5. The memory section, e.g. environment, command line arguments, code (or text), global initialised data, global uninitialed data, stack, heap etc.**

**In addition, you must use seven different background colours to highlight the following seven memory sections as indicated below::**

**[Green] initialised global variables (including constants and literals)**

**[Red] uninitialised global variables**

**[Blue] stack (containing the local variables and returning addresses of function calls)**

**[Magenta] heap (containing the dynamically allocated memories)**

**[Yellow] code (functions)**

**[Cyan] process environment**

**[White] command line arguments**

**Please also note that in your memory table, *the memory addresses must be strictly sequential, from the highest address to the lowest address* to reflect how different components of a running program are layout in the virtual memory. Your memory tablewill not be accepted if the addresses are not lined up sequentially in the table. If you find that components from one section aresplit in more than one continuous area of memory, it is a sure indication that there is something wrong with your memory map andyou should find out what went wrong and fix it.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **entity name such as global\_pointer1 or Hello, world!** | **address** | **Length of Storage (byte)** | **Nature of entity (Memory type) *such as function*, or *uninitialized global variable*** | **Memory section e.g. environment, command line arguments, code (or text), global initialised data, global uninitialed data, stack, heap etc** |
| Argc | 0x7fffc97cdcdc |  |  |  |
| Argv[0] | 0x7fffc97ce007 |  |  |  |
| Argv[1] | 0x7fffc97ce010 |  |  |  |
| Argv[2] | 0x7fffc97ce018 |  |  |  |
| Argv[3] | 0x7fffc97ce020 |  |  |  |
| first address command line argument | 0x7fffc97cddd8 |  |  |  |
| end address of command line argument | 0x7fffc97cddd8 |  |  |  |
| first address of environment | 0x7fffc97cde00 |  |  |  |
| last address of environment | 0x7fffc97cde38 |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**(5 marks) Based on the experiment and analysis you have carried out in 1 and 2 above, answer the following questions:**

1. **what is the size of the virtual address space of that process?**
2. **how does the operating system on your machine layouts the following process components in the virtual address space: command line arguments, environment, literals, initialised global variables, uninitialised global variables, functions, formal parameters and local variables of a function, and dynamically allocated variables (or memories)?**

My personal computer use segmentation instead of paging, dividing virtual address spaces into variable-length segments. Hence, a address consists of a segment number and an offset within the segment.

### Question 3 (25%): Executing Commands in Child Processes

**Write a program that takes a list of command line arguments, each of which is the full path of a command (such as /bin/ls, /bin/ps, /bin/date, /bin/who, /bin/uname etc). Assume the number of such commands is N, your program would then create N direct child processes (ie, the parent of these child processes is the same original process), each of which executing one of the N commands. You should make sure that these N commands are executed concurrently, not sequentially one after the other. The parent process should be waiting for each child process to terminate. When a child process terminates, the parent process should print one line on the standard output stating that the relevant command has completed successfully or not successfully (such as "Command /bin/who has completed successfully", or "Command /bin/who has not completed successfully"). Once all of its child processes have terminated, the parent process should print "All done, bye-bye!" before it itself terminates.**

### Source code listing

### The Linux executable, source code for the question along with the test files and output screenshots are attached.

The following code checks if the number of command line arguments are more than one because the first argument for executing a C program is the typical ./a.out or the executable name. The program creates an array of child processes which is equal to the number of command line arguments passed. After this, the program checks for the occurrence for the ‘/’ in the command and pass the entered command to the execl function which replaces an existing C function with another function. The program has another loop for managing the parent which waits for the child process to execute. It uses the macros WIFEXITED() which returns 1 if the child process executes normally, otherwise 0.

### The source code for the program is listed:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <string.h>

int main(int argc, char \*argv[]){

/\*\*

 \* the program needs at least one argument to execute

 \*/

    if(argc > 1){

        pid\_t pid[argc]; // a collection of number of arguments passsed to make the child concurrent

        char \*commandName; // stores the command name

        int status;        // gives the status

        for(int i = 1 ; i < argc; i++){

            pid[i-1] = fork();

            if(pid[i-1] < 0){

                perror("fork unsuccessful ");

                exit(1);

            } else {

                if(pid[i-1] == 0){

                    commandName = strchr(argv[i], '/'); // find the occurrence of '/'

                    // execl (path, commandName, NULL)

                    execl(argv[i], commandName, (char\*)0);

                    perror("execl failed...\n");

                    exit(1);

                }

            }

            printf("%s \n", argv[i]);

        }

        // for loop for controlling the parent process for concurrency

        for(int i=1; i<argc; i++){

            wait((int\*)0);

            // testing

    printf("Command %s has finished executing \n", argv[i]);

            sleep(2);

            if(WIFEXITED(status)){

                const int EXIT\_STATUS = WEXITSTATUS(status);

                // testing

                printf("exit status for the child was %d \n", EXIT\_STATUS);

            }

        }

    }

    else {

       printf("No path entered");

    }

    printf("All done bye-bye \n");

    return 0;

}

**Question 4 (25%): Reporting Information of Files**

**Write a C program, myls.c, that is similar to the standard Unix utility ls -l (but with much less functionality). Specifically, it takes a list of command line arguments, treating each command line argument as a file name. It then reports the following information for each file:**

1. **user name of the owner owner (*hints: Stevens & Rago, 6.2.*);**
2. **group name of the group owner; (*hints: Stevens & Rago, 6.4.*);**
3. **the type of file;**
4. **full access permissions, reported in the format used by the ls program;**
5. **the size of the file;**
6. **I-node number;**
7. **the device number of the device in which the file is stored, including both major number and minor number (*hints: Stevens & Rago, 4.23.*);**
8. **the number of links;**
9. **last access time, converted to the format used by the ls program (*hint: Stevens & Rago, 6.10.*);**
10. **last modification time, converted to the format used by the ls program;**
11. **last time file status changed, converted to the format used by the ls program;**