CPSC457_(p2017) _Assignment 2 Muzhou,Zhai_T02_10106810

Q1 - Written question (4 marks)

Define DMA and multiprogramming. Would the concept of multiprogramming be practical on a computer that does not support DMA? Why or why not?

-Direct memory access (DMA) is a methodology that allows an I/O device to send or receive data directly from or to the main memory, bypassing the CPU to speed up memory operations. This is a method of transferring data from the computer's RAM to another part of the computer without processing it using the CPU. What is more, the device controller transfers an entire block of data directly to the main memory without CPU intervention by using the method of DMA.

-Multiprogramming is a form of parallel processing in which several programs are run at the same time on a uniprocessor. In the multiprogramming system, when one program is waiting for I/O transfer, there is another program ready to utilize the CPU. So that makes it possible for many works share the time of CPU. What is more, multiprogramming is the rapid switching of the CPU between multiple processes in memory. It is done only when the currently running process requests I/O or terminates.

-Yes. The multiprogramming can be practical on a computer that does not support DMA by concepts. For example, the early computers have every byte of data read or written was handled by the CPU, and there was no DMA. However, the CPU will be fully occupied and there will be much lower efficient by this way.
-Explanation: Assuming that the major delay is the wait while data handling when a CPU is working, The prime reason for multiprogramming is to let CPU has some work to do while waiting for I/O to complete and. If there is no DMA, the multiprogramming can work by concept, but the CPU will be fully occupied doing I/O, so multiprogramming gains nothing. No matter how much I/O a program does, the CPU will be 100% busy. In another word, the multiprogramming will occupy the CPU ad DMA can solve this problem by allows multiple programs to run at the same time.

Q2 - Written question (3 marks)

Describe how a wrapped system call, e.g. read() in libc, invokes the actual system call in the kernel. Is it essential that a wrapper of a system call is named the same as the underlying system call?

-The system call is the fundamental interface between an application and the Linux kernel. A library provides a set of functions that are are available to an application programmer, including the parameters and the return values (APIs). Usually this API has a system call interface under the kernel mode, that contains a system call table. The system call table indexed by unique numbers associated with

each system calls. The library wrapper is only copying arguments and unique system call number to the registers where the kernel expects them, then trapping to kernel mode and setting the errno if the system call returns an error number. For example, The slides from the lecture gave an example of the printf() command. standard C library provides a portion of the system-call interface. for printf(), the C library intercepts the call and invokes the necessary system call write. the C library takes the value returned by write and pass it back to the user program.

To be more detailed, usually the library wrapped system call just do little work other than copying arguments to the right registers before invoking the system call, and then setting errno (errno is a header file in the standard library of the C programming language) after the system call has returned. These are the same steps that are performed by system call, which can be used to invoke system calls for which no wrapped system call is provided. System calls indicate a failure by returning a negative error number to the caller. For example, after a positive number has returned the wrapped system call negates the returned error number (to make it positive), copies it to errno, and returns -1 to the caller of the wrapper.

-No, it is not essential for a wrapper of a system call's name same as the underlying system call. System calls are generally not invoked directly, but rather via wrapper functions in libraries.Often, but not always, the name of the wrapper function is the same as the name of the system call that it invokes. So that there are some wrapper of system calls' name are different from the underlying system calls.

Q3 - Written question (4 marks)
Is it possible for a process to go from the blocking state to the running state? What about going from the ready state to the blocking state? If yes, describe an example, if not, describe why not.

-No, a processor cannot go from the blocking state to the running state. The running state is the process that is currently being executed and the blocking state is a process that cannot execute until some event occurs, such as the completion of an I/O operation. Processes entering the system must go initially into the ready state, and processes can only enter the running state via the ready state. So that it is obvious that the processor cannot go from the blocking state to the running state.

-No, a processor cannot go from the ready state to the blocking state. The ready state is a process that is queuing and prepared to execute when given the opportunity. Processes entering the system must go initially into the ready state. the blocking state is a process that cannot execute until some event occurs. If the processor want goes to the blocking state, it has to go to the running state from the ready state first. After the running state, the process is being executed then the processor can enter the blocking state that waitting for some events to occur.

Without the running state, the processor cannot directly go from the ready state to the blocking state.

Q4 - Written question (3 marks)
What is a context switch? What are the actions taken, including the information to be saved, during a context switch?

-A context switch is the switching of the CPU from one process or thread to another. It is an essential feature of multitasking operating system.

-The kernel performing the following activities with regard to processes (including threads) on the CPU when context switching:

-Retrieving the context of the next process from memory and restoring it in the CPU's registers.

-Suspending the progression of one process and storing the CPU's state (eg. text) for that process somewhere in memory.

-Returning to the location indicated by the program counter in order to resume the process. For example, returning to the line of code at which the process was interrupted.

-Saving the state of the old process in PCB as well as loads the saved state for the next process from PCB.

Q7 – Written question (3 marks)
Run "strace -c" and "time" on your bash script from Q5 and your C/C++ program from Q6 and compare the results. Explain why the results different? Include the output of the above commands in your report.
The output using the command time:
Firstly the c++ file, followed by the sh file.

```
[mzhai@zone29-wc Downloads]$ time ./scan png 10
./I1.png  1970
./I3.png  1959
./L1.png  1928
./T1.png  1921
./O0.png  1867
./O2.png  1866
./J3.png  1860
./Z3.png  1852
./Z1.png  1818
./T0.png  1810
Toal Size: 18851

real    0m0.025s
user    0m0.002s
sys     0m0.004s
```

```
[mzhai@zone29-wc Downloads]$ time ./scan.sh png 10
./I1.png        1970
./I3.png        1959
./L1.png        1928
./T1.png        1921
./O0.png        1867
./O2.png        1866
./J3.png        1860
./Z3.png        1852
./Z1.png        1818
./T0.png        1810
total bytes: 18851

real    0m0.015s
user    0m0.008s
sys     0m0.007s
```

The output using the command strace-c:
Firstly the c++ file, followed by the sh file.

```
[mzhai@zone29-wc Downloads]$ g++ scan.cpp -o scan
[mzhai@zone29-wc Downloads]$ strace -c ./scan png 10
./I1.png  1970
./I3.png  1959
./L1.png  1928
./T1.png  1921
./O0.png  1867
./O2.png  1866
./J3.png  1860
./Z3.png  1852
./Z1.png  1818
./T0.png  1810
Toal Size: 18851
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 93.03    0.000974           4       259         3 stat
  5.25    0.000055           5        11           write
  1.43    0.000015           2         8           read
  0.29    0.000003           0         7           fstat
  0.00    0.000000           0        28        23 open
  0.00    0.000000           0         6           close
  0.00    0.000000           0        16           mmap
  0.00    0.000000           0        10           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           clone
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           fcntl
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         1           pipe2
------ ----------- ----------- --------- --------- ----------------
100.00    0.001047                   355        27 total
```

```
[mzhai@zone29-wc Downloads]$ strace -c ./scan.sh png 10
./I1.png        1970
./I3.png        1959
./L1.png        1928
./T1.png        1921
./O0.png        1867
./O2.png        1866
./J3.png        1860
./Z3.png        1852
./Z1.png        1818
./T0.png        1810
total bytes: 18851
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 16.55    0.000114           4        26        18 open
 16.55    0.000114          29         4           clone
 14.22    0.000098          20         5         1 wait4
  9.14    0.000063           3        20         6 close
  8.42    0.000058           4        15           mmap
  8.27    0.000057           3        21           rt_sigprocmask
  5.52    0.000038           5         8           mprotect
  3.34    0.000023           4         6         3 stat
  2.61    0.000018           2        10           rt_sigaction
  2.47    0.000017           2         7           fstat
  2.32    0.000016           3         6           read
  1.60    0.000011           4         3           pipe
  1.31    0.000009           2         5           brk
  1.02    0.000007           7         1           munmap
  0.87    0.000006           2         3           lseek
  0.87    0.000006           2         3         1 fcntl
  0.58    0.000004           2         2           getrlimit
  0.58    0.000004           4         1           sysinfo
  0.44    0.000003           3         1           rt_sigreturn
  0.44    0.000003           3         1         1 ioctl
  0.29    0.000002           2         1           dup2
  0.29    0.000002           2         1           getpid
  0.29    0.000002           2         1           uname
  0.29    0.000002           2         1           getuid
  0.29    0.000002           2         1           getgid
  0.29    0.000002           2         1           geteuid
  0.29    0.000002           2         1           getegid
  0.29    0.000002           2         1           getppid
  0.29    0.000002           2         1           getpgrp
  0.29    0.000002           2         1           arch_prctl
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.000689                   160        31 total
```

Conclusion and explanation:

By comparing the two files' results after using the time command and the strace -c command, we can find that: Even though the results of the cpp and sh file are totally the same, but the total running time of the sh file is faster than the cpp file. What is

more, the number of calls from the cpp file is 2 times as much as the sh file. The reason is that: the shell script using all the shell commands that are very simple. However in the cpp files I created many functions written by c++ to accomplish the functionalities, such as filtering filenames based on the extension, sorting, and calculating the sum. Since I have only use the system call once, so that many of my instructions in my c++ file (eg. instructions using libraries) have to communicate with the kernel via API. However a shell script is a program written using the shells specific programming language to automate the use of the shell. At their most basic level they are like batch files which are just a list of instructions. That makes the sh file runs faster than the c++ file. And the sh file make fewer calls in the system.