Homework (Measurement)

In this homework, you’ll measure the costs of a system call and context switch. Measuring the cost of a system call is relatively easy. For example, you could repeatedly call a simple system call (e.g., performing a 0-byte read), and time how long it takes; dividing the time by the number of iterations gives you an estimate of the cost of a system call. One thing you’ll have to take into account is the precision and accuracy of your timer. A typical timer that you can use is gettimeofday(); read the man page for details. What you’ll see there is that gettimeofday() returns the time in microseconds since 1970; however, this does not mean that the timer is precise to the microsecond. Measure back-to-back calls to gettimeofday() to learn something about how precise the timer really is; this will tell you how many iterations of your null system-call test you’ll have to run in order to get a good measurement result. If gettimeofday() is not precise enough for you, you might look into using the rdtsc instruction available on x86 machines. Measuring the cost of a context switch is a little trickier. The lmbench benchmark does so by running two processes on a single CPU, and setting up two UNIX pipes between them; a pipe is just one of many ways processes in a UNIX system can communicate with one another. The first process then issues a write to the first pipe, and waits for a read on the second; upon seeing the first process waiting for something to read from the second pipe, the OS puts the first process in the blocked state, and switches to the other process, which reads from the first pipe and then writes to the second. When the second process tries to read from the first pipe again, it blocks, and thus the back-and-forth cycle of communication continues. By measuring the cost of communicating like this repeatedly, lmbench can make a good estimate of the cost of a context switch. You can try to re-create something similar here, using pipes, or perhaps some other communication mechanism such as UNIX sockets. One difficulty in measuring context-switch cost arises in systems with more than one CPU; what you need to do on such a system is ensure that your context-switching processes are located on the same processor. Fortunately, most operating systems have calls to bind a process to a particular processor; on Linux, for example, the sched setaffinity() call is what you’re looking for. By ensuring both processes are on the same processor, you are making sure to measure the cost of the OS stopping one process and restoring another on the same CPU.

Answer :

依照題意，我們需要分別測量system call 和 context switch的時間成本，

首先我們先使用題目推薦的方法，在line12 、 line14和line19宣告了該函式需要的區域變數，在line12中的timeval分別擁有tv\_sec(秒)以及tv\_usec(微秒) ，line14為開始記錄時間，line19為結束的時間，在line15~18中我們讓他做寫入line11中宣告的fd，並且執行nloops(1000000)次的寫入動作，但沒有寫入任何東西，執行完後我們將這段動作的starttime 以及endtime記錄下來，在line21~22中，將他們做出來的時間以及次數相除，即獲得平均一次幾微秒的結果，接下來做context switch，首先依照題意，他需要確保contesxt switch進程在同一個CPU上，在line25~27使用了cpu\_set\_t的結構體來操作進程的可調度CPU集合，在CPU\_SET中，第一個參數為pid，假設電腦為單核CPU4核心，那他的參數可以設定在CPU0 ~ 3之間，line27我設定為0表示設定CPU0，接下來依照題目的提示，sched\_setaffinity()為我們需要使用的，能確保兩個進程都在同一個CPU上，sched\_setaffinity()的第一個參數為pid，將後面參數寫入line25~27CPU\_SET的&set，接下來line45是在判定sched\_setaffinity()回傳的參數，如果傳入0的話就是會設定當前的執行緒，-1的話即錯誤，line46即離開，CPU設置好後，依照題意我們使用題目舉例的pipe來實作，題目說明需要2個pipe，所以in line29宣告了first\_pipe和second\_pipe，然後in line43~52 child中，在first\_pipe做讀取，在second\_pipe做寫入，之後in line53~69 為parent，in line61~64中，在first\_pipe做寫入的動作以及在second\_pipe做讀取的動作，他會來回重複以上動作nloops(1000000)次，之後分別in line59 and

65來獲取運行時間，之後由line67~68運算平均一次幾微秒之後輸出結果。

Code :

一張含有 文字 的圖片

自動產生的描述

一張含有 文字 的圖片

自動產生的描述