**OS第三次作业：**

3.1 Describe the differences among short-term, medium-term, and long-term scheduling.

**1 长程调度** 这个调度主要是为了解决需要选择哪些进程读入内存（选择哪些进程被同时执行），因为 计算机同时可以执行的进程数是有限的。通过长程调度我们可以控制计算机同时执行的进程数（多 道的程度,degree of multiprogramming)，同时计算机的进程类型主要分为 I/O-bound process 和 CPU-bound process，在同时执行的进程中，一个好的 I/O进程和CPU进程的比例可以提升I/O设备 和 CPU的效率（如果某一类进程过多，会导致另一种设备处于闲置状态）。长程调度可以很好地 控制这两类进程的比例。长程调度主要出现在批处理系统中，很多现代操作系统不存在长程调度 （比如linux)，长程调度的频率比较低。

**2 中程调度** 这个调度也成为swaping，主要是选择将哪些进程中途移出主存（移入硬盘），和将哪些 进程从硬盘移入主存，继续执行。中程调度和长程调度类似，但是本质上是不同的。两者的区别在 于，长程调度是选择哪些进程共同执行（可以理解为被选择的进程原本还在进程池中没有被执行） 而中程调度则是选择将一个执行到一半（正在执行）的进程，调出主存（或者反过来将进程调入主 存继续执行），中程调度的目的是为了，降低多道（multiprograming)的程度，减少进程对CPU的 争夺，释放主存资源。（中程调度可能是因为主存资源不足，需要释放主存资源），其频率比长程 调度高。

3 **短程调度** （CPU调度），就是从 ready queue中选择某些进程来执行（选择将CPU资源分配给ready queue中的某个进程），CPU调度的频率很高（可能每100毫秒就要执行一次）,因此CPU调度的选 择时间会很大程度影响CPU性能。

三种调度的目的不同，最大的区别是执行频率：

短程调度>中程调度>长程调度。

执行时间的要求：

短程调度<中程调度<长程调度。

3.2 Describe the actions taken by a kernel to context-switch between processes.

当context-switch 发生的时候，内核将当前进程的状态（状态由PCB描述）保存下来，然后读出并恢复被调度进程（下一个要执行进程）的状态（PCB）。保存状态的方式和时间取决于具体的操作系统和硬件。

3.6 The Fibonacci sequence is the series of numbers 0,1,1,2,3,5,8,.... Formally, it can be expressed as:

fib0 =0

fib1 =1

...

fibn = fibn−1 + fibn−2

Write a C program using the fork() system call that that generates the Fibonacci sequence in the child process. The number of the sequence will *be* provided in the command line. For example, if 5 is provided, the first five numbers in the Fibonacci sequence will be output by the child process. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the wait() call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a non-negative number is passed on the command line.

代码如下

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

15 if(argc!=2){

16 cout<<"请输入一个参数\n";

17 exit(-1);

18 }

19 int parm = std::stoi(string(argv[1])); // 将字符串转化为 整形

20 if(parm<0){

21 cout<<"参数不能小于0！"<<endl;

22 exit(-1);

23 }

24 // 开辟子进程

25 int pid = fork();

26 if(pid<0){

27 cout<<"fail to fork child process!\n";

28 exit(-1);

29 }else if(pid ==0){

30 //如果是子进程的话

31 if(parm ==0){

32 printf("fib%d = %d\n", 0 , 0);

33 }else if(parm == 1){

34 printf("fib%d = %d\n", 1 , 1);

35 }else{

36 long fb1 = 0 , fb2 = 1;

37 printf("fib%d = %d\n", 0 , 0);

38 printf("fib%d = %d\n", 1 , 1);

39 for(int i =2 ; i<=parm;i++){

40 long v = fb1 + fb2;

41 printf("fib%d = %ld\n",i , v);

42 fb1 = fb2;

43 fb2 = v ;

44 }

45 }

46 exit(0);

47 }else{

48 int status;

49 waitpid(pid,&status,WUNTRACED); // waitpid的作用和wait 类似等待 pid = pid的子进程结束;

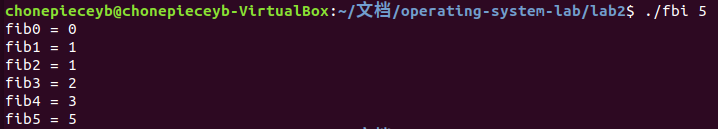
50 exit(0);

51 }

52

53 }

效果图：



54

**OS第四次作业：**

4.4 Which of the following components of program state are shared across threads in a multithreaded process?

a. Register values

b. Heap memory

c. Global variables

d. Stack memory

在多线程的进程中，全局变量是所以线程所共享的。每个线程有自己单独的寄存器值、堆和栈的值

4.5 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system?

对于 many-to-one 的模型来说，虽然说有多个用户线程，但是只有一个kernal threads，一旦一个进程的某个线程被挂起（blocking）那么这个进程的其它进程也被挂起了（因为这个进程的线程争夺一个kernal thread）在这中情况下就算有多CPU,也无法并行地执行其它线程（因为只有一个kernal thread）所以 多CPU并不能比单CPU取得更好的表先。

对于 many-to-many 的模型，在一个线程被挂起，其它线程仍然可以争夺其它的kernal thread，可以实现线程的并行执行，在这种情况下多CPU的表先比单CPU要好。