操作系统大作业

2018011090 于仲明 无83

1. 银行柜员服务问题

要求及描述

实验内容

问题描述:

银行有 n 个柜员负责为顾客服务，顾客进入银行先取一个号码，然后等着叫号。当某个 柜员空闲下来，就叫下一个号。

编程实现该问题，用 P、V 操作实现柜员和顾客的同步。

实现要求

1. 某个号码只能由一名顾客取得;
2. 不能有多于一个柜员叫同一个号
3. 有顾客的时候，柜员才叫号;
4. 无柜员空闲的时候，顾客需要等待
5. 无顾客的时候，柜员需要等待。

思路

根据提示，我们设置两个信号量，一个负责customer，一个负责counter。在customer的线程中P(customer)，V(counter)；在counter的线程中P(counter)，V(customer)，另外维护一个队列来解决同时到达的顾客冲突问题。为了解决何时结束counter的线程问题，首先要标记是否完成了所有customer的服务，即标记一个finish项；其次要设置另一个信号量counter_exit。每当一个线程快退出时，维护一个变量finish_counter使其加1。当finish_counter达到counter总数时，post信号量counter_all.size。在主线程中wait这个变量即可知道是否所有线程均进行完毕。

关键代码

本题目采用posix的信号量与互斥锁规范，并使用pthread进行并行线程编程。

下面是customer部分的代码

```
Customer &customer = *(Customer *)c;
sleep(customer.enter);      // 仿真顾客进入的时间
pthread_mutex_lock(&mutex); // 保护队列
wait_queue.push(customer);
pthread_mutex_unlock(&mutex);
sem_post(&sem_customer); //post customer
sem_wait(&sem_counter); // 要等待有没有新的柜台
sleep(customer.wait);   // 仿真被服务的时间
```

下面是counter部分的代码

```
int tryw = sem_trywait(&sem_customer); //尝试等待
int wait_time = 0;
if(tryw == 0)
{
  pthread_mutex_lock(&mutex); //进入临界区
  if (!wait_queue.empty())
  {
    Customer &customer = customer_all[wait_queue.front().id - 1];
    wait_queue.pop();
    customer.counter = counter.id;
    wait_time = customer.wait;
    customer.begin = time(NULL) - start_sim;
    finish_num++;
  }
  pthread_mutex_unlock(&mutex); //出临界区
  sem_post(&sem_counter); //post counter
  sleep(wait_time); // 仿真wait的时间
  pthread_mutex_lock(&mutex);
}
```

采用sem_trywait更能逼真的呈现多个柜台之间的平等性和随机性，而且理论上如果没有新顾客，不应该执行临界区的代码，也一直等待。

运行结果

假设有5个counter

输入

```
1 3 5
2 10 5
3 9 3
4 5 10
5 6 6
6 3 3
7 9 7
8 14 3
9 2 9
10 1 9
```

输出

```
1 3 3 8 1
2 10 11 16 3
3 9 10 13 5
4 5 5 15 4
5 6 6 12 2
6 3 3 6 2
7 9 9 16 1
8 14 14 17 2
9 2 2 11 3
10 1 1 10 5
```

可以看到上述结果和我们手动分析的一致。

思考题

1. 柜员人数和顾客人数对结果分别有什么影响?

采用上述输入的数据，改变柜员人数做出以下实验结果

柜员人数	总时间/min	平均等待时间/min
2	32	8.9
3	23	2.1
4	18	0.9
5	17	0.1
6	17	0

柜员人数越多，若多于顾客，顾客之间的独立性越强，即来了就可以被服务，顾客几乎不用等待。顾客人数越多，若多于柜员，则需要解决先来顾客和在服务顾客之间的互斥关系，顾客等待时间越长，则总的服务时间越长。

2. 实现互斥的方法有哪些?各自有什么特点?效率如何?

方法	优点	缺点	效率
禁止中断	简单	把禁止中断的权利交给用户进程，导致系统可靠性较差;不适用于多处理器	单处理器较高
锁变量		可能使两个进程同时处于临界区;忙等待	较低
严格轮转法		有可能使临界区外的进程阻塞其他进程;忙等待	低
PETERSON	解决了互斥访问问题，克服了强制轮转法缺点，可以正常工作	忙等待	低
硬件指令方法	适用于任意数目的进程;简单，易验证其正确性;支持进程中存在多个临界区	忙等待，耗费CPU时间	低
信号量	适用于任意数目的进程;解决忙等待问题	实现较复杂，需要系统调用;同步操作分散，易读性较差，不利于修改和维护，正确性难以保证	高
管程	提高代码可读性，便于修改和维护， 正确性易于保证	编译器必须识别管程并用某种方式对其互斥作出安排;多数语言不支持	高
消息传递	适用于分布式系统	更加复杂	高

实验总结

本次实验主要难点在于思考用什么信号量来编程，以及在什么位置P, V。思考清楚后，还需要考虑counter最后终止的问题，即如果不会再有顾客来了的话，应当结束所有线程。这个问题需要引入其他的信号量以及全局变量加以控制。总体上来说本实验增强我对了对信号量，互斥锁的整体认知和实践能力。

2.快速排序问题

要求及描述

(1)首先产生包含1,000,000个随机数（数据类型可选整型或者浮点型）的数据文件；

(2)每次数据分割后产生两个新的进程（或线程）处理分割后的数据，每个进程（线程）处理的数据小于1000以后不再分割（控制产生的进程在20个左右）；

(3)线程（或进程）之间的通信可以选择下述机制之一进行：管道（无名管道或命名管道）消息队列共享内存

(4)通过适当的函数调用创建上述IPC对象，通过调用适当的函数调用实现数据的读出与写入；

(5)需要考虑线程（或进程）间的同步;

(6)线程（或进程）运行结束，通过适当的系统调用结束线程（或进程）

思路

首先用一个随机数生成的代码生成数据。然后采用mmap将文件映射到内存，然后再利用共享内存的方式在多个线程中进行并行排序。由于是快速排序，因此分割数据很重要。本代码考虑在分割数据的时候采用nth_elements方法，记录头尾，并将这个定义成一个结构体。将该数据储存到一个全局vector中。刚开始的时候由于基本上不会产生小于1000的数据段，所以一边分裂数据，一边产生新的线程。根据nth_elements的特性，它可以保证分裂完毕的两端之间大块是有序的，即一边的最大值要小于另一边最小值。当线程数量到达我们设定的数据上限时，停止分裂线程。这时若线程负责的数据段中范围小于1000，则调用qsort完成快速排序。当仍大于1000时，继续分裂数据并把头尾push_back入全局vector。

关键代码

```
void *sort_thread(void *data)
{
    int *arr = (int *)data;
    while (finish != 1 || posStat.size() > 0) //退出循环的条件
    {
        pthread_mutex_lock(&mutex);
        pos tmp;
        if (posStat.size() != 0)
        {
            tmp = posStat.front();
            std::vector<pos>::iterator iter = find(posStat.begin(), posStat.end(), tmp);
            if (iter != posStat.end()) //寻找并删除
                posStat.erase(iter);
        }
        else
            tmp = {0, 0}; //若没有，则置零，这种情况会在最后数据段小于线程数的情况发生

        int start = tmp.min;
        int end = tmp.max;
        int stride = end - start;
        pthread_mutex_unlock(&mutex);

        if (stride > DIVIDE)
        {
            std::nth_element(arr + start, arr + start + stride / 2, arr + end);
            pthread_mutex_lock(&mutex); //进入临界区，保护posStat数据结构
            if (stride > DIVIDE)
            {
                posStat.push_back({start, start + stride / 2});
                posStat.push_back({start + stride / 2, end});
            }
            pthread_mutex_unlock(&mutex);
            if (threadNow < MAXTHREAD)
```

```
            {
                sem_wait(&threadSize);
                threadNow = threadNow + 2; //竞争资源threadNow标记当前的线程数量
                sem_post(&threadSize);
                pthread_t *thread_handles = (pthread_t *)malloc(thread_count *
sizeof(pthread_t));
                for (long thread = 0; thread < thread_count; ++thread)
                {
                    pthread_create(&thread_handles[thread], NULL,
                                   sort_thread, data);  //分裂出新的线程
                }
            }
        }
        else
        {
            if (stride != 0)
                std::qsort(arr + start, stride, sizeof(int), compare); //自定义排序compare函数
            pthread_mutex_lock(&Nsorted); //保护allCount
            allCount += stride;
            if (allCount == SIZE)
            {
                finish = 1;
                sem_post(&finished); //finished信号量标记全部排序完成
            }
            pthread_mutex_unlock(&Nsorted);
        }
    }
    pthread_exit(0);
}
```

这里有个trick时使用finished信号量标记是否排序结束。因为如果不等待线程做完的话，可能会出现未完全排好序而出错的场景。

思考题

1. 你采用了你选择的机制而不是另外的两种机制解决该问题，请解释你做出这种选择的理由

选择共享内存是一种高效的IPC通信机制，不过需要维护读写共享内存之间的互斥和同步。但总的来说，由于其共享的特性，是得过个线程之间协作十分方便。对于本快排来说，实际上共享的就是一个大的数据数组和一个记录首尾的数组，因此使用共享内存编程十分方便。另外的机制如管程我理解就是一个包装的信号量，而且对于C/C++语言不友好，所以不好实现。快速排序实际上并不需要点对点的线程之间进行消息传递，因此消息传递的模型可能并不适用。

2. 你认为另外的两种机制是否同样可以解决该问题？如果可以请给出你的思路；如果不能，请解释理由。

我认为管程对这个问题并不友好，因为实际上分裂出的首尾结构体需要让当前所有线程均知道，所以本质上还是一个共享的方法，与管程模型并不一致。对于消息传递的方法，则需要线程点对点的告知其他线程当前的排序状况，然后再完成同步。虽然理论上可行，但是通信代价使得编程开销大，而且实现复杂。

实验总结

本实验加强了我对共享内存的认知。通过使用mmap等函数，我知道了linux对文件映射和内存管理等系统调用方法。另外，这个编程中增强了我对信号量，互斥锁的进一步认知，深入理解了pthread并行编程的思路和原理，收获颇丰。

3. 文件字节倒放问题

要求及描述

生成一个由随机产生的字符型数据组成大的数据文件（例如，大小≥1GB）。将该文件中的所有字节进行倒放，存入原文件，即将文件中的首字节与尾字节对换，次字节与次尾字节对换，以此类推。编写两个程序，一个程序采用常规 的文件访问方法，另一个程序采用内存映射文件方法。请记录两种方法完成字节倒放所需要的时间，并进行比较。

思路

对于普通的文件读写，首先利用fopen打开一个文件，然后采用seek到不同的位置，进行read和write。先read首部的代码，然后再read尾部，再把首部临时变量写入，再返回首部写入尾部读到的数据。

关键代码

普通文件读写

```
for (int i = 0; i < SIZE / 2; i++)
{
  fseek(fp, i, SEEK_SET);
  fread(start, sizeof(char), 1, fp);
  fseek(fp, SIZE - i - 1, SEEK_SET);
  fread(end, sizeof(char), 1, fp);
  fseek(fp, SIZE - i - 1, SEEK_SET);
  fwrite(start, sizeof(char), 1, fp);
  fseek(fp, i, SEEK_SET);
  fwrite(end, sizeof(char), 1, fp);
  if (i % FLUSHPERIOD)
    fflush(fp);
}
```

采用mmap之后，直接对内存操作

```
char *data = (char *)mmap(NULL, SIZE * sizeof(char), PROT_READ | PROT_WRITE, MAP_SHARED, fd,
0);
for (int i = 0; i <= SIZE - i - 1; i++)
{
  char c = data[i];
  data[i] = data[SIZE - i - 1];
  data[SIZE - i] = c;
}
```

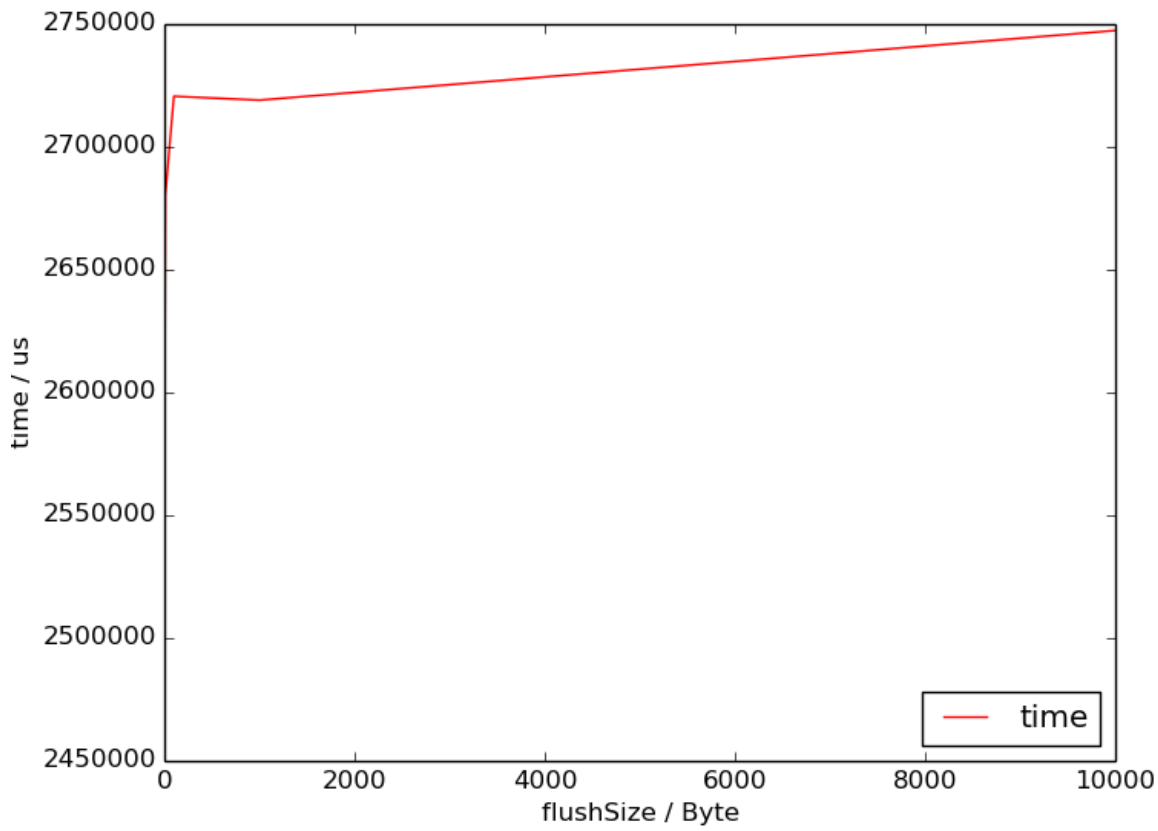
思考题

1. 采用常规的文件访问方法时，改变缓冲区的大小对程序的性能有什么影响？请用图表描述缓冲区的大小与程序性能之间的关系。

修改FLUSHPERIOD来探索，由于1G文件太大，运行时间过长，我们改成1M文件用来实验

FLUSHPERIOD	time(us)
1	2469385
2	2585858
5	2623551
10	2680261
100	2720610
1000	2719037
10000	2747389

可以看到，除了一些不稳定的随机波动，随着FLUSHPERIOD增大，也就是缓冲区增大，反而导致程序花费时间增大，性能下降，绘制图表如下：



2. 内存映射文件方法和常规的文件访问方法在性能上有什么差异，试分析其原因。

时间开销

内存映射(us)	文件访问(s)
1308491	2264

原因1 内存映射方法没用到缓冲区，避免IO操作，使得程序想内存一样访问磁盘文件。

常规文件访问需要用到文件IO，而且有缓冲区，文件字节倒换时还需要频繁进行磁盘上的seek，十分耗时。

实验总结

本实验加强了对文件操作，内存映射的认识和实践，对今后学习如何高效读写文件有很大的帮助。

代码地址： https://github.com/fishmingyu/THUEEOS