Project2 Introduction

1 Components

I create two threads in main function: ProcessSche and PageRep. These two threads work simultaneously so that they can simulate how CPU and memory work. After I read project2 problem statement file, the first idea of the project is that thread programming can achieve these requirements. But after I finished this project I exchanged my idea with some classmates and I found out that they did not use any thread. I reviewed my program and I found out that actually a single thread was absolutely able to achieve these requirements. But the time is not enough because I have to change lots of code to convert two threads to a single thread. So this is the final version. I will introduce these two threads in Section 3. All include files are as follows.

- Include files:
- Struct.h : includes all structures used in this project. Strucure PSarg is used to transmit parameters to thread ProcessSche. Structure PRarg is used to transmit parameters to thread PageRep. Structure _process includes all sufficient variables to describe a process, such as name, startTime, CPUTime and so on. Structure page is used to describe how page behaves in page replacement algorithms, which has three variables : refBit ,which is reference bit ,addr, which is the address of the page that result in page fault, and pid ,which the process id of the page, for example, the pid of P1 is 0, the pid of P2 is 1, the pid of P3 is 2 and so on.

- Queue.h : includes two classes: Queue and circularQueue. Queue is similar to STL class queue; circularQueue describes a circular queue. I add some functions in them in order to make them convenient to use.

- priorityQueue.h : includes a class priorityQueue, which is similar to STL class priority_queue. It is based on a array, and is used for arrivalQueue in thread processSche, because the element of arrivalQueue is sorted by the order of time.

- Mem.h : includes a class Mem, it simulates how memory deals with page fault. This class has a Queue named memory to store page address, and is used by fifo and lru algorithm; it also has a priorityQueue named pageQueue, and is only used by second-chance algorithm.

2 Poilicy decisions

2.1 The program first read the scheduling trace file, and push() all the processes to arrivalQueue, which is a priority queue. Then thread ProcessSche and PageRep starts to work.

2.2 There is a increasing counter in ProcessSche named currentCycle. If it equals to the startTime of the first process of arrivalQueue, then pop() arrivalQueue, and push() it to the end of readyQueue.

2.3 If a instruction results in a page fault, then the process will be removed from the scheduler, pushed to the end of blockQueue, and let the next process in readyQueue to execute. At the same time, thread PageRep will deal with the new process in blockQueue by different methods according to the replacement policy you enter.

2.4 In thread PageRep, the startTime of the process will be updated, and pushed to the end of arrivalQueue. So the next time this process starts up, a page faullt probably may happen again.

3 Introduction of threads and their synchronization

3.1 ProcessSche

This thread simulates how scheduler works. In each loop, a increasing counter currentCycle is increased by 1 to record the current cycle. If currentCycle equals to the first of arrivalQueue, it means a new process arrives, so push it to the end of readyQueue. Then follow these steps:

- 1. check whether status is CONTEXTSWITCH, if so, continue to the next loop; else go to step 2
- 2. if there is no runningProcess, go to step 3; else go to step 6
- 3. if readyQueue is empty, go to step 4; if not, go to step 5
- 4. increase noExecTime by 1, go to step 5
- 5. pop() the first process of readyQueue, and let it be runningProcess, go to step 6
- 6. read the current instruction from the memory traces, and check if it is in memory, if so, increase execTimeCycle; else push() it to blockQueue, and let runningProcess be NULL
- 7. increase currentCycle by 1, go to step 1.

  The thread will follow those steps until the number of finished process equals to the total number of processes.

3.2 PageRep

This thread simulate how memory deals with page fault. In each loop, follow these steps:
1. if blockQueue is not empty, then pop() blockQueue, and do page replacement according to the policy you enter, update the startTimeCycle of this process, push() it to arrivalQueue, and continue to the next loop
2. if blockQueue is empty, continue to the next loop.

The thread will follow those steps until the number of finished process equals to the total number of processes.

3.3 Synchronization

There are two threads running simultaneously in this program, so it is significant to synchronize them perfectly, otherwise error is likely to happen, such as deadlock, or wrong result may happen.

3.3.1

Because thread ProcessSche and PageRep work at the same time, and move page from disk to memory costs 1000 cycles, so in the 1000 cycles to come, the process must be pushed back to arrivalQueue again, or ProcessSche has to wait for it, otherwise the time will exceed 1000 plus the time when it blocked, and those processes which are supposed to be later than this process will be earlier than it. To implement this function, every time PageRep move a page from disk to memory, it signal a semaphore page_fault. In ProcessSche, after 1000 plus the time a page fault happens it wait the semaphore page_fault.

3.3.2

arrivalQueue blockQueue and validPage are shared by these two threads, so there are three mutex arrivalQueue_mutex blockQueue_mutex and validPage_mutex.

4 Experiment result presentation

4.1 page replacement algorithm

Memory size: 500, quntum: 50000, sche_traces_big.txt				
page replacement algorithm	total elapsed time/cycle	total idle time/cycle	total page faults	average elapsed time/cycle
fifo	7490260966	7477408972	7489479	4998531490
lru	6724043865	6711191871	6723185	4565145542
2ch-alg	6731502348	6718650354	6730612	4571226048

It is clear that lru performs better than the other two algorithms, while fifo is the worst. Yet second-chance algorithm is not that bad. FIFO is easy to implement but does not perform well. LRU algorithm needs more hardware and software support so that it performs better. And 2ch-alg is a compromise, because it is not complicated as LRU algorithm, but is more efficient than FIFO.

4.2 Time quntum

page replacement algorithm: 2ch-alg, memory size: 500, sche_traces_big.txt				
quntum/cycles	total elapsed time/cycles	total idle time/cycles	total page faults	average elapsed time/cycles
10000	6569195336	6556343342	6568447	4403496691
50000	6731502348	6718650354	6730612	4571226048
200000	6723008906	6710156912	6722138	4565657313

We can find out that larger time quntum results in worse performance. It is because if the time quntum is too large, round robin policy degenerates to FCFS policy, which is not efficient as RR policy.

4.3 Memory size

page replacement algorithm: 2ch-alg, quntum: 50000, sche_traces_big.txt				
memory size	total elapsed time/cycles	total idle time/cycles	total page faults	average elapsed time/cycles
150	11907315051	11894463057	11906770	8118111147
500	6731502348	6718650354	6730612	4571226048
5000	19346479	6494485	9973	3878692

I fail to run this program if the memory size is only 50. It is probably because the process that has page fault will fault again for the same page. 150 is much smaller than 500, I think it can take the place of 50 in this experiment, and is also able to represent a small memory in a computer.
From these data, it is clear that the bigger the memory size is, the better the program performs. It is because more memory size means less page fault. In extreme condition, if memory size is the same as disk, there will be only one page fault for each page.