project2_Report

I、Abstract

In this project, I write a paging simulator. It will read in a set of data files specifying the scheduling and memory access behavior of programs, and will simulate the paging requirements of those programs.
As input, there are two types of files: scheduling traces and memory traces (Scheduling files contain a trace of process start time, CPU time, and I/O count. Here we only look at start time and CPU time. Memory trace files contain a list of addresses accessed by a program. The addresses are truncated to virtual page numbers by removing the lower 12 bits, which makes the files smaller.).
According to the requirements, I simulate a fairly slow system, where there are 100,000 cycles in a second.

II、Coding Process

1、structures overview

Memory is constructed as an array whose elements are all structure 'page', which means the memory has 'memsize' pages. Each page contains its page process, page virtual address, count (The most initial definition,only consider FIFO algorithm first).
Count is a variable defined to protect the memory, if it doesn't exist, a situation may occur that the memory keeps taking page fault without ending. If the page is just exchanged into the memory and not ever be used, then its count is 0, which means the page can not be replaced by other pages. And if the counts of all the pages in the memory are all zeros, then no page can be replaced now and it will enqueue the process to the block queue.(This will be discussed later.)
Process is constructed to represent processes to deal with. The ready queue, block queue and finish queue are all consist of processes. Each process contains its name, arrive time, CPU time, need time, re-arrive time, elapsed time, Io ops, page Fault, state, src ,*next. The processes we need to deal with are initialized first in the main function.

2、functions description

==========

main function —— int main(int argc,char *argv[])

In this function, the parameters given by users are read and recorded. First, initialize the ready queue with the processes' information given in the traces file. Second, get value for page replacement policy, quantum, memory size. Third, construct an available array as our memory, and initialize

each page of the memory. Fourth, call the RounRobin function to execute our simulating for processes. Fifth, after executing the finishing results will be printed to the terminal, also before which some of the required results should be calculated including timings and page fault numbers.

==========

create function —— void create(char *openFile)

This function is used to read in the file scheduling traces and construct the processes into the ready queue. As the first function called in the main procedure, its only function is initialization.

==========

insert function —— void insert(struct process *p)

This function is used to insert a process to the ready queue. Every time by comparing the re-arrive time of the process needs to be inserted with that of the processes already in the ready queue we can find the position in the ready queue to insert our process, and then the process is inserted. This function is called when one quantum finishes and the need time of the process is still not zero then the process need to be inserted again, or a process in the block queue is unblocked.

==========

block function —— void block(struct process *p)

This function is used to add one process into the block queue. Every time it points the block tail pointer to the newly blocked process. This function is called when one process meet a page fault and also it can not replace any page of the memory, then the process is blocked.

==========

deblock function —— void deblock()

This function is used to remove one process of the block queue. Every time it inserts the process which the head pointer points to to the ready queue and points the block head pointer to the next process behind the head pointer. This function is called when the block queue is not null and one page of the memory has been used (the page can be replaced).

==========

compare function

—— bool compare(struct page* memoryi,char* temp,char*name)

With the calling parameters are one memory page, one virtual address array, and one process name array, this function is used to compare the given address and process name (process identifier) are whether or not

same as the page in the memory. This function is called when determining if one page needed is already in the memory or not. If the page is the same as the memory [i], the it will return true. Otherwise false.

==========

pagefault function —— int pagefault(struct page* memory,char *temp,char *name,int memsize,int len, char* pr)

The representatives of the parameters of the function are: memory-memory; temp-the virtual address need to be replaced into the memory; name-the process name need to be replaced into the memory; memsize-the size of the memory; len-the current used page number of the memory; pr-the chosen page replacement algorithm.

A boolean variable named replaceFlag is used in the function to reveal the page has succeeded in replacing or not. This will determine the process is then inserted to the ready queue or the block queue.

During page replacing process, first check if the number of the pages used in the memory is less than the total memory size, and if it is true, we can directly replace the first unused position and change the replaceFlag to true, otherwise we need to execute the page replacement algorithm according to the parameter.

FIFO —— The current position is defined as the global variable 'first', and a while loop is used to check if the 'first' position is available for page replacing. After the loop, the boolean variable 'shen' is used to reveal whether the replacement could take place. If 'shen' is true, then exchange the page to the 'first' position in the memory, 'first' increment one, its count element is initialized as zero, and replaceFlag is changed to true.

LRU —— First, find the first available position, and record it with 'least'. Second, find the position in the memory where the 'used' of the page is less than that of 'least' one, and also the page is available. (Attention: 'used' is a new element definition of the page structure.) Third, replace the 'least' page, and change replaceFlag to true.

2ch-alg —— According to the algorithm, a new element 'second' is added to the page structure. The current position is denoted by a global variable 'sec'. We need to find the position where its second is 1 and count is not zero. The boolean variable 'ch' is used to reveal whether such a position exists or not. If 'ch' is true, then we can replace the page on the position 'sec' and change replaceFlag to true.

count-alg —— My implemented algorithm is based on the number of the

page used before, and each time the least used (also the count is not zero) will be found and replaced. Here, a new element 'occur' is added to the page structure for recording the page's used count. Much like the LRU algorithm, eachtime we find the smallest 'occur' (also the count is not zero), if exist such a 'occur' position, then we replace the page and change replaceFlag to true.

my-alg ——my algorithm is a vision based on the least recent use algorithm. Here I first search the memory array if there is a page whose name is not the processing process's name. If there exists, then the page is prepared to be exchanged. Also, we should find the most recent used one. Otherwise, just use the least recent used algorithm.

After page replacement algorithm is finished, we should first move the page's file_pointer back four positions. If replacement succeeded, put the process back to the ready queue (seriously consider its re-arrive time and the clock time—the re-arrive time of the process should add 1000 cycles for page replacement, and check if it will be put to the ready head ->next or not to represent if there need a context switch). If replacement not succeeded, put the process to the block queue, and clock time add fifty cycles (for context switch).

This function will finally return the global integer variable 'len'. And the pagefault function is called in the RoundRobin function, used when a page fault is met by a process.

==========

RoundRobin function —— void RoundRobin(double quantum,int memsize,struct page* memory,char *pr)

The representatives of parameters of the function are: quantum-quantum; memsize-the size of memory; memory-memory; pr-page replacement algri-thm.

In this function, there is mainly a while loop whose condition is the ready queue is not null or the block queue is not null. (Actually there can not exist a situation that the ready queue is null and the block queue is not null. Because at the very beginning, all the count is initialized as one, and it means if one page's count is zero, then the page must experienced page faults and the corresponding process must have been put into the ready queue).

Then we will have two situations to discuss: (1) need time<=quantum; (2) need time >quantum.

Need time<=quantum —— 1>when clock time is less than the next re-arrive time, then the clock time is assigned as the re-arrive time. 2> check the startflag to record the start time which is used to calculate the elapsed time at last. 3> create a for loop loops need time loops, within each loop, it will read in one trace, and first check if the 'len' is zero (if it is zero, then faultflag is true and break the loop; if it is not zero, then continue checking) then check if the trace is in the memory (If the trace is in the memory, then the clock time increments one, need time minus one, and effect time increments one. If not, faultflag is true and break the loop.). 4> If faultflag is true, it means page fault takes place, and the integer for recording the number of the page fault of one process increments, and then call the pagefault function defined before. 5> If faultflag is false, it means the process has finished, then we should move the process out of the ready queue and put it into the finish queue. If the ready queue is not null after moving, that means context switch is needed, and clock time increments 50.
Need time>quantum —— 1>when clock time is less than the next re-arrive time, then the clock time is assigned as the re-arrive time. 2> check the startflag to record the start time which is used to calculate the elapsed time at last. 3> create a for loop loops quantum loops, within each loop, it will read in one trace, and first check if the 'len' is zero (if it is zero, then faultflag is true and break the loop; if it is not zero, then continue checking) then check if the trace is in the memory (If the trace is in the memory, then the clock time increments one, need time minus one, and effect time increments one. If not, faultflag is true and break the loop.). 4> If faultflag is true, it means page fault takes place, and the integer for recording the number of the page fault of one process increments, and then call the pagefault function defined before. 5> If faultflag is false, it means the process has finished this quantum, and it should be inserted to the ready queue again. Then we should check if a context switch is needed (check if its re-arrive time is larger than the next process in the ready queue.), if it is needed then clock time increments 50.
At the end of the function, it will print some of the results to the terminal.
==========

III、Policy Decisions Descriptions

(1) Do new processes go at the head of the ready list or the end?

>>>New processes go at the end of the ready list. The ready list is initialized in the create function as a queue.

(2) When a process starts up after a page fault, is it guaranteed to have the faulting page in memory, or could it fault again?

>>>It is guaranteed to have the faulting page in the memory. I used a variable named 'count' to represent whether the faulting page has been used or not.

(3) Do new processes or processes returning from a page fault get priority?

>>>In my program, new processes get priority when new processes and processes returning from a page fault have a same re-arrive time.

(4) When a process is blocked and when a process is unblocked?

>>>Here, page replacement can take effects only when there exists some available page that has been used since its last into memory. Therefore, when all the pages in the memory are newly replaced into memory and have not been used, then the process which needs page replacement will be blocked.

>>>When a process find its operating page has been in the memory, then it can use the page, which will make the page in the memory be available, then we will unblock one process into ready queue. Also, one unblock will happen when the ready queue is null and the block queue is not null.