Operating System Project Report

I The architecture of simulator

a) **General description**
The simulator system is splitted into two part, memory simulator and CPU simulator, just like the architecture as a real computer. There are two main class in this project:

simulator This class handle all kinds of interrupt and event, it implements the schedule algorithm, it maintains ready queue, CPU time and an event queue for simulation event(explained later).

mem_sim This is a virtual class, Since different replacement strategy are used in this simulator, a virtual memory simulator class is needed. The simulator class interact with memory simulator via a function clock_tick(). This function call will read the next memory reference and check if it is already loaded in memory. It will also generate page fault event in simulator.

b) **main function**
Entry of this project, parse command-line arguments and pass to simulator. Then it calls function in simulator class to drive the simulator until all process are finished.

c) **simulator class:**
This class handle all kinds of interrupt and event, and interact with memory simulator to simulator. There are several interfaces functions:

clock_tick main function drive the simulator via thist function call, this function will handle event happened in current clock(including page fault, page fetched into memory, process start, process finish, see event for detail) and handle a memory reference if there are process running.

schedule Scheduler implemented in this function, it will check the current task pointer and ready queue. After this function call, current process will be scheduled out(if there is running process) and pushed into the tail of ready queue. If context switch is need, it will add the context-switch time to system clock directly, otherwise don't change the system clock(for example, current process is not scheduled out). Process will not be moved out of ready queue.

switch_out Different from schedule function, this function move current process out of ready queue. That means that current is interrupted

and waiting for some event. Since I used an event queue mechanism in this project, there is no need to maintain a waiting queue in simulator. Then is set current task pointer to empty, and schedule function is called in this function, since current is set to NULL, scheduler will pick a new process in ready queue. The size of ready queue is reduced by 1 after this call.

Besides, several member variables are descripted here

current pointer point to current task_structure.

readyQueue Ready process queue.

finishedSet tasks that has finished, Since I used a event queue , which is separated from processes, Then event relative to a process may happen after the process has finished, causing problems, So a check of whether a process has finished is needed.

eventQueue This is a priority queue, sorted by the time of an event. This simulator is driven by this event queue. Event including a page loaded in to memory and process start. each clock the simulator will check whether some event happened in this clock circle, and handle the event. And memory simulator will generate page fetched event. On process start, a new process is added into the tail of ready queue. while on page fetched event, a process is set awake again and be pushed into ready queue.

d) **mem_sim class:**
This is a virtual class, defines interface of memory simulator system and CPU_os simulator system. There are two functions defined in this class:

clock_tick the interface provided to simulator, one memory reference should be handled or a page is generated. If a page fault is generated, then an page fetch event is added into the request of memory system. If memory is bot busy, then a new page fetched event is add into event queue of simulator, After certain time, the event will be handled by simulator and corresponding process is loaded into ready queue again. Thus the two part of this system is coupled. This function will call handle_page_event to help it to finish its job.

find_page this Function varies in different replacement algorithms, it returns the next page that should be replaced.

The three different algorithms are implemented by extending this class, and override the clock_tick function. The construction of simulator class will parse the input param and choose proper class in-

stance to initiate the mem_sim pointer.

e) **task_structure:**
This class stores information about the task. The construction finction of this class will read the mem_trace file and store memory reference in a queue. All process related information are stored in this class. There are exactly one task_structure for each process. The simulator will create them and refere them as pointer.

f) **event:**
This class stores information about events in simulator, they are stored in a priority queue in simulator as descripted in previous parts. C++ operator $<$ is implemented in this class to compare the time of each event.

This kind of architecture is flexible, especially for adding a new replaement algorithm. Only need to add corresponding entrence in simulator class and add corresponding class that extends the base class, then a new memory algorithm can run on the same simulator.

II policy decisions in this project

(a) A new process will always go to the tail of ready queue. So the new process will not interrupt the execution of current process, and will not affect anything except for ready queue.

(b) The process just weak from waiting will not generater another page fault again. This is garenteed by the implementation of simulator. There are two queues storing events, one is in mem_sim, called request queue, the other is in simulator, called event queue. Request queue will store all page fault event, keep records of whitch process need a page and in what order should the one page be loaded for the process. There can be many electments in request queue especially when the memory size is small, while there is only one page fetched in event queue. That is, the new loaded page will be at least refered once before its replaced. So a process with n memory refers may generate at most n page fault.

(c) Since any process will fault again, so any process returning from page will goes to the tail of readyqueue. Other process may gnerate page faults during this time, but they will be switched out immerdiately and there querry are queued.

(d) The page load queue is a FIFO queue, that is, the pages that are required earlier will be loaded into memory earlier.