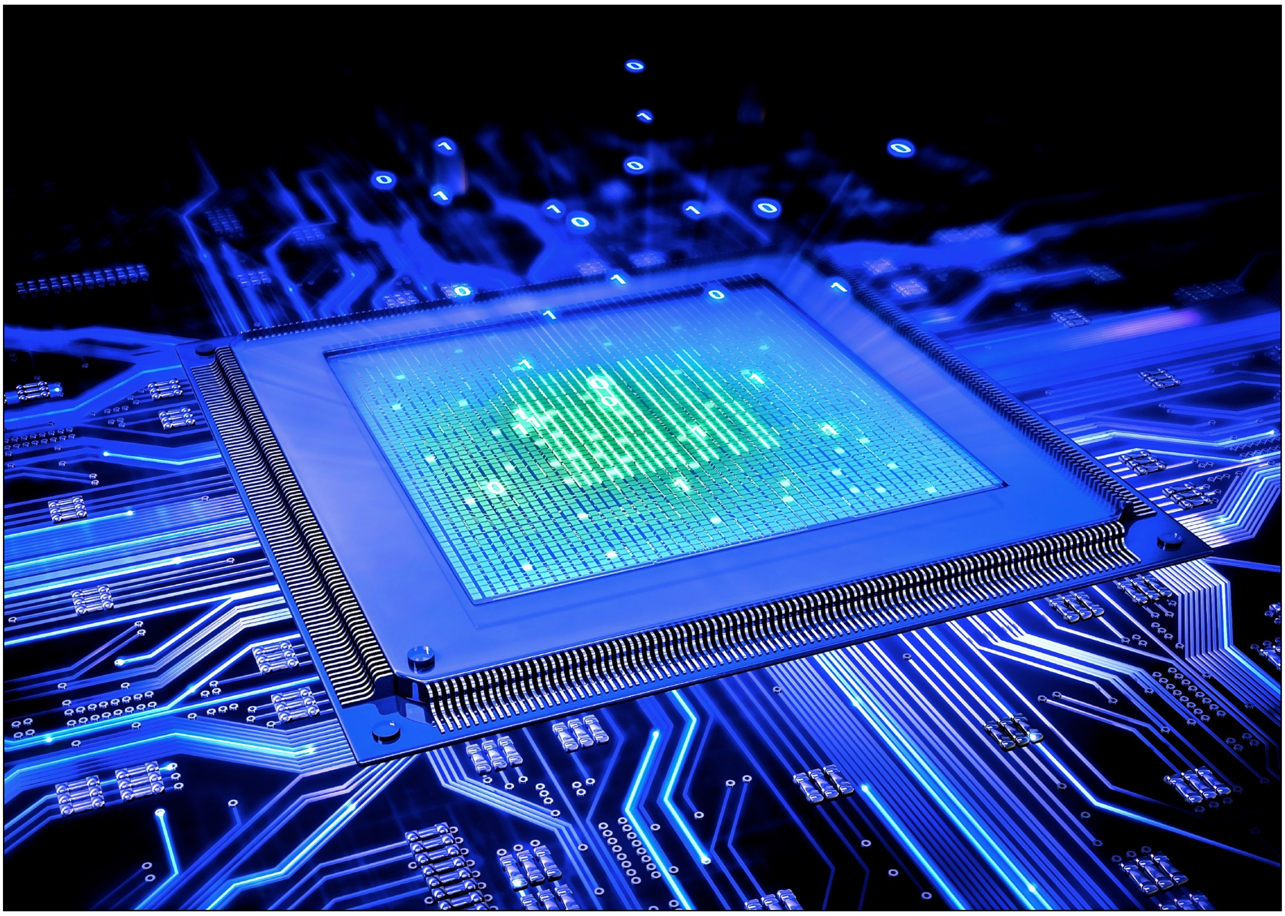
A tiny simulator

杨文杰 - 2015年6月9日



Abstract

This project focus on the simulation of a simple multi-threaded operating system. A tiny simulator should contain three main components: cpu, memory and scheduler.

The cpu is the kernel component of the simulator, which keeps track of which process is running and handle the interrupts. The cpu also contributes to context switching and fetching memory.

The memory focus on maintaining the page table, providing cpu which page is in the memory. When the memory is full, it should decide which page to kick out and spare space for the coming page. The replacement algorithm depends on the user's choice.

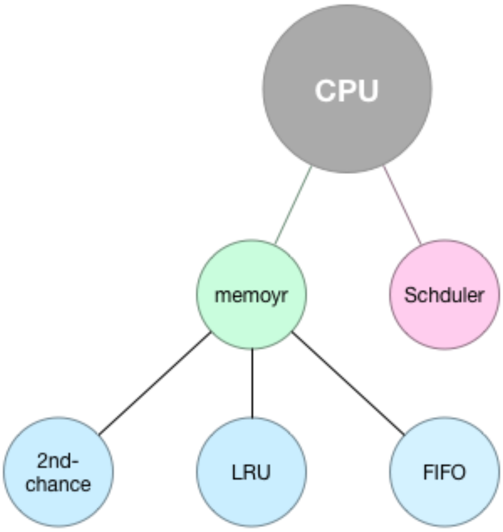
The scheduler should focus on the process switching. When the quantum expires or the process meet a page fault. The scheduler should pick up a process to run. If there is no ready process to run, it will switch to the idle process.

1. Introduction

A common operating system's main purpose is to abstract the hardware and make process feel like it doesn't share the hardware with other processes. In linux, the kernel maintain a process table and keep track of which process is running. When a interruption comes, the CPU will turn from user mode into kernel mode, and look up the interrupt vector table to find the interrupt function. If a page fault interrupt comes, the kernel will save the register information as a TSS and then pick up a ready process following reloading its TSS.

Our project is to simulate the behavior of the operating system during the multiple task switching. So the first component we should focus on is to write the simulator of the CPU and simulate its behavior.

2. Project structure



The structure of the project is on the left. The CPU aims to manage the core component of the simulator. Each component will be explained as follows.

First of all, I will explain the data structures I used in this project.

2.1 Data Structures

1. process

I don't use a global variable to store the process's information, for it's not necessary. The main purpose of the linux process table is to maintain the information of the process. The header file of the process table is greater than 2KB! But!! In this simulator, we only need to focus on the memory fetching. So there is no extra information we need to store in the program. We only need to know which page we need to fetch next, which can be done by the C++ standard IO features.

2. interrupts

In this project, there are four kind of interruptions: creation, disk, disk discard, timer. The main program is that how to store them. I use a map to store them. But how to distinct them? The idea I use here is make the map's key to be a pair, whose first element is the interrupts time and the second element is the interrupt's type represented by a integer. The advantage of this structure is that the closest interrupt can be picked very fast, O(1) time. And the search algorithm is O(log n). Because the STL map is store as red black tree. It will save a lot of time and space.

3. memory

In common implementation, the page table will be saved in the main memory. But in this simulator, it will be every expensive to do so. I use a map to store whether a page is in the main memory or not. It will save a lot of time and space. But the main disadvantage is that the random access time is long.

2.2 CPU

CPU is the core component of a computer. Its task is to perform the calculation of the program. The task of the operating system is to run multiple process synchronously and let the program feel like it does not share the CPU with other process. Here are the declaration of this component.

```
class CPU {
      Scheduler *scheduler;
      Memory *memory;
      //Process Name -----> The process memory trace file descriptor
      map<string, ifstream*> fd;
      //process Name -----> cycleNum, page fault number, terminate time.
      map<string, long long> cycleNum, pgFaultNum, terminateTime;
      //The process buffer.
      map<string, string> buffer;
      long long idleCycle;
      long long currentTime, currentProcessStartTime;
      string currentProcess, nextMem;
      bool readBuffer(string processName);
public:
      void initialize(Scheduler* s, Memory* m);
      void simulate();
```

```
	void ContextSwitch(string newProcess, long long 
newProcessStartTime);
	void pgFaultIncrease(string processName);
	void cycleIncrese(string processName);
};
```

I will explain the functions.

1. `void simulate()`

   This is the main function of the whole project. The function's main component is the while loop. In each loop, the CPU will first check out interrupt table to find if there comes interrupts. If so, handle the coming interrupts. And then it will check what process is running. If the current process is IDLE or context switching, it will go to the closest interrupt and then handle it. If not, the simulator must be running a process, so we need to read the next memory. If the memory page is not in the main memory (depends on the memory's response), we will go to page fault and perform context switching.

2. `void ContextSwitch(string newProcess, long long newProcessStartTime)`

   This function is to perform context switching. It's very simple. Just change the currentProcess to the newProcess.

2.3 Scheduler

Scheduler is one component of an operating system. Linux 0.12 kernel used the scheduler based on the priority queue. The scheduler function scan the task array, comparing every task's tick count, picking up the process with biggest count value. If every process's quantum has expired, the scheduler will calculus the counter and start to tick again.

In this project, our scheduler is simpler. It used the round-rubin algorithm to construct the scheduler. This algorithm's main idea is to distribute each process a time slice and each time when the CPU switch to the process. The process will reduce its quantum when it expires its quantum, it will make room for other processes.

```
class Scheduler {
	//Process's infomation : The rest time of the quantum and the 
process' next timer interrupt
	map<string, ProcessInfo> infoTable;
	//interrupts' time and type ----> the detailed information the 
interrupt, such as the process name or the interrupted page name
	map<interrupt_t, Interrupt> interrupts;
```

```
    //The queues which stores the faulting, hanged, ready process.
    deque<string> faultQueue, readyQueue;
    //blocked process
    //set<string> blockedQueue;
    int blockedNum;
    CPU* cpu;
    Memory* memory;
public:
    /*
     * return the closest interrupt from now.
     */
    long long cloestInterrupt() {
        if (interrupts.begin() == interrupts.end())
            return -1;

        return (interrupts.begin()->first).first;
    }

    void initilize(CPU* c, Memory* m) {
        cpu = c;
        memory = m;
    }

    void creationInterrupt(long long time, string processName);
    void diskInterrupt(long long time, string pgName);
    void diskDiscardInterrupt(long long time);
    void handleDiskDiscardInterrupt(long long time);
    void handleCreationInterrupttion(long long time, string
currentProcess);
    void handleDiskInterruption(long long time, string currentProcess);
    void handleTimerInterruption(long long time, string
currentProcess);

    bool handleInterrupts(long long time, string currentProcess);

    void processTermination(long long time, string currentProcess);

    void pgFault(long long time, string faultingProcess, string
faultingPage);

    bool hasInterrupts(long long time);
};
```

I did not choose array or linked list to store the interrupts' information. If we use array or linked list to store interrupts, we should choose to store in order or not. If we store the interrupts in order, the insertion operation will cost O(n) time. If we store the information in random order, when we want to search the closest interrupt, it will cost

O(n) time. So I choose STL map to store information. The inserting operation will be O(log n) time. The infoTable will store the quantum left and the process's next timer interrupt time.

1. `void creationInterruption(long long, string)`

    This function will create a creation interrupt and insert it into the interruption table. The other three are same like this.

2. `void handleDiskDiscardInterruption(long long)`

    This function will handle the disk discard interrupt when there comes such an interruption.

3. `void handleTimerInterruption(long long, string)`

    This function handles the timer interruption. Timer interruption happens when the process's time slice expires. So the most important point is that we should make sure that each process should only have one timer interrupts. When the process will switch to others. It should update its timer interrupts in the table.

4. `void pgFault(long long, string, string)`

    In fact, this function does not mean the common page fault. It means the scheduler's action when the running process meets a page fault. The scheduler should update the running process's timer interruption and then let the memory to swap a page. Finally it calls CPU to context switching.

Note: When the process terminates, the operating system will not perform context switching. But, the OS still need to pick up another process and reload its information. For that, I think we still need some time to switch to the new process.

2.4 Memory

The memory's main task is to keep track of which page is in the main memory. At each cycle, the cpu will ask the the memory to fetch a page. The memory will response to the CPU whether the page is in the main memory. And then the CPU will ask the scheduler to handle the page fault.

The main component of the memory is the mmu. If the memory pool is full and there comes a new page, the memory should decide which page we will kick out. In this project, I implement three algorithm.

There is one main function in this component.

1. `bool swapPage(long long, string processName, string pageName)`

This function handles the page fault. It will ask the scheduler to create a disk interruption and disk discard interruption. The function will also insert the new coming page into the page table and mark it busy, which means the page has not come yet.

There are three kind of memory I implement in this project. I will choose one to explain.

```
class FIFOMemory: MemoryBase {
      //page Name ------> page available time.
      map<string, long long> pageTime;
      //page Name ------> whether the page is marked.
      map<string, bool> pageMarked;
      //FIFO pages.
      deque<string> pages;
      int pgNum;
public:
      bool memoryFull();
      bool accessPage(long long time, string pgName);
      bool insertPage(long long time, string pgName);
      bool kickOut(long long time);
      void markBusy(string pgName);
      void unmarkBusy(string pgName);
};
```

The memory's in this model is stored in a queue. Three a two maps in this model. One of which stores the time when the page is available. The other one stores whether the page can be used now. The page which is marked means that this page have been in the blocked page queue, but it has not been carried from the disk to the main memory. When the disk interruption of this page occurs, the page will be unmarked.

`bool kickOut(long long time)`

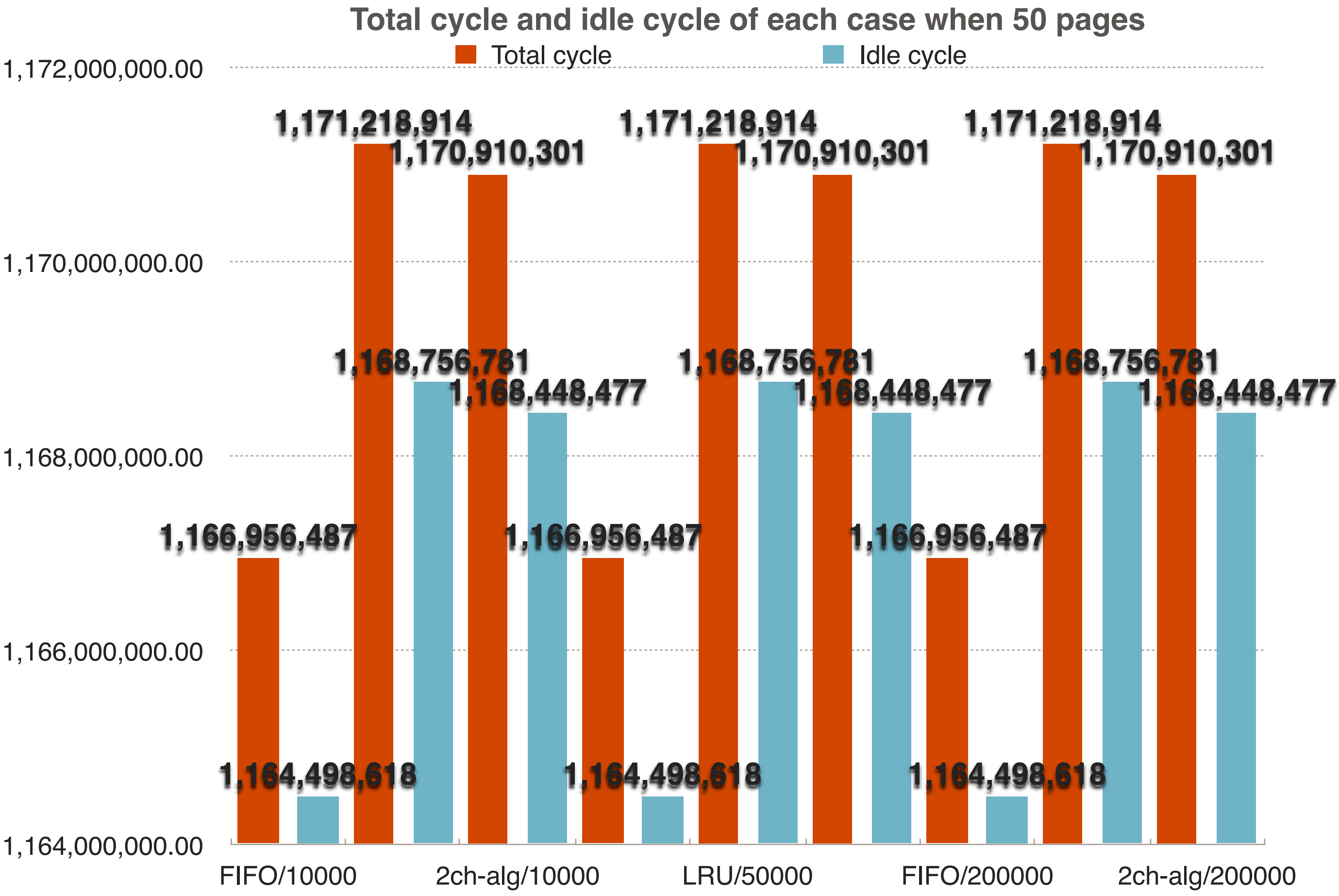
This is the main function of this component. It focus on which page to kick out. It scan the pages queue from begin to end, find the page which is unmarked and whose available time is smaller than current time.

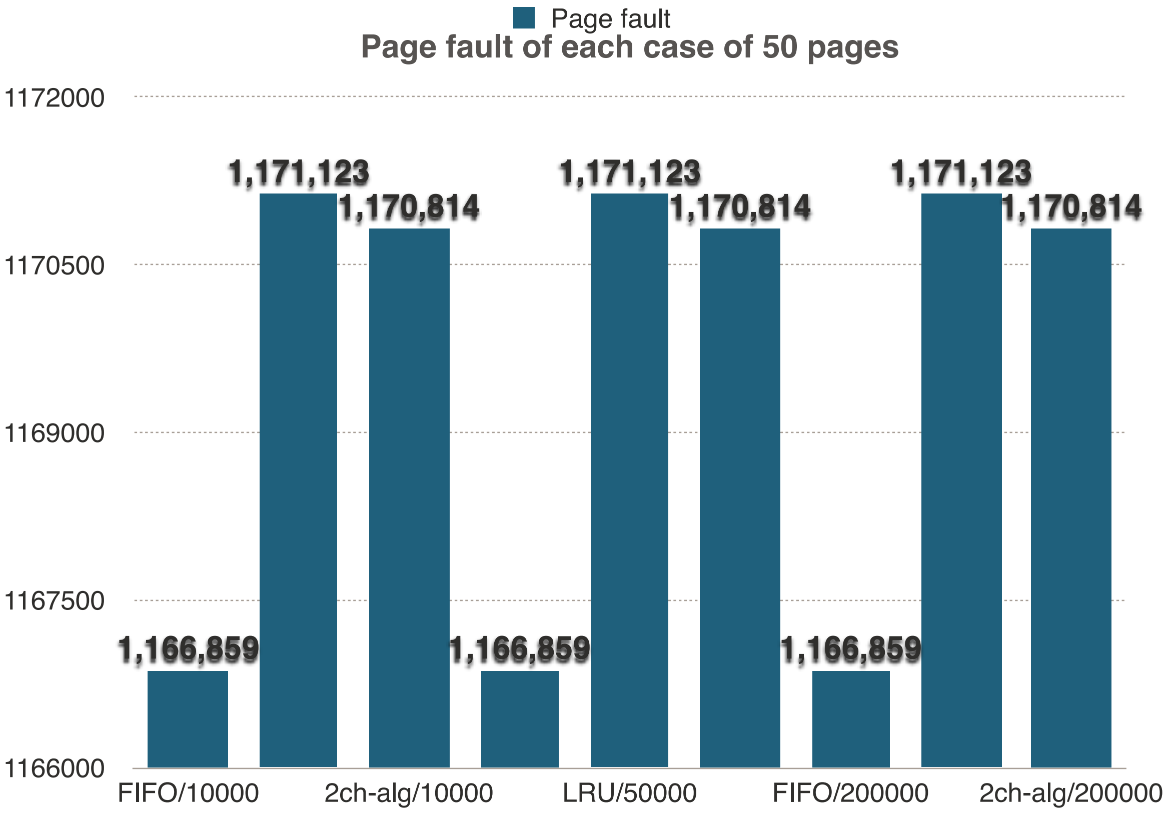
3. Result

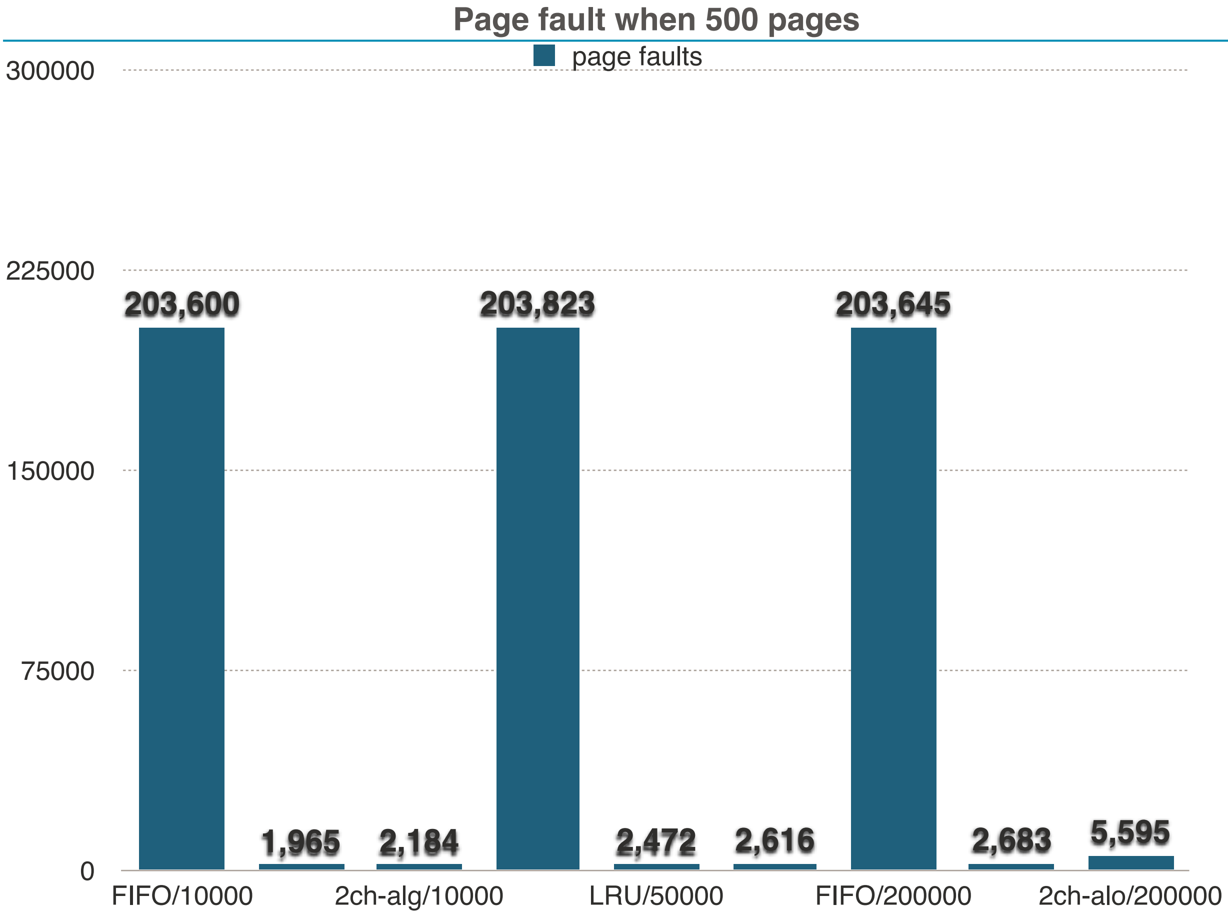
Here are the results of the project.

Small case:

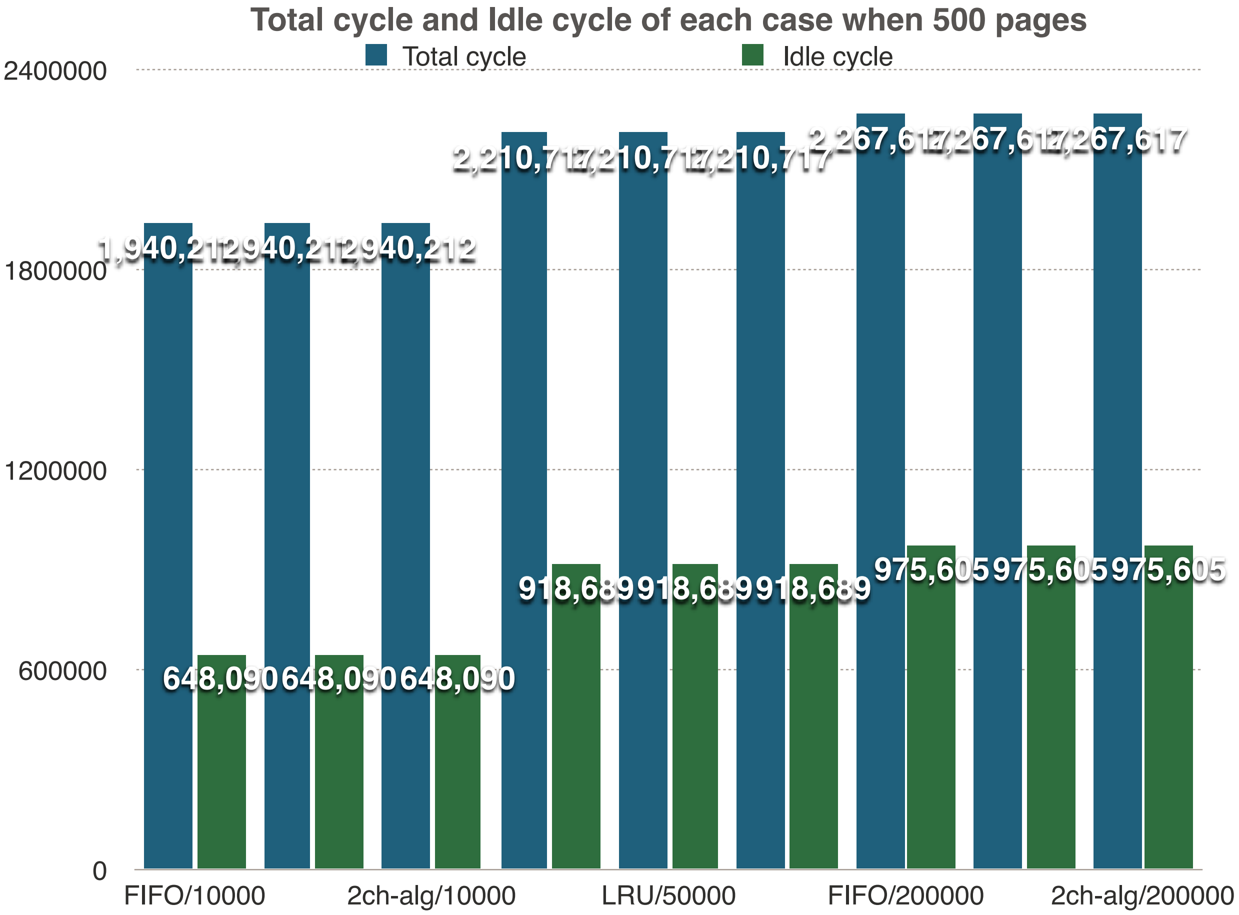
50 Pages :


Total cycle and idle cycle of each case when 50 pages
Total cycle
Idle cycle
1,172,000,000.00
1,170,000,000.00
1,168,000,000.00
1,166,000,000.00
1,164,000,000.00
1,171,218,914
1,170,910,301
1,171,218,914
1,170,910,301
1,171,218,914
1,170,910,301
1,168,756,781
1,168,448,477
1,168,756,781
1,168,448,477
1,168,756,781
1,168,448,477
1,166,956,487
1,166,956,487
1,166,956,487
1,164,498,618
1,164,498,618
1,164,498,618
FIFO/10000
2ch-alg/10000
LRU/50000
FIFO/200000
2ch-alg/200000

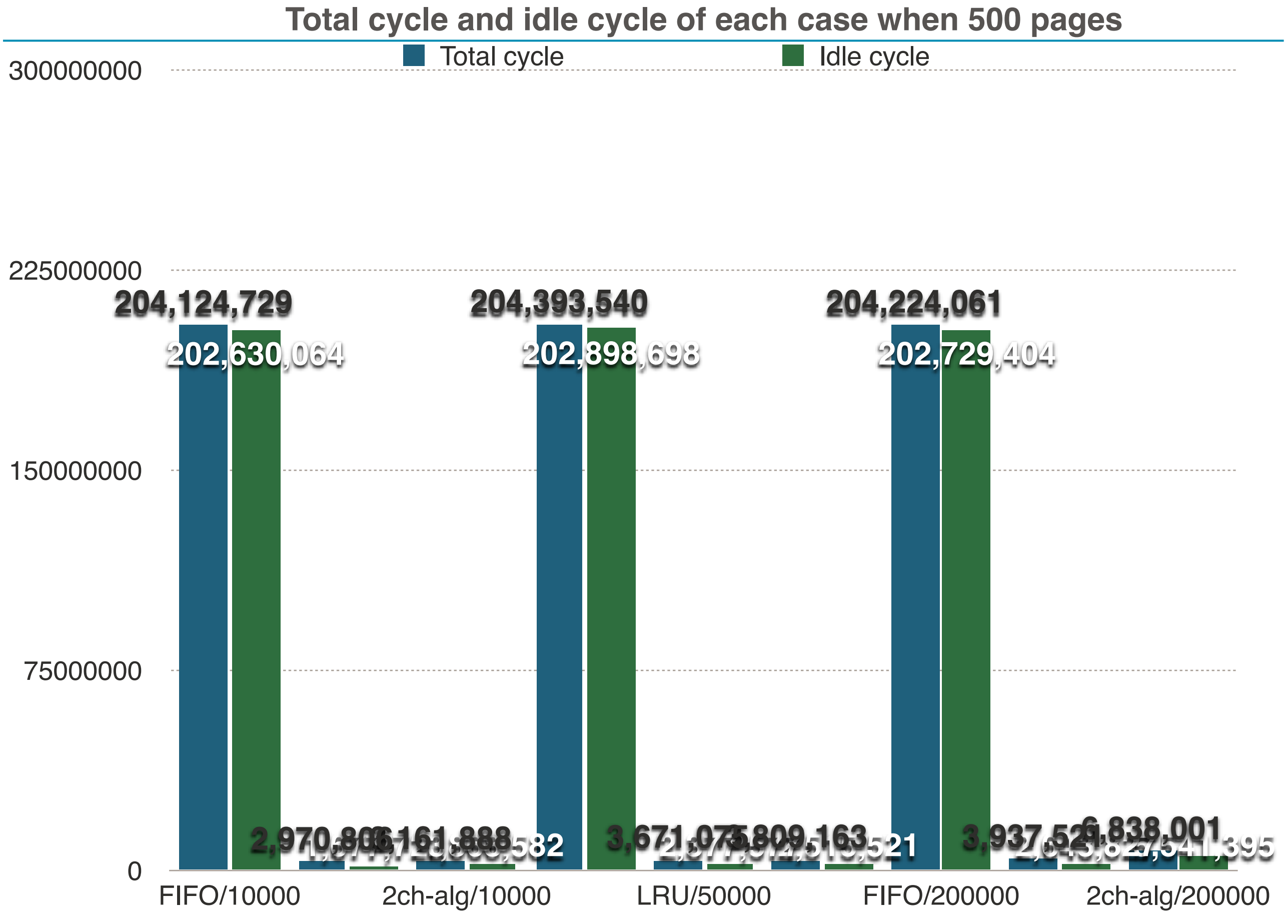

Page fault
Page fault of each case of 50 pages
1172000
1170500
1169000
1167500
1166000
1,171,123
1,170,814
1,171,123
1,170,814
1,171,123
1,170,814
1,166,859
1,166,859
1,166,859
FIFO/10000
2ch-alg/10000
LRU/50000
FIFO/200000
2ch-alg/200000

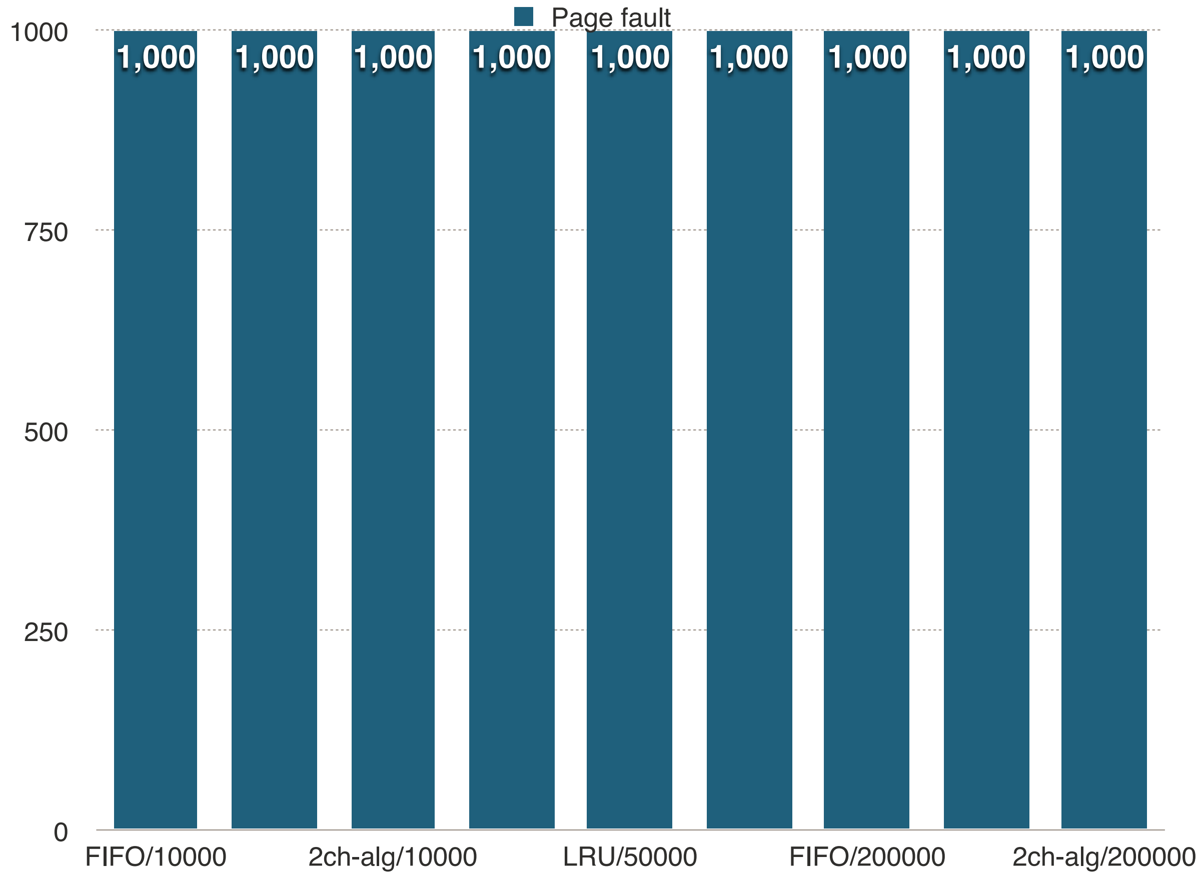


Page fault when 500 pages



Total cycle and Idle cycle of each case when 500 pages


Total cycle and idle cycle of each case when 500 pages
Total cycle
Idle cycle
300000000
225000000
150000000
75000000
0
204,124,729
202,630,064
204,393,540
202,898,698
204,224,061
202,729,404
2,970,80
6,161,888
3,671,07
5,809,163
3,937,52
6,838,001
FIFO/10000
2ch-alg/10000
LRU/50000
FIFO/200000
2ch-alg/200000


Page fault
1000
750
500
250
0
1,000
1,000
1,000
1,000
1,000
1,000
1,000
1,000
1,000
FIFO/10000
2ch-alg/10000
LRU/50000
FIFO/200000
2ch-alg/200000

5000 pages

It is easy to find that the result depends on the total pages in the main memory. There are three cases:

1. 50 pages. The pages in main memory is too small. The page fault happens very frequently. So the difference between the three algorithms is trivial. Total cycle, idle cycle and page fault are too large.
2. 500 pages. There are distinct difference between the three algorithms. The FIFO behaves very poor, for FIFO doesn't think over the which page should be replaced. It just pop a page from the queue. So FIFO memory may remove the page which will be used next, leading to extra page fault. So FIFO's page fault is much bigger than the other two. The LRU's page fault is a little smaller than 2nd chance algorithm.
3. 5000 pages. The difference between each case is trival. Because the memory size is so large that when a page is carried from the disk, it may never be kicked out. The memory size of each process is 100, there are 10 process. So the total amount of the pages is 1000, which is smaller than 5000. So the result is similar. In this case, we can find out how quantum affects the result. When the quantum becomes bigger, the cycles becomes bigger. It is because that long quantum leading to more context switching.

Next, I will analysis the page fault of the big data at each case. For each case's scheduler is same, so the total cycle and idle cycle depends on the

The result of big data when 50 pages.

	FIFO	LRU	2ch-alg
10000	11585374	11561445	11589895
50000	11585374	11561445	11589895
200000	11585374	11561445	11589895

The result of big data when 500 pages.

	FIFO	LRU	2ch-alg
10000	7136204	6150846	6347069
50000	7137953	6300222	6502302
200000	7138161	6301575	6503829

The result of big data when 5000 pages

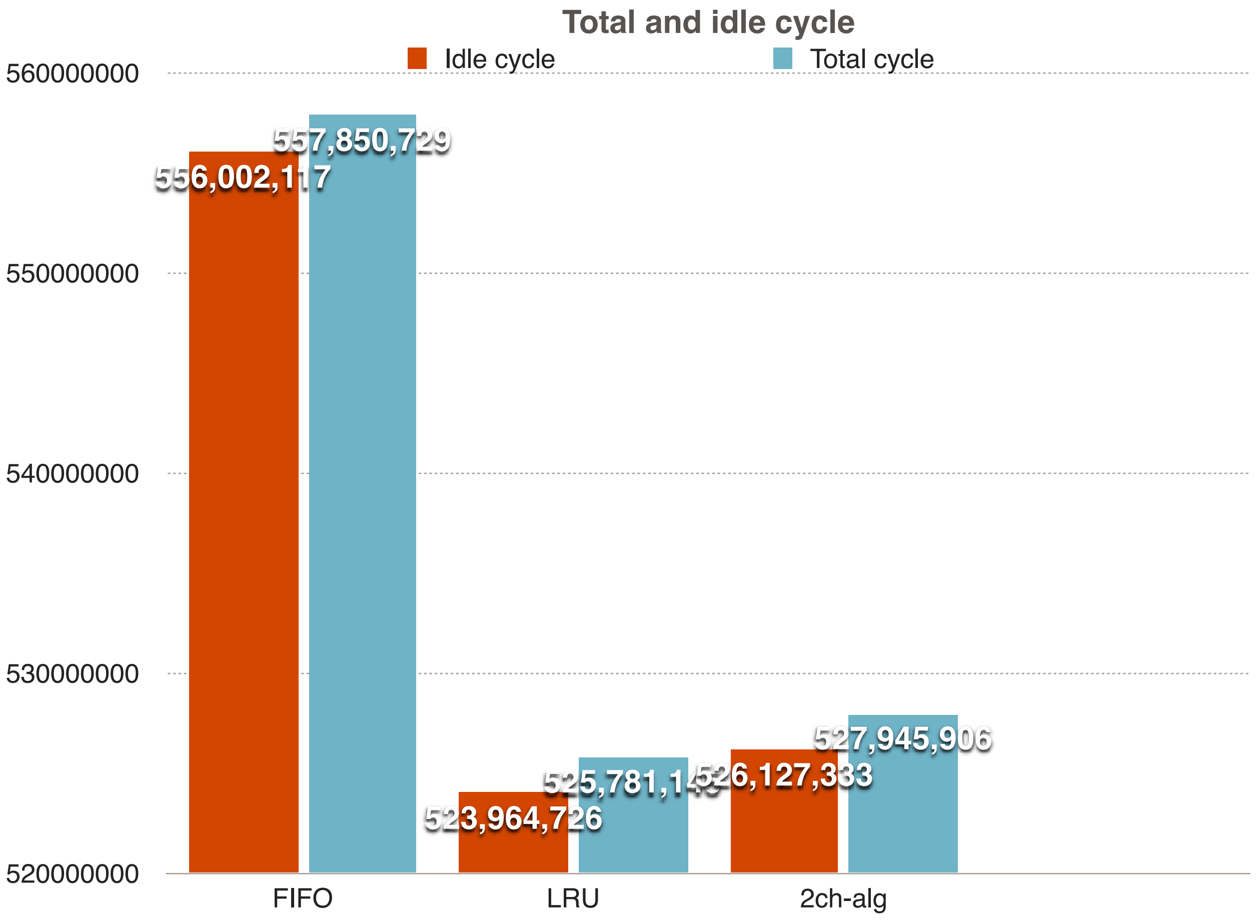
	FIFO	LRU	2ch-alg
10000	9973	9973	9973
50000	9973	9973	9973
2000000	9973	9975	9973

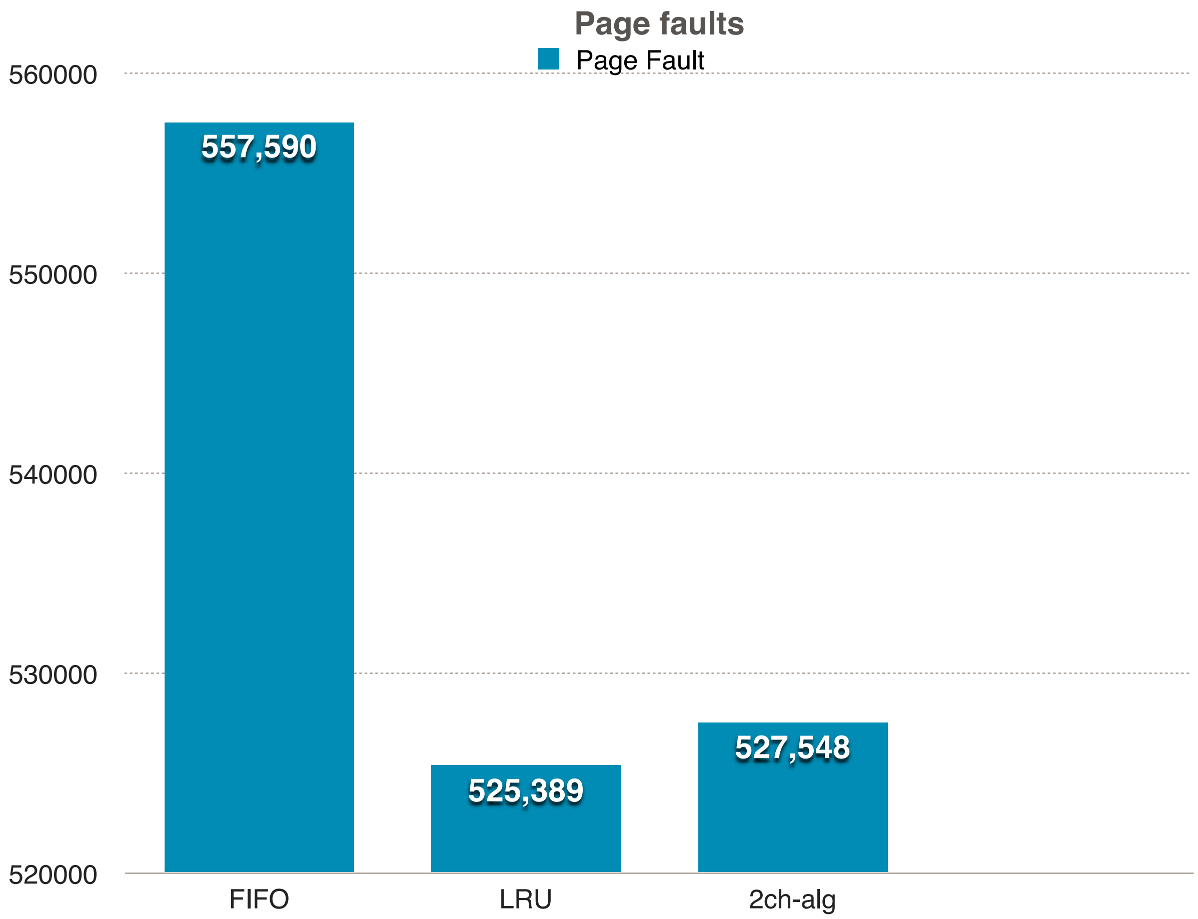
Note: The detailed result has been packed in the directory.
We can easily find out that the result of big data is similar to the result of small data. The main difference is that the difference of page fault of the three algorithm when page size is 500 is not distinct. I think it is because that the total memory of the data is larger. It turns into 100 * 100 = 10000 from 100 * 10 = 1000. Page faults will happen more frequently than small data. So the gap between FIFO and the two algorithm is not so distinct. But it's still obvious :)

4. Experiments

1. page Replacement algorithm.

Environment: 250 pages, 20000 quantum

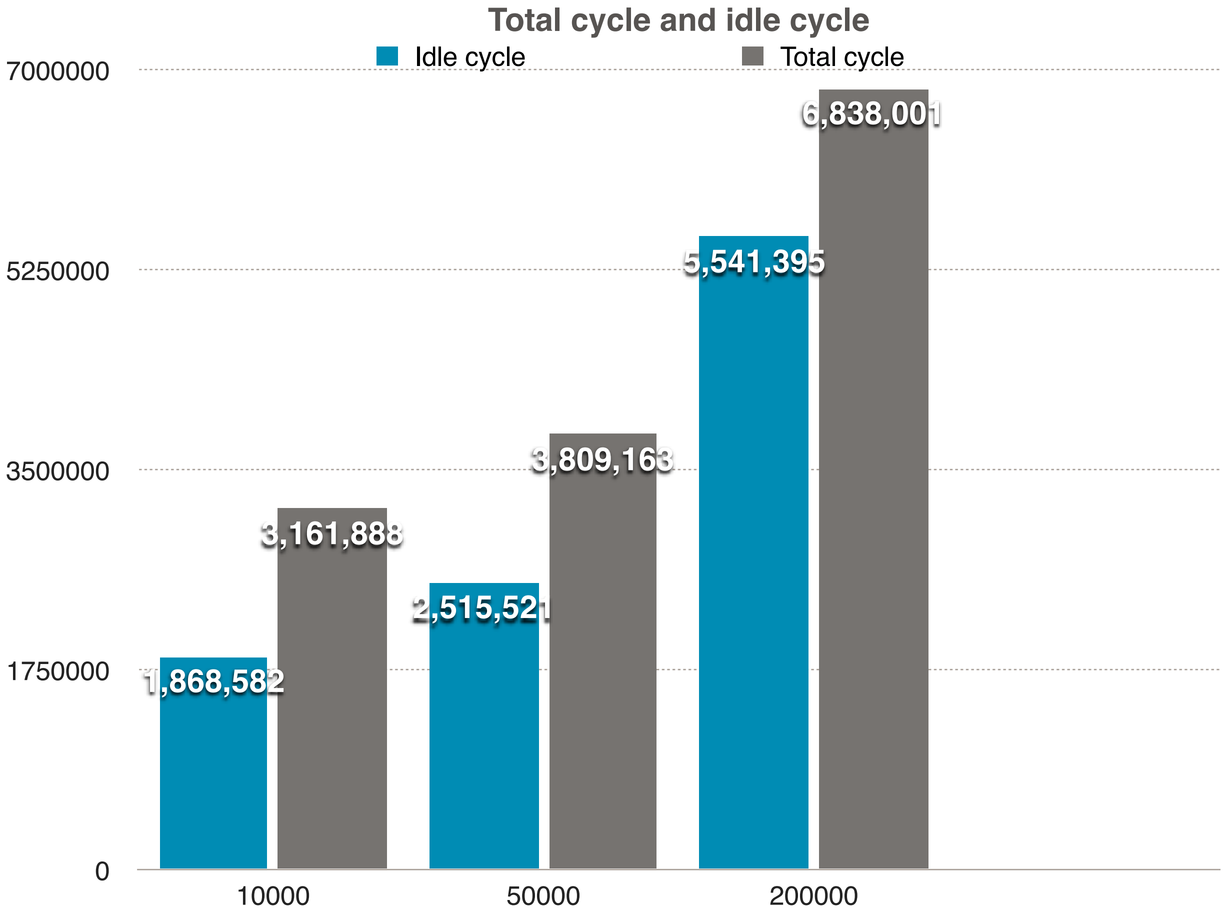


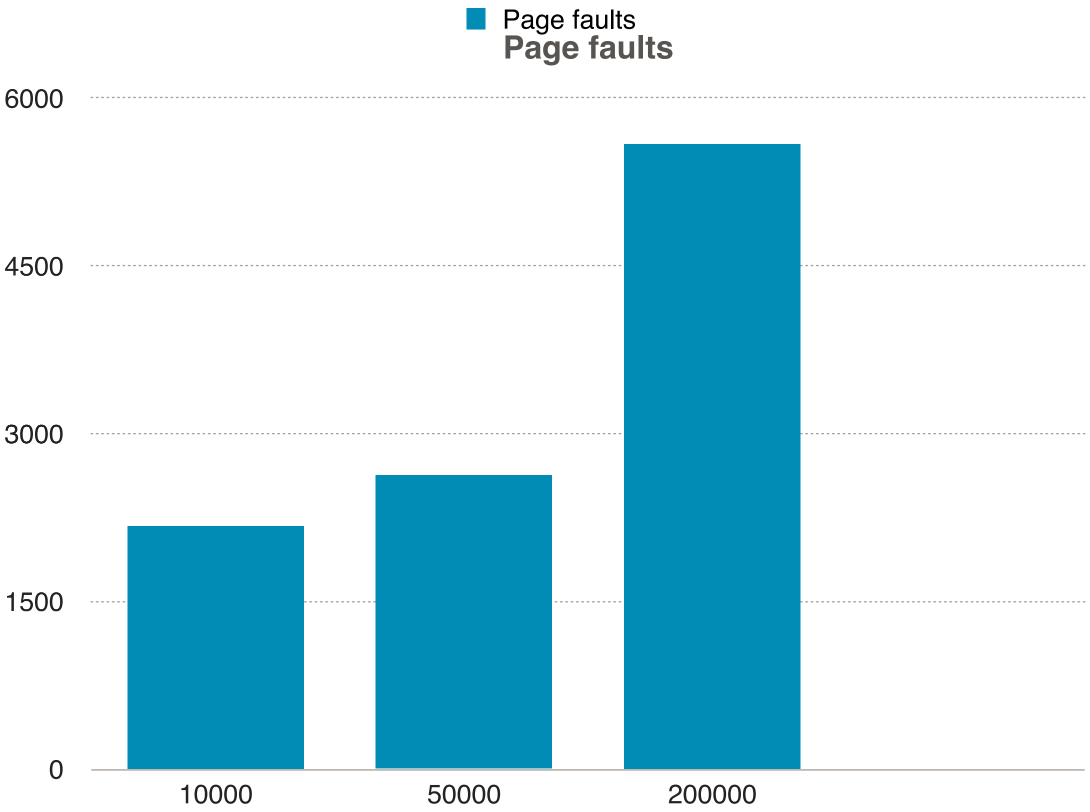


Conclusion: FIFO algorithm is worse than the other two. LRU algorithm is a little better than 2nd chance algorithm. But!! LRU algorithm is harder to be implemented than the 2nd chance algorithm. And it is more complex than 2nd chance algorithm.

2. Time quantum experiment.

Environment: 500 pages, second chance algorithm. 10000, 50000, 200000 quantums.

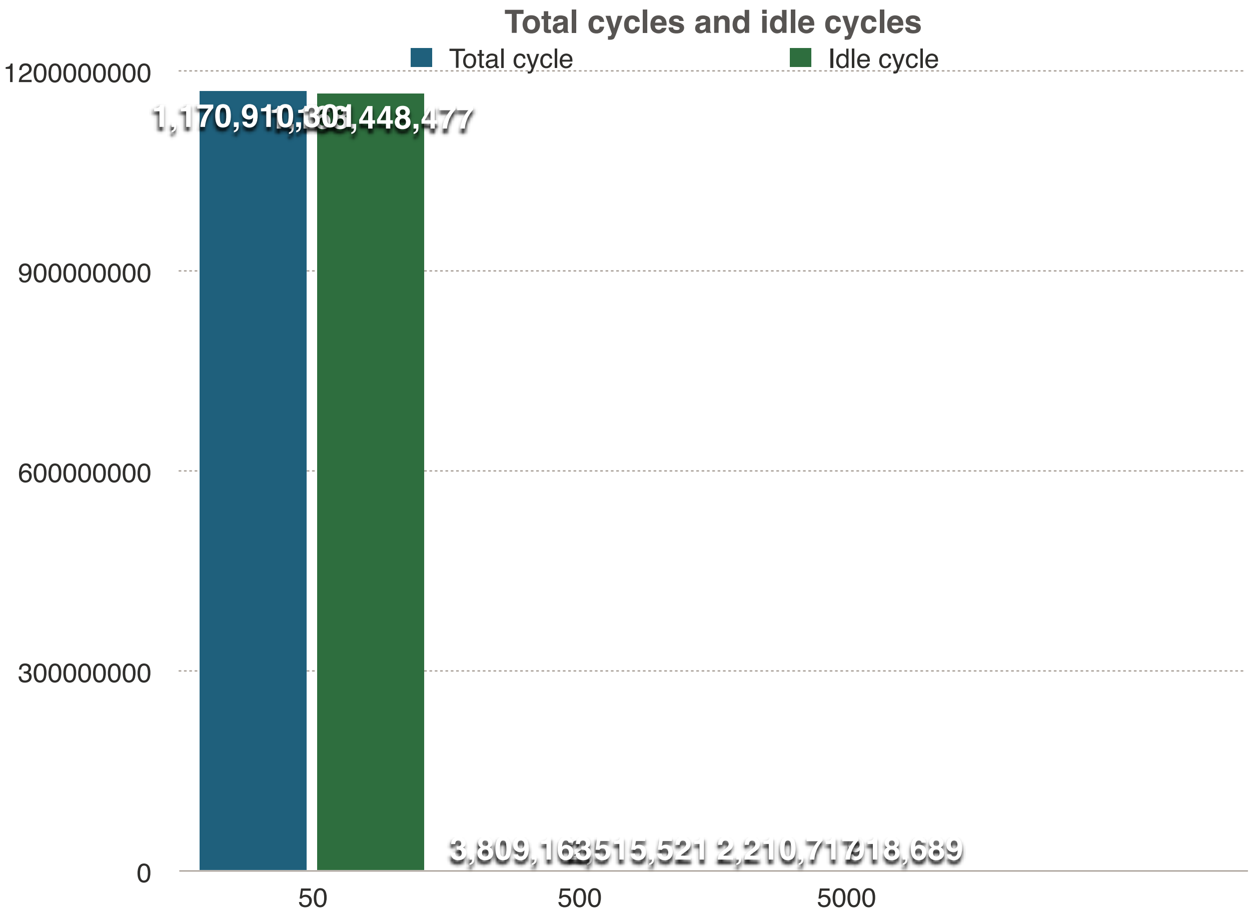

Total cycle and idle cycle
Idle cycle
Total cycle
7000000
5250000
3500000
1750000
0
1,868,582
3,161,888
2,515,521
3,809,163
5,541,395
6,838,001
10000
50000
200000


Page faults
Page faults
6000
4500
3000
1500
0
10000
50000
200000

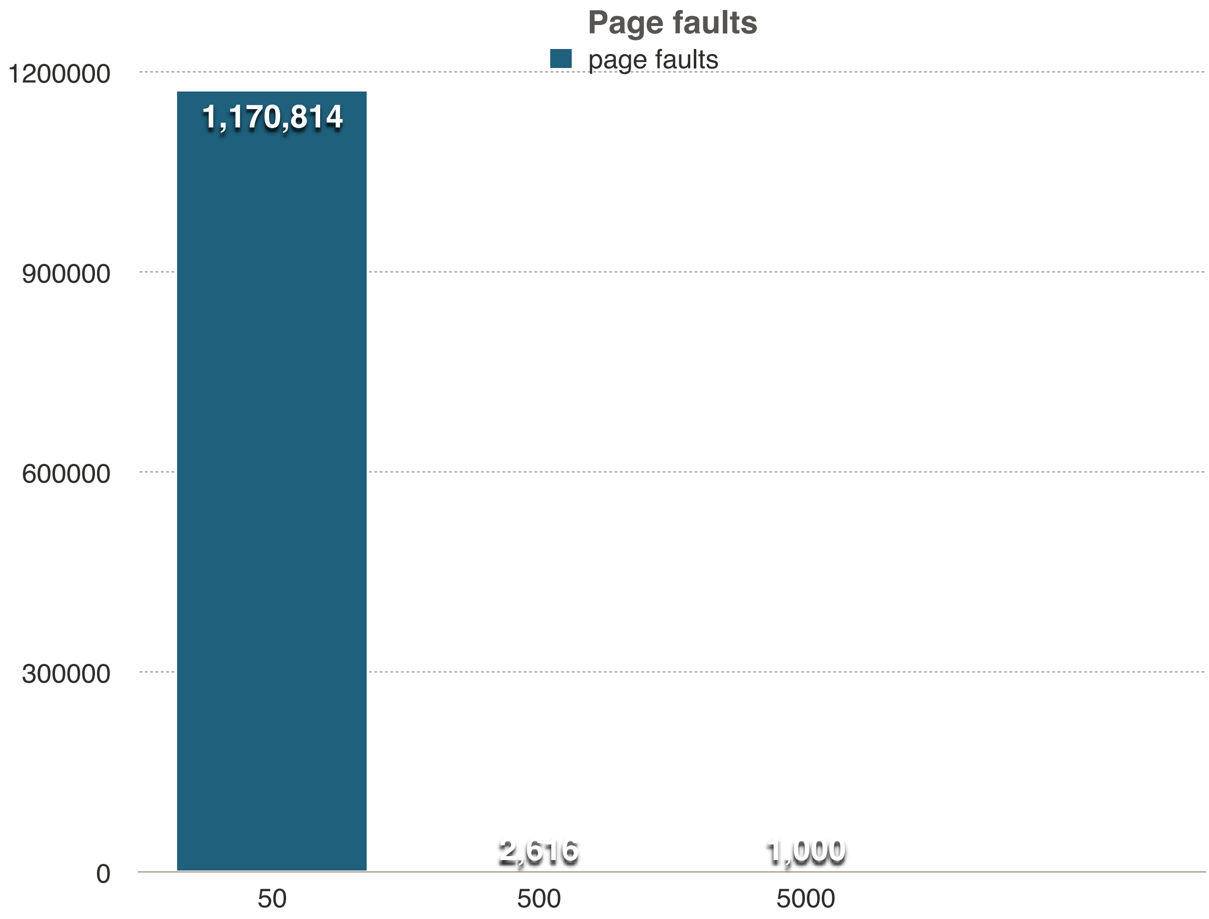
Conclusion: When the time raises, the total cycle, idle cycle and page faults become larger. Because when the quantum becomes larger, the process has more time to be running, leading to more page faults, so that the total cycle and the idle cycle will increase. But !!! total cycle minus idle cycle must be a constant value.

3. Memory size experiment

Environment: 50000 quantum, 2nd chance algorithm



Note: 500 pages: 3809163 total cycle of 2515521 idle cycle; 5000 pages:2210717 total cycles of 918689 cycles.



Conclusion: Memory size has great influence on the result. It is easy to to explain. The smaller memory will lead to more page fault, leading to more cycles.

5. Some notes

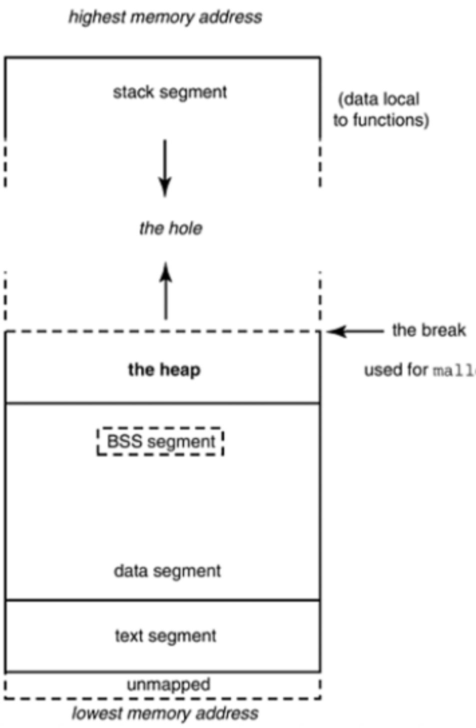
1. Do new process go at the head of the ready list or the end?
In this project, I create an extra structure to store process. A process can stay in the ready queue, blocked queue, or the *fault* queue. The fault queue stores the process which has receive the disk interrupt and know that its faulting page has been in the memory now. When the scheduler performs a context switching, it will firstly find the first process in the fault queue. If the fault queue is empty, then it refer to the ready queue. It means that the process which was blocked will get high priority to run.

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?
Unless the memory size is too small, the faulting page is guaranteed to be stored in the memory.

3. Does new process or process returning from a page fault get priority?
The new coming process gets the low priority. It will be pushed onto the end of the ready queue. The process returning from a page fault gets a high priority. It will be pushed onto

the end of the fault queue, which has a high priority than ready queue.

4. When a process terminates, will the scheduler perform a context switching?
No!!! But I think the CPU still need to reload the information of the process which will be switched to. So it still take some time. But, it does not matter, for the amount of process is trivial when compared with the total cycle.



6. Another algorithm I design

In this project, I design my own page replacement algorithm. I call it least replacement.

First of all, I introduce you the memory model of Linux. I found that the stack's address decreases from upper to bottom. So, when a process is running, the memory of higher address is the stack frame far away from now, it may be used late. So we can replace it.

My replacement algorithm is designed very easy. When we need to kick a page out of the memory, we find the page with highest address and kick it out. I use the set to store the pages, for it is implemented by red black tree.

The behavior of my algorithm is not good on the base of the given test data. But I believe, on the base of real linux, the algorithm will work well.

Here are some results of the algorithm:

Page faults

	10000	50000	200000
50	1217613	1217613	1217613
500	551898	551974	551980
5000	1000	1000	1000

7. Some thoughts and suggestions

In this project, I wrote a lot of code and construct a simple simulator project. But I think that only way to master the operating system course is to do more projects. We cannot master it unless we wrote a real operating system.

I find some information about how the operating system course organize in the excellent college around the world. MIT creating a simple OS named JOS. The main purpose of their OS course is to complete a real OS based on the skelton of JOS. Stanford has pintos. UCB invents Nachos. Tsinghua invented ucore. Peking university uses JOS for the students to complete. So I hope that SJTU's OS course can pay more attention to the practice. We can also write a simple OS, not a simulator.