Architectural Document

What is included in my system

1. In general, there are three main concerns in the phase 2 of my project. They are Memory Management, Disk Access, and Disk Management.

a) Memory Management

In this part, I focus on the handling of page fault. The whole process starts with **Fault Handler**. When a process attempts to access (by either read or write) a position in its virtual memory space while that page is not in main memory, an interrupt event is generated from hardware and captured by **Fault Handler**.

The **Fault Handler** firstly identifies which event is it about according to the interrupt number. If it is a page fault interrupt, hand it to **Page Fault Handler** to deal with it, otherwise deal with it accordingly.

When dealing with the page fault, system either simply allocates an empty frame to whom is requesting or runs sub-routine **Swap Page** to write an old page back to disk and make it available.

In the end, we renew the **Frame Table**, **Page Load List**, and the target page table entry.

Page Load List is a linked list which maintains the order of pages loaded into main memory. It is designed for the page replacing algorithm, in particular, the **Second Chance Algorithm**.

b) Disk Access

For disk access it is all about two system calls: **DISK_READ** and **DISK_WRITE**. The principle of doing disk access is that: If there is no disk request on specific disk that is handling or waiting to be handled, execute memory write sub-routine at once. Otherwise, put the disk request into the corresponding disk queue.

When it comes to disk requests that waiting to be handled, the system maintains a **Disk Queue** for every single disk to buffer them.

Sub-routines **Disk_R** and **Disk_W** are the routines which actually execute the disk read/write. They are invoked either by interrupt handler or by system calls.

Both DISK_READ and DISK_WRITE are implemented as pended-IO, so the system calls will not return until the disk request actually finished.

c) Disk Management

Disk management is about keeping track of data stored in disk and managing available space. I adopt extremely simple disk storage policy in my project when the content in the swapping-out page needs to be written back into disk. Since we have 12 disks with 1600 sectors each. I simply use process id as the index of the disk and virtual page number as the index of the sector. Since there are no more than 1024 pages in one process and the size of sector is the same as page entry. System can run normally as long as we don't create more than 12 processes, or use disk write/read independent of memory write/read.

Here is a list of the specific feathers that my system can do:

1. Call DISK_READ and DISK_WRITE.
2. Handle the page fault.
3. Use Second Chance Algorithm to find the victim.
4. Save the modified page into the corresponding disk before it is swapped out.
5. Load the page back to main memory when it is touched after it has been swapped out.
6. Guarantee multi-tasking.

High level design

1. Memory management design

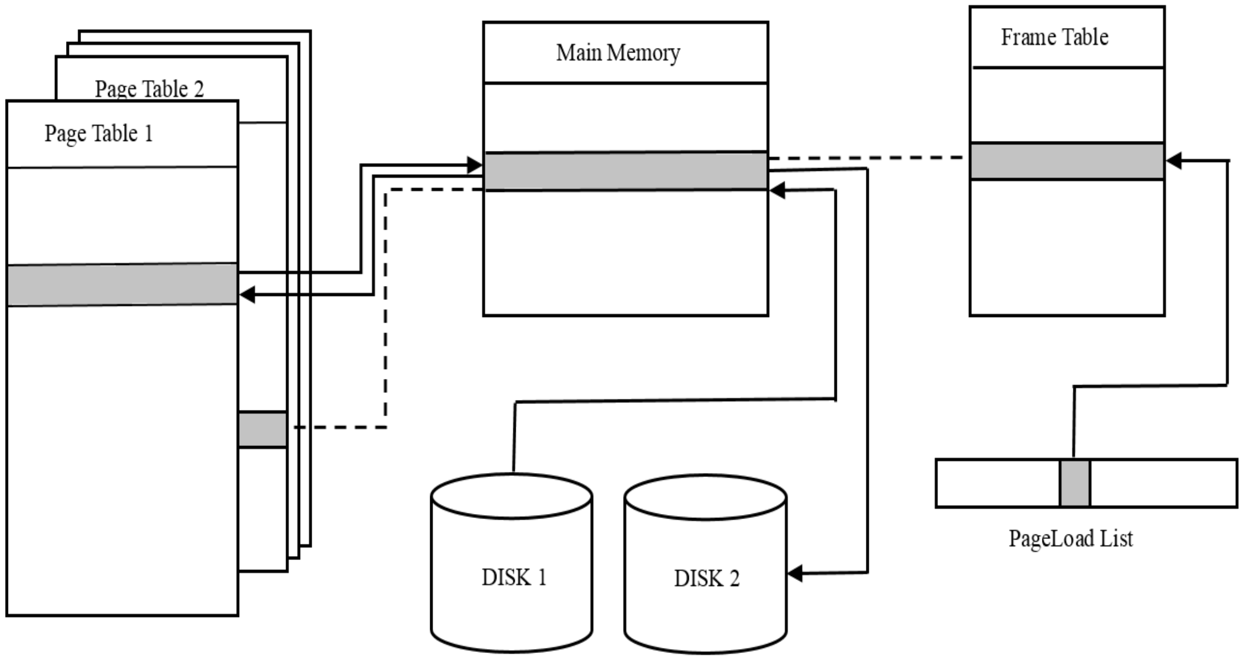


Figure 1 Memory management hierarchy

2. Disk management design

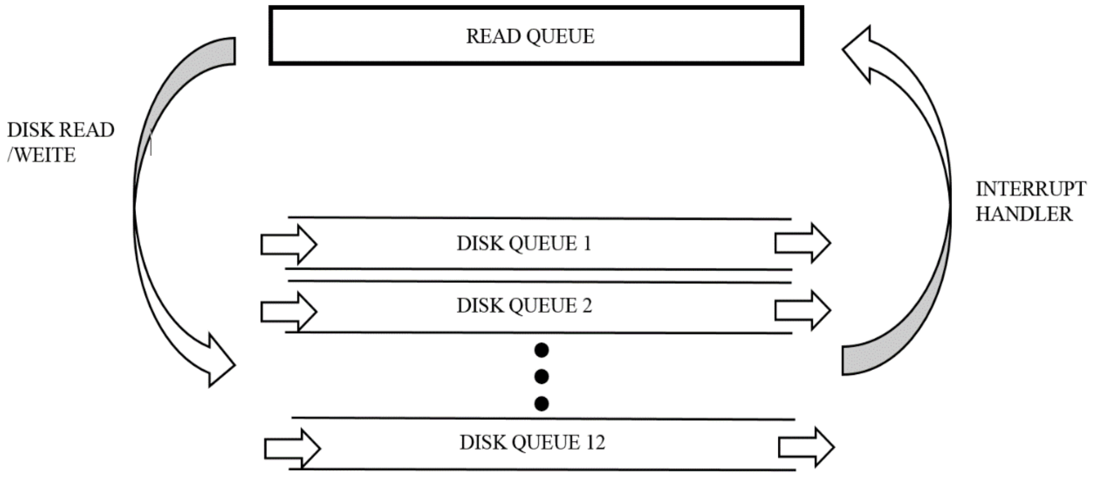


Figure 2 Disk queue

3. Page fault handle flows

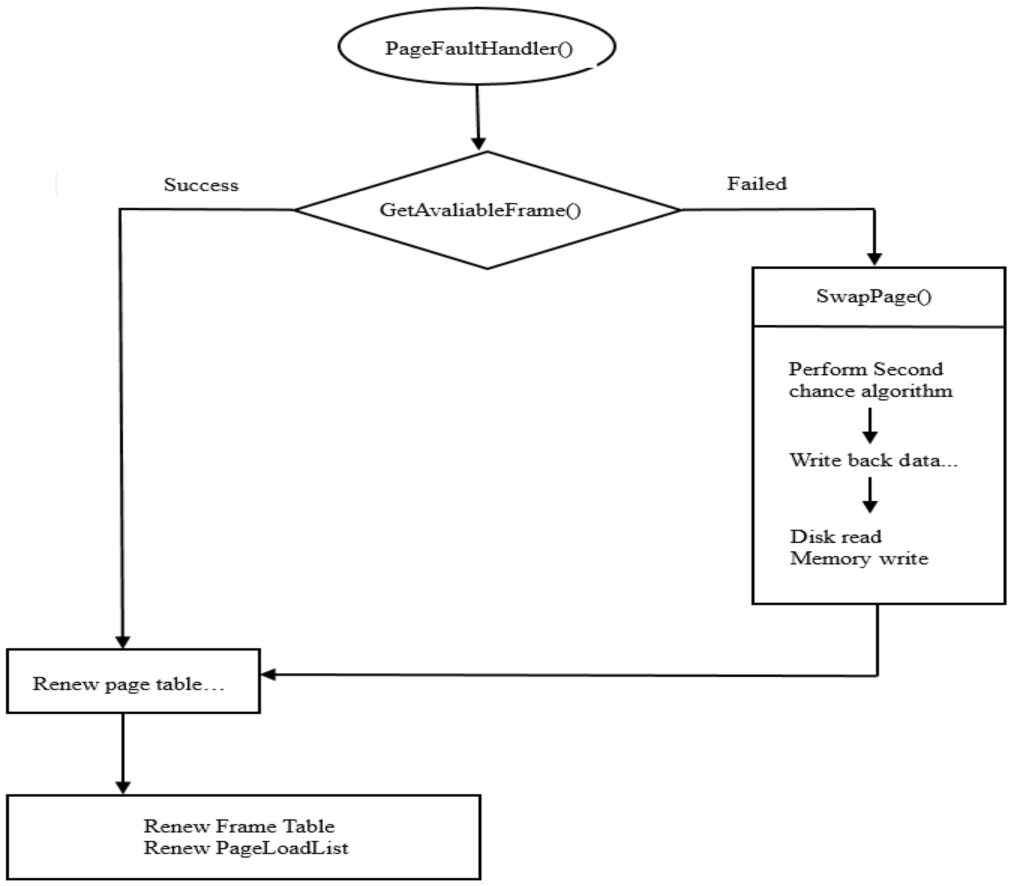


Figure 3 How page fault are handled

Justification of the High Level Design

1. Disk Interrupt

   When a disk interrupt occurs, system first takes out the first request from disk queue, which means that this request has been executed. Then add the corresponding PCB to the ready queue, making it ready to run. Then read the next disk request, if it is null, end interrupt handling, otherwise start dealing the request.

2. Disk Queue

   There is a disk queue for every single disk (see Figure 1), so each time we simply take the first one out of corresponding queue. If different disk requests are in a same queue, we have to scan the queue, which violates the principle of queue.

3. Page replacing policy

   When we want to find an alternative frame for a new page to get into, we scan the frame table (see Frame table entry in next section for more details) rather than scan the virtual page table. The reason is that if we only scan the virtual page table, we can only swap out page which belongs to the process itself. Suppose process 1 has used 63 frames out of total 64 frames in main memory, and process 2 just uses the rest 1 frame. When process 2 needs a new frame, it has no choice but swap out the only one frame that it has. It is not reasonable because the proportion that the frames contributing to different processes will remain the same even if process 2 is much more active than process 1. Scanning the whole frame table is fair enough, while it may cause more page faults in a certain situation. The Second Chance Algorithm is described in text book, so I decide not to explain it here.

4. Page writes back to disk.

   When we need to save a page on disk, we use DiskWrite to write the data in main memory to disk. After DiskWrite returned, system use DiskRead to read back the data at where we have put it above. In this way we can check if the data are stored in disk successfully and accurately.

5. Lock

   There are two kinds of lock in the system.

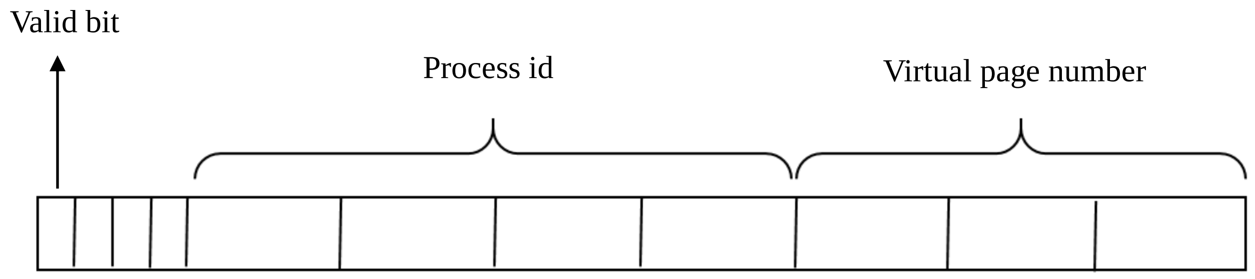
   MEMORY_INTERLOCK_DISK_QUEUE is used to guarantee the mutual exclusion of manipulating of the disk queue.

   MEMORY_INTERLOCK_DISK is used to guarantee that the executing of a sequence of instructions that perform disk read/write is atomic.

Unique and additional features

1. Frame table entry

   My frame table is an INT32 type array with a length of 64(same as the frame number). The index of array stands for the corresponding frame. From each array entry I stored the process id (pid) and virtual page number. I used the first 4 bits as mask bits, the middle 16 bits refer to pid, and the last 12 bits refer to the virtual page number (vpn).



   We use bit operation to retrieve the pid and vpn from frame table entry, which is much faster than using data structure.

2. Page load list

   It is a link list to maintain the order that page comes into the main memory. When we perform second chance algorithm, we take the head of the link and check if the P bit of this page is 1, if it is, then we clear it and move this page into the rear of the list, like a new comer. We keep doing this till we find a page which P bit is 0. Then we take it out.

3. Usage of Reserved bit in virtual page table entry

   I set reserved bit in page table entry as a mark which indicates the page has been swapped out and stored on disk. So when a page fault occurs on a page of which the reserved bit is 1, we need to read the data from disk to the main memory.

4. Disk read/write design controller - MyInterruptStatus

   There is a global variable in my system called MyInterruptStatus, which indicates if the interrupt handler is running in the background. If it is not, MyInterruptStatus = -1. Otherwise it equals to the disk id. It is used as a lock mechanism. One and only one condition must be met when disk read/write will be executed immediately is that disk queue is empty and MyInterruptStatus does not equal to the disk it acts on. The logic is like:

```
if(dq[disk_id]->head == NULL && MyInterruptStatus != disk_id+4){
    Disk_R(disk_id, sector, data,0);
}
```

```
addToDiskQueue(disk_id, sector, data, D_READ);
```

Note:

About the hardware statistic of the test 2d and test 2f.

Since I process id begins at 1, so the in test2d, process 2 and 3 act on disk 2, process 4 and 5 act on disk 3, process 6 act on disk 4.

For test 2f. Because I have identical policy in data write back (describe in “Page write back to disk” sector), the read number would be different than usual (128 as far as I know).

Since I use system(“pause”) instructions somewhere, if there are any runtime errors. Please include the original head files instead of mine.

Wrote by Yunhe Tang