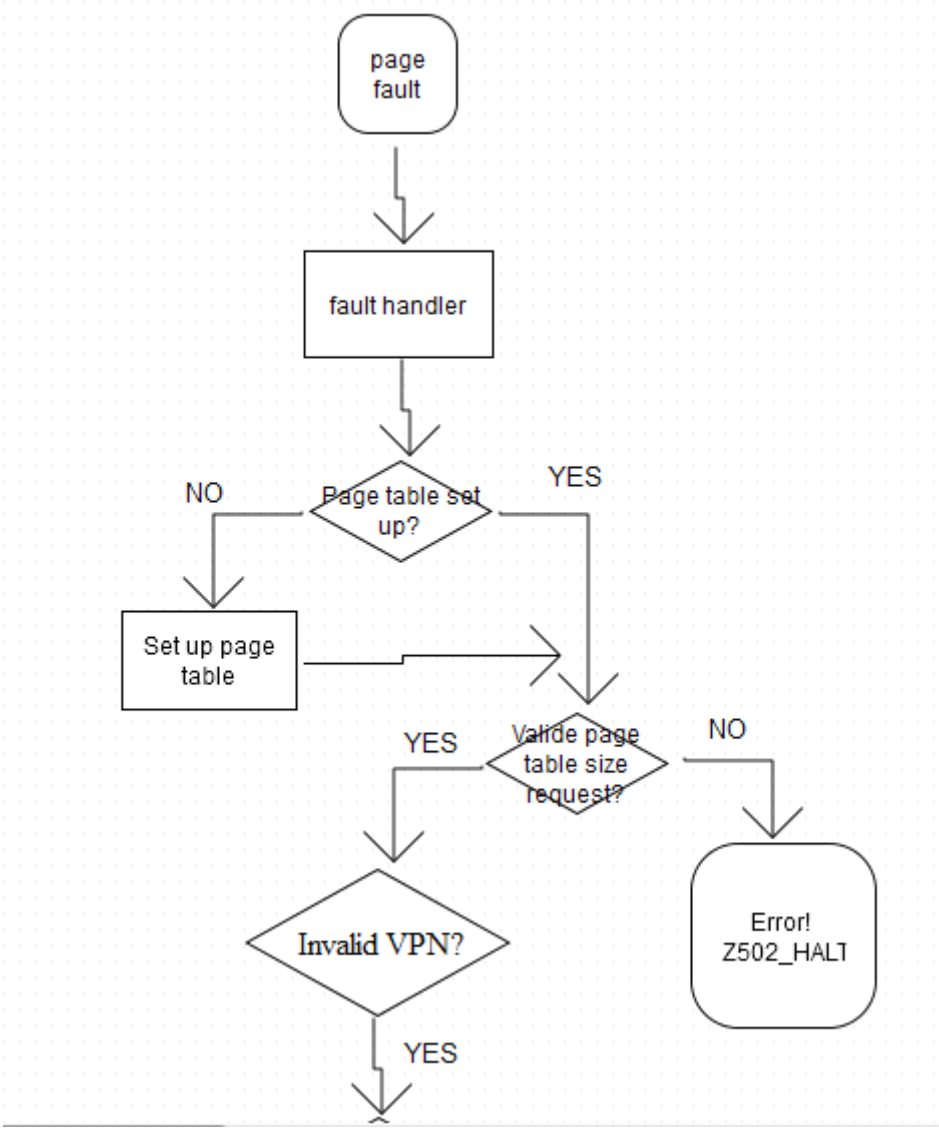
Architectural and policy overview

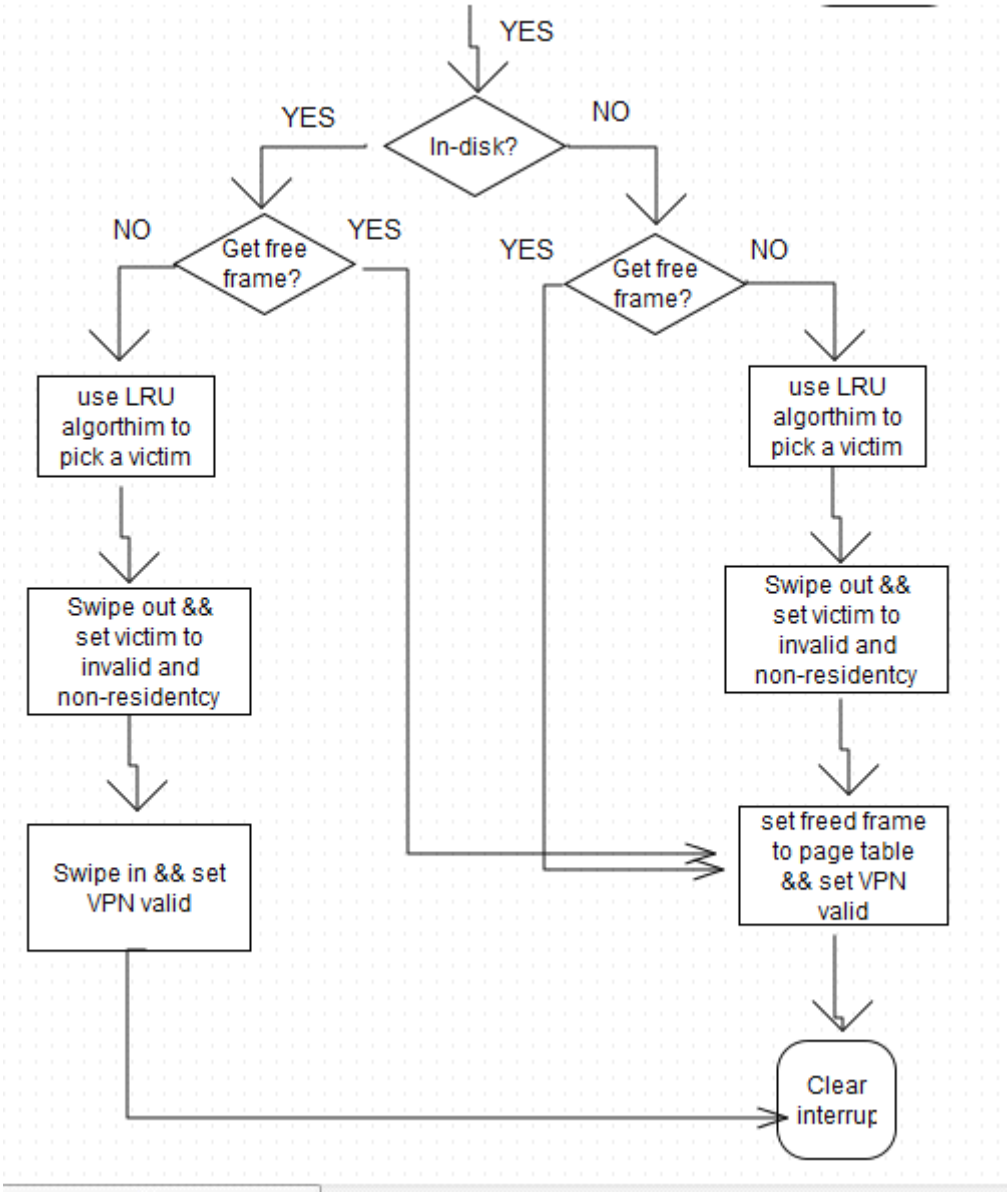
The system deals with memory reads and writes, disk calls and scheduling for multiple disk requests. "os_disk_write" and "os_disk_read" routine was written to respond to disk calls from user model. Page Fetch Algorithms and Page Replacement Algorithms was implemented in Fault_handler. Scheduling of multiple disk requests from different process was achieved in use of interrupt_handler and Waiting Queue. Advanced algorithms such as LRU was also implemented.

In sum, test2a, 2b, 2c, 2d, 2e, 2f was successfully solved. Test2h tries to implement 5 process of test2f, but failed.

High level design

Algorithm for test2a, 2b, 2e and 2f. (Picture 1.)





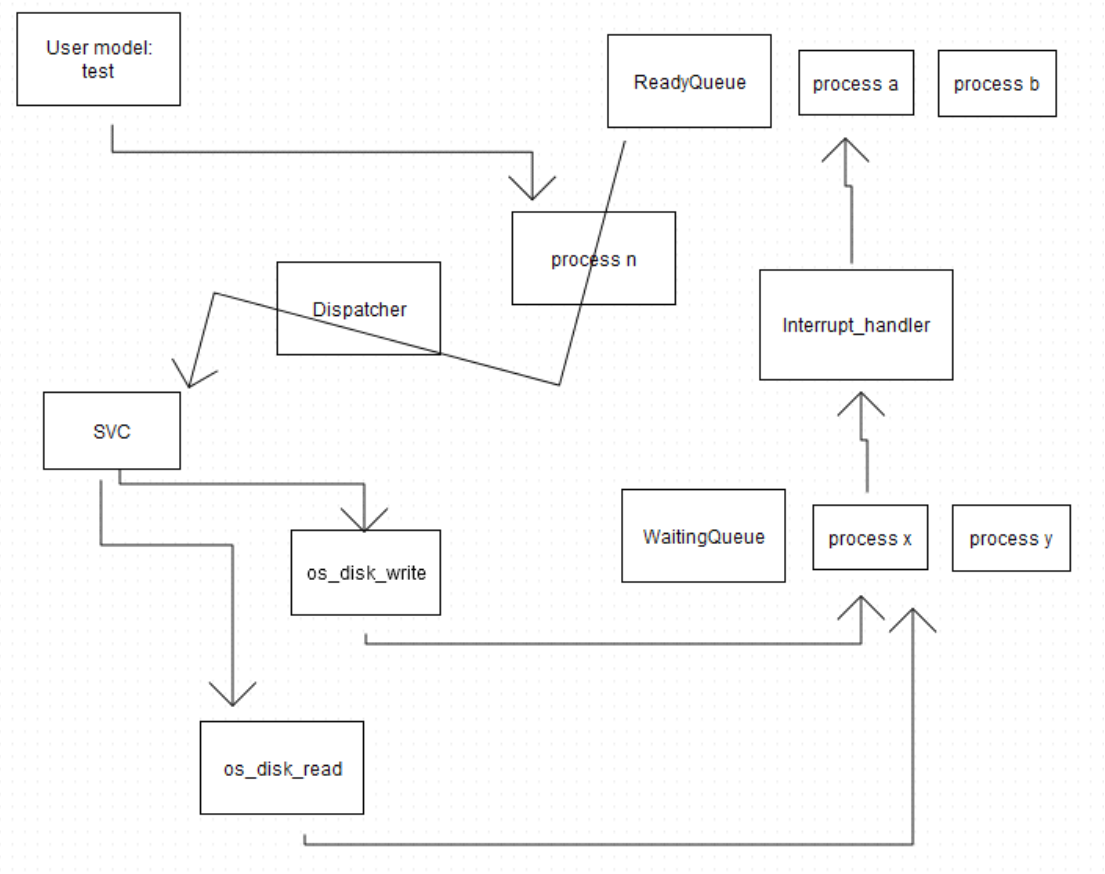
Picture 1.

When Swipe data in/out, we use the routine designed for test2c and test2d, which will be discussed later.

Picture 1 has omitted another situation that is when offset is larger 12. At this special case, as the virtual memory is treated continuous, we need also pick a free frame and set the next VPN to be valid. The algorithm for that is the same as Picture 1.

Algorithm for test2c and test2d.

As the real algorithm is too hard to express, the picture blow is a shallow version. Detailed discussion was under the picture.



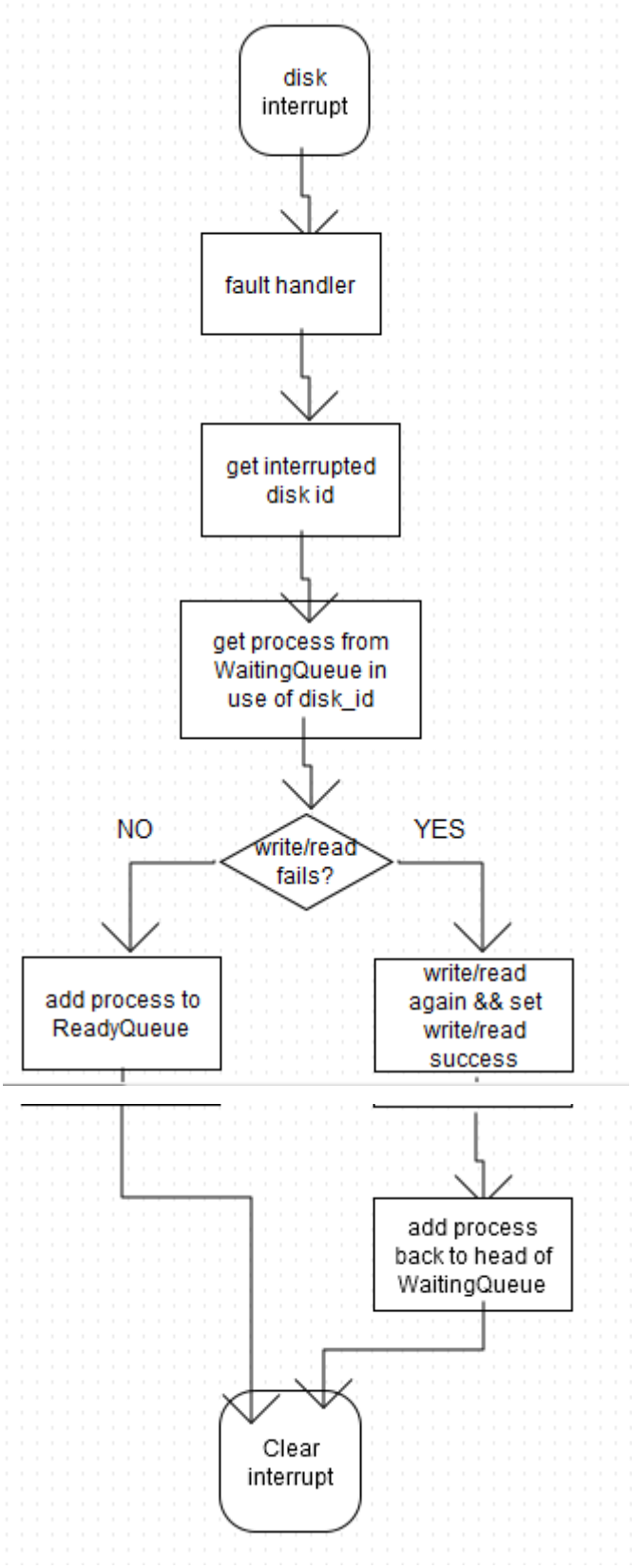
Picture 2.

Os_disk_write and os_disk_read routine was written to respond to system calls. Os_disk_write and os_disk_read are almost the same. Here only os_disk_write was discussed. When we go to this routine, it first check if certain disk is in use. If so, the requested process was put into the Waiting Queue and related buffer information is stored in the PCB, also we keep the status that the process failed this write. Otherwise, the process sets related information and start to write, then the process was put into Waiting Queue, but the status identifies that the process has written successfully.

As each time the disk successfully writes or reads, a disk interrupt would occurred and we'll go into the interrupt handler. In the interrupt hander, we fetch the related process, and check if the process has failed a write/read before. If not, we directly put the process to Ready Queue. Otherwise, we need to write/read again, then we add the process back to the head of the Waiting Queue. The next time an disk interrupt occurred because of this operation, we will then put the process to the Ready Queue.

Some thing need to mention is that in the os_switch_context_complete routine, we should copy the read information to the register. So that in the user model, the program could get the read information.

The algorithm for the interrupt handler is as the picture 3 showed below.


disk interrupt
fault handler
get interrupted disk id
get process from WaitingQueue in use of disk_id
write/read fails?
NO
YES
add process to ReadyQueue
write/read again && set write/read success
add process back to head of WaitingQueue
Clear interrupt

Picture 3.

Justification of High Level Design

For test2a and test2b, as the number of writes and reads is small so that we always have free frames to use. In the coding, I set up an free fame list in os_init, and each time use the routine RemoveFromFreeFrameList() to try to get a free frame. If there's a free frame, the routine returns the number of the frame, otherwise returns -1. Here the structure linked list is better than my first version which is a array, so that I don't need to go and search all frames each time.

As each time a disk write or read has finished, a interrupt will occur, and cause an interrupt. So, the scheduling is somehow alike sending/receive message in project phase 1. No matter a process writes/reads successfully or not, the process was put into the Waiting Queue. If success, the next interrupt of that disk will take the process out to Ready Queue. If not, the next interrupt will let the process to write/read again, and the interrupt of that operation will take the process out to Ready Queue. Hence, I can guarantee that each an process goes back to the ready queue, it has written/read successfully. Because, the next time the process will go to the next step, we must guarantee the previous operation has finished.

LRU algorithm, page fetch algorithm and page replacement algorithm are the same as in Jerry's power point, which is very straightforward. "When we over-allocate memory, we need to push out something already in memory. Over-allocation may occur when programs need to fault in more pages than there are physical frames to handle. If no physical frame is free, find one not currently being touched and free it." -- Jerry

Additional Features

LRU algorithm was implemented in two versions. The first version, which is not contained in the code, simply go through the whole page table, and find the valid VPN (we have 64 ones in this program). Each time we find a valid VPN, we check the if the reference bit is 1. If so, then it shows the VPN is used frequently somehow, we simply set to 0. If not, the VPN is used rarely, so we pick this VPN as the victim.

As I tries to implement the schedule for 5 test2f process (unfortunately failed, which will be discussed later). I changed the code of LRU to version 2. That is, I use a data structure, which is a array that contains 64 pointer. These pointer points to the valid VPN in the page tables of different process. So each time I directly go through the array. The other part is similar to version 1.

Data Analysis:

For test2e, I have 257 Faults. For test2f, I have 853 Faults. This is very a small number because in 2e only 128 writes occurred, but in 2f, the it writes loop_count (is 400) times, and reads all data NUMBER_OF_ITERATIONS(is 3) times. This verifies that my LRU algorithm works very well. Also, I compared with one of my classmates, who has 1335 Faults in test2f, which is larger than mine.

Discussion on failure of test2h, which tries to implement 5 test2f process.

Actually, one problem was met when I implemented my test2e and test2f. The hardware_interrupt thread will get stuck on the HardwareLock, and after debugging, the reason I can detect is that I use z502_switch_context in the dispatcher. And I realized that for test2e and test2f, only one process is used. So I use a tricky method, that is before the program use z502_switch_context, it first checks whether the next process is the same as the current process, if so, the program won't call this routine. This could avoid the stuck problem in 2e and 2f. But in 2h, as 5 process was called, the program must use z502_switch_context, then it is stuck, and I cannot fix that.

Jiefeng He
Dec 6, 2012