Architectural Document

Memory Management

One of the exercise of the second part of project is memory management . To execute any program, it must be brought in Physical **memory (RAM)**. Program can be larger than physical memory which is generally divided into small frames. Since physical memory is limited, program is generally breaks into parts and only some part is loaded into memory. This type of memory is called **virtual memory**. In this type memory management virtual address space is divided into pages. When program needs to access other part of code and is brought in memory when request it is made. This is called **demand paging.** If page is present into physical memory, its valid bit is set 1 otherwise 0. When demand page is not present in physical memory, then **page fault** is occurred. And control transfers to operating system. Operating system selects a page from memory by page replacement algorithm. The resident page in frame is transferred to secondary storage and set its valid bit to 0. This called **swapping**. After swapping, demand page is loaded into that frame and its valid bit set to 1 so that hardware can use it.

To support this management, I have added following structures in my design.

1. **Page Table**

Page table is just array of 8 bytes array with size MAX\_NUMBER\_OF\_PAGES (1024). At the very first time, when page fault is occurred for process, the size of MAX\_NUMBER\_OF\_PAGES (1024) of memory is allocated to page table. Each process has its own process table which is later saved in context. Each virtual page contains its frame if allocated and valid/referenced/modified bit. When page is in memory, its valid bit is set to 1.Referenced is set when page is only read by harware and when page is written its modified as well as referenced bit is set to 1.

1. **Shadow Page Table**

This table is array of structures with size of MAX\_NUMBER\_OF\_PAGES (1024). This table contains two fields. One is disk id and other one is disk location. The reason is to remember in which disk at which location data is present for virtual page. When physical memory is full, previous page content should have to transfer to secondary memory such as disk (swap area). To remember this disk id and sector for future operation, this table is used. In short, this table is helpful while swapping page from memory to swap area.

1. **Frame Table**

This table stores entry of each frame. Frame table contains process id to determine which process is using particular physical memory frame, logical page to determine whose content is present in frame. Third field of frame table is page Pointer which exactly points to logical page location in page table who currently owns particular frame. This field is useful in multi-processing environment to make logical page invalid when its frame is assigned to other fault page. In addition, to get location on disk to write it back if entry is present in process’s page table.

**Page Replacement Algorithms:**

When page fault occurs, operating system is responsible to replace previously resident page with new demand page. There is lot of algorithms for page replacement. I have implemented FIFO and Approximate Second chance LRU page algorithm.

FIFO algorithm selects victim frame in descending order and allocate to virtual page when memory frames are free. When memory gets full, this algorithm iterates circular and selects first used frame as victim frame and allocate this frame to virtual page to swap. This algorithm is not efficient. If some pages are referenced again and again, algorithm does not take this condition into consideration. So page faults are increased.

Approximate Second chance LRU page replacement algorithm selects victim frame in descending order when memory frames are available. When there is no frame available to load virtual page data, this algorithm is used. It checks all frame’s **referenced bit**. If any frame found unreferenced, it selects it as a victim to replace resident page with demand page and forwards pointer to next frame. When algorithm finds any frame referenced, it sets referenced bit to 0 and continues to search next victim. Initially it finds all entries referenced i.e. 1 so make each entry unreferenced i.e. 0. After last frame counter again set to 1st frame and finds frame unreferenced. Therefore, select it as a victim to replace page. Suppose in future any resident page is referenced, hardware set it its referenced bit to 1. In this way, if page is referenced so many times, it never gets replaced because it always gets second chance.

Test2f and test2g is good example of this Second chance LRU page replacement algorithm. There are fewer faults for test2f and test2g by using this algorithm. For test2f, I got 1084 page faults whereas for FIFO algorithm, there are 1109 page faults. This less number shows usefulness of this Second chance LRU page replacement algorithm. It not only reduces page faults but also reduces disk read and writes operations. There is no difference for test2d because it referenced one page only one time. You can see this to comment GetFrame() in MemoryManagement() function to check faults for FIFO algorithm**.**

Following function are used for above algorithms.

1. **int GetFIFOFrame()**

This routine is implementation of above FIFO page replacement algorithm. Selects victim is descending order and returns it to caller.

1. **int GetFrame()**

This routine selects frames in descending order when memory is available otherwise get frame from GetLRUFrame().

1. **int GetLRUFrame()**

This routine is implementation of Approximate Second chance LRU page replacement algorithm. It goes through all frames in circular manner. When it finds a frame is referenced, it set it’s referenced bit to 0 and continue search until finds first unreferenced frame. Select it as victim and returns it.

Disk Scheduling

Disk is secondary storage. Its content is generally non-volatile. User has system calls to read and write from disk. User provides disk id, location at which to read/write and buffer to read and write on disk. I have implemented **FCFS scheduling** to manage disk read and write operation. By this algorithm, first pending request is selected from queue and start disk for that request to complete its task as soon as disk is free.. To handle user system calls DISK\_READ and DISK\_WRITE, operating system has following routines:

1. **void DiskRead(long disk\_id,long sector, char \*data)**

This function does some error checking on disk id and on sector and directly returns from the function if invalid parameters are given. If each parameter is ok to proceed then add itself on Disk Queue and checks disk state (busy/free). If disk is free, then call StartDisk() routine to start disk to read. After starting disk, gives up CPU and call Dispatcher to schedule next process until its read operation is done. Otherwise only gives up CPU and call Dispatcher

1. **void DiskWrite(long disk\_id,long sector, char \*data)**

This function does some error checking on disk id and on sector and directly returns from the function if invalid parameters are given. Add current process PCB on Disk Queue and checcks disk availability to write on disk. If disk is free, then call StartDisk() routine to start disk to write, gives up CPU and call Dispatcher to schedule next process until its read operation is done. Otherwise only gives up CPU and call Dispatcher.

(NOTE: This above two DiskWrite() and DiskRead() routines stores disk id, sector and data pointer in requested process’s PCB in disk\_info structure to remember if disk is busy at the time of request specially for interrupt handler to start disk after finishing previous request.)

1. **void StartDisk(long disk\_id,long sector, char \*data, int readwrite)**

This function also checks disk id and sector and returns if parameter is wrongly given. Function sets each parameter for disk to start if disk is free otherwise return without doing anything.

*Interrupt handler and fault handler*

**Interrupt Handler**

This routine runs a separate thread in program after any I/O interrupt has finished with its task. There are two interrupts in project. One is TIMER\_INTERRUPT and other is DISK\_INTERRUPT. TIMER\_INTERRUPT is occurred after sleep time expires for process and call Make\_Ready\_To\_Run() to make process ready (for more detail of this interrupt, please refer project part1 document).

Interrupt occurs with device id after disk has finished with read/write operation. It calls HandleDiskInterrupt() to handle this type of interrupts.

1. **void HandleDiskInterrupt(INT32 device\_id)**

This routine first determines which disk has finished its task depending upon device id. After this,it checks Disk Queues of that disk and removes head - process’s PCB from Disk Queue and adds it into Ready Queue according to priority. After, it checks Disk Queue for next pending request for disk. If any finds, call StartDisk() to start disk to complete its task. Parameters of pending request for disk to start are found on requested process’s PCB in disk\_info structure.

**Fault Handler**

This routine is caused if PRIVILEGED\_INSTRUCTION, CPU\_ERROR or INVALID\_PHYSICAL\_MEMORY is found in simulator. This routine exits program if these faults occur. While paging, if page is not in memory, hardware passes control to fault handler as device id is equal to INVALID\_MEMORY. This fault is called page fault. This routine calls MemoryManagement() to handle such type of fault.

1. **void MemoryManagement(INT32 logical\_page)**

This routine first checks range of logical page. If not page number is less than 0 or greater than page table length, it exits program by showing error. Function gets physical frame from GetFrame() routine to load page into memory. If frame table has no previous record for particular frame, it simply assigns frame to virtual page, make virtual page valid in memory and add entry in frame table to remember which virtual page it is mapped to it and returns to hardware. Otherwise, search previous entry of virtual page in shadow table of current process to check if it is present on disk (swap area). If present, reads contents of virtual page from its location on disk (swap area). If physical memory frame was previously assigned to any virtual page, reads contents of resident virtual page. If resident virtual page entry is present on shadow table, then write on disk to its swap location otherwise write on new disk location (swap area location) and adds entry of it in its shadow table. Routine is also responsible to make previously physical memory resident page invalid by setting valid bit to zero. After this process routine write content of demand page on frame location to make sure that previously contents safely transferred to it location.

Physical memory frames related functions:

1. **void MakeFrameTableEntry(int frame\_num,int logical\_pg\_num,long pid)**

Routine stores entry in frame table to remember mapping between virtual pages and frames. Add entry of process who currently owns a frame as well as stores pointer to page in its page table.

1. **BOOL IsPhyMemoryFull()**

Returns TRUE if physical memory is full otherwise returns FALSE.

1. **void FreeFramesOfProcess(long id)**

At the time of process termination, operating system is responsible to free all frames of process owns and set to default so that other processes can use those to load their pages in future. Store each frame in free frames array.

File system Management

Every file has its own FCB to store its information like its filename, directory in which file is present (default is HOME), type of file, time of file creation, its disk location on disk and owner of file i.e. process id which created file. File has CREATE\_FILE, OPEN\_FILE, CLOSE\_FILE system calls. Operating system has following function to handle these calls

1. **FILE\_STRUCT \*IsFileExists(char \*name)**

This routine checks file existence in frame table. Return NULL if file is not present or returns file FCB pointer

1. **long CreateFile(char \*name)**

This routine first checks duplicate file name in file table. If file is already present, returns ERROR. Function only allows creating Max\_number\_of\_files otherwise returns ERROR. If conditions to create file satisfies then allocate memory for file FCB and store its name and other parameters of FCB. Adds an entry of newly created file in File Table, increments file counter by 1 and returns SUCCESS.

1. **long OpenFile(char \*name)**

This routine first checks file existence in file table. If file is not present, returns ERROR. If file is already open, then returns ERROR. File FCB is added in Open Files Table to remember which files are opened and returns SUCCESS.

1. **long CloseFile(char \*name)**

This routine first checks file existence in file table. If file is not present, returns ERROR. If file is not open, then returns ERROR. Otherwise, deletes entry from Open File Table and decrements openfilecounter by 1.

Function to show output requirements

1. **void state\_printer(char \*action, INT32 target, INT32 running\_proc\_id, INT32 done, INT32 new)**

This function prints all current states of processes. This function uses suspended queue to show target disk’s pending request in Disk Queue. It shows disk id under Target. For more detail please refer part1 document.

1. **int SetMemoryPrinter\_mode(char \*process)**

This function sets global variable memory\_printer\_mode to true if test is required to show memory printer.

1. **void memory\_printer()**

This function shows each frames present state in memory from frame table. 4 is for valid bit, 1 for referenced bit and 2 for modified bit. State is addition of these bits.

1. **void MemoryPrinter()**

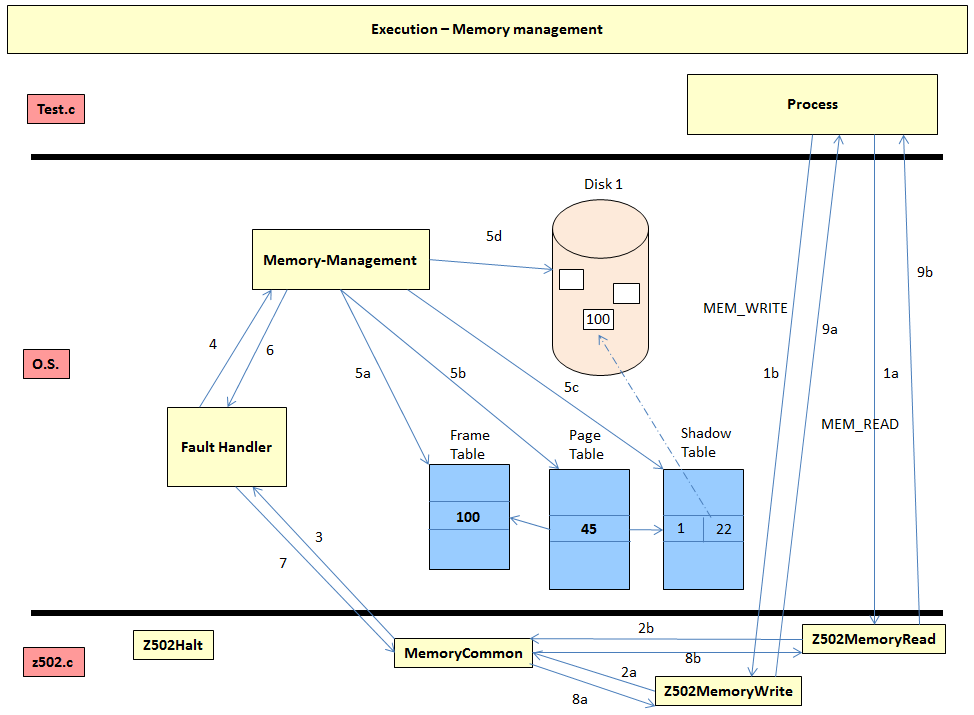
This function tries to complete requirement of memory printer which is mentioned in student manual.For test2a and test2b, it shows memory states for each page faults. For test2e, it shows states after 8 pages faults. For test2f, it shows states 50 pages fault. I believe it shows memory states in limited manner but you can see what is going on in memory very nicely.

***Execution of System Calls***

In part two there are two types of system calls. System calls like MEM\_READ and MEM\_WRITE talks with hardware directly. When hardware do not find page in memory it gives control to operating system to handle this memory management. System calls like DISK\_READ and DISK\_WRITE reads and writes on disk.

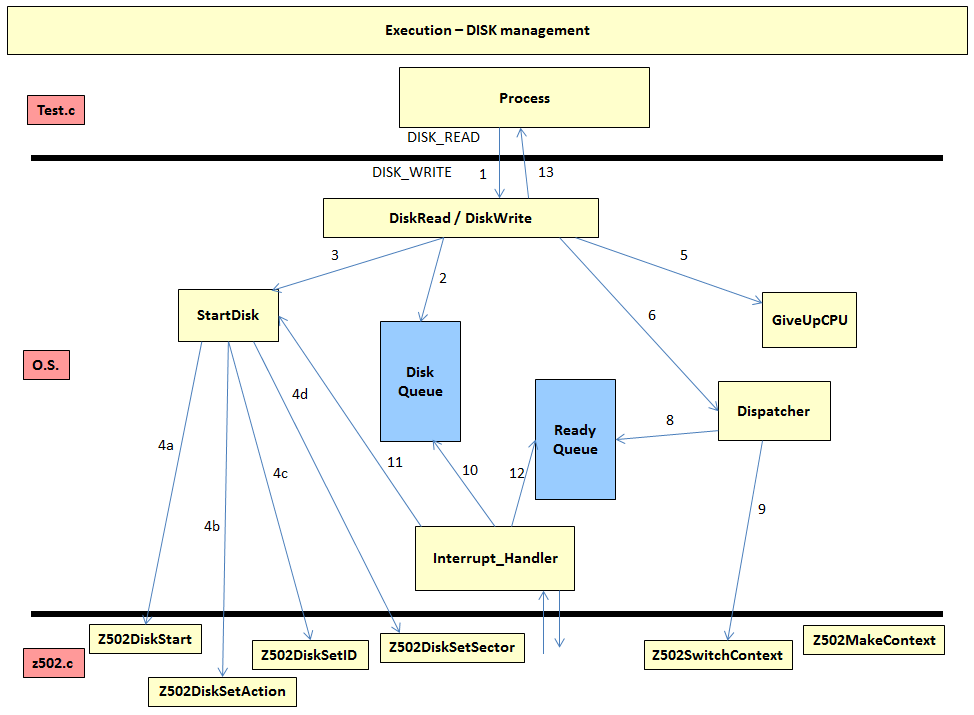
1. **Execution of MEM\_READ and MEM\_WRITE:**

When user calls these system calls, control transfers to hardware z502. Z502 calls Z502MemoryRead() or Z502MemoryWrite() according to READ/WRITE call. Hardware converts virtual address into virtual page. It searchs a page in memory. If it finds, it reads or write on that virtual page. But it page is not found then hardware calls fault handler indicating page fault is occurred. Fault handler calls MemoryManagement() routine to handle this fault. MemoryManagement() selects a victim frame according to replacement algorithm to replace a resident memory page with demand page. This routine checks shadow table for victim page’s swap area to transfer its content to that location. If table has an entry, it reads content of victim frame and write on disk and make shadow table entry to remember for future to use it. If shadow table has no entry, assign disk and disk location to transfer its contents. Make previous resident page invalid. If shadow table has entry for demand page, MemoryManagement() reads contents of demand page from disk and write on victim frame and make page valid to use. After making page valid, control goes back to fault handler and it passes control to hardware z502. Now hardware z502 finds virtual page valid so does read or write operation on that page and returns to user.



1. **Execution of DISK\_READ and DISK\_WRITE:**

When process from user mode calls DISK\_READ and DISK\_WRITE system calls, it gives control to SVC(). SVC() calls DiskRead() and DiskWrite() routines according to system call. DiskRead() and DiskWrite() adds a process who request to read/write in Disk Queue according to disk id. Every disk has its own Queue. After adding in queue, it then checks whether disk is free or not. If it finds that disk is free, then call StartDisk() function to read/write, gives up CPU nnd call Dispatcher to run next ready process in ready queue. If disk is not available only call GiveUpCPU() and Dispatcher(). After disk has finished reading / writing, it causes interrupt handler to run. Interrupt handler then calls HandleDiskInterrupt() which removes head from Disk Queue and add it into ready queue to run. HandleDiskInterrupt() checks if there is another request pending for the disk. If finds, calls StartTimer() to start disk for it to complete its task. After disk has finished its task, control is again passes to SVC() which return control to User.



***Additional features***

I have done some file system calls like CREATE\_FILE, OPEN\_FILE and CLOSE\_FILE. Please refer File System. To check these features, you can run test2i. Also I have implemented Approximate Second chance LRU page replacement algorithm which reduces page faults. Please refer Page replacement algorithm section for more details.

***BUG in simulator***

On Linux operating system, program sometimes gets stuck in Switchcontext() function while suspending previous thread. This happens in pthread\_cond\_wait() function. I was using previously Ubutu. I faced this problem. Then I try my code on windows and it never gets stuck in this function. I am not sure whether it is my logic problem or bug in simulator. But as it gets stuck in this pthread\_cond\_wait(), and we haven’t wrote this function, I feel it is probably a bug. I also faced this problem in part1 in multiprocessing tests programs.