

# CS 502 Project 2 Report

Che Sun

## 1 System Overview

For project 2, new system modules follow the same design pattern used in project 1. Figure 1 shows the final system structure after implementing memory manager and disk manager modules.

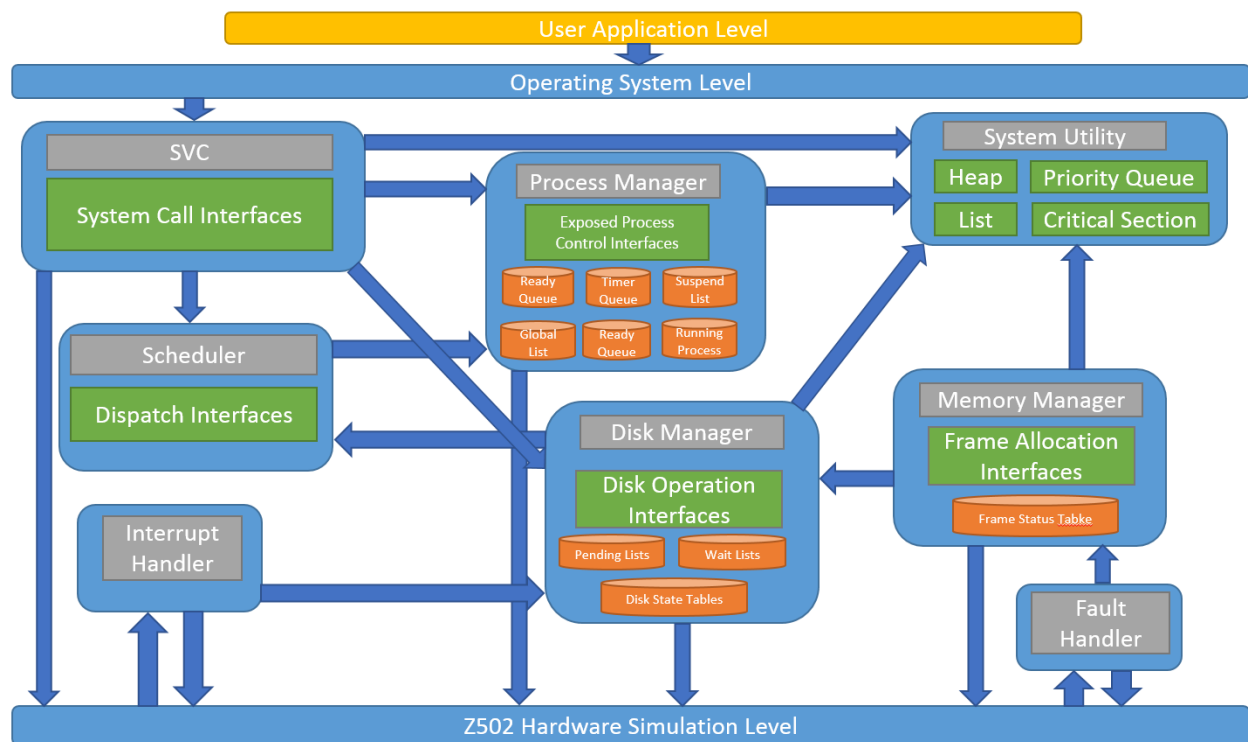


Figure 1. System Overview.

When the operating system starts, in addition to process manager and scheduler implemented in project 1, here it also initializes memory manager and disk manager, which are singleton manager classes for supporting functionalities of project 2. SVC, interrupt handler and fault handler are extended to support the new system functionalities.

## 2 Modules for Project 2

### 2.1 SVC

New interfaces are added to svc.h and svc.c to implement requests for disk operations.

```
void SVCWriteDisk(SYSTEM_CALL_DATA* SystemCallData);
void SVCReadDisk(SYSTEM_CALL_DATA* SystemCallData);
```

SVC is responsible for recording the user disk operation requests and giving those requests to disk manager. In order to track those requests, here, I use a structure called 'DiskOperation':

```
#define DISK_OP_READ          0
#define DISK_OP_WRITE        1
#define DISK_OP_READ_CACHE   2
#define DISK_OP_WRITE_CACHE  3

typedef struct DiskOperation
{
    long diskID;
    long sector;
    int operation;
    char* buffer;
    PCB* requester;
} DiskOperation;
```

This structure is also used for swap in and swap out operations when the memory manager tries to save some physical memory space for a user process. By using this unified data structure, all code dealing with disk operation queues are simplified.

### 2.2 Memory Manager

Memory Manager module is implemented in memory\_manager.h and memory\_manager.c files. It is a singleton class object created when operating system starts. Here is a bunch of interfaces of it:

```
// Memory manager is a global singleton object used to manage physical memory
// of Z502 machine.
typedef struct MemoryManager
{
    MemoryManagerMapPhysicalMemory MapPhysicalMemory;
    MemoryManagerSwapOut           SwapOut;
    MemoryManagerSwapIn            SwapIn;
} MemoryManager;

// Create memory manager when the OS boots.
void MemoryManagerInitialize();
void MemoryManagerTerminate();

// Memory manager global singleton object.
extern MemoryManager* gMemoryManager;
```

Following the same pattern I used in project 1, all data needed by memory manager are hidden from user. The main advantage is that other parts of the system will not mess up the data managed by memory manager.

Memory manager keeps track of the states of physical memory frames as well. So it must maintain a physical memory frame table:

```
// Track the current status of a physical memory page.
typedef struct PhysicalPageStatus
{
    int    used;          // 0 : free 1 : used
    INT32  virtualPageNumber;
    PCB*   user;          // The process that is using this page
} PhysicalPageStatus;

PhysicalPageStatus gPhysicalPageStatusTable[PHYS_MEM_PGS];
```

Each entry of the physical memory frame table is marked either as free or used. If an entry is used, the user and virtual page number of the user are recorded. Memory manager uses these information to find free physical memory frame and swap out frames when all frames have been used by user processes.

## Victim physical memory frame selection

Currently, memory manager selects the victim physical memory frame in a random fashion. This strategy is easy to implement and works well if the overall memory access pattern exhibits some degree of randomness, which is the case for our tests (Because test2f and test2g get virtual page numbers randomly).

## Turn on/off memory printer

For test2g, you can turn off memory printer to see the result more clearly. There is a macro defined in os\_common.h, comment it out will disable the memory printer:

```
// Print out physical memory state.
#define PRINT_MEMORY_STATE
```

## 2.3 Disk Manager

Disk manager is implemented in disk\_manager.h and disk\_manager.c files. The interfaces of it looks like this:

```
// Disk manager is a global singleton object used to manage disk operation
// of Z502 machine.
typedef struct DiskManager
{
    // Disk operation interfaces.
    DiskManagerPushToDiskOperationToDoList PushToDiskOperationToDoList;
    DiskManagerPopFromDiskOperationToDoList PopFromDiskOperationToDoList;
    DiskManagerPushToDiskOperationWaitList PushToDiskOperationWaitList;
    DiskManagerPopFromDiskOperationWaitList PopFromDiskOperationWaitList;
```

```

        DiskManagerGetDiskCache        GetDiskCache;
        DiskManagerFreeDiskCache       FreeDiskCache;

} DiskManager;

// Create disk manager when the OS boots.
void DiskManagerInitialize();
void DiskManagerTerminate();

// Disk manager global singleton object.
extern DiskManager* gDiskManager;

```

Disk manager maintains a disk operation todo queue and wait queue for each disk. So there are 'MAX\_NUMBER\_OF\_DISKS' pairs of them. This allows all disks to be used concurrently and subsequent requests to a disk that is busy enqueued. Whenever a disk finishes writing or reading, the disk manager could start a new disk operation requested earlier.

Another important functionality of disk manager is virtual memory cache. I use a structure called 'DiskSectorInfo' to keep track of the state of each disk:

```

// Disk usage.
#define DISK_UNUSED 0
#define DISK_STORE 1
#define DISK_CACHE 2

typedef struct _DiskSectorInfo
{
    PCB* user;
    int usage;
} DiskSectorInfo;

DiskSectorInfo* gDiskStateTable[MAX_NUMBER_OF_DISKS];

```

So each disk sector could be used as either user storage or virtual memory cache. By using these information, disk manager could manage disks efficiently. A disk sector will be marked as cache when it is chosen to be the temporary storage for a swap out operation. When the data is swapped into physical memory, the disk sector will be marked as unused again.

## 2.4 Interrupt Handler

Interrupt handler is extended to support disk interrupt.

## 2.5 Fault Handler

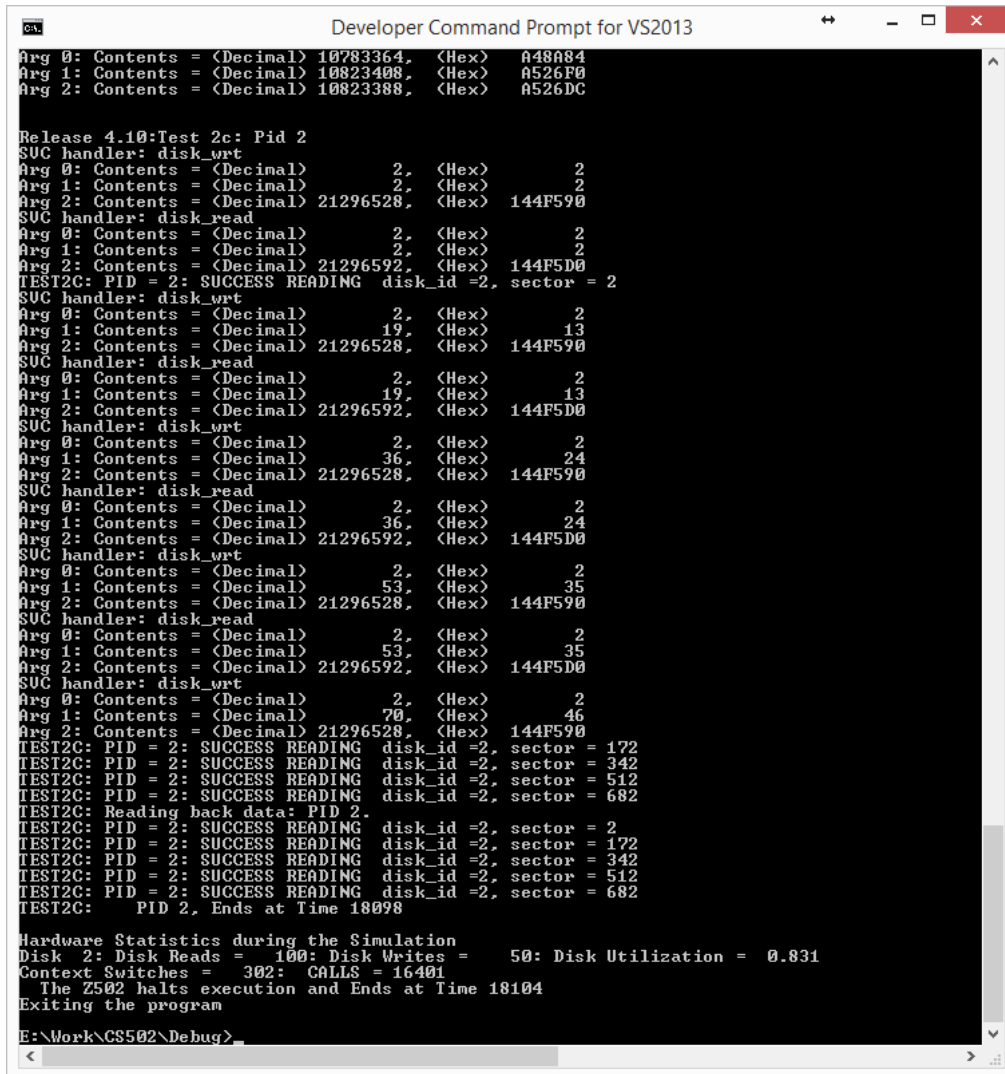
Fault handler is extended to support memory management. The first time a memory access fault happens on a user process, memory manager will allocate a virtual page table for the process.

### 3 Build

There is a Visual Studio 2013 project in my package which can be used directly to build the system. Also, I wrote a makefile using GCC compiler under Cygwin environment. It works on my Windows machine as well.

### 4 Tests

All tests except test2h (shared area) have been tested successfully. In order to run a specific test, type cs502 <test name> in a terminal or command-line window. For example, cs502 test2c, as is shown in Figure 2.



```
cs502
Developer Command Prompt for VS2013

Arg 0: Contents = (Decimal) 10783364, (Hex) 048084
Arg 1: Contents = (Decimal) 10823408, (Hex) 0526F0
Arg 2: Contents = (Decimal) 10823388, (Hex) 0526DC

Release 4.10:Test 2c: Pid 2
SUC handler: disk_wrt
Arg 0: Contents = (Decimal) 2, (Hex) 2
Arg 1: Contents = (Decimal) 2, (Hex) 2
Arg 2: Contents = (Decimal) 21296528, (Hex) 144F590
SUC handler: disk_read
Arg 0: Contents = (Decimal) 2, (Hex) 2
Arg 1: Contents = (Decimal) 2, (Hex) 2
Arg 2: Contents = (Decimal) 21296592, (Hex) 144F5D0
TEST2C: PID = 2: SUCCESS READING disk_id=2, sector = 2
SUC handler: disk_wrt
Arg 0: Contents = (Decimal) 2, (Hex) 2
Arg 1: Contents = (Decimal) 19, (Hex) 13
Arg 2: Contents = (Decimal) 21296528, (Hex) 144F590
SUC handler: disk_read
Arg 0: Contents = (Decimal) 2, (Hex) 2
Arg 1: Contents = (Decimal) 19, (Hex) 13
Arg 2: Contents = (Decimal) 21296592, (Hex) 144F5D0
SUC handler: disk_wrt
Arg 0: Contents = (Decimal) 2, (Hex) 2
Arg 1: Contents = (Decimal) 36, (Hex) 24
Arg 2: Contents = (Decimal) 21296528, (Hex) 144F590
SUC handler: disk_read
Arg 0: Contents = (Decimal) 2, (Hex) 2
Arg 1: Contents = (Decimal) 36, (Hex) 24
Arg 2: Contents = (Decimal) 21296592, (Hex) 144F5D0
SUC handler: disk_wrt
Arg 0: Contents = (Decimal) 2, (Hex) 2
Arg 1: Contents = (Decimal) 53, (Hex) 35
Arg 2: Contents = (Decimal) 21296528, (Hex) 144F590
SUC handler: disk_read
Arg 0: Contents = (Decimal) 2, (Hex) 2
Arg 1: Contents = (Decimal) 53, (Hex) 35
Arg 2: Contents = (Decimal) 21296592, (Hex) 144F5D0
SUC handler: disk_wrt
Arg 0: Contents = (Decimal) 2, (Hex) 2
Arg 1: Contents = (Decimal) 70, (Hex) 46
Arg 2: Contents = (Decimal) 21296528, (Hex) 144F590
TEST2C: PID = 2: SUCCESS READING disk_id=2, sector = 172
TEST2C: PID = 2: SUCCESS READING disk_id=2, sector = 342
TEST2C: PID = 2: SUCCESS READING disk_id=2, sector = 512
TEST2C: PID = 2: SUCCESS READING disk_id=2, sector = 682
TEST2C: Reading back data: PID 2
TEST2C: PID = 2: SUCCESS READING disk_id=2, sector = 2
TEST2C: PID = 2: SUCCESS READING disk_id=2, sector = 172
TEST2C: PID = 2: SUCCESS READING disk_id=2, sector = 342
TEST2C: PID = 2: SUCCESS READING disk_id=2, sector = 512
TEST2C: PID = 2: SUCCESS READING disk_id=2, sector = 682
TEST2C: PID 2, Ends at Time 18098

Hardware Statistics during the Simulation
Disk 2: Disk Reads = 100: Disk Writes = 50: Disk Utilization = 0.831
Context Switches = 302: CALLS = 16401
The Z502 halts execution and Ends at Time 18104
Exiting the program
E:\Work\CS502\Debug>
```

Figure 2. Running a specific test in command-line window.