

CS 502 Project 1 Report

Che Sun

1 System Overview

My operating system is designed and implemented in a highly modular fashion, as is shown in Figure 1. While the programming language is C, structures and function pointers are heavily used to achieve the goal of modularity and object-oriented programming. Here, process manager is a good example demonstrating this idea.

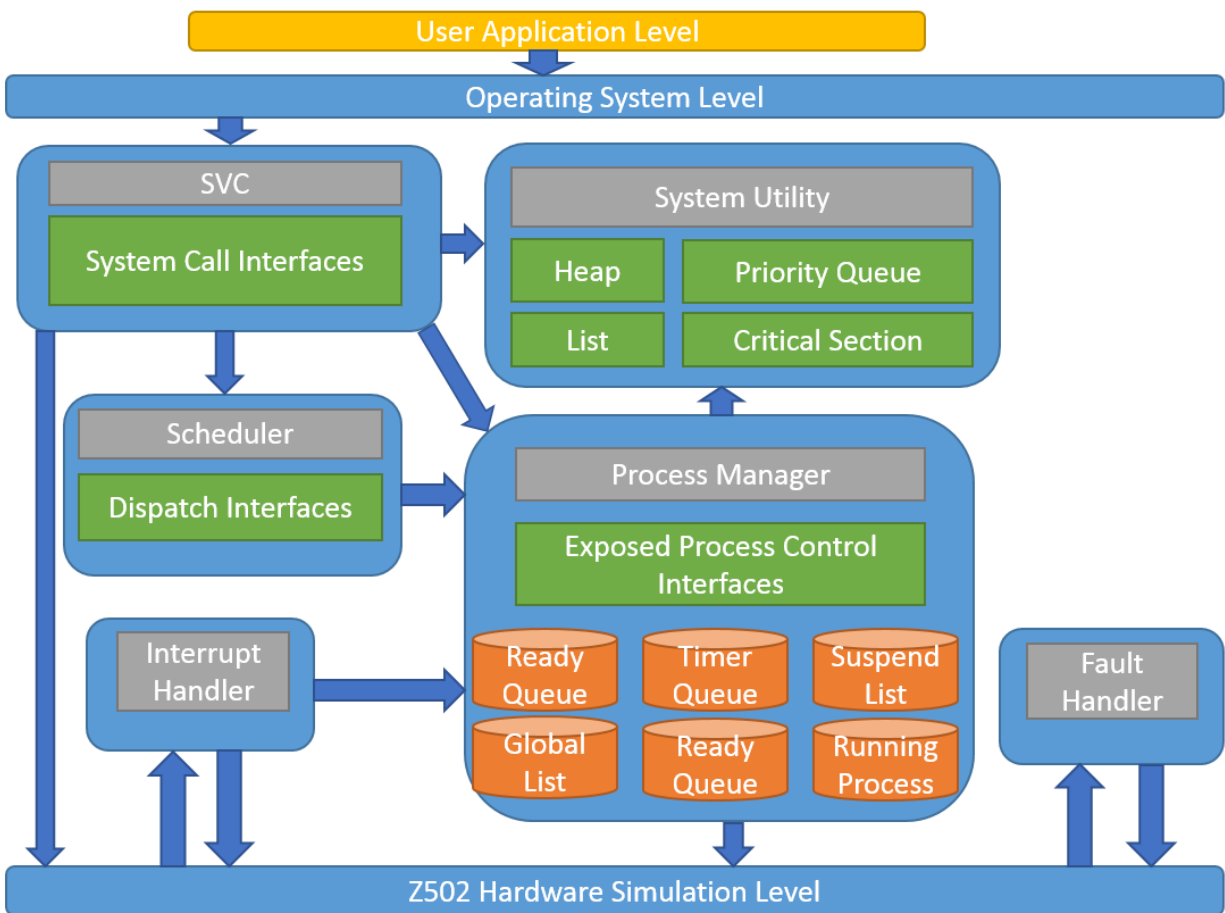


Figure 1. System Overview.

When the operating system starts, it initializes process manager and scheduler, which are singleton objects stored in global variables. Process Manager is in charge of all the process control operations

such as create, terminate and suspend a specific process. Other parts of the system must use interfaces exposed by the process manager to do process management related work. In this way, process data management is hidden by the process manager. So other modules do not need to worry about messing up the underlying process queues and lists. In Figure 1, process data are illustrated as orange cylinders.

Meanwhile, user processes employ svc's interfaces to do system calls. Then svc is the one who communicates with process manager and scheduler.

2 Modules

2.1 SVC

SVC module is implemented in svc.h and svc.c files. Interfaces defined here are:

```
void SVCGetProcessID(SYSTEM_CALL_DATA* SystemCallData);
void SVCTerminateProcess(SYSTEM_CALL_DATA* SystemCallData);
void SVCCreateProcess(SYSTEM_CALL_DATA* SystemCallData);
void SVCStartTimer(SYSTEM_CALL_DATA* SystemCallData);
void SVCSuspendProcess(SYSTEM_CALL_DATA* SystemCallData);
void SVCResumeProcess(SYSTEM_CALL_DATA* SystemCallData);
void SVCChangeProcessPriority(SYSTEM_CALL_DATA* SystemCallData);
void SVCSendMessage(SYSTEM_CALL_DATA* SystemCallData);
void SVCReceiveMessage(SYSTEM_CALL_DATA* SystemCallData);
```

These functions are implementations of system calls used by user level applications. They do necessary error checks and return calling results to the user level application.

2.2 Process Manager

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

```
// Process manager is a global singleton object used to manage processes.
typedef struct ProcessManager
{
    ProcessManagerGetProcessCount          GetProcessCount;
    ProcessManagerGetTimerQueueProcessCount GetTimerQueueProcessCount;
    ProcessManagerGetTimerQueueProcess     GetTimerQueueProcess;
    ProcessManagerGetReadyQueueProcessCount GetReadyQueueProcessCount;
    ProcessManagerGetReadyQueueProcess     GetReadyQueueProcess;
    .
    .
    .
    ProcessManagerPrintState               PrintState;
    ProcessManagerResetReadyQueueKeys      ResetReadyQueueKeys;
} ProcessManager;

void ProcessManagerInitialize();
void ProcessManagerTerminate();
```

```
extern ProcessManager* gProcessManager;
```

As you can see, all interfaces are implemented by using function pointers. This is the only way I know in which object-oriented programming is done using C language. The advantage of doing this is that underlying data such as queues and lists are hidden in the c file. Thus they are invisible to other parts of the system. So it is unlikely that other code mess up data by directly accessing them. Below are process queues and lists defined in process_manager.c.

```
// Global data managed by process manager.
List*      gGlobalProcessList;
MinPriQueue* gTimerQueue;
MinPriQueue* gReadyQueue;
List*      gSuspendedList;
PCB*       gRunningProcess;
```

In my implementation of process ready queue and timer queue, I use heap as underlying data structure. While linked list is easy to manipulate, insertion cost is $O(n)$, whereas heap-based priority queue has a better performance of $O(\lg n)$ on average. Every time we pop a PCB object from the heap, it needs $O(\lg n)$ to adjust itself. So the overall performance is $cO(\lg n)$, which is still $O(\lg n)$.

My PCB structure is defined in pcb.h, as follow:

```
#define PROCESS_STATE_UNKNOWN    0
#define PROCESS_STATE_READY     1
#define PROCESS_STATE_SLEEPING  2
#define PROCESS_STATE_RUNNING   3
#define PROCESS_STATE_SUSPENDED 4
#define PROCESS_STATE_SUSPENDING 5
#define PROCESS_STATE_DEAD      6

#define PROCESS_TYPE_SCHEDULER 0
#define PROCESS_TYPE_USER     1

typedef void(*ProcessEntry)(void);
typedef struct _PCB
{
    char*      name;
    int        type; // 0 : scheduler process 1 : user process
    long       processID;
    ProcessEntry entry;
    int        priority;
    int        timerQueueKey;
    int        readyQueueKey;
    int        state; // 1 : ready 2 : sleep 3 : running etc.
    void*      context;
    List*      messages;
} PCB;
```

Note that there is a readyQueueKey member variable in PCB structure. Initially, it is assigned as priority value of the process. Then it changes after a process is dispatched for running. By changing the value, a process will have lower and lower dynamic priority in the queue. So the scheduler could give other

process who have low priorities a chance running themselves. This readyQueueKey variable is the key implementing my preemptive algorithm.

2.3 Scheduler

Scheduler is implemented in scheduler.h and scheduler.c files. Currently, I am creating a system process for it. The motivation is that the scheduler should have the ability of interrupting a user process who does not sleep (i.e. taking too much CPU time running itself). The scheduler should be able to trigger a timer interrupt regularly to put the running process into ready queue and dispatch another process in ready queue. While this functionality is not required in any tests, it is very important in a real-world systems. Unfortunately, I have not got time to implement this functionality. You can observe my scheduler process switching with the user process back and forth in test1a. That is, when the user process sleeps, the scheduler process takes over and checks if there is another available user process to run.

Preemptive-based Dispatching

A preemptive algorithm is implemented for the scheduler. The basic idea is the concept of dispatching cycle. In each dispatching cycle, every user process's current priority will be modified before it is dispatched. When all processes' current priority values reach a threshold, the current dispatching cycle ends. Then the scheduler resets all the priority values in the read queue and starts a new dispatching cycle. In this manner, a process with higher priority will get dispatched many more times than a process with low priority. See mytest test case in mytest.c file for detail.

2.4 System Utilities

A bunch of helper functions are implemented here, such as list, heap, priority queue etc.

2.5 Interrupt Handler

Timer interrupt is handled in here. All processes who want to wake up at the same time must be woken together. See my code for details. It also checks if it is necessary to re-start the timer to wake up a later process.

2.6 Fault Handler

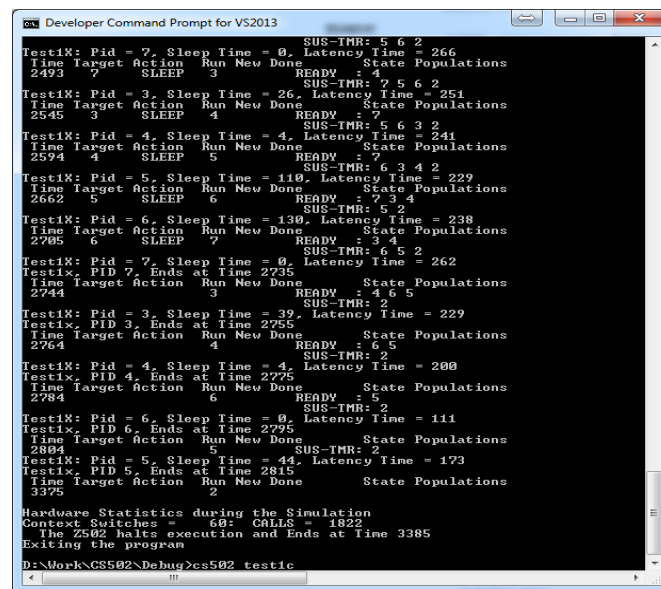
If a user process calls kernel mode functions, an error will be triggered and handled here. See test1k for details.

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 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 test1c, as is shown in Figure 2.



```
Developer Command Prompt for VS2013
Test1X: Pid = 7, Sleep Time = 0, Latency Time = 266
Time Target Action Run New Done State Populations
2493 7 SLEEP 3 READY 4
SUS-TMR: 5 6 2
Test1X: Pid = 3, Sleep Time = 26, Latency Time = 251
Time Target Action Run New Done State Populations
2545 3 SLEEP 4 READY 7
SUS-TMR: 7 5 6 2
Test1X: Pid = 4, Sleep Time = 4, Latency Time = 241
Time Target Action Run New Done State Populations
2594 4 SLEEP 5 READY 3
SUS-TMR: 5 6 3 2
Test1X: Pid = 5, Sleep Time = 110, Latency Time = 229
Time Target Action Run New Done State Populations
2662 5 SLEEP 6 READY 7
SUS-TMR: 6 3 4 2
Test1X: Pid = 6, Sleep Time = 130, Latency Time = 238
Time Target Action Run New Done State Populations
2705 6 SLEEP 7 READY 3
SUS-TMR: 6 5 2
Test1X: Pid = 7, Sleep Time = 0, Latency Time = 262
Test1X, PID 7, Ends at Time 2735
Time Target Action Run New Done State Populations
2744 7 SLEEP 3 READY 4
SUS-TMR: 4 6 5
Test1X: Pid = 3, Sleep Time = 39, Latency Time = 229
Test1X, PID 3, Ends at Time 2755
Time Target Action Run New Done State Populations
2764 3 SLEEP 4 READY 6
SUS-TMR: 6 5 2
Test1X: Pid = 4, Sleep Time = 4, Latency Time = 200
Time Target Action Run New Done State Populations
2784 4 SLEEP 6 READY 5
SUS-TMR: 5
Test1X: Pid = 6, Sleep Time = 0, Latency Time = 111
Test1X, PID 6, Ends at Time 2795
Time Target Action Run New Done State Populations
2804 6 SLEEP 5 READY 2
SUS-TMR: 2
Test1X: Pid = 5, Sleep Time = 44, Latency Time = 173
Test1X, PID 5, Ends at Time 2815
Time Target Action Run New Done State Populations
3375 5 SLEEP 2
SUS-TMR: 2

Hardware Statistics during the Simulation
Context Switches = 60; CALLS = 1822
The CS92 halts execution and Ends at Time 3385
Exiting the program
D:\Work\CS502\Debug>cs502 test1c
```

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

Preemptive-based Dispatching Test

Please type “cs502 test1l” and see my preemptive algorithm implemented in the scheduler module. Notice that processes 3 and 4 with high priorities get much more chances running than process 5, 6 and 7 who have low priorities.