CS 502 Support

Queue Management
State Printers and Output Philosophy

Release 4.50 - Revised 6/2018

This document contains the following sections:

1. Queue management – and overview of an object to make your coding easier.
2. An overview of State Printing – enabling you to describe in a simple way the contents of your various queues and the state of your physical memory.
3. Details of how to use the SchedulerPrinter, a way to easily view the state of the processes running in your operating system.
4. Details of how to use the MemoryPrinter, a way to view physical memory and who is using it.
5. A brief mention of the aprintf( ) method.

1. Queue Management

In previous releases of the CS502 code, students have been required to produce their own Queueing packages – code that they must debug and test, often requiring considerable time. The intent is to provide working code that will make this task easier. Note here that "Queueing" means a singly linked list. For your purposes, you don't care if the list is singly or doubly linked. The behavior of the queues should be the same. I would like to point out that "Queueing" is the only word in the English language with five consecutive vowels.

As it says in the code, there is NO locking provided by the Queue package – that's your job.

The methods supported by this queueing package named QueueManager.c are described detail in the package itself and aren't duplicated here. The names of the routines are:

```
int  QCreate(char *QNameDescriptor);
```
You create a Queue before you can insert or remove items from the Queue.

```
int  QInsert(int QID, unsigned int QueueOrder, void *EnqueueingStructure);
```
Enqueue an item on the designated Q.

```
int  QInsertOnTail(int QID, void *EnqueueingStructure);
```
Enqueue an item on the end of the designated Q.

```
void *QRemoveHead(int QID);
```
Dequeue an item from the head of the designated Q.

```
void *QRemoveItem(int QID, void *EnqueueingStructure);
```
Dequeue an item from the designated Q. May or may not be the head.

void *QNextItemInfo(int QID);
Get information about the item on the head of the designated Q.
The item is NOT removed from the Queue.

void *QItemExists(int QID, void *EnqueueingStructure);
Determine if the item is present on the Q.
The item is NOT removed from the Queue.

char *QGetName(int QID);

int GetNumberOfAllocatedQueues();
Returns the number of Queues that have been allocated;

void *QWalk(int QID, int QOrder);
Returns the address of the "QOrder-th" item on the Q.

DEBUGGING YOUR USE OF THESE METHODS:
This code has a constant Q_TRACE which is normally set to FALSE.
If you set it to TRUE, you will get additional trace information.

void QPrint(int QID)
Print out all details of the items on the designated Q. For Debugging.

2. State Printing Philosophy

The output generated from running your operating system can be overwhelming. The amount of data produced is abundant and in an effort to reduce the amount of data you generate, here are a few guidelines.

The output from your operating system comes in 4 categories, a) Scheduler Output, b) Memory Output, c) Output from the tests you ran, and d) Error and Trap handling. The requirements on what you are to hand in varies for each test. See the ***Student Manual*** for a detailed list.

Each of these categories are addressed below.

2.1 Scheduler Output

Attached are two sample outputs of the SchedulerPrinter. One is the output produced by sample.c. This can be very useful to you because you can look both at the output and at the code that produced that output. The scheduler has five actions that should be logged (you may have more); Dispatch, Create, Done (or Terminate), Ready, and Suspend (in multiple flavors).

You should highlight interesting things in your output, for example: if a process is suspended and resumed, or a change in priority.

2.2 Memory Output

You will want to print out significant memory events that show the work you have done in the OS. This means, for instance, you would want to show the first few memory operations - how you have associated a logical page with a physical page. A very significant memory event is when you have used up all of physical memory and must now steal a page. Showing how you have removed the data from a frame and then associated that frame with a new process/logical page is an EXTREMELY IMPORTANT thing to show. Many people simply log every time there is a change in a frame.

2.3 Test Output

The results from all test programs - that produced by "printf( ) statements in the tests themselves - should be handed in, in their entirety. For the tests provided to you, there is not a lot of output produced.

2.4 Error and Trap Handling

This section includes interrupt, fault, and SVC handling.

As you will see, this can be a lengthy amount of output. You should hand in enough output to demonstrate the proper handling of these errors and traps.

Consider using a similar scheme as the scheduler to show some error and trap handling, then disable it after some threshold has been exceeded. Whatever you set as a threshold, be sure it is high enough to show several instances of all the actions that might occur; in general, if you set the threshold so all of test1 is displayed, you'll be OK.

3. Using The SchedulerPrinter

3.1 Introduction

This tool is designed to save you time by giving a simple way to print out nicely formatted information about the state of the processes on your system. It should, perhaps, more aptly be named "process_printer". To use the printer, you need to:

1. Gather together all the information relating to the current state of all the processes in the simulation.
2. Fill in the data structure that will be used to communicate this information to the SchedulerPrinter.
3. Call SPPrintLine which has the effect of dumping out all the data you've just defined in the data structure.

The scheduler Printer code should be atomic. When you make the call to

SPPrintLine(&SPInput),

you cause that code to build the output on its stack and then print ALL of the data at one time.

3.2 SP Data Structure

This is the data structure used to convey information from your Operating System to the code that will print out the state of your processes. It is defined in syscalls.h.

```
typedef struct {
                // What action is performed
        char    TargetAction[SP_LENGTH_OF_ACTION+2];
                // Pid of process making request
        INT16   CurrentlyRunningPID;
                // Action is being done on what PID
         INT16   TargetPID;
                // At least 1 - duplicates CurrentlyRunningPID
         INT16   NumberOfRunningProcesses;
        INT16   RunningProcessPIDs[SP_MAX_NUMBER_OF_PIDS];
                // Number of Ready To Run Processes
        INT16   NumberOfReadyProcesses;
                // PIDs of those Ready To Run
        INT16   ReadyProcessPIDs[SP_MAX_NUMBER_OF_PIDS];
        INT16   NumberOfProcSuspendedProcesses;
        INT16   ProcSuspendedProcessPIDs[SP_MAX_NUMBER_OF_PIDS];
         INT16   NumberOfMessageSuspendedProcesses;
        INT16   MessageSuspendedProcessPIDs[SP_MAX_NUMBER_OF_PIDS];
        INT16   NumberOfTimerSuspendedProcesses;
        INT16   TimerSuspendedProcessPIDs[SP_MAX_NUMBER_OF_PIDS];
        INT16   NumberOfDiskSuspendedProcesses;
        INT16   DiskSuspendedProcessPIDs[SP_MAX_NUMBER_OF_PIDS];
        INT16   NumberOfTerminatedProcesses;
        INT16   TerminatedProcessPIDs[SP_MAX_NUMBER_OF_PIDS];
} SP_INPUT_DATA;
```

Here is an explanation of the fields in the SP_INPUT_DATA data structure.

TargetAction - This is a string made up of at most SP_LENGTH_OF_ACTION characters. You can use any string you want. The PURPOSE of this field is to define what is happening at any given point. For instance, it's very useful to be able to describe the state of your system when you do a process creation, a schedule, a suspend, etc.

CurrentlyRunningPID - The process ID of the process that's generating the Scheduler Printer request.

TargetPID - The PID of the process that's being effected. For instance, if you are creating a new process, then "CurrentlyRunningPID" is creating a new process "TargetPID".

NumberOfRunningProcesses - Ignored until Multiprocessor hardware is implemented.

RunningProcessPIDs - Ignored until Multiprocessor hardware is implemented.

NumberOfReadyProcesses - The number of processes on the Ready Queue

ReadyProcessPIDs - The list of PIDs of processes on the Ready Queue

NumberOfProcSuspendedProcesses - The number of processes that have been suspended as a result of the SUSPEND_PROCESS system call. Note that this is an EXCELLENT way to be able to show that your OS can implement this system call.

ProcSuspendedProcessPIDs - The list of PIDs of processes that have been suspended as a result of the SUSPEND_PROCESS system call.

NumberOfMessageSuspendedProcesses - The number of processes that are waiting for a message. This could be processes that have executed a RECEIVE_MESSAGE system call.

MessageSuspendedProcessPIDs - The list of PIDs of processes that are waiting for a message.

NumberOfTimerSuspendedProcesses - The number of processes on the Timer Queue. These are processes that have executed a SLEEP system call and are waiting for the timer to interrupt.

TimerSuspendedProcessPIDs - The list of PIDs of processes on the Timer Queue.

NumberOfDiskSuspendedProcesses - The number of processes on the Disk Queues. These are processes that have executed a DISK_READ or DISK_WRITE and are waiting for the disk to interrupt.

DiskSuspendedProcessPIDs - The list of PIDs of processes on the Disk Queues.

NumberOfTerminatedProcesses - This field is currently not used by the Print Manager.

TerminatedProcessPIDs - This field is currently not used by the Print Manager.

Format: SPPrintLine(SP_INPUT_DATA *OutputData);

This routine prints out all the data that has been defined when setting up the output structure. Here is an example of the output generated when running the sample code "z502 sample". You can see how the values used in this example are prepared by looking at sample.c.

```
 Time Target  Action Run   State    Populations
17614    3    Create   2   READY  : 0 1 2
                           SUS-PRC: 3 4 5  6
                           SUS-TMR: 8 9 10 11 12 13
                           SUS-MSG: 16 17
                           SUS-DSK: 15
```

3.3. An Example of SchedulerPrinter Output

Here is an example of the output from SchedulerPrinter. This is an abbreviated version of what might be printed out when running “z502 test3”.

```
    Time Target   Action Run    State    Populations
    65   1     Create    0
Time Target   Action Run   State    Populations
  102   2     Create    0  READY   : 1
Time Target   Action Run   State    Populations
  139   3     Create    0  READY   : 1 2
Time Target   Action Run   State    Populations
  176   4     Create    0  READY   : 1 2 3
Time Target   Action Run   State    Populations
  213   5     Create    0  READY   : 1 2 3 4
Time Target   Action Run   State    Populations
  236   0       Sleep   0  READY   : 1 2 3 4 5
                           SUS-TMR: 0
Time Target   Action Run   State    Populations
  242   1    Dispatch   0  READY   : 1 2 3 4 5
                           SUS-TMR: 0
Time Target   Action Run   State    Populations
  290   1       Sleep   1  READY   : 2 3 4 5
                           SUS-TMR: 1 0
Time Target   Action Run   State    Populations
  296   2    Dispatch   1  READY   : 2 3 4 5
                           SUS-TMR: 1 0
Time Target   Action Run   State    Populations
  344   2       Sleep   2  READY   : 3 4 5
                           SUS-TMR: 2 1 0
Time Target   Action Run   State    Populations
  350   3    Dispatch   2  READY   : 3 4 5
                           SUS-TMR: 2 1 0
```

4. Using The Memory Printer

4.1 Introduction

This tool is designed to save you time by giving a simple way to print out nicely formatted information about the state of physical memory on your system. To use the printer, you repetitively do the following:

1. Gather all information about the physical memory frames in the provided data structure.
2. Call MPPrintLine which outputs the entire contents of the MP_INPUT_DATA structure as a single print statement.

The ultimate goal is to have a simple and compact way to display the memory state at any point in time.

4.2 MP Data Structure

This is the data structure that defines EACH of the frames in the physical memory.

```
typedef struct {
    INT16    InUse;          // TRUE == in use, FALSE == not in use
    INT16    Pid;            // The Process  holding this frame
    INT16    LogicalPage;    // The logical  page in that process
    INT16    State;          // The  state of the frame.
} MP_FRAME_DATA;
```

And this is the structure that aggregates the information from all the frames in a single large strucure

```
typedef struct {
    MP_FRAME_DATA   frames[PHYS_MEM_PGS];
} MP_INPUT_DATA;
```

The information required for each frame is defined as follows:

InUse: If TRUE, then this frame is in use and the data in the remainder of the structure is valid. If FALSE, then information about this frame is ignored and the printout assumes the frame is not used.

Pid: The process id of that process that has this frame in its page table. In the case of shared memory where more than one process' page table aims at this frame, you'll have to pick one. Legal values are 0 - 9; sorry but there's only one digit available in the printout.

LogicalPage: This identifies which logical page within the page table of process PID is associated with this frame. In the output this is called VPN = virtual page number. Legal values are 0 to (VIRTUAL_MEM_PGS - 1) or as currently implemented, 0 to 1023.

State: The current state of this page. Values possible here are:

```
FRAME_VALID  Frame is Valid - the physical frame contains real data
FRAME_MODIFIED  Frame is Modified - it has been written to making it dirty
FRAME_REFERENCED Frame is Referenced - some process has written or read it
```

The state you enter is the sum of these possible values. So, if a frame is valid and has been read, you would enter FRAME_VALID + FRAME_REFERNCED as the value of its state. Note that these are the same values (and in the same order) as in the page table.

4.3 Using MPPrintLine

Format: MPPrintLine(MP_INPUT_DATA *input);

This routine prints out all the data that has been defined when filling in the MP_INPUT_DATA structure. Here is an example of the output that is printed by Sample.c

```
                          `
                        PHYSICAL MEMORY STATE
Frame 000000000011111111112222222222333333333344444444445555555555666 6
Frame 012345678901234567890123456789012345678901234567890123456789012 3
PID   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
VPN   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
VPN   0 0 0 1 1 1 2 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7 8 8 8 9 9 9 0
VPN   1 4 7 0 3 7 0 3 6 9 3 6 9 2 5 9 2 5 8 1 5 8 1 4 7 1 4 7 0 3 7 0
VPN   0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2
VMR   0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
```

5. Using aprintf

Format: void aprintf(const char *format, ...);

This routine is exactly like a printf, except it has a lock around it. This protects the output from being divided into multiple pieces. Linux especially has a way of splitting text in a way you didn't want or expect.