CS2106 Operating Systems
2017/18 Semester 2
Term Assignment

INTRODUCTION

In this term assignment you will implement two schedulers: A LINUX-style scheduler with 140 priority levels, and a Rate-Monotonic-Scheduling scheduler.

Several files are enclosed in this ZIP file in addition to this writeup:

llist.h/llist.cpp: A FIFO queue implementation.
prioll.h/prioll.cpp: A priority queue implementation.
kernel.h/kernel.cpp: The OS "kernel" where you are to implement your schedulers.
main.cpp: A test program.
assgansbk.docx: Your answer book.

TEAM SETUP

You are to work in teams of 2 or 3. Single-person submissions are not allowed.

DELIVERABLES

Zip up kernel.cpp, kernel.h, and assgansbk.docx together into a ZIP file named Exxxxxxx.ZIP where Exxxxxx is the student number of the team's leader. Please ensure that every student's name and Student Number are in kernel.cpp and assgansbk.docx.

ASSIGNMENT DEADLINE IS SUNDAY 22 APRIL 2018 2359 HRS

CODE WALKTHROUGH

We will examine the three main files: kernel.h, kernel.cpp and main.cpp. This walkthrough is very brief and you are advised to read it together with the source code.

The kernel.h file contains the following constants:

SCHEDULER_TYPE	You should set this to 0 to generate code for the LINUX compiler, and 1 to generate code for the RMS compiler.
NUM_PROCESSES	Maximum number of processes allowed. Default is 10.
NUM_RUNS	Number of full scheduler runs. In the LINUX scheduler, each process will complete execution once in each run. Default is 2 runs. In the RMS scheduler, each run consists of LCM(p1, p2, … pN), where p1, ... pN is the period of each process.
PRIO_LEVELS	Number of LINUX priority levels. Default is 140.
QUANTUM_STEP	Used in calculating the quantum for each priority level in the LINUX scheduler.
QUANTUM_MIN	Quantum size for the lowest priority process in the LINUX scheduler.

The code in kernel.cpp is fairly complex and consists of two independent sections; one section consists of the LINUX code and is compiled when SCHEDULER_TYPE is 0, and the other consists of RMS code and is compiled when SCHEDULER_TYPE is 1. There is also a third section of code that is shared by both the LINUX and RMS schedulers.

a. The Shared Part

The shared code consists of:

i) Data Structures

There is a table called “processes” which is the Process Table. Each entry of the Process Table is of type TTCB.

TTCB’s definition is either:

```
/* Process Control Block for LINUX scheduler*/
typedef struct tcb
{
    int procNum;
    int prio;
    int quantum;
    int timeLeft;
} TTCB;
```

For the LINUX scheduler, or:

```
/* Process Control Block for RMS scheduler*/

typedef struct tcb
{
    int procNum;
    int timeLeft;
    int deadline;
    int c;
    int p;
} TTCB;
```

For the RMS scheduler.

More details on TTCB can be found later in this document.

There are also several shared variables:

a) timerTicks: Number of 1 ms jiffies that have past since the OS was started.
b) numProcesses: Number of processes created.
c) currProcess: ID of currently running process.

ii) Functions:

timerISR(): Calls either linuxScheduler or RMSScheduler to get the process ID of the process that should be running, and increments timerTicks. It also prints out which process is running.

startTimer(): Calls timerISR() once per millisecond, for 2 complete runs.

startOS(): Sets up data structures important to the LINUX and RMS schedulers, then calls startTimer()

initOS(): Initializes the OS.

addProcess(): Adds in a new process. Process IDs are automatically allocated by the OS. There are two versions:

LINUX:

addProcess(prio): Add a process with the given priority level.

RMS:

addProcess(period, cpuTime): Adds a process with period "period", running for "cpuTime" clock cycles each time.

b. <u>LINUX Portion</u>

The LINUX parts of the code are enclosed with:
#if SCHEDULER_TYPE == 0
… LINUX code …
#elif SCHEDULER_TYPE == 1

i) Data Structures

Processes are managed using two arrays of linked lists of type TNode, defined in llist.h:

```
// This file implements a FIFO queue
typedef struct ll
{
    struct ll *next, *prev;
    int procNum;
    int quantum;
}TNode;
```

Note that TNode doesn't actually contain much process data. Main process data is in the TTCB entry in "processes". procNum is an index into "processes", and "quantum" is the quantum allocated to the process, used in computing how long each run lasts in milliseconds.

There are two arrays of linked lists, each consisting of 140 linked lists. Each linked list corresponds to one priority level. E.g. queueList[0] is a linked list of processes at priority level 0, etc.

queueList1: First array.
queueList2: Second array.

There are two variables that tell us which array is the active set, and which is the expired set:

activeList: Points to array of queues containing active processes.
expiredList: Points to array of queues containing expired processes/

ii) Functions

findNextPrio: Finds next priority level in the active set that has processes ready to run. Runs through the array of queues in the index set looking for the first entry that is not NULL.

linuxScheduler: The main LINUX Scheduler. You need to implement this.

findQuantum: Computes the quantum in milliseconds for each priority level using the equation:

$$\text{findQuantum}(\text{prio}) = 2 \times (139 - \text{prio}) + 20$$

c. RMS Portion

The RMS code is enclosed between:

#elif SCHEDULER_TYPE == 1
… RMS code …
#endif

i) Data Structures

The basic data stricture used to manage RMS processes is TPrioNode, defined in prioll.h.

```
// Node that lets us sort by priority
typedef struct t
{
    struct t *prev, *next;
    int procNum;
    int prio;
    int p;

} TPrioNode;
```

Once again this structure is only used to tell us which process is ready to run, which is blocked, etc., it does not contain task information like the # of CPU cycles per run, deadlines, etc.

The procNum field is an index into the “processes” table, prio is a priority field that is used to sort the linked list, and p is the process period. Generally we would set p and prio to the same value, so that the list is sorted by process period.

There are several important variables:

readyQueue: Queue of processes ready to run. In our simple case processes are always ready to run in multiples of their process periods p. So for a process with period of 4, the process is always ready at cycles 0, 4, 8, 12, 16, etc.

blockedQueue: Processes that have run, and are not due to run again.

currProcessNode: The TPrioNode entry of the currently running process. You need to assign this within your RMS scheduler.

suspended: The TPrioNode entry of a process that has been pre-empted. You need to assign this within your RMS scheduler when a process gets pre-empted by a higher priority process.

ii) Functions

RMSScheduler: Implement your RMS Scheduler here.

Finally main.cpp is fairly simple:

```
#include <stdio.h>
#include "kernel.h"

#define MISS_DEADLINE       0
int main()
{
    initOS();

#if SCHEDULER_TYPE == 0
    addProcess(15);
    addProcess(106);
    addProcess(109);
    addProcess(139);
    addProcess(109);
    addProcess(15);
    addProcess(139);
    addProcess(109);
#elif SCHEDULER_TYPE == 1

#if MISS_DEADLINE==0
    addProcess(4, 1);
    addProcess(8, 2);
    addProcess(12, 3);
#elif
    addProcess(3, 1);
    addProcess(6, 2);
    addProcess(8, 3);
#endif
#endif
    startOS();
}
```

The main.cpp file begins with a definition for MISS_DEADLINE, which is used by the RMS part of main.cpp.

This file contains just one function, which is main. The main function calls initOS to initialize the operating system. In the LINUX portion it calls addProcess to add in 8 processes of various priority levels.

The RMS portion is divided into two parts. When MISS_DEADLINE is 0, main creates three processes with the following properties:

Process	Ci	Pi
1	1	4
2	2	8
3	3	12

These processes are schedulable under RMS.

When MISS_DEADLINE is 1, main creates processes with the following characteristics:

Process	Ci	Pi
1	1	3
2	2	6
3	3	8

TASK 1 – LINUX SCHEDULER

The details of the LINUX scheduler are covered in pages 24-26 of Lecture 3. We will build a simplified scheduler with processes that do not block, and without any RT-FIFO and RT Round Robin processes.

All process information is stored in the Process Table, called “processes”. Process information stored for the LINUX scheduler include:

```
/* Process Control Block for LINUX scheduler*/
typedef struct tcb
{
    int procNum;
    int prio;
    int quantum;
    int timeLeft;
} TTCB;
```

procNum: The process number.
Prio: Process priority number.
Quantum: Assigned process quantum in milliseconds.
timeLeft: Number of millseconds left before this process gets swapped out.

The two arrays queueList1 and queueList2 maintain linked lists of processes at each priority level. You may find the following functions from llist.h/llist.cpp useful:

Function Name	Parameters	Description

Insert(head, procNum, quantum)	head: Pointer to the head variable for the linked list. This has to be of type TNode **.
	
	procNum: Process number.
	
	quantum: Process quantum in millseconds
		Inserts a new process into the linked list pointed to by “head”.
		
		E.g.
		
		activeList = queueList1;
		
		insert(&activeList[1], 15, 300);
		
		This will insert a new process with process ID of 15, and a quantum of 300 ms into the linked list for priority level 1.

remove(head)	head: Pointer to the head variable for the linked list.	Removes the first item in the linked list. E.g.: int processID = remove(&activeList[15]); This will remove the first process in the queue for priority level 15, and return the process number.

totalQuantum	head: The head variable for the linked list.	Sums up all the quantum in the linked list.
		
		E.g.:
		
		This finds the total quantum of all processes across all priority levels.
		
		total=0;
		
		for(i=0; i<PRIO_LEVELS; i++)
		total += totalQuantum(activeList[i]);

destroy	Pointer to the head function.	De-allocates entire linked list.
		
		E.g.
		
		// De-allocates linked list for
		// priority level 15.
		destroy(&queueList1[15]);

Implement a LINUX style scheduler in the linuxScheduler function. You should not make any changes outside of the linuxScheduler function, unless otherwise noted here.

To compile:

i. Edit kernel.h and ensure that SCHEDULER_TYPE is set to 0:

```
#define SCHEDULER_TYPE                0
```

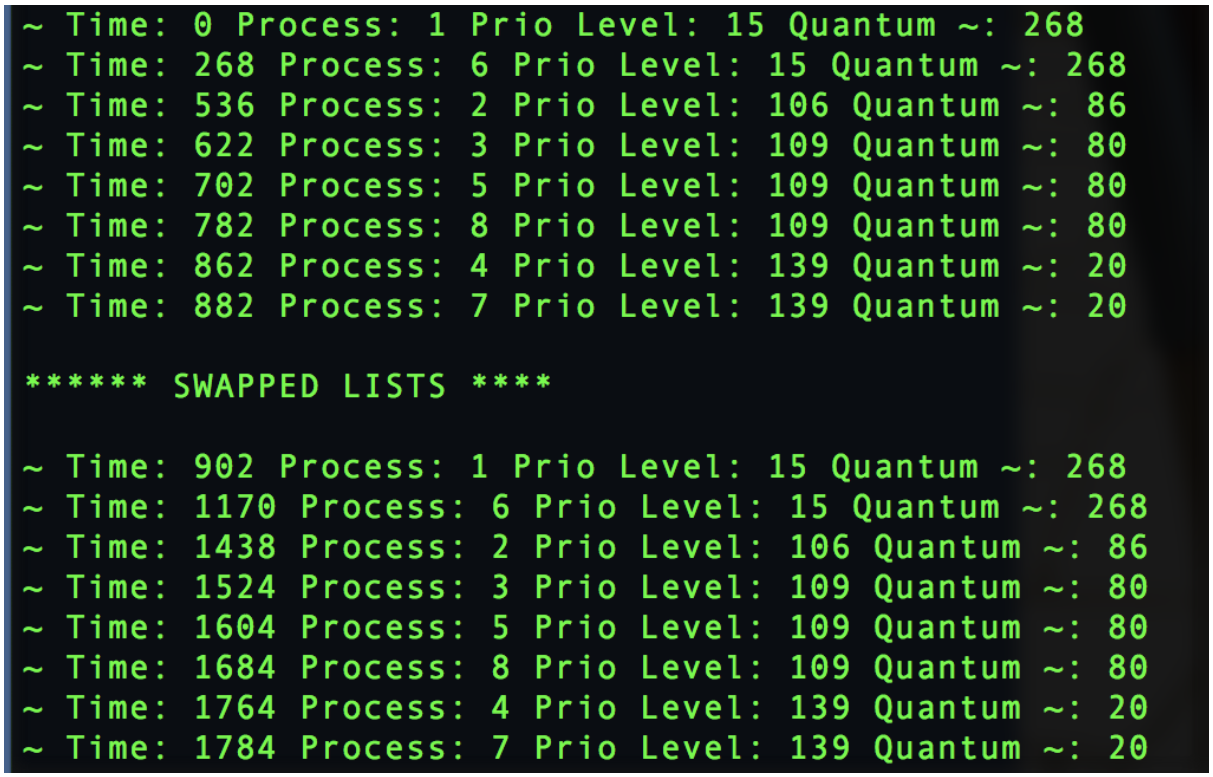
ii. Compile with:

```
gcc kernel.cpp main.cpp llist.cpp –o linux.
```

iii. Run with:

./linux

If you have implemented the linuxScheduler function correctly, your output will look like this:



(Note: Your own output WILL NOT have the "~" characters you see here, but the timing and sequence of process execution will be exactly as shown)

Answer the following questions:

Question 1 (15 marks)

Cut, paste and explain your linuxScheduler code in your answer bookAND EXPLAIN YOUR CODE. You can use comments to explain each step.

Also cut and paste a screen shot of your program running into your report.

Question 2 (5 marks)

LINUX stores processes in an array of 140 queues, with each queue for one priority level. Give one advantage and one disadvantage of this approach compared to storing all the processes in a priority queue sorted in descending order by quantum.

Question 3 (5 marks)

You want to implement a "renice" function that can change the priority level of a process and its quantum. Your function signature is:

```
void renice(int adjust);
```

Adjust can be anything from -15 to 15.

Give pseudocode for how you would implement renice in this operating system. YOU DO NOT ACTUALLY HAVE TO IMPLEMENT IT, JUST GIVE AND EXPLAIN THE PSEUDOCODE.

TASK 2 – RMS SCHEDULER

The Rate-Monotonic-Scheduler (RMS) is a fixed priority scheduling policy for periodic processes, where processes with the shorter periods have higher priorities. Please see pages 16 to 19 of Lecture 3 for more details about how RMS works.

Process information is as before stored in the Process Table "processes". Information for each process is shown below:

```
/* Process Control Block for RMS scheduler*/

typedef struct tcb
{
    int procNum;
    int timeLeft;
    int deadline;
    int c;
    int p;
} TTCB;
```

The fields are:

procNum: Process ID
timeLeft: Number of milliseconds left for this process to run
deadline: Next deadline in milliseconds
c: CPU time this process will consume at each run.
p: Process period in milliseconds.

You may find the following functions in prioll.h / prioll.cpp useful:

Function Name	Parameters	Description

prioInsert(head, procNum, p, prio)	head: Pointer to the queue head.
	procNum: Porces ID of the process to insert.
	p: Process period
	prio: Process priority. The list is sorted by priority in ascending order. For convience you can set p and prio to the same value.
		Inserts a new process.
		
		E.g.
		// Insert process 1 with period 4 and priority 4 into the ready queue.
		prioInsert(&readyQueue, 1, 4, 4);

prioInsertNode(head, TPrioNode *node);	head: Pointer to the queue head.
	node: Node to be inserted to the queue.
		Inserts a node into a queue.
		
		E.g.
		
		// Remove from ready queue and place into blocked queue.
		
		TPrioNode *node = prioRemove(&readyQueue);
		
		prioInsertNode(&blockedQueue, node);

prioRemove(head)	head: Pointer to the queue head.	Removes and returns the node at the head of the queue. E.g. shown above in prioInsertNode. Returns NULL if queue is empty.

prioRemoveNode(head, node)	head: Pointer to queue head.
	node: Node to be removed. This node must be in the queue pointed to by head.
		Removes a node from the queue. Returns NULL if no processes are ready.
		
		E.g.: The following code will check if there's any processes in the blocked queue that are due to run at the given timerTick. If so remove from the blocked queue and insert into the ready queue.
		
		TPrioNode *node = checkReady(blockedQueue, timerTicks);
		
		while(node != NULL)
		{
		prioRemoveNode(&blockedQueue, node)
		prioInsertNode(&readyQueue, node);
		node = checkReady(blockedQueue, timerTicks);
		}

checkReady(head, timerTick)	head: Queue's head variable.
	timerTick: Current timertick.
		Returns a pointer to a node for a process that is ready at time "timerTick", but does not remove the node.
		
		E.g.: Moves all processes that are ready at time timerTick to the ready queue:
		
		TNode *node = checkReady(blockedQueue, timerTick);
		
		while(node)
		{
		prioRemoveNode(&blockedQueue, node);
		
		prioInsertNode(&readyQueue, node);
		}

printList(head)	head: Head variable for the queue.	Prints contents of the list pointed to by head
		
		E.g.:
		
		// Print details of all processes in the ready queue.
		printList(readyQueue);

peek(head)	head: Head variable for the queue.	Returns the first node of the queue pointed to by head without removing it, or NULL if queue is empty.
		
		E.g. See if priority of process in the ready queue is higher than our current process. Pre-empt current process if so.
		
		TPrioNode *node = peek(&readyQueue);
		
		if(node->p < currentProcessNode->p)
		{
		// Pre-empt
		suspended = currentProcessNode;
		currentProcessNode = prioRemove(&readyQueue);
		}

prioLCM(head)	head: Head variable for the queue.	Returns LCM of all process periods in the queue.
prioDestroy(head)	head: Pointer to head variable for the queue.	Destroys the queue by releasing all memory, and sets head to NULL.

Implement your RMS scheduler in the RMSScheduler function. You should not modify anything outside of this function.

To compile:

i) Edit kernel.h and set SCHEDULE_TYPE to 1:

```
#define SCHEDULER_TYPE 1
```

ii) Edit main.cpp, and set MISS_DEADLINE to 0:

```
#define MISS_DEADLINE 0
```

iii) Compile:

```
gcc kernel.cpp prioll.cpp main.cpp –o rms
```

iv) Run:

```
./rms
```

If you have implemented your schedule properly, your output will be very similar to what is on the next page, except that your output will NOT have the “~” characters:

```
Time: 0 ~ P1 Deadline: 4 ~
Time: 1 ~ P2 Deadline: 8 ~
Time: 2 ~ P2 Deadline: 8 ~
Time: 3 ~ P3 Deadline: 12 ~

--- PRE-EMPTION ---

Time: 4 ~ P1 Deadline: 8 ~
Time: 5 ~ P3 Deadline: 12 ~
Time: 6 ~ P3 Deadline: 12 ~
Time: 7 ---
Time: 8 ~ P1 Deadline: 12 ~
Time: 9 ~ P2 Deadline: 16 ~
Time: 10 ~ P2 Deadline: 16 ~
Time: 11 ---
Time: 12 ~ P1 Deadline: 16 ~
Time: 13 ~ P3 Deadline: 24 ~
Time: 14 ~ P3 Deadline: 24 ~
Time: 15 ~ P3 Deadline: 24 ~
Time: 16 ~ P1 Deadline: 20 ~
Time: 17 ~ P2 Deadline: 24 ~
Time: 18 ~ P2 Deadline: 24 ~
Time: 19 ---
Time: 20 ~ P1 Deadline: 24 ~
Time: 21 ---
Time: 22 ---
Time: 23 ---
Time: 24 ~ P1 Deadline: 28 ~
Time: 25 ~ P2 Deadline: 32 ~
Time: 26 ~ P2 Deadline: 32 ~
Time: 27 ~ P3 Deadline: 36 ~

--- PRE-EMPTION ---

Time: 28 ~ P1 Deadline: 32 ~
Time: 29 ~ P3 Deadline: 36 ~
Time: 30 ~ P3 Deadline: 36 ~
Time: 31 ---
Time: 32 ~ P1 Deadline: 36 ~
Time: 33 ~ P2 Deadline: 40 ~
Time: 34 ~ P2 Deadline: 40 ~
Time: 35 ---
Time: 36 ~ P1 Deadline: 40 ~
Time: 37 ~ P3 Deadline: 48 ~
Time: 38 ~ P3 Deadline: 48 ~
Time: 39 ~ P3 Deadline: 48 ~
Time: 40 ~ P1 Deadline: 44 ~
Time: 41 ~ P2 Deadline: 48 ~
Time: 42 ~ P2 Deadline: 48 ~
Time: 43 ---
Time: 44 ~ P1 Deadline: 48 ~
Time: 45 ---
Time: 46 ---
Time: 47 ---
```

Question 4. (20 marks)

Cut, paste and explain your RMSScheduler code in your answer book. Also cut and paste your screen output.

Question 5. (5 marks)

Compute the CPU utilization using the following formula:

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i}$$

Estimate also the CPU utilization based on the output of the program. Do the two utilizations agree? Explain your observations.

Question 6. (10 marks)

Sketch modifications and write down pseudocode for how to modify the RMS scheduler you have built into an EDF scheduler. You do not have to build the EDF scheduler, only explain in as much detail as possible how you can convert your RMS scheduler into an EDF scheduler.

Now we will look at missing deadlines!

i) Edit main.cpp and set MISS_DEADLINE to 1:

```
#define MISS_DEADLINE   1
```

This will create a new set of processes with tighter periods.

ii) Recompile:

```
gcc kernel.cpp prioll.cpp main.cpp –o rms
```

iii) Run:

```
./rms
```

If you have implemented your scheduler correctly, you will see an output like what is shown on the next page (note: Your own output WILL NOT show the ~ characters):

```
Time: 0 ~ P1 Deadline: 3 ~
Time: 1 ~ P2 Deadline: 6 ~
Time: 2 ~ P2 Deadline: 6 ~
Time: 3 ~ P1 Deadline: 6 ~
Time: 4 ~ P3 Deadline: 8 ~
Time: 5 ~ P3 Deadline: 8 ~

--- PRE-EMPTION ---

Time: 6 ~ P1 Deadline: 9 ~
Time: 7 ~ P3 Deadline: 8 ~
Time: 8 ~ P2 Deadline: 12 ~

--- PRE-EMPTION ---

Time: 9 ~ P1 Deadline: 12 ~
Time: 10 ~ P2 Deadline: 12 ~
Time: 11 ---
Time: 12 ~ P1 Deadline: 15 ~
Time: 13 ~ P2 Deadline: 18 ~
Time: 14 ~ P2 Deadline: 18 ~
Time: 15 ~ P1 Deadline: 18 ~
Time: 16 !! ~ P3 Deadline: 16 ~ !!
Time: 17 !! ~ P3 Deadline: 16 ~ !!

--- PRE-EMPTION ---

Time: 18 ~ P1 Deadline: 21 ~
Time: 19 !! ~ P3 Deadline: 16 ~ !!
Time: 20 ~ P2 Deadline: 24 ~

--- PRE-EMPTION ---

Time: 21 ~ P1 Deadline: 24 ~
Time: 22 ~ P2 Deadline: 24 ~
Time: 23 ---
Time: 24 ~ P1 Deadline: 27 ~
Time: 25 ~ P2 Deadline: 30 ~
Time: 26 ~ P2 Deadline: 30 ~
Time: 27 ~ P1 Deadline: 30 ~
Time: 28 !! ~ P3 Deadline: 24 ~ !!
Time: 29 !! ~ P3 Deadline: 24 ~ !!

--- PRE-EMPTION ---

Time: 30 ~ P1 Deadline: 33 ~
Time: 31 !! ~ P3 Deadline: 24 ~ !!
Time: 32 ~ P2 Deadline: 36 ~

--- PRE-EMPTION ---

Time: 33 ~ P1 Deadline: 36 ~
Time: 34 ~ P2 Deadline: 36 ~
Time: 35 ---
Time: 36 ~ P1 Deadline: 39 ~
Time: 37 ~ P2 Deadline: 42 ~
Time: 38 ~ P2 Deadline: 42 ~
Time: 39 ~ P1 Deadline: 42 ~
Time: 40 !! ~ P3 Deadline: 32 ~ !!
Time: 41 !! ~ P3 Deadline: 32 ~ !!

--- PRE-EMPTION ---

Time: 42 ~ P1 Deadline: 45 ~
Time: 43 !! ~ P3 Deadline: 32 ~ !!
Time: 44 ~ P2 Deadline: 48 ~

--- PRE-EMPTION ---

Time: 45 ~ P1 Deadline: 48 ~
Time: 46 ~ P2 Deadline: 48 ~
```

Notice that processes that miss their deadlines are marked with !!.

Question 7. (2 marks)

Cut and paste the output of your program to your report.

Question 8. (8 marks)

Perform Critical Instance Analysis on this second set of tasks, and also find the CPU utilization of this second set.

Comment on the outcome of both analyses.