Lab 3: Group 6
Jordan Hutcheson
Walter Conway
Date: 4/8/2014
Your report must:

A.) State whether your code works.

B.) Report/Analyze the results about the missed events (based on the filled table), the average response time, and the average turnaround time. Contrast lab3 values with those measured in lab 2 and lab 1.

The quality of analysis and writing is critical to your grade. Contrast A, B, C values based on their priorities. What do you conclude?

C.) Address Part 3) "Code Analysis" (quality of writing (content and form) is of utmost importance)

Code Analysis:

A.)You must explain why your code misses events and how the number of misses is related to **lambda** and **Mu.**
Explain also the variations of the response time and turn-around time.
Compare the values (missed events, average response time, and average turnaround time) for Lab3in comparison to lab1 and lab 2.
Compare the metrics for devices with highest and lowest priority.
Comment and explain the differences.

B.)Open question (needs some reading/research): Does your code guarantee that you will always BE PROCESSING the event with the highest priority among all stored events? Justify your response.

PART A: *State whether your code works.*
The code that we wrote works. It is also available to view at the end of this report.

PART B: Report/Analyze the results about the missed events (based on the filled table).
Q=2

NbrDevices	lambda	Mu	A_0 (%)	B_0 (s)	C_0 (s)	A_L (%)	B_L (s)	C_L (s)
2	2	30	4.000000%	0.001801	0.178947	10.000000%	0.001828	0.222222
2	2	90	34.000000%	0.002827	1.307692	43.000000%	0.003284	1.210526
4	2	30	1.000000%	0.002037	0.030303	1.000000%	0.002380	0.091837
4	2	60	5.000000%	0.002199	0.189474	9.000000%	0.002679	0.555556
4	2	90	16.000000%	0.002125	0.440476	20.000000%	0.002466	1.051948
8	4	30	0.000000%	0.004023	0.020000	0.000000%	0.004394	0.060000
8	6	60	2.000000%	0.005598	0.397959	9.000000%	0.007323	0.879121
8	6	90	6.000000%	0.005624	0.925532	32.000000%	0.009315	2.691176

Q=8

NbrDevices	lambda	Mu	A_0 (%)	B_0 (s)	C_0 (s)	A_L (%)	B_L (s)	C_L (s)
2	2	30	0.000000%	0.001862	0.191919	1.000000%	0.001766	0.393939
2	2	90	18.000000%	0.002381	1.456790	35.000000%	0.002799	1.938462
4	2	30	0.000000%	0.001751	0.050000	0.000000%	0.002141	0.121212
4	2	60	0.000000%	0.001750	0.260000	0.000000%	0.002092	0.676768
4	2	90	3.000000%	0.001927	0.567010	29.473682%	0.003198	1.359375
8	4	30	0.000000%	0.003479	0.020000	0.000000%	0.003669	0.060000
8	6	60	0.000000%	0.005010	0.460000	0.000000%	0.005435	1.400000
8	6	90	0.000000%	0.005034	1.070000	27.000002%	0.008111	5.671233

Note: Analysis of results are covered more thoroughly in Part C.

PART C: Code Analysis:

A.)*You must explain why your code misses events and how the number of misses is related to* ***lambda*** *and* ***Mu.***

Lambda – Events are generated on each device at an average inter-arrival (in seconds). Frequency of your events.

Mu – Each event requires an average service time (percentage of lambda). The required service time.

Explanation:

The reason why we are missing events is that when Mu is high this means that the required service time for each event will take longer. This correlates to the missed events as well as the low Lambda, which means each specified lambda that is when an event will occur seeing that we are giving low lambdas in respect of the high Mu specified this means that when we are servicing an event that has a high mu and a system that has a low lambda then the percentage of missed events will be higher. Since when we are servicing an event an interrupt occurs which makes the service take longer than it should this in turn makes the queue not de-queue and we miss events.

Explain also the variations of the response time and turn-around time.

Explanation of variations of the response time:

Response time = Time Stamp at First Response - Event Time Stamp.

The response time remains largely unaffected and you don't really see any huge fluctuations. From our results as mu changes the response time increases slightly.

Explanation of the variation of the turn-around time:

Turn-Around time = Time Stamp at End of Service – Event Time Stamp.

As mu increases, the turn-around time increases exponentially. It increases worst for the higher bit however since it has a lower priority and must wait it's turn to be processed, sometimes for much larger amounts of time.

Compare the values (missed events, average response time, and average turnaround time) for Lab3 in comparison to lab1 and lab 2.

Missed events:

lab 3 vs lab 1:

Lab3 with a queue size of 2 most closely resembled lab1 with the missed events for the lowest priority events.

lab 3 vs lab 2:

However, lab3's queue of 8 is much more closely related to lab2's missed events since whenever the mu became a high value missed events might occur. Generally, though both tended, so long as mu was low, to never miss events.

Average response time:

lab 3 vs lab 1:

Lab3 is faster compared to lab1. This is almost certainly due to the use of interrupts.

lab 3 vs lab 2:

Almost identical since they both use interrupts.

Average turnaround time:

lab 3 vs lab 1:

The average turnaround time is all around better. However, there could be that one event waiting to be serviced, on the back burner, that never gets serviced.

lab 3 vs lab 2:

The turnaround time on average for lab3 is better. Lab2 after all is Lab1 but with interrupts. We started to notice as the number of devices and lambda increased for a larger queue size the two began to almost become identical.

Compare the metrics for devices with highest and lowest priority. Comment and explain the differences.

Queue size 2:

Low vs High:

Response time:

The response time for the highest priority is better than the lowest since it takes precedence over the lowest.

Missed Events:

The number of missed events of the lowest priority would be more since the highest priority must process its events.

Turn Around Time:

The turn-around time of the highest priority is much better since it takes precedence over the lower priority.

Queue size 8:
Low vs High:
Response time:
The response time for the highest priority is better than the lowest since it takes precedence over the lowest.

Missed Events:
The number of missed events of the lowest priority would be more since the highest priority must process its events before the lowest.

Turn Around Time:
The turn-around time of the highest priority is much better since it takes precedence over the lower priority.

B.) Open question (needs some reading/research):
Does your code guarantee that you will always **BE PROCESSING** the event with the highest priority among all stored events? Justify your response.
No, since if a low priority task is currently being processed then a higher priority task occurs then it has to wait until the lower priority task finishes before it switches to the higher priority task.

Source Code lab3.c

```
/*****************************************************************************\
* Laboratory Exercises COMP 3510                                              *
* Author: Jordan Hutcheson                                                    *
* Author: Walter Conway                                                       *
* Date  : April 6, 2013                                                       *
\*****************************************************************************/
/*****************************************************************************\
*                        Global system headers                                *
\*****************************************************************************/
#include "common.h"
/*****************************************************************************\
*                        Global data types                                     *
\*****************************************************************************/
/*****************************************************************************\
*                        Global definitions                                   *
\*****************************************************************************/
#define MAX_QUEUE_SIZE 2 /* Max Queue Size */
#define MAX_LAB_DEVICE_SIZE 8 /*Max devices in lab exercise*/
/*****************************************************************************\
*                       Global data structures                                *
\*****************************************************************************/
struct node{
        Event event;
        struct node *next;
};

struct queue{
        struct node * tail;
        struct node * head;
        int size;
};
/*****************************************************************************\
*                          Global data                          *
\*****************************************************************************/
struct queue que0;
struct queue que1;
struct queue que2;
struct queue que3;
struct queue que4;
struct queue que5;
struct queue que6;
struct queue que7;


float totalServiceTime = 0 ;
float totalResponseTime = 0;
float totalTurnAroundTime = 0;
int totalEvents = 0;
int totalServicedEvents = 0;
int totalMissedEvents = 0;
int totalGeneratedEvents = 0;


struct queue* queueAddresses[8] = {&que0,&que1,&que2,&que3,&que4,&que5,&que6,&que7};
int numberOfDevices=0;
float totalResponseTimeByDevice[8] =  {0,0,0,0,0,0,0,0};
float totalTurnAroundTimeByDevice[8] = {0,0,0,0,0,0,0,0};
int totalEventsGeneratedByDevice[8] =  {0,0,0,0,0,0,0,0};
int totalMissedEventsByDevice[8] =  {0,0,0,0,0,0,0,0};
int totalEventServicedByDevice[8] = {0,0,0,0,0,0,0,0};
int nextEventThatShouldBeServiced[8] = {0,0,0,0,0,0,0,0};

/*****************************************************************************\
*                         Function prototypes                                 *
\*****************************************************************************/
void Control(void);
void InterruptRoutineHandlerDevice(void);
void BookKeeping();

void calculate(int deviceNumber, int eventNumber, float eventTime, float endOfServiceTime);
```

```
void enq(struct queue *q, Event *eventAddr);
Event *deq(struct queue *q);
int isEmpty(struct queue *q);
void create(void);

/*****************************************************************************\
* function: main()                                                            *
* usage:    Create an artificial environment for embedded systems. The parent *
*           process is the "control" process while children process will gene- *
*           generate events on devices                                        *
*****************************************************************************
* Inputs: ANSI flat C command line parameters                                 *
* Output: None                                                                *
*                                                                             *
* INITIALIZE PROGRAM ENVIRONMENT                                              *
* START CONTROL ROUTINE                                                       *
\*****************************************************************************/
int main (int argc, char **argv) {
  if (Initialization(argc,argv)){
    Control();
  }
} /* end of main function */

/*****************************************************************************\
 * Input : none                                                                *
 * Output: None                                                                *
 * Function: Monitor Devices and process events (written by students)          *
 \*****************************************************************************/
void Control(void){
//Local Variables
create();
Event* currentEvent;
float eventTime = 0;
float endOfServiceTime = 0;
int eventNumber = 0;
int deviceNumber = 0;
        while (1) {
                deviceNumber = 0;
                while(queueAddresses[deviceNumber]->size == 0){
                deviceNumber = (deviceNumber+1)%MAX_LAB_DEVICE_SIZE;
                }
                currentEvent =deq(queueAddresses[deviceNumber]);
                eventTime = currentEvent->When;
                eventNumber = currentEvent->EventID;
                Server(currentEvent);
                endOfServiceTime = Now();
                calculate(deviceNumber, eventNumber,eventTime, endOfServiceTime);
                }


}
/*****************************************************************************\
* Input : None                                                                *
* Output: None                                                                *
* Function: This routine is run whenever an event occurs on a device          *
*           The id of the device is encoded in the variable flag              *
\*****************************************************************************/
void InterruptRoutineHandlerDevice(void){
totalGeneratedEvents++;
float startTime = 0;
float eventTime = 0;
float reponseTime = 0;
Event *currEvent;
int tempFlags = Flags;
Flags = 0;
int deviceNumber = 0;
int eventNumber;
        while(tempFlags != 0){
                if(tempFlags & 1){
                        currEvent = &BufferLastEvent[deviceNumber];
                        eventTime = currEvent->When;
                        eventNumber = currEvent->EventID;
                        startTime = Now();
```

```
                                        DisplayEvent('A'+deviceNumber, currEvent);
                                        totalEventsGeneratedByDevice[deviceNumber]++;
                                        reponseTime = (startTime-eventTime);
                                        totalResponseTimeByDevice[deviceNumber] += reponseTime;
                                        totalResponseTime += reponseTime;
                                        fprintf(stderr,"The queue for Device %c is: %d\n", 'A'+deviceNumber ,queueAddresses[deviceNumber]-
>size);
                                        enq(queueAddresses[deviceNumber], currEvent);

                                }
                tempFlags >>= 1;
                deviceNumber++;

                }

//This finds out how many devices there are
//throughout the life cycle of the program.
if(deviceNumber > numberOfDevices) {
                numberOfDevices = deviceNumber;
}
}

/*************************************************************************\
* Input : None                                                            *
* Output: None                                                            *
* Function: This must print out the number of Events buffered not yet     *
*         not yet processed (Server() function not yet called)            *
\*************************************************************************/
void BookKeeping(void){
fprintf(stderr,"Total Generated Events: %d\n", totalGeneratedEvents);
fprintf(stderr,"Total Unserviced Events: %d\n", totalMissedEvents);
fprintf(stderr,"Total Serviced Events: %d\n", totalServicedEvents);

fprintf(stderr,"Average percentage of missed events: %10.6f%\n", ((float)totalMissedEvents/totalGeneratedEvents)*100);
fprintf(stderr,"Average response time: %10.6f\n",totalResponseTime/totalServicedEvents);
fprintf(stderr,"Average turn around time: %10.6f\n",totalTurnAroundTime/totalServicedEvents);
int i;
                for(i = 0; i < (numberOfDevices +1); i++) {
                if(i == 0 || i == numberOfDevices-1){
                                fprintf(stderr,"----------Information for Device: %d----------\n",i);
                                fprintf(stderr,"Total Generated Events: %d\n", totalEventsGeneratedByDevice[i]);
                                fprintf(stderr,"Total Unserviced Events: %d\n", totalMissedEventsByDevice[i]);
                                fprintf(stderr,"Total Serviced Events: %d\n", totalEventServicedByDevice[i]);
                                fprintf(stderr,"Average percentage of missed events: %10.6f%\n",
((float)totalMissedEventsByDevice[i]/totalEventsGeneratedByDevice[i])*100);
                                fprintf(stderr,"Average response time: %10.6f\n",totalResponseTimeByDevice[i]/totalEventServicedByDevice[i]);
                                fprintf(stderr,"Average turn around time: %10.6f\n",totalTurnAroundTimeByDevice[i]/totalEventServicedByDevice[i]);
                                }
                }
}

/* Create an empty queue */
void create()
{
                que0.size = 0;
                que0.head = NULL;
                que0.tail = NULL;

                que1.size = 0;
                que1.head = NULL;
                que1.tail = NULL;

                que2.size = 0;
                que2.head = NULL;
                que2.tail = NULL;

                que3.size = 0;
                que3.head = NULL;
                que3.tail = NULL;

                que4.size = 0;
                que4.head = NULL;
                que4.tail = NULL;
```

```
	que5.size = 0;
	que5.head = NULL;
	que5.tail = NULL;

	que6.size = 0;
	que6.head = NULL;
	que6.tail = NULL;

	que7.size = 0;
	que7.head = NULL;
	que7.tail = NULL;
}

/* En-queuing the queue */
void enq(struct queue *q, Event *eventAddr)
{

	if((q->size) < MAX_QUEUE_SIZE){

		struct node * newPtr;
		newPtr = (struct node *)malloc(1*sizeof(struct node));

		if (newPtr != NULL) {
			newPtr->event = *eventAddr;
			newPtr->next = NULL;
			if(isEmpty(q))
			{
				q->head = newPtr;
			} else {
					q->tail->next = newPtr;
				}
			q->tail = newPtr;
			q->size++;

			} else {
				printf("not inserted\n");
			}

} else{
fprintf(stderr, "missed one\n");
}

}



/* Dequeing the queue */
Event *deq(struct queue *q)
{
Event *tempEvent;
tempEvent = (struct EventTag *)malloc(sizeof(struct EventTag));
struct node * tempPtr;
*tempEvent = q->head->event;
tempPtr = q->head;
q->head = q->head->next;
if(q->head == NULL){
	q->tail = NULL;
}
free (tempPtr);
q->size--;
return tempEvent;
}

int isEmpty(struct queue *q){
return q->head == NULL;
}


void calculate(int deviceNumber, int eventNumber, float eventTime, float endOfServiceTime){
int missedEvents=0;
int turnAroundTime = 0;
//This finds the number of missed events since last serviced events of the particular device
```

```
//and updates two variables: Total missed and total missed by device.
if((nextEventThatShouldBeServiced[deviceNumber]) != eventNumber) {
                missedEvents = eventNumber - (nextEventThatShouldBeServiced[deviceNumber]);
                totalMissedEvents += missedEvents;
                totalMissedEventsByDevice[deviceNumber] += missedEvents;
}

nextEventThatShouldBeServiced[deviceNumber] = (eventNumber) + 1;
totalEventServicedByDevice[deviceNumber]++;
turnAroundTime = endOfServiceTime - (eventTime);
totalTurnAroundTimeByDevice[deviceNumber] += turnAroundTime;

totalServicedEvents++;
totalTurnAroundTime += turnAroundTime;

}
```