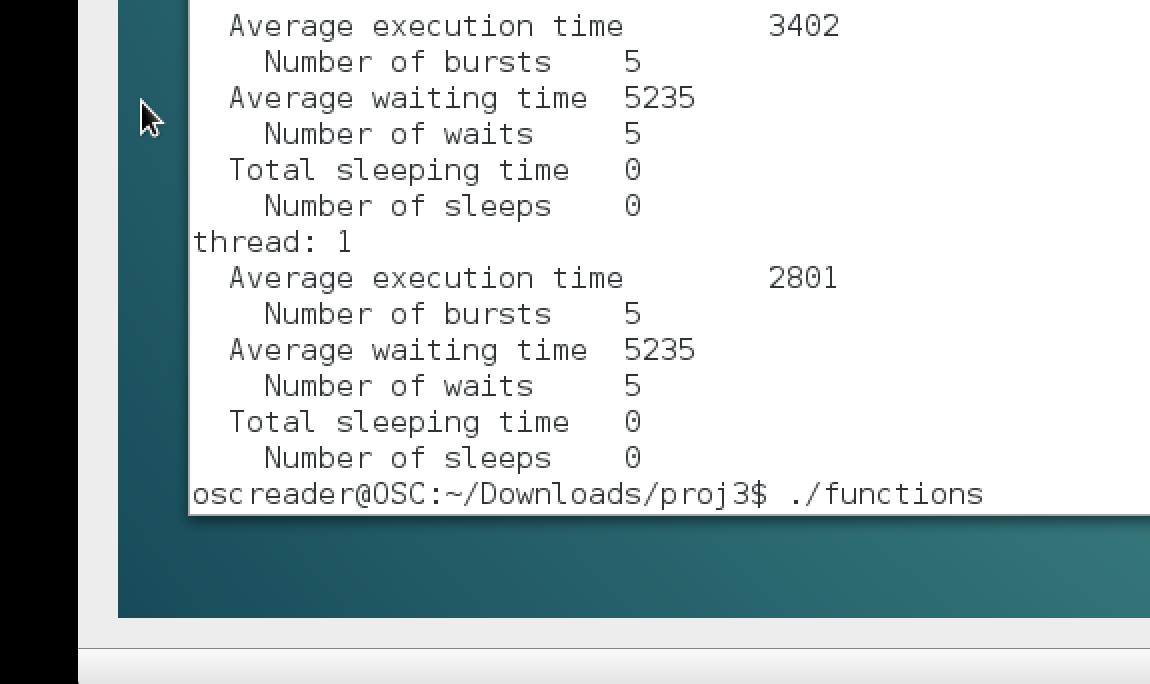
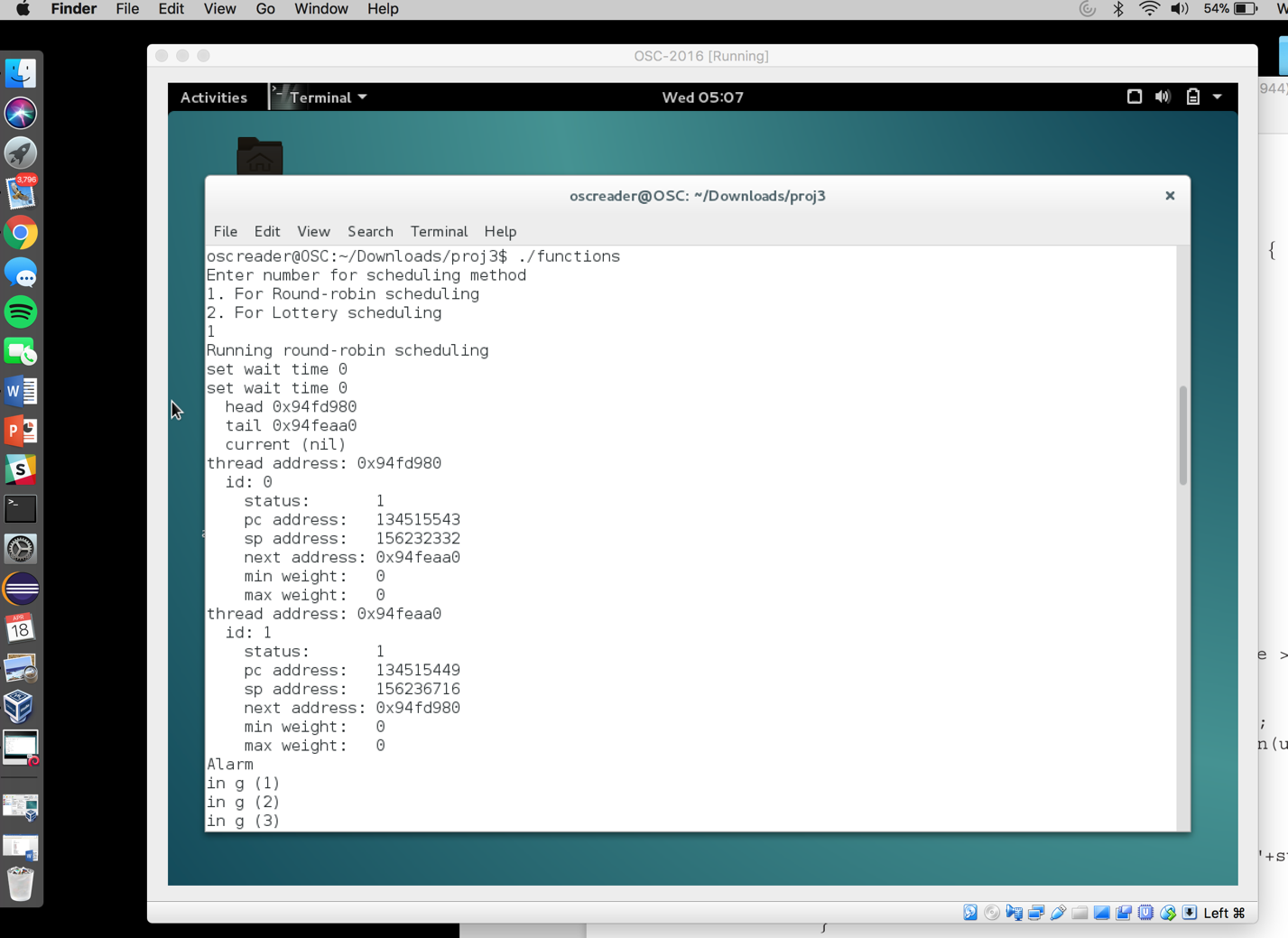
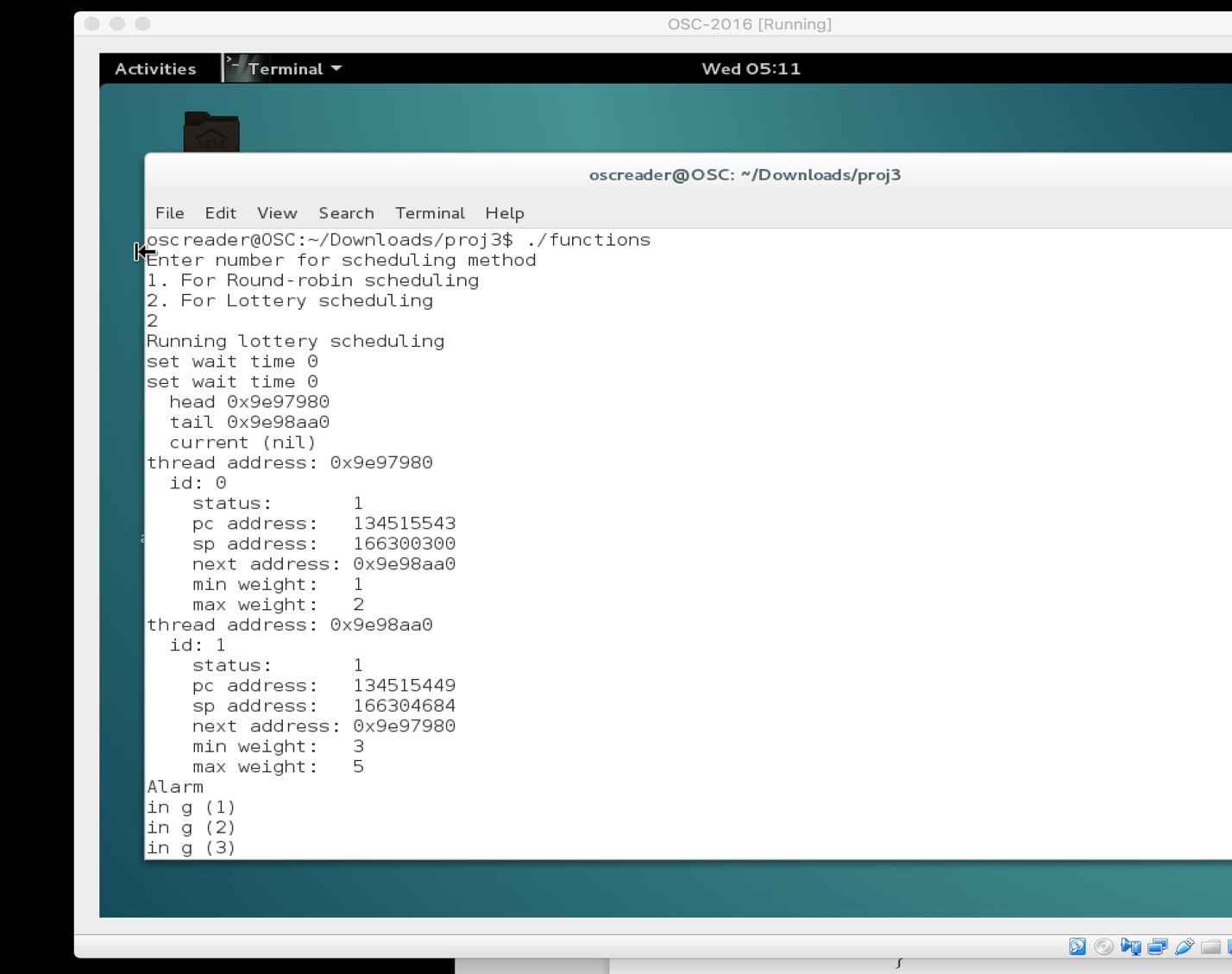
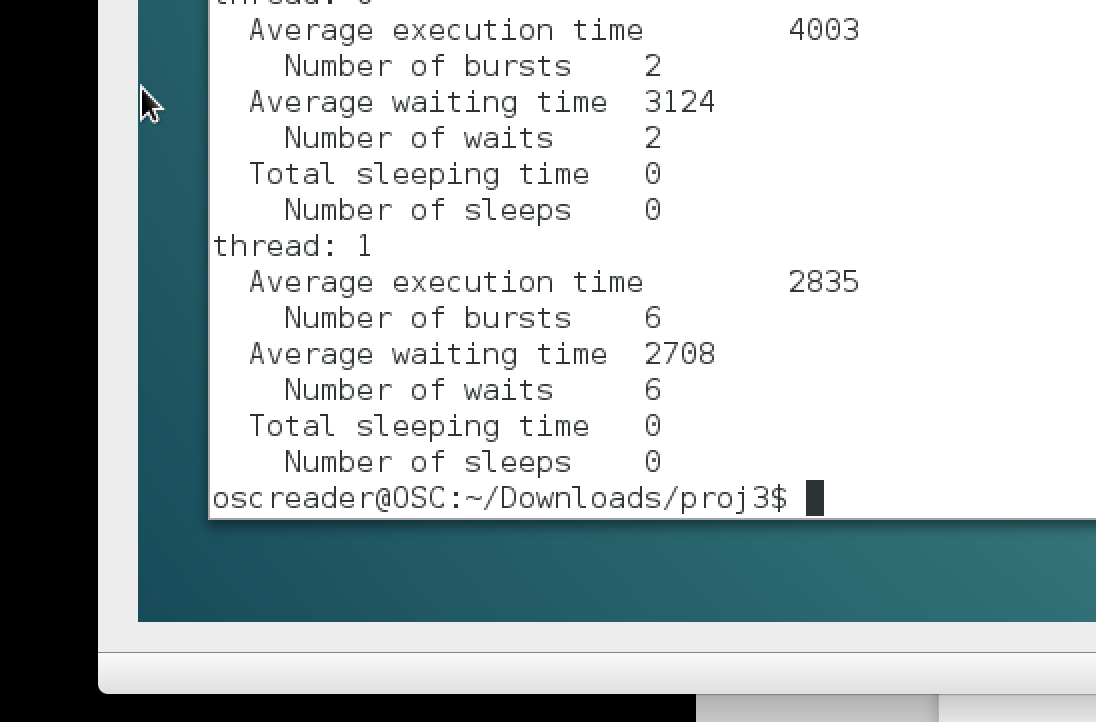
Assignment #3 – User Level Threading

(This report contains: source code, make-file code and necessary screenshots.)

**Screenshots:** 





**source code:**

/\*\*

\* Assignment 3 - User Level Threading

\* @author: Ali H. Iqbal

\*\*/

#include <stdio.h>

#include <setjmp.h>

#include <signal.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/time.h>

#include <math.h>

#include <sys/timeb.h>

#include <setjmp.h>

#include <time.h>

// defined constants

#define JB\_SP 6

#define JB\_PC 7

#define SECOND 1000000

#define MAX\_NO\_OF\_THREADS 100

#define STACK\_SIZE 4096

#define STATUS\_SLEEPING 0

#define STATUS\_READY 1

#define STATUS\_SUSPENDED 2

#define STATUS\_RUNNING 3

#define TIME\_QUANTUM 1\*SECOND

static int run\_time = 15000; /// max instructions for thread

static int curr\_schedule = 1; // 1 for round robin, 2 for lottery scheduling

// global

static int num\_of\_threads = 0;

static int curr\_thread\_count = 0;

static int weight\_total = 1;

struct timeb t\_start, t\_stop;

typedef unsigned long address\_t;

//thread weight for lottery scheduling

typedef struct thread\_weight

{

int min\_weight;

int max\_weight;

}thread\_weight;

// thread sleeping time

typedef struct sleep\_info

{

int start\_sleeping;

int sleep\_to;

struct timeb start\_s;

int total;

}sleep\_struct;

// thread waiting time

typedef struct wait\_info

{

int start\_waiting;

int stop\_waiting;

struct timeb start\_w, stop\_w;

int total;

}wait\_struct;

// thread control block

typedef struct TCB

{

address\_t pc;

address\_t sp;

thread\_weight weight;

sleep\_struct sleep\_time;

wait\_struct wait\_time;

sigjmp\_buf jbuf;

int thrd\_id;

int thrd\_status;

int num\_bursts;

int num\_waits;

int num\_sleeps;

int exec\_time;

struct TCB \*next;

}TCB;

TCB \*head\_list = NULL;

TCB \*tail\_list = NULL;

TCB \*curr\_thread = NULL;

//functions

void appendList(TCB \*new\_tcb);

void dispatch(int sig);

void printList();

void yieldCPU(void);

void status(TCB \*current);

void CleanUp();

void SleepThread(int sec);

int GetId();

void f( void );

void g( void );

int createThread (void (\*f) (void));

TCB \*findTcb(int random\_number);

void go();

#ifdef \_\_x86\_64\_\_

address\_t tr\_address(address\_t addr)

{

address\_t ret;

asm volatile("xor %%fs:0x30,%0\n"

"rol $0x11,%0\n"

: "=g" (ret)

: "0" (addr));

return ret;

}

#else

#define JB\_SP 4

#define JB\_PC 5

address\_t tr\_address(address\_t addr)

{

address\_t ret;

asm volatile("xor %%gs:0x18,%0\n"

"rol $0x9,%0\n"

: "=g" (ret)

: "0" (addr));

return ret;

}

#endif

//function used to get status of thread, accepts TCB thread pointer as parameter

void status(TCB \*current)

{

int ex\_avg;

int wait\_avg;

int sleep\_total;

printf("thread: %d\n", current->thrd\_id);

if(current->num\_bursts > 0)

{

ex\_avg = (current->exec\_time)/(current->num\_bursts);

}

else

{

ex\_avg = 0;

}

if(current->num\_waits > 0)

{

wait\_avg = (current->wait\_time.total)/(current->num\_waits);

}

else

{

wait\_avg = 0;

}

if(current->num\_sleeps > 0)

{

sleep\_total = (current->sleep\_time.total);

}

else

{

sleep\_total = 0;

}

printf(" Average execution time\t%d\n", ex\_avg);

printf(" Number of bursts\t%d\n", current->num\_bursts);

printf(" Average waiting time\t%d\n", wait\_avg);

printf(" Number of waits\t%d\n", current->num\_waits);

printf(" Total sleeping time\t%d\n", sleep\_total);

printf(" Number of sleeps\t%d\n", current->num\_sleeps);

}

//function prints info, removes list and exits program

void CleanUp()

{

TCB \*index = head\_list;

int count = 0;

//suspend threads

while(count < num\_of\_threads)

{

index->thrd\_status = STATUS\_SUSPENDED;

status(index);

index = index->next;

count++;

}

index = head\_list;

TCB \*next;

while(index != tail\_list)

{

next = index->next;

free(index);

index = next;

}

exit(0);

// exits

}

//function puts thread to sleep for given time

void SleepThread(int sec)

{

printf(" SLEEPING\n");

struct timeb temp\_time;

ftime(&temp\_time);

curr\_thread->num\_sleeps++;

curr\_thread->sleep\_time.start\_s = temp\_time;

curr\_thread->sleep\_time.sleep\_to = temp\_time.millitm + sec;

curr\_thread->thrd\_status = STATUS\_SLEEPING;

yieldCPU();

}

//fuction to return thread id

int GetId()

{

return curr\_thread->thrd\_id;

}

// function handles thread context switching

void yieldCPU(void)

{

printf("switching threads\n");

ftime(&t\_stop);//stop exectuting

curr\_thread->exec\_time += ( 1000.0 \* (t\_stop.time - t\_start.time) + (t\_stop.millitm - t\_start.millitm));

// print thread info

printf(" Current Thread: %d\n", curr\_thread->thrd\_id);

printf(" Thread Status: %d\n", curr\_thread->thrd\_status);

printf("Execution Time: %d\n", curr\_thread->exec\_time);

usleep(2\*SECOND);//pauses

raise(SIGVTALRM);

}

//function represents thread f

void f(void)

{

int i=0;

while(1)

{

++i;

printf("in f (%d)\n",i);

if (i % 3 == 0)

{

yieldCPU();

}

usleep(SECOND);

}

}

//function represents thread g

void g( void )

{

int i = 0;

while(1)

{

++i;

printf("in g (%d)\n",i);

if (i % 3 == 0)

{

yieldCPU();

}

usleep(SECOND);

}

}

//function create threads along with given arguments

int createThread (void (\*f) (void))

{

TCB\* curr\_tcb = malloc(sizeof(TCB));

if(curr\_tcb == NULL)

{

curr\_tcb->thrd\_id = -1;

num\_of\_threads++;

}

else

{

curr\_tcb->thrd\_id = curr\_thread\_count++;

curr\_tcb->pc = (address\_t)f;

curr\_tcb->sp = (address\_t)malloc(STACK\_SIZE);

curr\_tcb->sp = curr\_tcb->sp + STACK\_SIZE - sizeof(address\_t);

curr\_tcb->num\_bursts = 0;

curr\_tcb->num\_waits = 0;

curr\_tcb->num\_sleeps = 0;

curr\_tcb->exec\_time = 0;

curr\_tcb->sleep\_time.sleep\_to = 0;

curr\_tcb->sleep\_time.start\_sleeping = 0;

curr\_tcb->sleep\_time.total = 0;

curr\_tcb->wait\_time.start\_w.millitm = 0;

curr\_tcb->wait\_time.stop\_w.millitm = 0;

curr\_tcb->wait\_time.total = 0;

printf("set wait time %d\n", curr\_tcb->wait\_time.total);

curr\_tcb->next = NULL;

curr\_tcb->weight.min\_weight = 0;

curr\_tcb->weight.max\_weight = 0;

struct timeb temp\_time;

ftime(&temp\_time);

curr\_tcb->wait\_time.start\_w = temp\_time;

curr\_tcb->thrd\_status = STATUS\_READY;

num\_of\_threads++;

// calculate weight for RR

if(curr\_schedule == 2)

{

curr\_tcb->weight.min\_weight = weight\_total;

int exponent = curr\_tcb->thrd\_id;

curr\_tcb->weight.max\_weight = weight\_total + pow(2,exponent);

weight\_total = curr\_tcb->weight.max\_weight;

weight\_total++;

}

// threads exceeded

if(num\_of\_threads >= MAX\_NO\_OF\_THREADS)

{

CleanUp();

}

}

sigsetjmp(curr\_tcb->jbuf,1);

(curr\_tcb->jbuf->\_\_jmpbuf)[JB\_SP] = tr\_address(curr\_tcb->sp);

(curr\_tcb->jbuf->\_\_jmpbuf)[JB\_PC] = tr\_address(curr\_tcb->pc);

sigemptyset(&curr\_tcb->jbuf->\_\_saved\_mask);

appendList(curr\_tcb);

return curr\_tcb->thrd\_id;

}

// find thread with chosen "lottery ticket"

TCB \*findTcb(int random\_number)

{

TCB \*index = head\_list;

int count = 0;

while((count < num\_of\_threads))

{

if((random\_number >= (index->weight.min\_weight)) && (random\_number <= (index->weight.max\_weight)))

{

break;

}

index = index->next;

count++;

}

if(count == num\_of\_threads)

{

return NULL;

}

else

{

return index;

}

}

//function to check sleeping threads which will get woken up

void sleeping\_threads()

{

TCB \*index = head\_list;

int count = 0;

while(count < num\_of\_threads)

{

struct timeb temp\_time;

ftime(&temp\_time);

if((index->thrd\_status == STATUS\_SLEEPING) && (temp\_time.time > curr\_thread->sleep\_time.sleep\_to))

{

curr\_thread->thrd\_status = STATUS\_READY;

curr\_thread->wait\_time.start\_w = temp\_time;

curr\_thread->sleep\_time.sleep\_to = 0;

curr\_thread->sleep\_time.total += ( 1000.0 \* (temp\_time.time - curr\_thread->sleep\_time.start\_s.time ) + (temp\_time.millitm - curr\_thread->sleep\_time.start\_s.millitm));

}

index = index->next;

count++;

}

}

//function which runs thread scheduler and dispach a signal

void dispatch(int sig)

{

sleeping\_threads();

printf("Alarm\n");

//Round-robin scheduling

if(curr\_schedule == 1)

{

if(curr\_thread == NULL)

{

curr\_thread = head\_list;

head\_list->thrd\_status = STATUS\_RUNNING;

ftime(&t\_start);

curr\_thread->wait\_time.stop\_w = t\_start;

curr\_thread->wait\_time.total += ( 1000.0 \* (curr\_thread->wait\_time.stop\_w.time - curr\_thread->wait\_time.start\_w.time) + (curr\_thread->wait\_time.stop\_w.millitm - curr\_thread->wait\_time.start\_w.millitm));

siglongjmp(head\_list->jbuf, 1);

}

else

{

if( (curr\_thread->exec\_time) > run\_time )

{

CleanUp();

}

if(sigsetjmp(curr\_thread->jbuf, 1) == 1)

{

ftime(&t\_start);

return;

}

struct timeb temp\_time;

ftime(&temp\_time);

curr\_thread->thrd\_status = STATUS\_READY;

curr\_thread->wait\_time.start\_w = temp\_time;

curr\_thread = curr\_thread->next;

while(curr\_thread->thrd\_status != STATUS\_READY)

curr\_thread= curr\_thread->next;

curr\_thread->thrd\_status = STATUS\_RUNNING;

ftime(&t\_start);

curr\_thread->wait\_time.stop\_w = t\_start;

if(curr\_thread->wait\_time.stop\_w.millitm != 0)

{

curr\_thread->wait\_time.total+= ( 1000.0 \* (curr\_thread->wait\_time.stop\_w.time - curr\_thread->wait\_time.start\_w.time) + (curr\_thread->wait\_time.stop\_w.millitm - curr\_thread->wait\_time.start\_w.millitm));

curr\_thread->num\_waits++;

}

curr\_thread->num\_bursts++;

siglongjmp(curr\_thread->jbuf, 1);

}

} // Lottery scheduling

else if (curr\_schedule == 2)

{

if(curr\_thread == NULL)

{

curr\_thread = head\_list;

head\_list->thrd\_status = STATUS\_RUNNING;

ftime(&t\_start);

curr\_thread->wait\_time.stop\_w = t\_start;

curr\_thread->wait\_time.total += ( 1000.0 \* (curr\_thread->wait\_time.stop\_w.time - curr\_thread->wait\_time.start\_w.time) + (curr\_thread->wait\_time.stop\_w.millitm - curr\_thread->wait\_time.start\_w.millitm));

siglongjmp(head\_list->jbuf, 1);

}

else

{

if( (curr\_thread->exec\_time) > run\_time )

{

CleanUp();

}

if(sigsetjmp(curr\_thread->jbuf, 1) == 1)

{

return;

}

curr\_thread->thrd\_status = STATUS\_READY;

struct timeb start\_wait\_time;

ftime(&start\_wait\_time);

curr\_thread->wait\_time.start\_w = start\_wait\_time;

int mod\_value = weight\_total - 1;

int chosen\_number = ( rand() % mod\_value )+ 1;

TCB \*selected\_thread = NULL;

do

{

selected\_thread = findTcb(chosen\_number);

chosen\_number = ( rand() % mod\_value ) + 1;

}while(selected\_thread->thrd\_status != STATUS\_READY);

curr\_thread = selected\_thread;

ftime(&t\_start);

curr\_thread->wait\_time.stop\_w = t\_start;

if(curr\_thread->wait\_time.stop\_w.millitm != 0)

{

curr\_thread->wait\_time.total += ( 1000.0 \* (curr\_thread->wait\_time.stop\_w.time - curr\_thread->wait\_time.start\_w.time) + (curr\_thread->wait\_time.stop\_w.millitm - curr\_thread->wait\_time.start\_w.millitm));

curr\_thread->num\_waits++;

}

curr\_thread->thrd\_status = STATUS\_RUNNING;

curr\_thread->num\_bursts++;

siglongjmp(curr\_thread->jbuf, 1);

}

}

}

//function to print the list

void printList()

{

TCB \*index;

if(head\_list == NULL)

{

printf("\*Empty list \n");

}

printf(" head %p\n", head\_list);

printf(" tail %p\n", tail\_list);

printf(" current %p\n", curr\_thread);

int i = 0;

for(index = head\_list; i < num\_of\_threads; index = index->next)

{

printf("thread address: %p\n", index);

printf(" id: %d\n", index->thrd\_id);

printf(" status: %d\n", index->thrd\_status);

printf(" pc address: %lu\n", index->pc);

printf(" sp address: %lu\n", index->sp);

printf(" next address: %p\n", index->next);

printf(" min weight: %d\n", index->weight.min\_weight);

printf(" max weight: %d\n", index->weight.max\_weight);

i++;

}

}

//function add/append newly created thread

void appendList(TCB \*new\_tcb)

{

if( head\_list == NULL)

{

head\_list = new\_tcb;

tail\_list = new\_tcb;

new\_tcb->next = head\_list;

}

else

{

TCB \*index;

for( index = head\_list; index != tail\_list; index = index->next);

index->next = new\_tcb;

tail\_list = new\_tcb;

new\_tcb->next = head\_list;

}

}

//function gets called by main, creates threads

void go()

{

signal(SIGVTALRM, dispatch);

srand(time(NULL));

struct itimerval tv;

tv.it\_value.tv\_sec = 2; //first timer

tv.it\_value.tv\_usec = 0;

tv.it\_interval.tv\_sec = 2; //all timers but first one

tv.it\_interval.tv\_usec = 0;

setitimer(ITIMER\_VIRTUAL, &tv, NULL);

CreateThread(g);

CreateThread(f);

printList();

while(1);

}

//main function chooses scheduling method

//return int

int main()

{

printf("Enter number for scheduling method\n");

printf("1. For Round-robin scheduling\n");

printf("2. For Lottery scheduling\n");

scanf("%d",&curr\_schedule);

if(curr\_schedule == 1){

printf("Running round-robin scheduling\n");

} else if(curr\_schedule == 2){

printf("Running lottery scheduling\n");

} else {

curr\_schedule = 1;

printf("Running round-robin scheduling\n");

}

go();

return 0;

}

**make-file code:**

CC = gcc

SRC1 = demo.c

SRC2 = timer.c

SRC3 = functions.c

TRGT1 = demo

TRGT2 = timer

TRGT3 = functions

CFLAGS = -lm

all: T1 T2 T3

T1:

$(CC) $(SRC1) -o $(TRGT1)

T2:

$(CC) $(SRC2) -o $(TRGT2)

T3:

$(CC) $(SRC3) $(CFLAGS) -o $(TRGT3)

clean:

@rm $(TRGT1) $(TRGT2) $(TRGT3)