Light Weight User-Level Thread Scheduling System

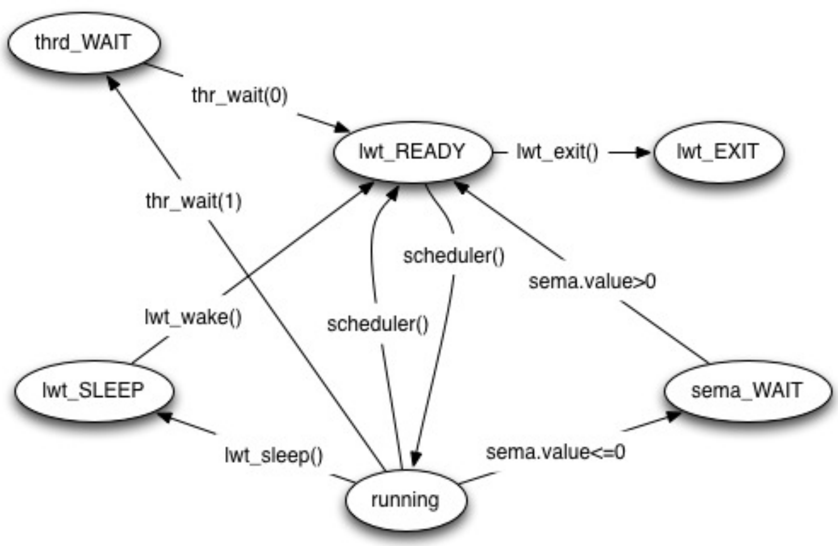
Hanying Huang

1. Introduction

The purpose of this project is to implement a lightweight user-level thread scheduling system without the need to modify the Linux kernel. The system allows the creation and concurrent execution of threads of control within a Linux task, in a true preemptive manner. And the LWT system is capable of creating threads and performing mini context switches to share the CPU time among the task threads, based on the round-robin scheduling algorithm.

2. Design

a) State Change of a Thread



b) Primitives

```
int lwt_init(int sec, int count);       //allocate and initialize the required data structure and setup the initial thread
int lwt_create(void (*fn)(void*),int argc, void *arg, char *name, int waitingflag);  //create a new thread
void scheduler(int sig);                //scheduling the threads, context switch
void P(sema *s);                        //decrease the semaphore's value to synchronize the threads.
void V(sema *s);                        //increase the semaphore's value.
void lwt_sleep(double sleepTimeSec);    //let the running thread sleep for specified seconds
void sema_init(int buffersize);         //initiate the semaphores
void lwt_exit();                        //exit the thread
```

Other details:

- scheduler(int sig) – If sig is not 0, it is ualarm signal to call the scheduler.
- lwt_init(int sec, int count) – **sec**: time quantum for scheduling, **count**: the number of threads need to be created(not including the child threads)
- **int waitingflag** in lwt_create() indicates whether the thread is a sub thread which means other thread is waiting for its termination to continue.

c) Structures

i. Thread

```
struct thread_t {
int thr_id;
char *name;
int state;              //thread's state:0-exit  1-lwt_READY  2-thrd_WAIT  3-sem_WAIT  4-lwt_SLEEP
int wait_count;         //the amount of threads of which it is waiting of the termination
int argc;               //count of arguments
jmp_buf env;            //context
uchar *sp;              //stack pointer
void    (*fn)(void*);   //routine
void    *arg;           //argument
thread_t *waiting_t;    //the thread which is waiting for its termination
thread_t *previous;     //the previous thread in the link list
thread_t *next;         //the next thread in the link list
long sleepTime;         //remain sleeping time
};
```

ii. Semaphore

```
struct semaphore{
        int value;
        link_list *sem_q;
};
```

All threads, which are waiting for the semaphore, will be added to sem_q.

iii. Data Structures

```
struct thr_link{        //link list
        thread_t *first;
        thread_t *last;
};
```

3. Algorithm

The system performs context switches based on the round-robin scheduling algorithm.

A small unit of time, called a time quantum, is defined. The ready queue is treated as a circular queue. The scheduler goes around the ready queue, switch context to next ready thread for a time interval of up to 1 time quantum.

The thread may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next thread in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt. A context switch will be executed and the process will be put at the tail of the ready queue. The scheduler will then select the next thread in the ready queue.

In the RR scheduling algorithm, no thread is allocated the shared CPU for more than 1 time quantum in a row. If a thread burst exceeds 1 time quantum, that thread is preempted and is put back in the ready queue.

4. Implementation

a) Link List

```
thr_wait_q=(link_list *)malloc(sizeof(link_list));
ready_q=(link_list *)malloc(sizeof(link_list));
sleep_q=(link_list *)malloc(sizeof(link_list));
```

There are three link lists in the system which hold threads in different states. thr_wait_q holds threads which are in thrd_WAIT state. ready_q holds threads in lwt_READY state and sleep_q holds sleeping threads whose state is lwt_SLEEP.

b) Stack Allocation

```
thread_t *thr;
thr=(thread_t *)malloc(sizeof(thread_t));
thr->thr_id=++idgen;
thr->name=name;
thr->state=1;
thr->wait_count=0;
thr->sp=(uchar *)(malloc(REQUIRED_STACK)+REQUIRED_STACK-64);
```

After initiating the structure for the thread, use malloc to allocate the required stack (16KB) for each thread. The important thing is to set the right pointer to the top of the stack, because the growth direction of the stack in Linux is from top to bottom.

```
//read,write sp
asm ("movq %%rsp, %0;" :"=r"(pre_sp) );
asm ("movq %0, %%rsp;"
:
:"r"(main_thr->sp)
:"%rsp"
);
```

After getting the stack pointer, use assembly language to read previous sp from and write new sp to the register rsp. The previous sp will be used while exit the process

c) SIGALARM Handler

```
quant=x;
requiredThr=y;
signal(SIGALRM,scheduler);
ualarm(quant,quant);
```

quant is the time quantum for scheduling based on round-robin algorithm.

d) Context Switch

```
if(running_thr&&running_thr->name!="Main")//save context
{
        if(sig){ //sig!=0:ualarm signal, add running thread to ready queue
                running_thr->state=1;
                addThr(ready_q, running_thr);
                printf("Add a thr to ready q:%s%d\n",
                ready_q->last->name, ready_q->last->thr_id);
        }
        printf("Save context for %s%d\n", running_thr->name, running_thr->thr_id);

        temp=sigsetjmp (running_thr->env,1); //save context

}
```

The above code section is for saving the context of the running thread.

```
if(running_thr&&temp){// temp!=0:called by longjmp, returned to the saved context
                printf("here!\n");
                return;
}

if(!ready_q->first){//no other ready thread
        printf("Empty ready q\n");
        if(!sleep_q->first){
                lwt_exit(); //also no sleeping thread, exit the process
        }
        else{ //continue sleeping
                onlySleep=1;
                running_thr=nil;
                scheduler(0);
        }
}
else { //jmp to next ready thread
        running_thr=ready_q->first;
        printf("Current running thread: %s%d\n",running_thr->name,running_thr->thr_id);
        delThr(ready_q,ready_q->first);
        siglongjmp (running_thr->env,1);
}
```

This section shows the use of siglongjmp for switching to the next ready thread if there is any.

e) Thread Wait

```
void thrd_wait(int isWaiting, thread_t *thr ){
        if(isWaiting) {//need to wait
                (thr->wait_count)++;
                thr->state=2;
                addThr(thr_wait_q,thr);
                printf("add %s %d to wait q\n",thr->name,thr->thr_id);
        }
        else {//one child thread is terminated
                (thr->wait_count)--;
                if(!thr->wait_count) {//no more child threads
                        delThr(thr_wait_q,thr);
                        printf("Sub threads are done, move to ready q:%s %d\n\n",thr->name,thr->thr_id);
                        thr->state=1;
                        addThr(ready_q,thr);
                }
        }
}
```

If the value of isWaiting is not 0, the thread needs to wait other thread. Or it means the child thread is terminated and the thread can be added to ready queue for execution. This primitive will be invoked when the child thread is exiting shown in the following figure.

```
printf("\n*************start exiting %s%d***************\n",
while(running_thr->waiting_t){
        isChild=1;
        thrd_wait(0,running_thr->waiting_t);
        running_thr->waiting_t=nil;
}
```

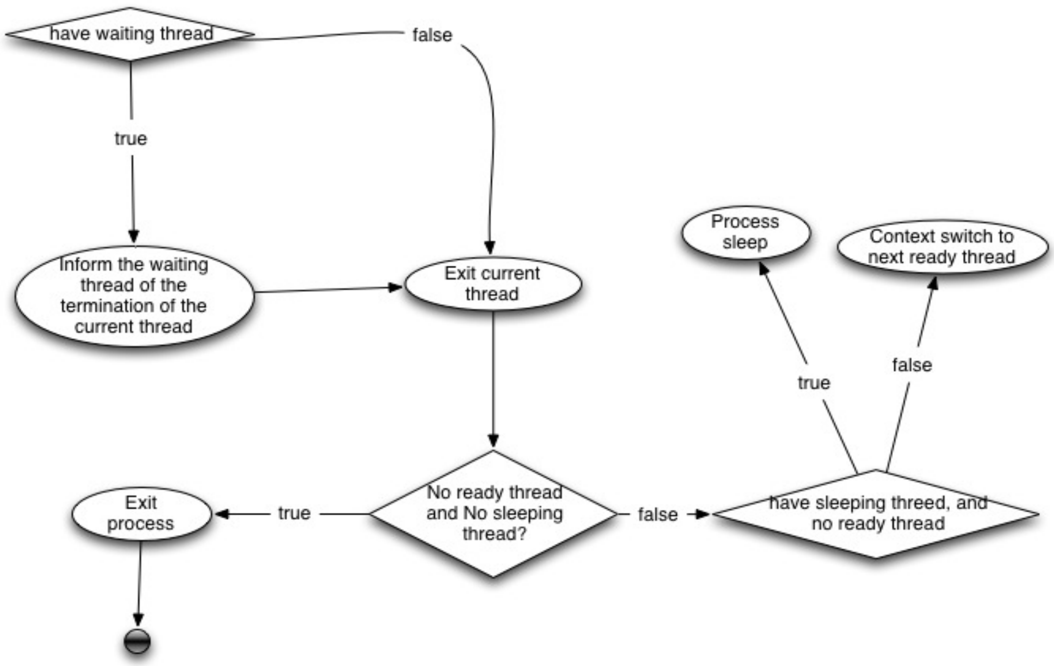
f) Sleeping Time Counting

```
if(flag) //get time
{
        gettimeofday(&tv_begin, NULL);flag--;
}
else
{
        gettimeofday(&tv_end, NULL);flag++;
}
//calculate time span from last switch to this switch in micro secs
rs = abs(1000000 * (tv_end.tv_sec - tv_begin.tv_sec) + tv_end.tv_usec - tv_begin.tv_usec);
if(sleepingCount){ //check sleeping threads
        updateSleep();
        while(onlySleep){ //no ready thread, skip switching
                signal(SIGALRM, SIG_IGN); //ignore the sigalrm
                usleep(minSleep); //sleep for the minimum sleeping time in all sleeping threads in us
                rs=minSleep;
                updateSleep();
                signal(SIGALRM, scheduler); //reinstall the sigalrm
                ualarm(quant,quant);
        }
        //restore the time
        gettimeofday(&tv_begin, NULL);
        gettimeofday(&tv_end, NULL);

}
```

It is very likely that there is no ready thread while there are some sleeping threads. In this situation, the process will continue sleeping for **minSleep microseconds**. minSleep is the minimum sleeping time in all sleeping threads. Then the time is updated and the thread with minimum sleeping time will wake up and the process will also wake up. In this situation, the variation from the accurate waking time is pretty small. While there is any running thread, the **maximum** variation from the accurate waking time for a sleeping thread is **one quantum** time.

g) lwt_exit() System Call

When the lwt_exit() is called, different conditions will be checked shown as the following figure.



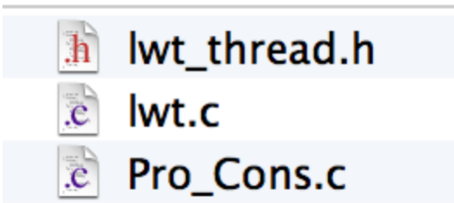
h) Synchronization

```
void P(sema *s){
        if(s->value>0){
                --(s->value);
        }
        else
        {
                if(s->value<=0){//block
                        running_thr->state=3;
                        addThr(s->sem_q,running_thr);
                        printf("wait in sema:%s id:%d\n",s->sem_q->last->name, s->sem_q->last->thr_id);
                        scheduler(0);
                        P(s);
                }
        }
}

void V(sema *s){
        s->value++;
        if((s->value>0)&&(s->sem_q->first)){//unblock the waiting thread

                thread_t *wait_sema=s->sem_q->first;
                printf("%s%d\n",s->sem_q->first->name,s->sem_q->first->thr_id);
                delThr(s->sem_q,s->sem_q->first);
                wait_sema->state=1;
                addThr(ready_q,wait_sema);
                printf("unblock sem wait t:%s id:%d\n",ready_q->last->name,ready_q->last->thr_id);

        }
}
```

The system uses global semaphores to synchronize the threads. In the sample program, there is a bounded buffer that the producers create an integer and save it into the buffer while the consumers read/consume an integer from the buffer. The semaphores can synchronize the remaining items in the buffer. P() is for decreasing semaphore's value and block the current thread if the value is not positive. V() is for increasing semaphore's value and unblock one thread from the waiting queue if there is any.

5. Result

a) Files



b) Creating Threads

```
****************create thread, name:Main  id:1**********
Save context for main thread!

************Context Switch***********
continue creating
Initiate Semaphore!

****************create thread, name:Producer  id:2**********
Pro_Sema producer:7
Con_Sema consumer:1
Input:1

************Context Switch***********
Add a thr to ready q:Producer2
Save context for Producer2
continue creating

****************create thread, name:Producer  id:3**********
Pro_Sema producer:6
Con_Sema consumer:2
Input:2
```

c) Context Switching and LWT Sleeping

```
>>>>>>>>>>>>>>Current running thread: Consumer6<<<
Con_Sema consumer:2
Pro_Sema producer:6
Output: 4

************Context Switch***********
Add a thr to ready q:Consumer6
Save context for Consumer6
Switching ....

>>>>>>>>>>>>>>Current running thread: Producer2<<<
Producer2 need to sleep for 2000000 micro secs

************Context Switch***********
time passed:21 micro sec
remain sleeping micro sec for Producer2:1999979
Save context for Producer2
Switching ....
```

d) Waking Up

Remaining time <0

```
>>>>>>>>>>>>>>Current running thread: Producer2<<<<<<<<<<<<<<<<<<
return from sleeping
Pro_Sema producer:7
unblock sem wait t:Consumer id:5
Con_Sema consumer:1
Input:7

************Context Switch***********
time passed:50978 micro sec
remain sleeping micro sec for Producer3:-50964
Producer3 wake up
remain sleeping micro sec for Producer4:-50956
Producer4 wake up
Add a thr to ready q:Producer2
Save context for Producer2
Switching ....
```

e) Synchronization

```
>>>>>>>>>>>>>>Current running thread: Consumer5<<<<<<<<<<<<<<<<<<
Con_Sema consumer:0
Pro_Sema producer:8
Output: 12

************Context Switch***********
time passed:49993 micro sec
remain sleeping micro sec for Producer3:1899998
remain sleeping micro sec for Producer4:1900012
Add a thr to ready q:Consumer5
Save context for Consumer5
Switching ....

>>>>>>>>>>>>>>Current running thread: Consumer6<<<<<<<<<<<<<<<<<<
wait in sema:Consumer id:6
```

Buffer size is 8.
Con_Sema is 0.
Block the thread.

```
>>>>>>>>>>>>>>Current running thread: Producer3<<<<<
return from sleeping
Pro_Sema producer:7
unblock sem wait t:Consumer id:6
Con_Sema consumer:1
Input:13
```

Con_Sema > 0. Unblock one thread from the wait queue.

f) Waiting Other Thread

```
>>>>>>>>>>>>>>Current running thread: Producer2<<<<<<<<<<<<<<<<<<
Pro_Sema producer:5
Con_Sema consumer:3
Input:10

****************create thread, name:SubProducer  id:7**********
add Producer 2 to wait q
Save Context for Producer 2

****************create thread, name:SubProducer  id:8**********
add SubProducer 7 to wait q
Save Context for SubProducer 7
testwait:0
testwait:1
testwait:2
testwait:3

**************start exiting SubProducer8****************
Sub threads are done, move to ready q:SubProducer 7

......SubProducer8 exit successfully!

>>>>>>>>>>>>>>Current running thread: SubProducer7<<<<<<<<<<<<<<<<<<
testwait:0
testwait:1
testwait:2
testwait:3

**************start exiting SubProducer7****************
Sub threads are done, move to ready q:Producer 2

......SubProducer7 exit successfully!
```

g) Process Sleeping

No ready thread, but there are sleeping threads

```
Save context for Consumer5
Empty ready q

************Context Switch***********
time passed:3 micro sec
remain sleeping micro sec for Producer3:1899948
remain sleeping micro sec for Producer4:1899962
remain sleeping micro sec for Producer2:1999962
Process is sleeping!
time passed:1899948 micro sec
remain sleeping micro sec for Producer3:0
Producer3 wake up
remain sleeping micro sec for Producer4:14
remain sleeping micro sec for Producer2:100014
Switching ....

>>>>>>>>>>>>>>Current running thread: Producer3<<<<<<<<<<<<<<<<<<
return from sleeping
```

h) Exiting Thread

```
>>>>>>>>>>>>>>Current running thread: Producer3<<<
Pro_Sema producer:5
Con_Sema consumer:3
Input:18

************Context Switch***********
Add a thr to ready q:Producer3
Save context for Producer3
Switching ....

>>>>>>>>>>>>>>Current running thread: Producer2<<<
Finish producing required 18 items!Exiting...

**************start exiting Producer2************
```

```
>>>>>>>>>Current running thread: Consumer6<<<<<<<<<
Con_Sema consumer:0
Pro_Sema producer:0
Output: 18

************Context Switch***********
Add a thr to ready q:Consumer6
Save context for Consumer6
Switching ....

>>>>>>>>>>>>>>Current running thread: Consumer5<<<<<
Finish Consuming All 18 items!Exiting...

**************start exiting Consumer5***************
```