Junxiang Wu
juw48@pitt.edu 10/31/2013

CS2510 Computer Operating Systems –Project1

Description

The purpose of this project is to implement a light weight user-level thread (LWT) scheduling system without the need to modify the Linux kernel. Your system must allow the creation and concurrent execution of threads of control within a Linux task, in a true preemptive manner. The LWT system must be capable of creating threads and performing mini context switches to share the CPU time among the task threads, based on the specified scheduling policy.

Design

The basic components of the LWT system include:

- **A thread scheduler**: thread scheduler is implemented as a SIGALRM handler that after a piece of quantum time it selects next thread to execute and perform a mini context switch.
- **A thread context** every thread has its own context; specifically SP (stack pointer), BP (base pointer) and PC (program pointer) contribute to the least context I care about most in this project.

When a new thread is created, space for stack needed to be allocated (by using "malloc") and PC need to be assigned (by using function pointer). While in a context switch, we need to save current context and rescore next context. For SP and BP, we need to do it manually by using inline assembly language. As for PC, setjmp and longjmp will do that for us.

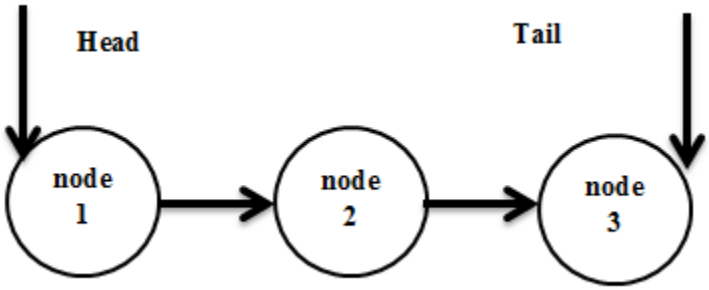
- **Semaphores** Semaphore is used for mutual exclusive and synchronization. The initial value of a Semaphore indicates the number of resources available. P() and V() is used to request and release a recourse respectively. In P() operation, if the values is negative, indicating that there is no recourse available at present, then the thread will be blocked and put into the waiting queue of the semaphore until it is release by others.

Implementation

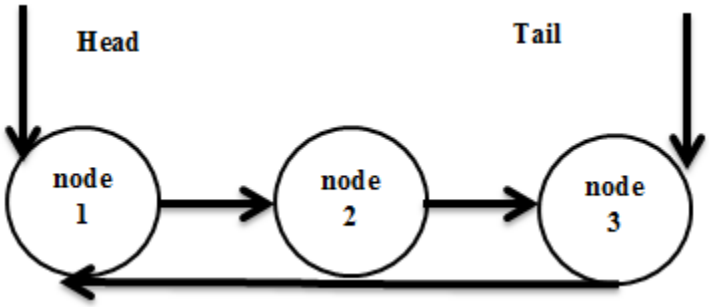
1. Queue design

Because lots of operation in this project involves queue operation, I first design and implement a queue library provide common queue operation. The implantation is originated from [1] (http://blog.csdn.net/hopeyouknow/article/details/6736987).

The original follow the FIFO scheme. As shown below:



To implement the ready queue for round robin scheme, I extend the library to support a circle list. This circle queue only adds and deletes node at head. Also there is a move to next function for selecting next thread.



2. Context switch

SP and BP need to save and restore mamually by using inline assembly language coede. As shown below, the load store macro is used for save & restore sp (stack pointer) and bp (base pointer). By using the macro to choose which part of code to compile, this program can be complied and run regardless of 32bit or 64bit machine.

```
#ifdef __x86_64__


    #define lwt_store(pthread) \
        do { \
            asm volatile ( \
                "movq %%rsp, %0\n\t" \
                "movq %%rbp, %1" \
                :"=a"(pthread->t_sp),"=b"(pthread->t_bp) \
                ); \
        } while(0)

    #define lwt_load(pthread) \
```

```
        do { \
            asm volatile ( \
                "movq %0, %%rsp\n\t" \
                "movq %1, %%rbp" \
                : \
                :"a"(pthread->t_sp),"b"(pthread->t_bp) \
                );\
        } while(0)
#else
    #define lwt_store(pthread) \
        do { \
            asm volatile ( \
                "movl %%esp, %0\n\t" \
                "movl %%ebp, %1" \
                :"=a"(pthread->t_sp),"=b"(pthread->t_bp) \
                ); \
        } while(0)

    #define lwt_load(pthread) \
        do { \
            asm volatile ( \
                "movl %0, %%esp\n\t" \
                "movl %1, %%ebp" \
                : \
                :"a"(pthread->t_sp),"b"(pthread->t_bp) \
                );\
        } while(0)

#endif
```

3. Thread control block and thread initialization

The lwt_struct is the thread control block structure that maintain the information of a thread.

```
typedef struct lwt_struct {
    int t_id;
    int t_slpc;//sleep cycle

    void* t_sp;
    void* t_bp;
    struct lwt_struct* t_father;
    struct lwt_struct* t_wthread;
    jmp_buf t_env;
    lwt_func t_func;
    lwt_state t_state;
}lwt_struct;
```

It at least should have sp, bp and env buffer for saving the thread context. t_slpc is for sleep function and f_father , t_wthread is for wait & exit function which will be discussed later.

The lwt_init() is to initialize the first thread and ready queue.

4. Create a thread

When a new thread is created, space for stack needed to be allocated (by using “malloc”) and PC need to be assigned (by using function pointer). In lwt_create() function, current thread state is been saved first, then memory is been allocated and new thread execute immediately.

There are two tricky things in implementation:

1. Signal mask:

Interrupted by scheduler is not acceptable during thread creation (same to other lwt_functions), and thus

```
sigprocmask(SIG_BLOCK, &blockset, NULL);
```

is used to block the SIGALRM and it should be unblocked when function is complete or return from longjmp (because longjmp will restore the previous setjmp saved signalmask).

2. Argument access

The lwt_create() switch stack to a new place during its execution, meaning the its argument stored in old stack is no longer available after the switch. I use a declared global variable to temporally save the function pointer so that it can be accessed after the switch. I tried to make the thread function support multiple arguments by using an argument array, but not went well because things get tricky when I want try to find a scheme guarantee it can be access after the switch.

5. Sleep function

Sleep function is to let thread sleep for couple of seconds then wakeup. The idea is that as the scheduler implemented as SIGALARM handler, the times scheduler activated can be count as indicator of time. For example, if the quantum for a thread is 500ms, after 2 times scheduler activation means 1 second has past.

To implement this idea:

1) Once entering sleep function, number of second is translated to number of scheduler cycles maintained in thread control block lwt_struct and thread will be put into sleep_queue (circle queue, same as ready queue)

2) Every time scheduler is activated, clock cycle ticking, meaning that the t_slpc of each thread in sleep queue will decrease 1. and pose a flag a thread need to wake up

3) If detect a wake up flag, scheduler will delete it from sleep queue and add it back to ready queue, execute immediately.

Because these operations is closely related with scheduler, I leave the hard work to the scheduler, let it figure out if current thread want to sleep and perform proper actions mentioned above.

```
#define lwt_quantum 500000
#deint wakeup_flag;

fine lwt_factor 2


/* ticking the sleep time*/
void ticktock(lwt_struct* pthread){
    pthread->t_slpc--;
    if(pthread->t_slpc<=0)
        wakeup_flag=true;
}
/* thread sleep sec seconds*/
void lwt_sleep(int sec) {
    GetFront(ready_queue,&temp_thread);
    temp_thread->t_state=lwt_SLEEP;
    temp_thread->t_slpc=sec*lwt_factor;
    pause();
}
```

6. Wait and exit

wait() and exit() function is correlated.

For exit(): thread want to know if it's been waited by father thread.

If it's waited by father thread, it need to wake up father: 1) set state to ready 2) put father thread back to ready queue 3) exit and execute father thread

If not, then it set the lwt_state to EXIT and delete itself from ready queue, then execute next ready thread.

For wait(): check the thread if it is already EXIT. If not, then perform wait: 1) set set to wait and delete itself from ready queue 2) select and execute next thread.

7. Semaphore and producer/consumer problem

A semaphore basically contains a values and a waiting list. Anyone request resource by using P(), which decrease the value by 1 and test if the value is less than 0: If true, meaning that the resource is not available, the thread will be put into the waiting list of the semaphore and remove from ready queue. While V() is called when thread want to

release a resource, during V() it will check if there is any threads waiting for that and pick one from the list execute it.

Producer and Consumer problem is classic case to demonstrate semaphore function. In my program, number of producer/consumer, time of producing/consuming, size of buffer is defined by macro. Producer will produce items into buf[0],buf[1]…and consumer will consume them accordingly. The expected output will like this:

Producer A Produced: 0

Producer B Produced: 1

Producer A Produced: 2

Producer B Produced: 3

Producer A Produced: 4

Consumer A Consumed: 0

Consumer A Consumed: 1

Consumer B Consumed: 2

The number refers to the buffer number it produced to/consumed from. There semaphore is declared and initialized: full, empty, mutex;

- “full” indicate the number of buffer holds item, initialized to 0;
- “empty” indicate the number of empty buffer ,initialized to buf_size;
- “mutex” is for synchronization, initialized to 1.

Junxiang Wu
juw48@pitt.edu 10/31/2013

Appendix

prodcon.c

```
#define buf_size 500

#define producer_num 2
#define consumer_num 2

#define produce_times 1000
#define consume_times 1000
int buffer[buf_size];
int in=0;
int out=0;
lwt_semaphore mutex,empty,full;

void producer(int p_num,char **argv) {
    int i;
    p_num--;
    printf("producer %c create\n",'A'+p_num);
    for(i = 0; i < produce_times; i++ ){
        lwt_semP(&empty); //get a empty slot
        lwt_semP(&mutex);
        /*critical section begin*/
        buffer[in]=1;
        printf("producer %c: produced %d \n",'A'+p_num,in);
        in = (in+1) % buf_size;
        /*critical section end*/
        lwt_semV(&mutex);
        lwt_semV(&full);
    }
    printf("producer %c exit\n",'A'+p_num);
    lwt_exit();
}
void consumer(int c_num,char **argv) {
    int i;
    c_num--;
    printf("consumer %c create\n",'A'+c_num);
    for(i = 0; i < consume_times; i++ ){
        lwt_semP(&full); //get a full slot
        lwt_semP(&mutex);
        /*critical section begin*/
        buffer[out]=0;
        printf("consumer %c: consumed %d \n",'A'+c_num,out);
        out = (out+1) % buf_size;
        /*critical section end*/
        lwt_semV(&mutex);
        lwt_semV(&empty);
    }
    printf("consumer %c exit\n",'A'+c_num);
    lwt_exit();
}

int main(){
    int i;
    lwt_struct * producer_thread[producer_num];
```

```
    lwt_struct * consumer_thread[consumer_num];

    lwt_sem_init(&mutex,1);
    lwt_sem_init(&empty,buf_size);
    lwt_sem_init(&full,0);

    lwt_init();


    for(i = 0; i < producer_num; i++){
        producer_thread[i]=lwt_create(producer,i+1,NULL);
        printf("producer %d created\n",i);
    }
    printf("producers created\n");

    for(i = 0; i < consumer_num; i++){
        consumer_thread[i]=lwt_create(consumer,i+1,NULL);
        printf("consumer %d created\n",i);
    }
    printf("consumers created\n");

    for(i = 0; i < producer_num; i++){
        lwt_wait(producer_thread[i]);
    }
    for(i = 0; i < consumer_num; i++){
        lwt_wait(consumer_thread[i]);
    }

    printf("producer comsuer complete\n");
    return 0;
}
```