Implementation of a user level thread library

Soumen Basu(2013MCS2584)

Assignment Description:

In this assignment we implement a user level thread library. The deliverable includes the following:

1. The thread library with the Round-Robin Scheduler.
2. Our own "semaphore".
3. To show that our threads library and semaphore works, we implemented
   - *Bounded buffer producer consumer* problem.
   - Few demo threads that calls the functions defined in thread library.

Data Structures:

```
typedef void* (*funWithArg_t)(void*);
typedef void (*fun_t)(void);
typedef  short bool;
typedef  unsigned threadid_t;

/* Data structure for Thread Control Block */
typedef struct _threadCB{
      threadid_t tid;                  /* Thread id */
      status_t stat;                   /* Status of the thread */
      sigjmp_buf env;                  /* Context of the thread*/
      void* retval;                    /* Return value of the thread*/
      bool withArg;                    /* Check if a thread with arguments was created */
      fun_t fun;                       /* Address of the function associated with thread */
      funWithArg_t funWithArg;         /* Address of the function with arguments associated with thread */
      struct _threadCB* right_th;      /* Pointer to next/right thread control block*/
      struct _threadCB* left_th;       /* Pointer to previous/left thread control block*/
      char* tStack;                    /* Execution Stack associated with the thread*/
}tcb_t;

/* Data structure of Queue: for listing Thread Control Blocks.*/
typedef struct _tqueue{
      int count;                       /* Number of thread control blocks in Queue*/
      tcb_t* head;                     /* Pointer to Queue Head */
      tcb_t* tail;                     /* Pointer to Queue Tail */
}queue_t;
```

```
/* Semaphore structure. */
typedef struct _tsem{
      int value;              /* Value of the semaphore */
      queue_t list;           /* Queue of blocked threads */
}semaphore;

/* Data structure for recording thread status. */
typedef struct _tstatus{
      threadid_t id;
      enum {RUNNING, READY, SLEEPING, SUSPENDED, TERMINATED} state;
      unsigned no_of_bursts;
      unsigned total_exec_time;
      unsigned total_sleep_time;
      unsigned avr_exec_time;
      suseconds_t sleeptime;
      struct timeval wakeuptime;
      struct timeval last_exec_time;
}status_t;
```

Note on *setjmp(), longjmp()* and *jmp_buf*:

C99 defines the setjmp() macro, longjmp() function and jmp_buf tyoe, which are used to bypass the normal functiom call and return discipline.
The setjmp() macro saves its calling environment for later use by longjmp() function. the longjmp() function restores the environment saved by the most recent invocation of the setjmp() macro.
Following is a implementation of jmp_buf structure.

```
#if defined(__x86_64__)   /* For x86_64 Architecture */
#define JB_BP 1
#define JB_SP 6
#define JB_PC 7
  typedef long int __jmp_buf[8];

#elif defined(__i386__)    /* For i386 Architecture */
#define JB_BP 3
#define JB_SP 4
#define JB_PC 5
  typedef int __jmp_buf[6];
#endif
struct __jmp_buf_tag{

    __jmp_buf __jmpbuf;             /* Calling environment.  */
    int __mask_was_saved;           /* Saved the signal mask?  */
    __sigset_t __saved_mask;        /* Saved signal mask.  */
};

typedef __jmp_buf_tag jmp_buf[0];
```

The jmp_buf structure contains three fields. The calling environment is stored in `__jmpbuf`. The `#define` statements indicate which values are stored in each array element. In case of ***i386*** architecture, the *Stack Base Pointer* (BP) is is stored in `__jmpbuf[3]`. The *Stack Pointer* (SP) and *Program Counter* (PC) are stored in `__jmpbuf[4]` and `__jmpbuf[5]` respectively.
For ***x86_64*** architecture, the *Stack Base Pointer* (BP) is is stored in `__jmpbuf[1]`. The *Stack Pointer* (SP) and *Program Counter* (PC) are stored in `__jmpbuf[6]` and `__jmpbuf[7]` respectively.

Following is the assembly instructions generated from longjmp() on Linux.

longjmp(env, i)

```
1. movl  i ,        %eax                 /* Return i */
2. movl  env.__jmpbuf[JB_BP], %ebp       /* Load env.__jmpbuf[JB_BP] to stack base pointer register "ebp" */
3. movl  env.__jmpbuf[JB_SP], %esp       /* Load env.__jmpbuf[JB_SP] to stack top pointer register "esp" */
4. jmp   (env.__jmpbuf[JB_PC])           /* Jump to address specified by env.__jmpbuf[JB_PC] */
```

The trick to switch contexts of user levels threads is overwriting the values of SP and PC during the thread creation time. These values are overwritten with the thread's own stack pointer and pointer to the function associated with the thread during thread creation.
But there is one catch. These values can't be directly fed into the `__jmpbuf[JB_SP]` and `__jmpbuf[JB_PC]` fields. The reason is that these fields are encoded to fortify against hacks/ abuse by programmers. The encoding scheme is simple though.

1. X = (Address of SP/PC) **xor** (Address of *Process Environment Block*)
2. Rotate (X) 17 bits Left.
3. __jmpbuf[JB_SP/JB_PC] = X

In ***x86_64*** architecture, **FS:[0x30]** refers to the **4** bytes starting from **[0x30]** in the *segment register* **FS**. These 4 bytes store the linear address of ***Process Environment Block*** (PEB). In ***i386*** architecture, **GS:[0x18]** does the same.
Following is the implementation of the encoding scheme,

```
p1 = (unsigned long)func;  /* func is a pointer to function associated with the thread */
p2 = (unsigned long)stack; /* stack is a pointer to the execution stack */
#if defined(__x86_64__)
  __asm__ __volatile__("xorq %%fs:0x30, %0\n\trolq $0x11, %0" : "=r"(p1) : "r"(p1));
  __asm__ __volatile__("xorq %%fs:0x30, %0\n\trolq $0x11, %0" : "=r"(p2) : "r"(p2));
#elif defined(__i386__)
  __asm__ __volatile__("xorl %%gs:0x18, %0\n\troll $9, %0" : "=r"(p1) : "r"(p1));
  __asm__ __volatile__("xorl %%gs:0x18, %0\n\troll $9, %0" : "=r"(p2) : "r"(p2));
#endif
```

This implementation can be done by inline assembly code also. A mix of inline assembly and high level C instructions can also do the trick.

```
typedef unsigned long address_t;

/* Mix of Inline Assembly and High level C instructions. */

address_t rotl(address_t ret, int shift)
{
	if((shift  &= 31)==0)
		return ret;
	return (ret << shift) | (ret >> (32 - shift));
}

address_t encode_addr(address_t addr)
{
	address_t ret;
	asm volatile("\tmov	%%fs:0x30,%0" : "=g" (ret));
	ret = ret^addr;
	ret = rotl(ret,17);

return ret;
}

/* Pure Inline Assembly */

address_t encode_addr(address_t addr)
{
	address_t ret;
	asm volatile("xorq	%%fs:0x30,%0\n"
		     "rolq	$0x11,%0\n"
		     : "=g" (ret)
		     : "0" (addr));
return ret;
}
```

Functions & their descriptions:

int CreateThread(void (*f)(void))	This function creates a new thread with **f()** being its entry point. The function returns the created thread id (>= 0) or (-1) to indicate failure. It is assumed that f() never returns. A thread can create other threads! The created thread is appended to the end of the ready list (state = READY). Thread ids are consecutive and unique, starting with 0.
void Go()	This function is called by the main process to start the scheduling of threads. It is assumed that at least one thread is created before **Go()** is called. This function is called exactly once and it never returns.
int GetMyId()	This function is called within a running thread. The function returns the thread id of the calling thread.
int DeleteThread(int tid)	Deletes a thread which has id = *tid*. The function returns 0 if successful and -1 otherwise. A thread can delete itself, in which case the function doesn't return.
void Dispatch(int sig)	The thread scheduler. This function is called by the interval clock interrupt and it schedules threads using FIFO order. It is assumed that at least one thread exists all the time - therefore there is a stack in any time. It is not assumed that the thread is in ready state.
void YieldCPU()	Called by the running thread. The function transfers control to the next thread. Next thread will have a complete time quantum to run.
int SuspendThread(int tid)	This function suspends a thread until the thread is resumed. The calling thread can suspend itself, in which case the thread will **YieldCPU()** as well. The function returns id on success and -1 otherwise. Suspending a suspended thread has no effect and is not considered an error. Suspend time is not considered waiting time.
int ResumeThread(int tid)	Resumes a suspended thread. The process is resumed by appending it to the end of the ready list. Returns id on success and -1 on failure. Resuming a ready process has no effect and is not considered an error.
int GetStatus(int tid, status *stat)	This call fills the status structure with thread data. Returns id on success and -1 on failure.
void SleepThread(int sec)	The calling thread will sleep until (current-time + sec). The sleeping time is not considered wait time.
void CleanUp()	Shuts down scheduling, prints the statistics, deletes all threads and exits the program.
void Wait (semaphore *S)	The calling thread will perform wait operation on Semaphore S.
void Signal (semaphore *S)	The calling thread will perform signal operation on Semaphore S.

Bounded Buffer Producer Consumer Problem (Solution with Semaphore):

Chosen **Buffer_Size = 5**
Mutex: Binary Semaphore initialized to 1. This ensures the mutual exclusion of critical sections.
Full: Counting Semaphore initialized to 0. This keeps a track of no. of filled slots in buffer.
Empty: Counting Semaphore initialized to Buffer_Size. This keeps track of no. of empty slots in buffer.

Producer	Consumer

`while(true)`
`{`
`//Produce item V`
`Wait(&Empty);`
`Wait(&Mutex);`
`//Put item V into buffer`
`Signal(&Mutex);`
`Signal(&Full);`
`}`
	`while(true)`
	`{`
	`Wait(&Full);`
	`Wait(&Mutex);`
	`//Get item V from buffer`
	`Signal(&Mutex);`
	`Signal(&Empty);`
	`//Consume item V`
	`}`

Running the Files:

1. Open terminal.
2. Make sure all the files (*thread.h mythread.c Makefile & your own .c file*) are in same folder.
3. Open Makefile.
4. Type your own .c file name (without extension) in place of FILE.
5. Type your desired output file name in place of OUT.
6. To clear all the previous build, run the following command on terminal
   $ `make clean`
7. To build new executable, run the following command,
   $ `make`
8. Now, run your executable.

Alternatively,

1. Open terminal.
2. Make sure all the files (*thread.h mythread.c Makefile & your own .c file*) are in same folder.
3. Run the following set of commands,
   $ `gcc -c mythread.c`
   $ `gcc -c <yourfilename>.c`
   $ `gcc <yourfilename>.o mythread.o -o <outputname>`
   $ `./<outputname>`

The Makefile:

```
COM=gcc
FLAG=-c
REM=rm -rf *o
FILE=test
OUT=tdemo

all: $(OUT)

$(OUT): $(FILE).o mythread.o
	$(COM) $(FILE1).o mythread.o -o $(OUT) -g

$(FILE).o: $(FILE).c
	$(COM) $(FLAG) $(FILE).c

mythread.o: mythread.c
	$(COM) $(FLAG) mythread.c

clean:
	$(REM) $(OUT)
```