CSP 701 : ASSIGNMENT 2

Implementation of User Level Thread Library

Amit Kumar

Gaurav Singh

Introduction : This program is a library which a client can use to create multi threaded program. This is implementation of user level multi threading , that means that the kernel has no information about multiple threads running inside the process . This means the threads are completely managed in user space .
This is implemented using setjmp and longjmp functions of the gcc library. These functions are used to store the current context (setjmp) and restore the store context (longjmp) . These are basically used for complex control strategies.

Library : the library contains the following headers :

ThreadStructure.h

ThreadQueue.h

ThreadManipulation.h

User only needs to include ThreadManipulation.h in order to use multithreading.

ThreadStructure.h : This file contains the structure of thread control block and the structure of nodes of the lists.

ThreadQueue.h : This file contains the function which are used to manipulate different lists .

ThreadManipulation.h : This file contains functions which are interface to the user and are used to write multi threaded applications . These are as follows :

Functions to be used by user :

```
int CreateThread(void (*f)(void));
void Go();
int GetMyId();
int DeleteThread(int thread_id);
void Dispatch(int sig);
void YieldCPU();
int SuspendThread(int thread_id);
int ResumeThread(int thread_id);
int GetStatus(int thread_id, status *stat);
void SleepThread(int sec);
void CleanUp();
int CreateThreadWithArgs(void *(*f)(void *), void *arg);
void *GetThreadResult(int tid);
```

- **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 thread_id)** - deletes a thread. 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()** - this function is called by the running thread. The function transfers control to the next thread. The next thread will have a complete time quantum to run.
- **int SuspendThread(int thread_id)** - 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 thread_id)** resumes a suspended process. 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 thread_id, 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 process will sleep until (current-time + sec). The sleeping time is not considered wait time.
- **void CleanUp()** - shuts down scheduling,deletes all threads and exits the program.

How to run : to run simply create your program and run the file Compile.sh or CompileWS.sh (if you want stats then use CompileWS.sh)with the argument which is your main file name(which consist main function).
sh Compile.sh yourfilename.c or sh CompileWS.sh yourfilename.c
Then the output will be Output or OutputWithStat respectively then run the output file using command
./Output or ./OutputWithStat

Explanation of mangle function (convert_address):

The function convert_address is used to encrypt / mangle the address before it is stored in the jmpbuf structure because the jmpbuf structure can be altered by the user using sigjmp and longjmp function resulting in undefined behaviour of program and it can be used for hacking / attacking the system.This encryption scheme can detect modified addresses and hence can prevent these attacks.

In Linux,libc provides a pointer guard in gs:0x18 . The idea behind the pointer
guard is the following: to encrypt a sensitive address p , a program can compute
s=p xor gs:0x18
, optionally add some bit rotations, and store it in a structure that gets
passed around. Decryption can simply invert any bit rotations, and then compute
p=s xor gs:0x18
back.

Pointer Guard protects the jmpbuf structure.
The libc pointer guard has different values across multiple runs of a program.

In 64 bit systems the pointer guard is stored in fs:0x30

```
//Architecture Dependent Code

#ifdef __x86_64__
// code for 64 bit architecture

typedef unsigned long enc_addr;

#define JB_SP 6         //location of stack pointer and Instruction Pointer in
jump buffer
#define JB_IP 7

enc_addr convert_address(enc_addr addr)
{
      enc_addr ret , key;
      asm volatile("movq    %%fs:0x30,%0\n"
//getting the value of key which is used to mangle/encrypt the address
```

```
                    : "=r" (key));
      ret = key ^ addr;                           //xor the key with addr
      ret = (ret << 17) | (ret >> (64 - 17));   //left shift 17 times or 0x11 times
      return ret;
}


#else

// code for 32 bit architecture

typedef unsigned int enc_addr;

#define JB_SP 4         //location of stack pointer and Instruction Pointer in
jump buffer
#define JB_IP 5

enc_addr convert_address(enc_addr addr)
{
    enc_addr ret , key;
//    int size;
//    size = sizeof(enc_addr)*8;
    asm volatile("mov   %%gs:0x18,%0\n"
                      : "=r" (key));
    ret = key ^ addr;
    ret = (ret << 0x9) | (ret >> (32-0x9));
    return ret;
}


#endif
```