Using SDL Threads in Computer Science and Engineering Courses

Tong Lai Yu and David A. Turner
School of Computer Science and Engineering
California State University San Bernardino
San Bernardino, CA 92407
tyu@csusb.edu, dturner@csusb.edu
909-537-5326

Abstract

This paper describes the experience of the authors in using SDL threads to develop computer science and engineering course materials that cover multi-threaded programming. The courses include data structures, operating systems, computer graphics and video game programming. The techniques developed are also used in the work of students' independent study projects and master's projects. In particular, they have been used to support an NSF CPATH grant to revitalize computer science education and promote computational thinking. This paper includes 3 simple example programs that illustrate threading concepts.

Key Words: computer science education, concurrency, Linux, multi-threaded. video games

1. Introduction

Most new personal computers are now built with multi-core processors and therefore provide computational parallelism at the hardware level. As older machines are retired and replaced, concurrency at the hardware level will gradually become ubiquitous in desktop and laptop personal computers. Distributed computing in which separate computers collaborate in parallel on computing tasks is another example of the presence of computational parallelism in current practice. Correspondingly, there is an increased need for students of computing programs to master the principles of parallel algorithms and multi-threaded programming. In 2008 a task force representing the Association of Computing Machinery and the IEEE Computer Society published computing curriculum recommendations [1] that revise the widely regarded 2001 Computer science curriculum recommendations [2]. The 2008 report presents concurrency as a cross-cutting concern that students should see as a recurring theme throughout their courses and suggests that concurrency -- along with security and entertainment software -- could serve as one possible explicit theme around which to organize a degree program curriculum.

The concern of the task force is reflected in real-world software development trends. As the number of multi-media applications continue to grow, multi-threaded programming has become ever more important in software applications. For example, without the use of threads it is impossible to write even a commercial-grade multi-media application that needs to decode data, render graphics, stream audio and handle other features simultaneously. Teaching students multi-threaded programming has become an increasingly more important component of the computer science and engineering curriculum. Though the concept and principles of multi-threaded programming are discussed in some computer science courses, such as operating systems and data communications, they are often presented in an abstract aspect without executable example programs that are small enough to be accessible to students. For example, textbooks on operating systems written by Professor Tanenbaum are widely used in universities, but the algorithms and concepts are illustrated in C-like pseudo code that are not complete executable programs. Thus students can not easily gain hands-on experience through experimentation. Some textbooks present threading concepts in Java that are small executable examples. While using Java is one way to let students gain hands-on experience and carry out experiments with the code, not every computer science or engineering program uses Java as the primary language for teaching. Many programs still use C/C++ as the primary programming language. Students in these programs may not be familiar or even know Java. In programs that emphasize video game programming, Java is not generally used.

For teaching purposes, we would ideally like to use a thread API that is simple, easy to learn, cross platform, and open-source. The cross platform approach in teaching the courses is important because most students nowadays have a personal computer and use one of the three popular operating systems, Windows, Mac OS/X, and Linux. Cross platform software allow students to run in their own personal computers without making modifications to the programs or purchasing any new software. Table 1 compares some commonly used thread APIs used in various computing platforms. In Table 1, the symbols W, M, L, S, and B represent the operating systems MS

Windows, Mac OS/X, Linux, Sun Solaris and BSD, respectively.

Table 1. Comparing thread APIs

API	*W*	*M*	*L*	*S*	*B*	*Open*	*Simple*
Windows	Y	N	N	N	N	N	N
Solaris	N	N	N	Y	N	Y	N
POSIX	Y	Y	Y	Y	Y	Y	N
Java	Y	Y	Y	Y	Y	Y	Y
SDL	Y	Y	Y	Y	Y	Y	Y

The SDL library is written in C and can be easily integrated with OpenGL [3]; it is a cross-platform open-source multimedia library designed to provide low level access to audio, keyboard, mouse, joystick, 3D hardware and 2D video framebuffer.

IEEE defines a POSIX standard API, referred to as Pthreads (IEEE 1003.1c), for thread creation and synchronization [4]. Many contemporary systems, including Linux, Solaris, and Mac OS X implement Pthreads. The Pthreads specification has a rich set of functions, allowing users to develop sophisticated multi-threaded programs. However, in most applications many of the Pthread features are not needed. For this reason, we use SDL threads for educational purposes because it provides only those functions that are most commonly used. More importantly, SDL is platform independent and can run in the three popular platforms, Linux, Mac OS/X and MS Windows without any modifications to the source code. Though simplified, SDL threads provide semaphores and locking mechanisms that allow us to do proper synchronization of various events.

2. Operating systems

Operating systems is a common course where principles and applications of multi-threading are taught along with synchronization. In particular, semaphores are simple and commonly used to synchronize tasks. They are taught in practically all introductory operating system courses. Table 2 presents C-like pseudo code for the synchronization of two tasks to access a critical section (CS) using two semaphores. If the semaphores are not carefully used, they can easily create deadlocks, a situation illustrated in Table 3.

The code presented in Table 2 and Table 3 looks very simple and its meaning is clear. However, we found that they are simple and clear only to educators who have thoroughly understood the concept of multi-threaded programming. When we present the code of Table 3 to students who have taken a course in operating systems without hands-on experiences with multi-threaded programming, most of the students are confused. When we ask these students the following two questions, we are surprised to find that roughly 90% can not answer the questions correctly without further hints and guidance:

- Does the code of Table 3 achieve mutual exclusion?
- Is deadlock possible when executing the code? If yes, could you modify the code so that it becomes deadlock free?

This confirms that abstract concepts are difficult to understand without actual hands-on experience.

Table 2. Synchronization using semaphores

```
Semaphore S1;       // shared variables
Semaphore S2;

Task 1:             Task 2:
wait( S1 );         wait( S1 );  // acquire lock S1
wait( S2 );         wait( S2 );  // acquire lock S2
CS( );              CS( );
signal( S2 );       signal( S2 );  // release lock S2
signal( S1 );       signal( S1 );  // release lock S1
```

Table 3. Deadlock created by semaphores

```
Semaphore S1;       // shared variables
Semaphore S2;

Task 1:             Task 2:
wait( S1 );         wait( S2 );  // acquire lock S2
wait( S2 );         wait( S1 );  // acquire lock S1
CS( );              CS( );
signal( S2 );       signal( S1 );  // release lock S1
signal( S1 );       signal( S2 );  // release lock S2
```

Program Listing 1 shows a complete C/C++ program that is an implementation of the code of Table 3 using SDL threads. When students compile and execute the program, they will see the following two statements displayed and the program hangs.

```
I am Arnold. Arnold has locked a.
I am Groper. Groper has locked b.
```

Program Listing 1.
Deadlock created with SDL threads

```
// Compile: g++ deadlock.cpp -lSDL -lpthread
#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#include <stdlib.h>

using namespace std;

SDL_mutex *s1, *s2;     // semaphores
```

```
bool quit = false;
int a = 0, b = 0;
int thread1 ( void *data )
{
  char *tname = ( char * )data;

  while ( !quit ) {
    printf("I am %s. ", tname );
    SDL_mutexP ( s1 );  // lock before processing a
    printf("%s has locked a.\n", tname);
    ++a;              // dummy instruction
    // do somehting
    SDL_Delay( 10 );

    // also need b value
    SDL_mutexP ( s2 ); // lock before accessing b
    printf("%s has locked b.\n", tname);
    a += b++;         // dummy instruction

    // release lock 2
    SDL_mutexV ( s2 );
    // release lock 1
    SDL_mutexV ( s1 );

    // delay for a random amount of time
    SDL_Delay ( rand() % 3000 );
  }
  printf("%s quits\n", tname );

  return 0;
}

int thread2 ( void *data )
{
  char *tname = ( char * )data;

  while ( !quit ) {
    printf("I am %s. ", tname );
    SDL_mutexP ( s2 );  // lock before processing b
    printf("%s has locked b.\n", tname);
    ++b;              // dummy instruction
    // do somehting
    SDL_Delay( 10 );

    // also need a value
    SDL_mutexP ( s1 );  // lock before accessing a
    printf("%s has locked a.\n", tname);
    a += b++;         // dummy instruction

    // release the lock 1
    SDL_mutexV ( s1 );
    // release the lock 2
    SDL_mutexV ( s2 );

    // delay for a random amount of time
    SDL_Delay ( rand() % 3000 );
  }
  printf("%s quits\n", tname);

  return 0;
}

int main ()
{
  SDL_Thread *id1, *id2;  // thread identifiers
  // thread names
  char *tnames[2] = {"Arnold","Groper"};

  s1 = SDL_CreateMutex();
  s2 = SDL_CreateMutex();

  // create the threads
  id1 = SDL_CreateThread ( thread1, tnames[0] );
  id2 = SDL_CreateThread ( thread2, tnames[1] );

  // experiment with 20 seconds
  for ( int i = 0; i < 10; ++i )
     SDL_Delay ( 2000 );

  quit = true;  // signal the threads to return

  // wait for the threads to exit
  SDL_WaitThread ( id1, NULL );
  SDL_WaitThread ( id2, NULL );

  return 0;
}
```

This simple, concrete illustration allows students to easily recognize that deadlock has occurred. They can further solidify their understanding by modifying the program shown in Table 2 to be deadlock free. By doing this simple experiment, students can now grasp the concept of the usage of semaphores and the occurrence of deadlock. Of course, they could learn other basic principles such as mutual exclusion by making further modifications to the program, recompiling and executing to observe the results.

Using SDL threads, we can easily implement simple executable programs that simulate classical synchronization problems, including the Producer-Consumer Problem, Barber Problem, and Dining Philosophers Problem. A more detailed discussion of using these programs in lab sessions of an operating systems course can be found at [8]. As an example, Listing 2 presents the simulation of the barber problem using SDL threads. In the barber problem, the barber shop has one barber and N chairs for waiting customers. If no customers have arrived, the barber sleeps. If a customer arrives and the shop is full, he leaves otherwise he sits on a chair and waits. A customer has to wake up the barber to get served.

Program Listing 2. Barber problem

```
#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#include <stdlib.h>
#include <string>
```

```
#include <deque>

using namespace std;

const int N = 5;  // # chairs for waiting customers
SDL_sem           *mutex;  // for access to critical region
SDL_sem *ncustomers;  // # cust. waiting for service
SDL_sem *nbarbers;  // # barb. waiting for cust.
int nwaiting = 0;   // # of waiting customers
bool dinner_time = false;

void cut_hair()
{
  int n = rand() % 1000;
  printf("\nBarber cutting hair");
  SDL_Delay ( n );  // simulate hair cutting
}

int barber ( void *data )
{
  while ( true ) {
    if (dinner_time && nwaiting==0) break;  // end
    SDL_SemWait( ncustomers);  // sleep if 0 cust.
    SDL_SemWait(mutex);  // access 'waiting'
    nwaiting=nwaiting-1;  // one less waiting cust.
    SDL_SemPost (nbarbers);  // one barber is ready
    SDL_SemPost ( mutex );  // leave C.S.
    cut_hair();  // cut hair (outside C.S.)
  }
  printf("\nShop closing! Its dinner time!");
  return 0;
}

void get_haircut( const char *cname )
{
  int n = rand() % 1000;
  printf("\n%s is being served", cname );
  SDL_Delay ( n );    //simulate hair cutting
}

int customer ( void *data )
{
  SDL_SemWait ( mutex );  // enter C.S.
  deque<string> cqueue;    // customer names
  string s = string ( (char *) data );
  printf("\nnwaiting = %d ", nwaiting );
  if (nwaiting < N) {  // seats available,enter shop
    cqueue.push_back ( s );
    nwaiting += 1;  // one more waiting customers
    //wake up barber if necessary
    SDL_SemPost ( ncustomers );
    SDL_SemPost(mutex);  // release 'waiting'
    SDL_SemWait ( nbarbers );  // sleep
    string s1 = cqueue.front();
    cqueue.pop_front();
    get_haircut( s1.c_str()  );
  } else {    // if no more free chairs, leave
    printf("\nShop full, %s leaving", s.c_str());
    SDL_SemPost (mutex);  // shop full; don't wait
  }

  return 0;
}

int main ()
{
  SDL_Thread *barber_id;    // thread identifiers
  // simulate up to one hundred customers
  SDL_Thread  *customer_id[100];
  char tname1[] = { "Barber" };  // thread names
  char tname2[20];

  // initialize mutex to 1,no customer,no barber
  mutex = SDL_CreateSemaphore ( 1 );
  ncustomers = SDL_CreateSemaphore ( 0 );
  nbarbers  = SDL_CreateSemaphore ( 0 );
  barber_id = SDL_CreateThread(barber, tname1);
  for ( int i = 0; i < 50; ++i ) {
    SDL_Delay ( rand() % 500 );
    sprintf( tname2, "Customer %d", i );
    customer_id[i]=
       SDL_CreateThread(customer,tname2);
  }
  dinner_time=true;  // end of day, no more cust.
  // wait for the threads to exit
  SDL_WaitThread ( barber_id, NULL );
  for ( int i = 0; i < 50; ++i )
    SDL_WaitThread ( customer_id[i], NULL );

  return 0;
}
```

3. Data communications

A course on data communications and networking is taught in most computer science and engineering degree programs of this country. Though socket programming may be taught in such a course, multi-threaded programming may not be covered as frequently because of its complexity. Many educators may not be aware of the simple cross-platform SDL threads that can make this job easier. With the emergence of the Web and the explosive growth of multi-media applications, there is a compelling need to teach students to use threads to make programs more robust and efficient when developing applications in this area. Again, SDL threads can serve this purpose.

As an example, let us consider a typical application example in data communications. Suppose we want to receive a long stream of encoded data over a network, decode the data, and render the decoded data on the screen. The best way to write such an application is to separate cleanly the tasks of data-receiving, data-decoding, and data-rendering. We can use one thread for receiving data (receiver), one for decoding data (decoder) and one for rendering data (player). To separate these three tasks cleanly, we can apply the producer-consumer paradigm [5]. In general, a producer-consumer problem consists of a set of producer threads supplying resources to a set of consumer threads. These threads share a common buffer pool where resources are deposited by producers and removed by consumers. All the threads are asynchronous in the sense that producers may attempt to

deposit and consumers may attempt to remove resources at any instant. Since producers may outpace consumers and vice versa, two constraints must be satisfied. First, consumers are forbidden to remove any resource when the buffer pool is empty, and second, producers are forbidden to deposit any resources when the buffer pool is full. A conventional way to solve a producer-consumer problem is to use semaphores to block and wake up the threads at appropriate moments. we can use the SDL semaphore functions to achieve this. In our example, the receiver thread acts as the producer and the decoder thread acts as a consumer, and both share a common buffer. The receiver deposits data in the buffer and the decoder removes the data. In this way, the data-receiving process is cleanly separated from further data processing.

On the other side, the decoder thread shares another buffer pool with the player thread. The decoder puts the decoded data in the buffer and the player removes the data for rendering. In this case, the decoder acts as the producer and the player acts as the consumer. The decoder can use whatever method or techniques it wants to decode the encoded data and put the decoded data in the buffer. Similarly, the player can do whatever it wants with the decoded data; it can render them on the screen directly, or it can treat the data as a sequence of textures and render them on an arbitrary surface. Listing 3 shows a complete executable program that simulates the producer-consumer problem, which is ready to be used in the data-communication problem.

Program Listing 3.
Simulation of producer-consumer problem

```
#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#include <stdlib.h>

using namespace std;

#define  N   5    // number of slots in buffer
SDL_sem *mutex;  // controls access to C.S.
SDL_sem *nempty;  // counts number of empty slots
SDL_sem *nfilled;  // counts number of filled slots
int buffer[N];

int produce_item()
{
  int n = rand() % 1000;
  return n;
}

void insert_item ( int item )
{
  static int tail = 0;

  buffer[tail] = item;
  printf("insert %d at %d", item, tail );
  tail = ( tail + 1 ) % N;
}

// a producer thread
int producer ( void *data )
{
  int item;
  char *tname = ( char *) data;  // thread name
  while ( true ) {   // infinite loop
    SDL_Delay ( rand() % 1000 );  // random delay
    printf("\n%s : ", tname );
    item = produce_item();  // produce an item
    SDL_SemWait ( nempty );  // decrement empty count
    SDL_SemWait ( mutex );   // entering C.S.
    insert_item ( item );
    SDL_SemPost ( mutex );  // leave C.S.
    SDL_SemPost ( nfilled);   // increment nfilled
  }

  return 0;
}

int remove_item ()
{
  static int head = 0;
  int item;

  item = buffer[head];
  printf("remove %d at %d", item, head );
  head = ( head + 1 ) % N;
  return item;
}

// a consumer thread
int consumer ( void *data )
{
  int item;
  char *tname = ( char * ) data;//thread name

  while ( true ) {         // infinite loop
    SDL_Delay ( rand() % 1000 );  // random delay
    SDL_SemWait(nfilled);  // decrement filled count
    SDL_SemWait (mutex);  // entering C.S.
    printf("\n%s : ", tname );
    item =remove_item ();  // take item from buffer
    SDL_SemPost( mutex ); //  leave C.S.
    // increment count of empty slots
    SDL_SemPost ( nempty );
    // can do something with item here
  }
}

int main ()
{
  SDL_Thread *id1, *id2; // thread identifiers
  char *tnames[2] = { "Producer", "Consumer" };

  // initialize mutex to 1
  mutex = SDL_CreateSemaphore ( 1 );
  // initially all slots empty
  nempty = SDL_CreateSemaphore ( N );
  // initially no slot filled
```

```
  nfilled  = SDL_CreateSemaphore ( 0 );
  id1 = SDL_CreateThread (producer, tnames[0]);
  id2 = SDL_CreateThread (consumer, tnames[1]);

  // wait for the threads to exit
  SDL_WaitThread ( id1, NULL );
  SDL_WaitThread ( id2, NULL );

  return 0;
}
```

4. Computer graphics and games

Video games is a relatively new topic in the computer science and engineering curriculum. The use of threads in teaching computer graphics and video games are just as important as in other courses, if not more. Again, SDL threads can be used to facilitate learning in related courses. As an example, let us consider graphical scenes generated by ray-tracing that will be used in a video game. Ray-tracing is a typical topic covered in computer graphics. In general, it takes a large amount of computing time to generate a complex graphical scene. Even a simple scene that may be used as a texture or a terrain in a video game is time consuming to generate by ray-tracing. However, the difficulty can be eased or solved with the use of multi-threaded programming that delegates the tasks of ray-tracing to a number of machines. Typically, the scenes generated by ray-tracing in the client machines are related; for example, scene 1 could be a cube illuminated by a light source and scene 2 could be a scene with the cube rotated by a small angle about a certain axis; all the scenes combined could show an illuminated rotating cube. Figure 1 shows an image of a simple movie produced by such a method with the help of the open-source ffmpeg utility.

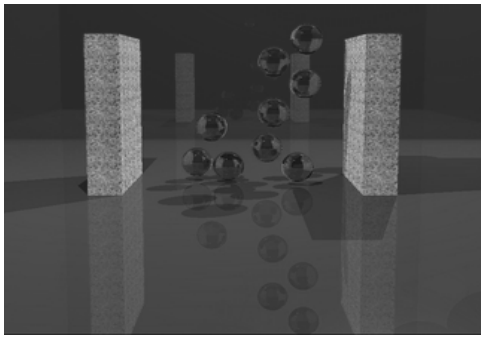

Figure 1. A movie created by ray-tracing

5. Independent Study and Student Projects

SDL threads are particularly useful for students to do independent studies or projects on contemporary topics of computer science and engineering. With the use of SDL threads, students can develop applications that are more sophisticated and interesting as compared to single-threaded applications. Figure 1 is one example of programs written by students through independent studies with use of SDL threads. Without threads, it is difficult to create programs that play music and render 3D scenes. Using threads properly is important in embedding video and audio in video games [7].

6. Conclusions

We have presented our experience of teaching various courses in computer science and engineering using SDL threads, which are simple, practical and easy for students to master. We use SDL threads to illustrate various concepts and give students a chance to do hands-on experiments; the students can understand the concepts a lot easier with this approach. Moreover, students find that using SDL threads to create simple cross-platform applications is interesting and satisfying.

7. References

[1] Association for Computing Machinery and IEEE Computer Society. *Computer Science Curriculum 2008: An Interim Revision of CS 2001*, ACM, Dec 2008.

[2] Association for Computing Machinery and IEEE Computer Society. *Computing Curricula 2001 Computer Science*, ACM Dec 2001.

[3] John R. Hall, *Programming Linux Games*, Linux Journal Press, 2001.

[4] B. Lewis, and D. Berg, *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, 1995.

[5] M. Singhal and N.G. Shivaratri. *Advanced Concepts in Operating Systems*, McGraw-Hill, 1994.

[6] T.L. Yu, *Chess Gaming and Graphics using Open-Source Tools*, Proceedings of ICC2009, p.253-256, Fullerton, California, IEEE Computer Society Press, April 2-4, 2009.

[7] T.L. Yu, D. Turner, D. Stover, and A. Concepcion, *Incorporating Video in Platform-Independent Video Games Using Open-Source Software*, Proceedings of ICCSIT, Chengdu, China, July 9-11, IEEE Computer Society Press, 2010.

[8] http://cse.csusb.edu/tongyu