An Introduction to Video Compression in C/C++

Fore June

Chapter 10 Video Programming

10.1 Introduction

In order to experiment with the encoding and decoding of video data, we need to have a way to play the video on a PC. We need some well-developed tools to help us achieve this goal. Ideally, the tools we are going to use are free and platform independent, and hopefully are simple to use. Surprisingly, such tools are available, thanks to the open-source community that provides many high-quality free useful software applications. The main tool that we shall use to play video is the Simple DirectMedia Layer (SDL) (*http://www.libsdl.org*). We also need some tools to process some video files that you can find in Internet and download them to carry out the tests and experiments. There exists a lot of video formats but we do not intend to address all of them; exploring video formats is **not** a goal of this book. Rather, we shall only discuss in detail the relatively simple AVI (Audio Video Interleaved) format and we use it as an intermediate format that we can read and save video data. There exists free utilities that allow us to change AVI files to other video formats and vice versa. To simplify things, we shall also make use of an open-source AVI (Audio Video Interleaved) library that can help us process AVI files.

In this chapter, we shall first use the SDL library to develop a simple video player (without audio part). Our goal of course is to decode the compressed data and render them on the screen. *Can we integrate the decoder and player seamlessly? Could we separate the decoder and the player functionalities so that changing the decoder would not affect the rendering and vice versa?* It turns out that this can be easily handled by the concept of the producer-consumer problem, which is a well-studied synchronization problem in Computer Science. In this case, the decoder is the producer which provides data and the player is the consumer that consumes the data. To accomplish these, they have to be run using different threads. SDL provides simple thread functions that allow us to implement all these with relative ease. We shall discuss the principles and implementations of the player and related issues in the following sections.

10.2 Simple DirectMedia Layer (SDL)

Simple DirectMedia Layer (SDL) is a cross-platform multimedia library designed to provide low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video framebuffer. It is used by MPEG playback software, emulators, and many popular games. It supports Linux, MS Windows and many other platforms. It is ideal to be used in embedded Linux as it is small and does not require X-window to do graphics. Embedded systems frequently have highly constrained resources and can afford neither the program storage space nor the memory footprint of desktop graphics software like X Window, KDE and GNOME. Because of these, SDL has been particularly popular in embedded Linux. It has been applied to a number of Embedded Linux implementations such as Microwindows, Paragui, and Superwaba.

SDL was designed and written by a group of experienced and highly professional game programmers and is supported by a huge collection of good programmers. In summary, it is a lean, portable, well-defined and reliable API (Application Programming Interface) , which is simple to learn and use. It is reported in an article (*http://gameprogrammer.com/sdl.html*) that the author spent several months writing test programs both as a way to learn SDL and to try to find bugs in SDL. In all that time he only found a couple of minor documentation bugs and what he would call a miss-feature for which there was an easy work around. He tried ridiculous things with SDL, such as writing a program that used 10,000 timers, and wasn't able to break it. Every time he found what he thought was a bug it turned out to be either the result of minor errors in the documentation or the result of his misunderstanding of the documentation. The one time he thought he had found a serious bug in SDL it turned out that it had already been fixed.

SDL can be used in games, emulators, and multimedia applications including the following:

- video: it can set a video at any depth (8 bits per pixel or greater) with optional conversion; write directly to a linear graphics framebuffer; create surfaces with colorkey or alpha blending attributes,
- events: provides event change detection like keyboard input, mouse input; each event can be enabled or disabled with SDL_EventState(),
- audio: set audio playback of 8-bit and 16-bit audio, mono or stereo; audio runs independently in a separate thread, filled via a user callback mechanism; provides complete CD audio control API,
- threads: provides simple thread creation API and simple binary semaphores for synchronization, and
- timers: gets the number of milliseconds elapsed and waits a specified number of milliseconds.

The most important application of SDL, though, is game programming.

The SDL library consists of several sub-APIs, providing cross-platform support for video, audio, input handling, multithreading, OpenGL rendering contexts and other amenities.

10.3 SDL API

The complete SDL API can be found at *http://www.libsdl.org/cgi/docwiki.cgi/SDL_20API*. We discuss here only some of the basic SDL functions that relate to playing videos.

10.3.1 Initializing the Library

We use SDL_Init() to dynamically load and initialize the library. This function takes a set of flags corresponding to the portions you want to activate:

SDL_INIT_AUDIO
SDL_INIT_VIDEO
SDL_INIT_CDROM
SDL_INIT_TIMER
SDL_INIT_EVERYTHING

We use SDL_Quit() to clean up the library when we are done with it. The following are the synopsis and explanations of the the usage of the functions SDL_init(), SDL_Quit(), and SDL_QuitSubSystem(), where "Uint32" refers to data type 32-bit unsigned integer.

Synopsis	#include “SDL.h”
	int **SDL_Init**(Uint32 flags);

Description	Initializes SDL. This should be called before all other SDL functions. The flags parameter specifies what part(s) of SDL to initialize. (Flags should be bitwise-ORed together, e.g. “SDL_INIT_AUDIO | SDL_INIT_VIDEO”.)
Returns	-1 on error, 0 on success. When error occurred, you can obtain more information about it by calling SDL_GetError().

SDL_INIT_TIMER	Initializes the timer subsystem.
SDL_INIT_AUDIO	Initializes the audio subsystem.
SDL_INIT_VIDEO	Initializes the video subsystem.
SDL_INIT_CDROM	Initializes the cdrom subsystem.
SDL_INIT_JOYSTICK	Initializes the joystick subsystem.
SDL_INIT_EVERYTHING	Initialize all of the above.
SDL_INIT_NOPARACHUTE	Prevents SDL from catching fatal signals.
SDL_INIT_EVENTTHREAD	Run the event manager in a separate thread.

Synopsis	#include “SDL.h”
	void **SDL_Quit**(void);

Description	Shuts down all SDL subsystems and frees the resources allocated to them. You can set SDL_Quit as your atexit call for simplicity, like: atexit(SDL_Quit); However, it is not advisable to use atexit for large programs or dynamically loaded code.
Returns	none

Synopsis	#include “SDL.h”
	void **SDL_QuitSubSystem** (Uint32 flags);

Description	Shuts down a particular component of SDL, leaving others untouched.
Returns	none

The following example shows the initialization of SDL audio and video:

```
#include <stdlib.h>
#include "SDL.h"

main(int argc, char *argv[])
{
    if ( SDL_Init(SDL_INIT_AUDIO|SDL_INIT_VIDEO) < 0 ) {
        fprintf(stderr, "Unable to init SDL: %s\n", SDL_GetError());
        exit(1);
    }
    //atexit(SDL_Quit);
    SDL_Quit();
    ...
}
```

10.3.2 Video API

Every personal computer has a video card to process graphical data. There are numerous brands of video cards which differ in structures and functions. To hide the low-level programming details of the video hardware, people introduce the concept of **framebuffer** device which is an abstraction for the graphic hardware. It represents the frame buffer of some video hardware, and allows application software to access the graphic hardware through a well-defined interface, so that the software doesn't need to know anything about the low-level interface stuff. We may regard framebuffer as an area of memory that describes the image on the computer screen, with each screen pixel corresponding to a memory location.

SDL uses structures called **surfaces** to handle graphical data. A surface can be regarded as a memory block that stores a rectangular region of pixels. Like a framebuffer, a surface has widths, heights, and specific pixel formats. The rectangular region of data of a surface is often referred to as **bitmaps** or **pixmaps**.

SDL surfaces can be copied onto each other efficiently with one-to-one pixel mapping, which is referred to as **block image transfer** or **blit**. Blit operations are important in graphics programming as they allow a portion or complete image to be transferred from a memory buffer working in the background to the display screen effectively. Since the framebuffer can be also regarded as a surface, one can display the entire image on the screen with a single blitting operation. In practice, graphics applications such as video games rely mainly on blits to show the graphics rather than writing to individual pixels one by one. As an example, a video game's artwork is created by artists and saved in files. The game program assembles the images on screen from those predrawn graphics.

Choosing and setting video modes

Before writing to the frame buffer, we need to tell the video card what features we need. We can choose our desired bit-depth and resolution, and set it using the SDL_SetVideoMode() function, which returns an SDL_Surface where all graphics data, including the screen, are stored. You can choose a video mode based on the following information: *Full screen* or *windowed, Screen size*, *Window properties* (has-border, resize, ..), *Bits per Pixel*, and *Surface type* (software surface, hardware surface).

The following two examples ask for a 640 × 480 screen software surface with a pixel format that is the same as the current display setting:

```
//Example:
int options = (
    SDL_ANYFORMAT  |
    SDL_FULLSCREEN |
    SDL_SWSURFACE
);

SDL_Surface *screen = NULL;

screen = SDL_SetVideoMode(640, 480, 0, options);
if (NULL == screen)
{
    printf("Can't set video mode");
    exit(1);
}
```

```
//Example:

//setvideo.cpp
//compile by: g++ -o setvideo setvideo.cpp -I/usr/include \
//            -L/usr/local/lib -lSDL
#include <SDL/SDL.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
  SDL_Surface *screen;

  //initialize video system
  if ( SDL_Init( SDL_INIT_VIDEO ) < 0 ) {
  fprintf(stderr, "Unable to init SDL: %s\n", SDL_GetError());
    exit(1);
  }
  //ensure SDL_Quit is called when the program exits
  atexit(SDL_Quit);

  //set video mode of 640 x 480 with 16-bit pixels
  screen = SDL_SetVideoMode(640, 480, 16, SDL_SWSURFACE);
  if ( screen == NULL ) {
    fprintf(stderr,"Unable to set video:%s\n",SDL_GetError());
    exit(1);
  }

  SDL_Delay ( 2000 ); //delay 2 seconds before exit
  printf("Setting video mode successful!\n");
  return 0;
}
```

Drawing pixels on the screen

Drawing to the screen is done by writing directly to the graphics framebuffer, and calling the screen update function. Nothing we draw on a software surface is visible until it has been copied from memory to the display buffer on the video card. SDL provides two ways to do that: **SDL_Flip**() and **SDL_UpdateRect**(). **SDL_Flip**() copies the entire software surface to the screen. If the screen is set to 640×480 at 4 bytes per pixel, **SDL_Flip**() will copy 1.2 megabytes per frame and the frame rate will be limited by how fast our computer can copy images to the screen.

SDL_UpdateRects() is designed to let us use a “dirty pixels” scheme. It lets us specify a list of rectangular areas that have been changed and only copies those areas to the screen. This technique is ideal for a game with a complex background but only a small number of moving or changing items. Tracking dirty pixels can give us a dramatic improvement in performance. **SDL_UpdateRect**() updates one rectangular area.

Synopsis	#include "SDL.h"
	int **SDL_Flip**(SDL_Surface *screen);

Description	Swaps the background and foreground buffers for systems that support double-buffering. For systems not supporting double-buffering, it is the same as SDL_UpdateRect(screen, 0, 0, 0, 0).
Returns	none

Synopsis	#include "SDL.h"
	void **SDL_UpdateRect**(SDL_Surface *screen, Sint32 x, Sint32 y, Sint32 w, Sint32 h);

Description	Makes sure the given area is updated on the given screen. The rectangle must be confined within the screen boundaries (no clipping is done). If 'x', 'y', 'w' and 'h' are all 0, SDL_UpdateRect will update the entire screen.
Returns	none

Program Listing 10-1 presents an example that draws a pixel on a screen with arbitrary format. In the program, Uint8, Uint16, and Uint32 represent data types 8-bit, 16-bit and 32-bit unsigned integer (unsigned char, unsigned short, and unsigned int) respectively.

Program Listing 10-1: Drawing a Pixel on Screen

```
//Example: Drawing a pixel on screen
void DrawPixel(SDL_Surface *screen, Uint8 R, Uint8 G, Uint8 B){
  Uint32 color = SDL_MapRGB(screen->format, R, G, B);

  if ( SDL_MUSTLOCK(screen) )
    if ( SDL_LockSurface(screen) < 0 )
      return;
  switch (screen->format->BytesPerPixel) {
  case 1: { /* Assuming 8-bpp */
    Uint8 *bufp;
    bufp = (Uint8 *)screen->pixels + y*screen->pitch + x;
    *bufp = color;
  } break;
  case 2: { /* Probably 15-bpp or 16-bpp */
    Uint16 *bufp;
    bufp = (Uint16 *)screen->pixels + y*screen->pitch/2 + x;
    *bufp = color;
  } break;
  case 3: { /* Slow 24-bpp mode, usually not used */
    Uint8 *bufp;
    bufp = (Uint8 *)screen->pixels + y*screen->pitch + x;
    *(bufp+screen->format->Rshift/8) = R;
    *(bufp+screen->format->Gshift/8) = G;
    *(bufp+screen->format->Bshift/8) = B;
  } break;
  case 4: { /* Probably 32-bpp */
    Uint32 *bufp;
    bufp = (Uint32 *)screen->pixels + y*screen->pitch/4 + x;
    *bufp = color;
  } break;
  }
  if ( SDL_MUSTLOCK(screen) )
    SDL_UnlockSurface(screen);
  SDL_UpdateRect(screen, x, y, 1, 1);
}
```

Loading and displaying images

SDL provides one single image loading function, **SDL_LoadBMP**() for users to load a Windows BMP image onto an SDL surface. The function returns a pointer to an SDL_Surface structure containing the image, or a NULL pointer for failure of loading the image. The user can then display the loaded image by using **SDL_BlitSurface**() to blit it into the graphics framebuffer. SDL_BlitSurface() automatically clips the blit rectangle, which should be passed to SDL_UpdateRect() to update the portion of the screen which has changed. Bitmaps use dynamically allocated memory; if no longer needed they should be freed by using the function **SDL_FreeSurface**(). Conversely, the function **SDL_SaveBMP**() allows us to save an SDL surface as a BMP file.

If we need to process images in other popular formats like **.png, .jpg** or **.gif**, we need to install the **SDL_image**- library. After we have installed it, our program needs to include SDL/SDL_image.h and link with "-lSDL_image". Documentation of this library can be found at *http://jcatki.no-ip.org/SDL_image/SDL_image.html*. The library supports BMP, PNM (PPM/PGM/PBM), XPM, LBM, PCX, GIF, JPEG, PNG, TGA, and TIFF formats.

We can use the function **IMG_Load**() to load an image onto an SDL surface. It is best to call this function outside of event loops, and keep the loaded images around until you are really done with them. This is because the loading process could be time-consuming. When we have finished using the image, we should use **SDL_FreeSurface**() to free the allocated resources. The usage of **IMG_Load()** is shown below.

Synopsis	#include "SDL_image.h"
	SDL_Surface *IMG_Load (const char **ifile*);
Description	Loads *ifile* for use as an image onto a new SDL_Surface.
Returns	A pointer to the new SDL_Surface where the image is loaded.

If we are loading an image to be displayed many times, we can improve blitting speed by converting it to the format of the screen. The function **SDL_DisplayFormat**() does this conversion.

Listing 10-2 presents a complete program that loads an image using **IMG_Load**(). You also need to include the following headers in your program:

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

Program Listing 10-2: Loading Image using IMG_Load()

```
//Example: Loading an Image using IMG_Load()
/*
loadimage.cpp
```

```
compile by: g++ -o loadimage loadimage.cpp -I/usr/include \
            -L/usr/local/lib -lSDL -lSDL_image
*/
bool load_image(SDL_Surface *screen, char *image_name,int x,int y){
  SDL_Surface *image;
  SDL_Rect source, offset; //offset is the destination

  image = IMG_Load(  image_name );
  if ( image == NULL ) {
    printf ( "Unable to load image\n" );
    return false;
  }
  source.x = 0;    source.y = 0;
  source.w = image->w; source.h=image->h;//display the whole image
  offset.x = x;    offset.y = y; //position to display the image
  offset.w = image->w;   //width and height here actually NOT used
  offset.h = image->h;

  //Draws image data to the screen:(image,source) is the source,
  //  (screen, offset) is the destination
  SDL_BlitSurface ( image, &source, screen, &offset );
  //free the resources allocated to image
  SDL_FreeSurface ( image );
  return true;
}

int main(){
  SDL_Surface *screen;
  //initialize video system
  if ( SDL_Init( SDL_INIT_VIDEO ) < 0 ) {
        fprintf(stderr, "Unable to init SDL: %s\n", SDL_GetError());
        exit(1); }
  //ensure SDL_Quit is called when the program exits
  atexit(SDL_Quit);
  //set video mode of 640 x 480 with 16-bit pixels
  screen = SDL_SetVideoMode(640, 480, 16, SDL_SWSURFACE);
  if ( screen == NULL ) {
        fprintf(stderr, "Unable to set video: %s\n",SDL_GetError());
        exit(1);
  }
  //put image near center of screen
  if ( !load_image ( screen, "test-image.gif", 320, 240 ) )
        exit ( 1 );
  //update the entire screen
  SDL_UpdateRect ( screen, 0, 0, 0, 0 );
  SDL_Delay ( 4000 );   //delay 4 seconds before exit
  return 0;
}
```

10.4 SDL Events

Programs that operate in a GUI environment are event-driven. An event is an action that takes place within a program when something happens. Part of writing a GUI application is to create event listeners. An event listener is an object or a loop that triggers certain action when a specific event occurs. In the SDL programming environment, an event is produced whenever we move or click the mouse, press a key, or resize the SDL video window. Event handling allows our application to receive input from the user. SDL stores unprocessed events in an internal event queue which allows SDL to collect as many events as possible each time it updates an event. Using functions like **SDL_PollEvent, SDL_PeepEvents** and **SDL_WaitEvent** we can observe and handle waiting input events. There are four main categories of events: keyboard, mouse, window, and system-dependent events. Window events handle gaining and losing focus, as well as exit requests. System-dependent events process raw messages from the windowing system that SDL otherwise would ignore. Information of an event is stored in the structure type SDL_Event. The event queue itself is composed of a series of **SDL_Event** unions, one for each waiting event. **SDL_Event** unions are read from the queue with the **SDL_PollEvent** function and it is then up to the application to process the information stored with them. Its definition is shown below.

Synopsis	SDL_Event : General event structure, which is a union of all possible event types for receiving events from SDL.
Definition	see below
Description	For reading and placing events on the event queue.

```
typedef union{
   Uint8 type;
   SDL_ActiveEvent active;
   SDL_KeyboardEvent key;
   SDL_MouseMotionEvent motion;
   SDL_MouseButtonEvent button;
   SDL_JoyAxisEvent jaxis;
   SDL_JoyBallEvent jball;
   SDL_JoyHatEvent jhat;
   SDL_JoyButtonEvent jbutton;
   SDL_ResizeEvent resize;
   SDL_ExposeEvent expose;
   SDL_QuitEvent quit;
   SDL_UserEvent user;
   SDL_SysWMEvent syswm;
} SDL_Event;
```

The SDL event subsystem is intertwined with the video subsystem. We basically cannot separate the use of them. Therefore, they are both initialized with **SDL_INIT_VIDEO** parameter to **SDL_Init**().

10.4.1 Event Processing

To process an event, we need to read the **SDL_Event** unions from the event queue using the **SDL_PollEvent**() or **SDL_WaitEvent**() function. We can also add events onto the event queue using **SDL_PushEvent**(), which returns 0 on success or -1 on failure. We can wait for events to occur using the SDL_WaitEvent() function, which waits indefinitely and returns only if an event or an error has occurred as explained below. On the other hand, we can peek at events in the event queue without removing them by passing the **SDL_PEEKEVENT** action to **SDL_PeepEvents**().

Synopsis #include “SDL.h”
int **SDL_WaitEvent**(SDL_Event **event*);

Description Waits indefinitely for the next available event.

Returns 0 if there was an error while waiting for events, 1 otherwise. Information of the event detected will be stored in the structure pointed by event and the event is removed from the event queue.

The following example, **waitevent.cpp** of Program Listing 10-3 shows how to use **SDL_Wait_Event**() to wait for various events.

Program Listing 10-3: Waiting for an event

```
//waitevent.cpp: Waiting for an event to occur.
bool wait_for_events () {
  SDL_Event event;
  int status;  char *key;
  bool quit = false;
  printf("waiting for events, press 'q' or 'ESC' to quit\n");
  while ( !quit ) {
    //wait indefinitely for an event to occur
    status = SDL_WaitEvent(&event);
                        //event will be removed from event queue
    if ( !status ) {     //Error has occurred while waiting
       printf("SDL_WaitEvent error: %s\n", SDL_GetError());
       return false;
    }
    switch (event.type) {            //check the event type
      case SDL_KEYDOWN:              //if a key has been pressed
        key = SDL_GetKeyName(event.key.keysym.sym);
        printf("The %s key was pressed!\n", key );
        if (event.key.keysym.sym == SDLK_ESCAPE)//quit if 'ESC'
          quit = true;
        else if ( key[0] == 'q'  )    //quit if 'q'  pressed
          quit = true;
        break;
      case SDL_MOUSEMOTION:          //mouse moved
        printf("Mouse motion x:%d, y:%d\n",
               event.motion.x, event.motion.y );   break;
      case SDL_MOUSEBUTTONUP:        //mouse button pressed
        printf("Mouse pressed x:%d, y:%d\n",
                event.button.x, event.button.y );  break;
      case SDL_QUIT:                 //'x' of Window clicked
         exit ( 1 );
    }
  } //while
  return true;
}
int main() {
  SDL_Surface *screen;
  if ( SDL_Init( SDL_INIT_VIDEO ) < 0 ) {
    fprintf(stderr, "Unable to init SDL: %s\n", SDL_GetError());
    exit(1);
  }
  //ensure SDL_Quit is called when the program exits
  atexit(SDL_Quit);
  //set video mode of 640 x 480 with 16-bit pixels
  screen = SDL_SetVideoMode(640, 480, 16, SDL_SWSURFACE);
  if ( screen == NULL ) {
    fprintf(stderr, "Unable to set video: %s\n", SDL_GetError());
    exit(1);
  }
  wait_for_events();        return 0;
}
```

We can also poll for events using the **SDL_PollEvent**() function. In addition to handling events directly, each type of event has a function which allows us to check the application event state. If we use this exclusively, we should ignore all events with the **SDL_EventState**() function, and call **SDL_PumpEvents**() periodically to update the application event state. The following piece of code shows how to poll events and pump events:

```
//Example: Polling Event State
{ SDL_EventState(SDL_MOUSEMOTION, SDL_IGNORE);}

void CheckMouseHover(void)
{ int mouse_x, mouse_y;
  SDL_PumpEvents();
  SDL_GetMouseState(&mouse_x, &mouse_y);
  if ( (mouse_x < 32) && (mouse_y < 32) )
   printf("Mouse in upper left hand corner!\n");
}

//Example: Pumping events.
{ SDL_EventState(SDL_MOUSEMOTION, SDL_IGNORE); }

void CheckMouseHover(void) {
  int mouse_x, mouse_y;
  SDL_PumpEvents();
  SDL_GetMouseState(&mouse_x, &mouse_y);
  if ( (mouse_x < 32) && (mouse_y < 32) )
    printf("Mouse in upper left hand corner!\n");
}
```

10.5 SDL Threads

The effective use of threads is very important in modern programming. It allows a program to execute multiple parts of itself simultaneously in the same address space. In many cases, we basically cannot accomplish the tasks without using threads. For instance, consider a game program that needs to accept inputs from the mouse and keyboard, and at the same time has to play music at the background and generate some special sounds at various stages; it will be extremely difficult if not impossible to achieve these effects in our program without using threads. Of course, we use threads only when we have to. We are not replacing simple nonthreaded programs with fancy, complex, threaded ones. Threads are just one more way we can use to make our programming tasks easier. The main benefits of using threads in programming include the following:

- gaining performance from multiprocessor hardware,
- easier programming for jobs with multi-tasks,
- increasing job throughput by overlapping I/O tasks with computational tasks,
- more effective use of system resources by sharing resources between threads,
- using only one binary to run on both uniprocessors and multiprocessors,
- creating well-structured programs, and
- maintaining a single source for multiple platforms.

Multithreaded applications are ubiquitous in modern computing systems. For example, a Web server makes use of threads to dramatically improve performance and interactivity. Typically, a Web server receives requests from remote clients for Web pages, images, multimedia data and other information; the Web server has one main thread listening to the requests and creates a seperate thread to service each request. Upon receiving a request, the main thread spawns a new thread that interprets the request, retrieves the specified Web pages and transmits the data to the client (typically a Web browser). After spawning a new thread, the main thread (the parent) can continue to listen for new requests. If the Web server runs on a multiprocessor system, it can receive and fulfill several requests at the same time by creating different threads, and thus improves both the throughput and response time of the system. Another commonly cited example that makes use of threads to enhance user productivity and improve interactivity is a word processor, which uses one thread to accept commands from a user and another thread to carry out the commands at the background. For example, many modern word processors detect mispelled words as they are typed and periodically save a copy of the document to disk to prevent loss of data. Each feature is implemented with a separate thread; consequently, the word processor can respond to keyboard interrupts even if one or more of its threads are blocked because of other I/O activities (e.g. saving a copy of the document to disk).

10.5.1 What Are Threads

A thread is also referred to as a **light weight process** (**LWP**). A process is a program in execution. It is a unit of work in a modern time-sharing system. You can create several processes from the same program. A process not only includes the program code, which is sometimes referred to as the text section, but also the current activities and consumed resources including the program counter, processor registers, the process stack (which contains temporary data such as function parameters, return address, and local variable), and the data section, which contains global variables. A process may also have a heap, which is the memory dynamically allocated to it during run time.

A **thread** is a basic unit of CPU utilization, comprising a **thread ID**, a **program counter**, a **register set**, and a **stack**. It shares with its other threads of the same process its code section, data section, and other operating-system resources, such as open files, signals, and global variables. The various states of a thread can be represented by Figure 10-1, where quantum refers to the time the computer allocated to run a thread before switching to running another thread.

10.5.2 Pthreads

IEEE defines a POSIX standard API, referred to as **Pthreads** (IEEE 1003.1c), for thread creation and synchronization. Many contemporary systems, including Linux, Solaris, and Mac OS X implement Pthreads. To use Pthreads in your program, you must include **pthread.h** and link with **-l pthread**. The following exmaple, **pthreads_demo.cpp** of Listing 10-4 shows how to use Pthreads.

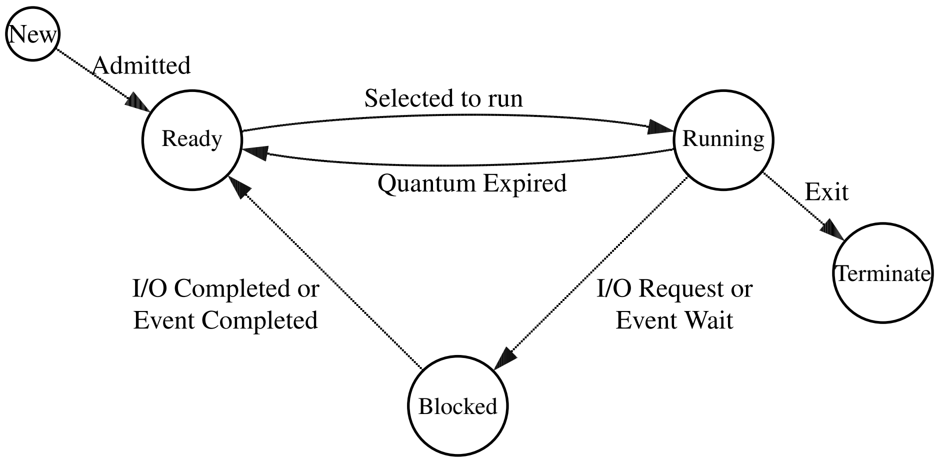

Figure 10-1. States of a Thread

Program Listing 10-4: Pthread Example

```
/*
  pthreads_demo.cpp
  A very simple example demonstrating the usage of pthreads.
  Compile: g++ -o pthreads_demo pthreads_demo.cpp -lpthread
  Execute: ./pthreads_demo
*/

#include <pthread.h>
#include <stdio.h>

using namespace std;

//The thread
void *runner ( void *data )
{
  char *tname = ( char * )data;

  printf("I am %s\n", tname );

  pthread_exit ( 0 );
}

int main ()
{
  pthread_t id1, id2;            //thread identifiers
  pthread_attr_t attr1, attr2;   //set of thread attributes
  char *tnames[2] = { "Thread 1", "Thread 2" }; //names of threads

  //get the default attributes
  pthread_attr_init ( &attr1 );
  pthread_attr_init ( &attr2 );

  //create the threads
  pthread_create ( &id1, &attr1, runner, tnames[0] );
  pthread_create ( &id2, &attr2, runner, tnames[1] );

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

  return 0;
}
```

In the example of **pthreads_demo.cpp** shown in Program Listing 10-4, we use **pthread_tid** to declare the identifiers for the threads we are going to create. Each thread has a set of attributes containing information about the thread like stack size and scheduling information. We use **pthread_attr_t** to declare the attributes of the threads and set the attributes in the function called by **pthread_attr_init**(). As we did not explicitly set any attributes, the default attributes will be used. The function **pthread_create**() is used to create a separate thread. In addition to passing the thread identifier and the attributes to the thread, we also pass the name of the function, *runner*, where the new thread will begin execution. The last argument passed to **pthread_create**() in the example is a string parameter containing the name of the thread. At this point, the program has three threads: the initial parent thread in **main**() and two child threads in **runner**(). After creating the child threads, the **main**() thread will wait for the **runner**() threads to complete by calling **pthread_join**() function.

Pthreads specification has a rich set of functions, allowing users to develop very sophisticated multithreaded programs. In the applications discussed here, we do not need to use many of the Pthread functions. So instead we will use SDL threads, which are much simpler and only consist of a few functions. Moreover, SDL is platform independent. Besides POSIX threads, different crucial thread programming schemes exist in the market. MS Windows has its own threading interface which is very different from POSIX threads. Though Sun's Solaris supports Pthreads, it also has its own thread API. Other UNIX systems may also have their own thread APIs. SDL solves this inconsistency with its own set of portable threading functions.

10.5.3 SDL Thread Programming

The mechanisms of using SDL Threads are basically the same as that of Pthreads. The thread functions are similar except that the names are different. We start new threads with the **SDL_CreateThread**() function, which returns a thread handle of type **SDL_Thread** for subsequent thread operations. The above Pthreads example can be rewritten using SDL threads as shown in Listing 10-5. The code is similar to and a little simpler than the Pthreads code of Listing 10-4. The **SDL_WaitThread**() function works in the same way as the

Pthreads **pthread_join**(), which waits a thread to complete. We need to do a **-lpthread** link in compilation because some of the SDL thread implementations are based on the Pthreads libraries.

As mentioned above, the SDL's threading API is a simplified set of threading functions. Its implementation is somewhat incomplete. For example, SDL does not allow a program to change a thread's scheduling priority or other low-level attributes. Actually, these features are highly system-dependent and supporting them would be difficult to make SDL platform independent. For many video applications, the SDL's threading API is sufficient. The following shows the detailed usage of some of the basic threading functions of SDL.

Synopsis	#include "SDL.h"
	#include "SDL_thread.h"
	SDL_Thread ***SDL_CreateThread**(int (**fn*)(void *), void **data*);

Description	Creates a new thread of execution that shares all of its parent's global memory, signal handlers, file descriptors, etc, and runs the function fn(), which utilizes the void pointer data passing to it. The thread quits when fn() returns.
Returns	A pointer to the thread of type SDL_Thread.

Synopsis	#include "SDL.h"
	#include "SDL_thread.h"
	void **SDL_KillThread** (SDL_Thread **thread*);

Description	racelessly terminates the thread associated with the thread. You should avoid using it to terminate a thread. If possible, use some other form of IPC to signal the thread to quit.
Returns	None

Synopsis	#include "SDL.h"
	#include "SDL_thread.h"
	void **SDL_WaitThread**(SDL_Thread **thread*, int **status*);

Description	Waits for a thread to finish (timeouts are not supported).
Returns	The return code for the thread function is placed in the area pointed to by status, if status is not NULL.

Program Listing 10-5: Example of SDL Thread Programming

```
/*
  sdlthreads_demo.cpp
  A very simple example demonstrating the usage of sdl threads.
  Compile:g++ -o sdlthread_demo sdlthread_demo.cpp -lSDL -lpthread
  Execute:  ./sdlthread_demo
*/

#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#include <stdio.h>

using namespace std;

//The thread
```

```
int runner ( void *data )
{
  char *tname = ( char * )data;
  printf("I am %s\n", tname );
  return 0;
}

int main ()
{
  SDL_Thread *id1, *id2;                //thread identifiers
  char *tnames[2] = { "Thread 1", "Thread 2" }; //names of threads

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

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

  return 0;
}
```

The following table lists all the SDL Thread functions:

Table 10-1 SDL Threading Functions

Functions	**Descriptions**
SDL_CreateThread()	Creates a new thread that shares its parent's properties.
SDL_ThreadID()	Gets the 32-bit thread identifier for the current thread.
SDL_GetThreadID()	Gets the SDL thread ID of a SDL_Thread
SDL_WaitThread()	Waits for a thread to finish.
SDL_KillThread()	Gracelessly terminates the thread.
SDL_CreateMutex()	Creates a mutex.
SDL_DestroyMutex()	Destroys a mutex.
SDL_mutexP()	Locks a mutex.
SDL_mutexV()	Unlocks a mutex.
SDL_CreateSemaphore()	Creates a new semaphore and assigns an initial value to it.
SDL_DestroySemaphore()	Destroys a semaphore that was created by SDL_CreateSemaphore.
SDL_SemWait()	Locks a semaphore; suspends the thread if semaphore value is zero.
SDL_SemTryWait()	Attempts to lock a semaphore but does not suspend thread.
SDL_SemWaitTimeout()	Locks a semaphore; waits up to a specified maximum time.
SDL_SemPost()	Unlocks a semaphore.
SDL_SemValue()	Returns the current value of a semaphore.
SDL_CreateCond()	Creates a condition variable.
SDL_DestroyCond()	Destroys a condition variable.
SDL_CondSignal()	Resumes a thread waiting on a condition variable.
SDL_CondBroadcast()	Resumes all threads waiting on a condition variable.
SDL_CondWait()	Waits on a condition variable.
SDL_CondWaitTimeout()	Waits on a condition variable, with timeout Time.

10.6 A Simple PPM Viewer

We have discussed the reading and writing of PPM files in Chapter 4. Here we want to develop a program that displays the image of a PPM file on the screen so that we can visually compare images before and after compression directly. It is surprisingly simple to use SDL to render an image. The program **ppmviewer.cpp** presented in Listing 10-6 accomplishes the task. The program is a slight modification of the PPM demo program, **ppmdemo.cpp** of Listing 4-1 discussed in Chapter 4. A few lines of codes related to SDL are added to display a PPM image. An SDL surface is created using the SDL function **SDL_SetVideoMode**(), and we use the variable *screen* to point to the newly created surface. After loading the image data into the memory buffer *ibuf*, we simply point the framebuffer to the data buffer, which is done by the statement "screen→pixels = ibuf;". There is a catch here. While PPM saves image data in the order red, green, and blue, SDL processes data in the opposite order, blue, green and red. To make things consistent, we provide the function **ppm2sdl**() that converts the PPM format to SDL format. After making the conversion, the function **SDL_UpdateRect**() is used to send the data to the screen efficiently. **SDL_Delay**() is called to delay the program exit by four seconds. In other words, the image will be displayed for four seconds on the screen.

Program Listing 10-6: Viewing PPM Images Using SDL

```
/* ppmviewer.cpp
 * Demostrate rendering a PPM file.
 * Slight modification of ppmdemo.cpp; SDL is used to display image.
 *
 * Compile: g++ -o ppmviewer ppmviewer.cpp -L/usr/local/lib -lSDL
 * Execute: ./ppmviewer
 */

#include <stdio.h>
#include <stdlib.h>
#include <SDL/SDL.h>

//A public class is the same as a 'struct'
class CImage {
public:
  unsigned char red;
  unsigned char green;
  unsigned char blue;
};

void ppm_read_comments ( FILE *fp )
{
  int c;
  while ( (c = getc ( fp ) )  == '#' ) {
    while (  getc( fp )  != '\n' )
;
  }
  ungetc ( c, fp );
}

class ppm_error
{
  public:
    ppm_error() {
      printf("\nIncorrect PPM format!\n");
      exit ( 1 );
    }
};


//change from (R, G, B) to (B, G, R)
void  ppm2sdl ( CImage *ibuf, int width, int height )
{
  unsigned char temp;

  for ( int i = 0; i < height; ++i ) {
    int row_offset = i * width;
    for ( int j = 0; j < width; ++j ){
      int offset =  row_offset + j;
      temp = ibuf[offset].red;
      ibuf[offset].red = ibuf[offset].blue;
      ibuf[offset].blue = temp;
    }
  }
}

int main()
{
  int ppmh[20], c;             //PPM header
  int width, height;        //image width and height
  SDL_Surface *screen;
  char filename[] = "../data/beach.ppm"; //hard-coded filename

  FILE *input = fopen (filename, "rb");  //PPM file for testing read
  if ( !input ) {
    printf("\nError opening input file!\n");
    return 1;
  }

  //read PPM input file
  ppm_read_comments ( input );  //read comments
  char temp[100];
  fscanf ( input, "%2s", temp );
  temp[3] = 0;
  if ( strncmp ( temp, "P6", 2 ) )
    throw ppm_error();
  ppm_read_comments ( input );
  fscanf ( input, "%d", &width );
  ppm_read_comments ( input );
  fscanf ( input, "%d", &height );
  ppm_read_comments ( input );
  int colorlevels;
  fscanf ( input, "%d", &colorlevels );
  printf("\n%s PPM file: ", temp );
  printf(" \n\twidth=%d\theight=%d\tcolorlevles=%d\n", width,
             height,colorlevels+1 );
  ppm_read_comments ( input );
  while (( c = getc ( input )) == '\n');//get rid of extra returns
  ungetc ( c ,input );

  // May use CImage ibuf[width][height] if we do not use SDL_QUIT;
  CImage *ibuf = (CImage *) malloc ( width * height * 3 );
  fread ( ibuf,  3, width * height, input );   //read image data
  fclose ( input );

  //initialize video system
  if ( SDL_Init( SDL_INIT_VIDEO ) < 0 ) {
    fprintf(stderr, "Unable to init SDL: %s\n", SDL_GetError());
    exit(1);
  }
  //ensure SDL_Quit is called when the program exits
  atexit(SDL_Quit);//if not use this, we need to
                  //  do house cleaning manually when program ends

  //set video mode of width x height with 24-bit pixels
  screen = SDL_SetVideoMode( width, height, 24, SDL_SWSURFACE);
  if ( screen == NULL ) {
        fprintf(stderr, "Unable to set %dx%d video: %s\n", width,
                  height, SDL_GetError());
        exit(1);
  }

  //convert PPM format (R, G, B)  to SDL format (B, G, R)
  ppm2sdl ( ibuf,  width, height );

  screen->pixels = ibuf;  //point framebuffer to data buffer
  //  ibuf needs to be dynamically allocated if SDL_QUIT is used
  SDL_UpdateRect ( screen, 0, 0, 0, 0 );   //blit data to screen

  SDL_Delay ( 4000 );     //delay 4 seconds before exit
  printf("Displaying PPM image %s successful!\n", filename );

  //do NOT free(ibuf) if use SDL_QUIT which does the house cleaning
  return 0;
}
```

10.7 The Producer-Consumer Problem

The program **ppmviewer.cpp** presented in the previous section is a single-threaded program. We can modify it to a "video player" by adding a loop in **main**() to display a sequence of images: it reads in the image data from a file and saves it in the buffer *ibuf*, points the framebuffer to the data buffer, blits the data to the screen, waits for a fixed period of time, and repeats the process by reading in another set of image data. Such a "video player" is very inflexible. The whole program is dedicated to a single task, reading and displaying images. It cannot do other things like playing music or accepting inputs. If we add a decoder in the loop, it becomes difficult to synchronize the displaying speed and the decoding speed. We can overcome these shortcomings by changing the program to a multi-threaded program, and using the producer-consumer concept to handle the synchronization between decoding and rendering.

The producer-consumer problem is a common paradigm for thread synchronization. A **producer** thread produces information which is consumed by a **consumer** thread. This is in analog with whats happening in a fast-food restaurant. The chef produces food items and put them on a shelf; the customers consume the food items from the shelf. If the chef makes food too fast and the shelf is full, she must wait. On the other hand, if the customers consume food too fast and the shelf is empty, the customers must wait.

To allow producer and consumer threads to run concurrently (simultaneously), we must make available a buffer that can hold a number of item and **shared** by the two threads; the producer fills the buffer with items while the consumer empties it. A producer can produce an item while the consumer is consuming another item. Trouble arises when the producer wants to put a new item in the buffer, which is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and finds it empty, it goes to sleep until the producer puts something in the buffer and wakes the consumer up. The **unbounded-buffer** producer-consumer problem places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items without waiting. The **bounded-buffer** producer-consumer problem puts a limit on the buffer size; the consumer must wait when it is empty, and the producer must wait when the buffer is full.

The approach sounds simple enough, but if not properly handled, the two threads may **race** to access the buffer and the final outcome depends on who runs first. There is a simple technique to resolve the *race conditions*. E.W. Dijkstra introduced the concept of **semaphore** to handle synchronization problems in 1965. A semaphore is an integer variable associated with two operations, *down* and *up*, a generalization of *sleep* and *wakeup*, respectively. The *down* operation checks if the semaphore value is greater than 0. If yes, it decrements it and continues; if the semaphore value is 0, the thread is put on sleep. Checking the value, changing it, and possibly going to sleep are all done in a single, indivisible, **atomic action**. This it to guarantee that once a semaphore operation has started, no other thread can access the semaphore until the operation has completed or blocked. SDL provides semaphore operations, **SDL_SemPost**(), and **SDL_SemWait**() corresponding to the *up* and *down* operations we just mentioned.

The program **proconsumer.cpp** presented in Listing 10-7 is a simple example of using SDL threads as well as an example of solving the producer-consumer problem using semaphores. The program uses three semaphores, *nfilled*, *nempty*, and *mutex* to do synchronization, with *nfilled* used for counting the number of slots that are filled, *nempty* for counting the number of empty slots, and *mutex* to ensure the producer and consumer do not access the buffer at the same time. *nfilled* initially has a value of 0; *nempty* has an initial value equal to the number of slots in the buffer, and *mutex* has an initial value of 1. In general, the variable *mutex* refers to **mutual exclusion**, which is initialized to 1 and used by two or more threads to ensure only one of them can access a certain piece of code referred to as a critical section. This kind of special semaphore is called a **binary semaphore**; if each thread does a *wait* (down) operation just before entering the critical section and a *post* (up) operation just after leaving it, mutual exclusion is guaranteed.

In the program, the buffer we use is a circular queue. The producer inserts an item at the tail of the queue and the consumer removes an item at the head of it. We advance the tail and head pointers after an insert and a remove operation respectively. The pointers wrap around when they reach the "end" of the queue. This concept is illustrated in Figure 10-2.

Program Listing 10-7: Solving Producer-Consumer Problem Using Semaphores

```
/*
  proconssumer.cpp
  An example of using SDL threads and semaphores.
  The producer-consumer problem using semaphores.
  Compile:  g++ -o proconsumer proconsumer.cpp -lSDL -lpthread
  Execute:  ./proconsumer
*/
#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#include <stdio.h>
#include <stdlib.h>
using namespace std;

#define N     5          //number of slots in buffer
SDL_sem *mutex;          //controls access to critical region
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 ) {
    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 critical section
    insert_item ( item );
    SDL_SemPost ( mutex );         //leave Critical Section
    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 ) {
    SDL_Delay ( rand() % 1000 );    //random delay
    SDL_SemWait ( nfilled );        //decrement filled count
    SDL_SemWait ( mutex );          //entering critical section
    printf("\n%s : ", tname );
    item = remove_item ();          //take item from buffer
    SDL_SemPost ( mutex );          //leave Critical Section
    SDL_SemPost ( nempty );         //increment empty slot count
    //can do something with item here
  }
}

int main ()
{
  SDL_Thread *id1, *id2, *id3;        //thread identifiers
  char *tnames[3] = { "Producer", "Consumer" }; //names of threads

  mutex = SDL_CreateSemaphore ( 1 );  //initialize mutex to 1
  nempty = SDL_CreateSemaphore ( N ); //initially all slots empty
  nfilled  = SDL_CreateSemaphore ( 0 );  //no slot filled
  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;
}
```

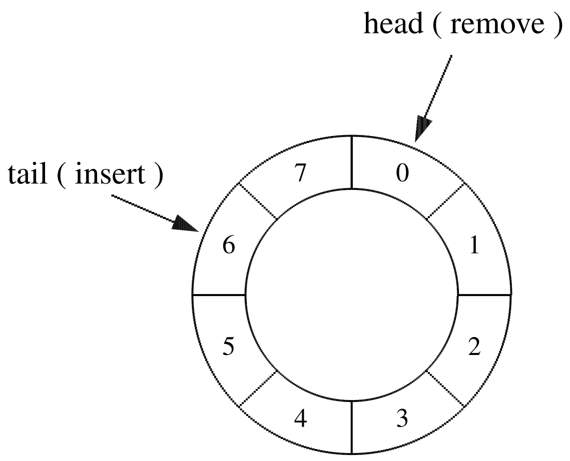

Figure 10-2. Circular Queue with Eight Slots

The program can be compiled with the command,

```
$g++ -o proconsumer proconsumer.cpp -lSDL -lpthread
```

The following are sample outputs of the program when we execute **proconsumer**:

Table 10-2 Sample Outputs of **proconsumer**

```
Producer : insert 777 at 0
Consumer : remove 777 at 0
Producer : insert 335 at 1
Consumer : remove 335 at 1
Producer : insert 649 at 2
Producer : insert 362 at 3
Producer : insert 690 at 4
Consumer : remove 649 at 2
Producer : insert 926 at 0
Producer : insert 426 at 1
Producer : insert 736 at 2
Consumer : remove 362 at 3
Producer : insert 567 at 3
Consumer : remove 690 at 4
Producer : insert 530 at 4
Consumer : remove 926 at 0
Consumer : remove 426 at 1
Consumer : remove 736 at 2
Producer : insert 929 at 0
Consumer : remove 567 at 3
Consumer : remove 530 at 4
```

10.8 A Multi-threaded Raw Video Player

In this section, we put together what we have learned to develop a simple multi-threaded video player. Our concern here is to illustrate the concept of playing video data in an effective way. To simplify things, we hard-code the dimensions of a frame and assume that there's no compression in the data. We shall see that we can easily generalize the player to accommodate encoded data and data attributes.

A single-thread raw video player is easy to implement: it just sits in a main loop to read in the video data and blits them to the screen, waits for a while and repeats the data reading and blitting. In practice, a player has to handle various tasks besides blitting data on the screen. It may have to decode the data or to process the audio; a single-threaded player tangles all the tasks together and one needs to worry about the coordination between various tasks. On the other hand, a multi-threaded program can handle these tasks much better, as playing data (sending data to screen) can be cleanly separated from other tasks. The following section describes how to develop a multi-threaded program named **tplayer.cpp** to play raw video data. The program and the sample raw data can be downloaded from this book's web site at *http://www.forejune.com/vcompress*.

In its simplest form, besides **main**(), our multi-threaded player needs two threads, one for sending data to the screen and one for 'decoding' data (at this moment, 'decoding' is simply reading from the file); suppose we name these threads **player**() and **decoder**() respectively. This becomes a classical producer-consumer problem. Here, **decoder**() is the producer which produces resources (video data) and **player**() is the consumer which consumes resources (video data).

As discussed above, in the producer-consumer problem, to allow producer and consumer threads to run concurrently, we must have available a buffer for holding items that can be filled by the producer and emptied by the consumer. A producer can produce zero or more items while the consumer is consuming an item; the number of items that can be produced or consumed depends on the production rate as well as the consumption rate and other factors that may influence the production and consumption operations. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced and the producer suspends production when the buffer is full as there will not be any space to hold the produced item. Therefore, the producer (**decoder**()) must wait when the buffer is full and the consumer (**player**()) must wait when the buffer is empty.

Typically, when the buffer is empty, the consumer goes to sleep and when the producer has finished producing an item and put it in the buffer, it is responsible to wake up the consumer. On the other hand, when the buffer is full, the producer goes to sleep and the consumer is responsible to wake up the producer after it has consumed an item. These can be implemented using the SDL semaphores discussed above. However, to slightly simplify our implementation of **tplayer.cpp**, each thread is responsible to wake up itself after sleeping (**SDL_Delay**()) for a certain period of time and check the buffer condition again. In our implementation, we create a buffer that can hold four frames of video data:

```
char *buf[4];
for ( int i = 0; i < 4; ++i ) {
    buf[i] = ( char * ) malloc ( frameSize );
    assert ( buf[i] );
}
```

The buffer *buf* acts as a circular queue. As usual, we maintain two pointers, *head* and *tail* like those in Figure 10-2 to manage the queue. Data are entered into the queue at the 'rear' where *tail* points at and are deleted (consumed) at the front where *head* points at. Each time, an item is produced, *tail* is advanced by 1 and when an item is consumed, *head* is advanced. When *head* equals *tail*, indicating that the buffer is empty, the **player**() thread must wait:

```
while ( !quit ) {
   if ( head == tail ) { //buffer empty (data not available yet)
     SDL_Delay ( 30 );   //sleep for 30 ms
     continue;
   }
   ....
 }
```

When *tail* is 4 ahead of *head*, it implies that the buffer is full and **decoder**() must wait:

```
while ( !quit ) {
  if ( tail >= head + 4 ) {  //buffer full
     SDL_Delay ( 30 );
     continue;
  }
  ....
}
```

Modulo 4 is taken on *head* and *tail* to point to the actual buffer location. Also, the SDL function **SDL_GetTicks**() is used to give an estimate of the time delay between two frames as shown in the following piece of code where a frame rate of 20 frames per second (fps) is assumed:

```
Uint32 prev_time, current_time;
current_time = SDL_GetTicks();          //ms since library starts
if ( current_time - prev_time < 50 )    //20 fps ~ 50 ms / frame
   SDL_Delay ( 50 - ( current_time - prev_time ) );
prev_time = current_time;
```

The variable *quit* is global. All the threads check the state of *quit* periodically; if it is **true**, the threads terminate. It is set to **true** when the key 'ESC' or 'q' is pressed or when there are no more data. The complete code of **tplayer.cpp** is listed in Listing 10-8 below. You may compile it with the command:

```
$g++ -o tplayer tplayer.cpp  -L/usr/local/lib -lSDL
```

Simply executing the command **tplayer** palys the raw video "../data/sample_video.raw".

Program Listing 10-8: Multi-threaded Simple Raw Video Player **tplayer.cpp**

```
/*
  tplayer.cpp
  A simple multi-threaded program to demonstrate playing raw video
  using SDL.  Assume that you have a raw video clip named
  "sample_video.raw" in directory "../data/" and assume that
  the image size is 320 x 240 and is 24 bits/pixel.
  Solution for producer-consumer problem is used to handle
  interface between "decoder" and  player. The player acts as a
  consumer and the "decoder" plays the role of a producer.
  Pressing 'ESC' ends the program.
  Compile by: g++ -o tplayer tplayer.cpp -L/usr/local/lib -lSDL
  Execute by: ./tplayer</b>
*/
#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <assert.h>

//shared variables
bool quit = false;
```

```
char *buf[4];
unsigned long head = 0,  tail = 0;
unsigned int frameSize;

//Consumer
int player ( void *scr )
{
  SDL_Surface *screen = ( SDL_Surface * ) scr;
  Uint32 prev_time, current_time;
  current_time = SDL_GetTicks();//ms since library starts
  prev_time = SDL_GetTicks();   //ms since library starts

  while ( !quit ) {
    if ( head == tail ) {  //buffer empty (data not available yet)
      SDL_Delay ( 30 );    //sleep for 30 ms
      continue;
    }
    //consumes the data
    screen->pixels = buf[head%4];
    current_time = SDL_GetTicks();      //ms since library starts
    if (current_time - prev_time < 50)  //20 fps ~ 50 ms / frame
      SDL_Delay ( 50 - (current_time - prev_time) );
    prev_time = current_time;

    SDL_UpdateRect ( screen, 0, 0, 0, 0 );//update whole screen
    head++;
  } //while
  return 0;
}

//Producer
int decoder ( void *data )
{
  static FILE *fp = NULL;
  long n;

  if ( fp == NULL )
    fp = fopen ( "../data/sample_video.raw", "rb" );
  if ( fp == NULL ) {
    printf("\nError opeing file\n" );
    quit = true;
    return 1;
  }
  while ( !quit ) {
     if ( tail >= head + 4 ) { //buffer full
        SDL_Delay ( 30 );
        continue;
     }
     //produce data
     n = fread ( buf[tail%4], frameSize, 1, fp );
     if ( n <= 0 )
       quit = true;
     else
       tail++;
  } //while
  return 0;
}

int main()
{
  SDL_Surface *screen;
  frameSize = 320 * 240 * 3;
  SDL_Thread *producer, *consumer;
  SDL_Event event;
  int status;
  char *key;

  //initialize video system
  if ( SDL_Init( SDL_INIT_VIDEO ) < 0 ) {
        fprintf(stderr, "Unable to init SDL: %s\n",SDL_GetError());
        exit(1);
  }
  //ensure SDL_Quit is called when the program exits
  atexit(SDL_Quit);

  //set video mode of 320 x 240 with 24-bit pixels
  screen = SDL_SetVideoMode(320, 240, 24, SDL_SWSURFACE);
  if ( screen == NULL ) {
        fprintf(stderr, "Unable to set 320x240 video: %s\n",
                         SDL_GetError());
        exit(1);
  }
  //create buffers to hold at most 4 frames of video data
  for ( int i = 0; i < 4; ++i ) {
    buf[i] = ( char * ) malloc ( frameSize );
    assert ( buf[i] );
  }
  consumer = SDL_CreateThread ( player, screen );
  producer = SDL_CreateThread ( decoder, ( void * ) "decdoing" );
  while (!quit)
  {
    //wait indefinitely for an event to occur
    status = SDL_WaitEvent(&event);
                          //event will be removed from event queue
    if ( !status ) {      //Error has occured while waiting
        printf("SDL_WaitEvent error: %s\n", SDL_GetError());
        quit = true;
        return false;
    }
    switch (event.type) {             //check the event type
      case SDL_KEYDOWN:               //if a key has been pressed
        key = SDL_GetKeyName(event.key.keysym.sym);
        printf("The %s key was pressed!\n", key );
        if ( event.key.keysym.sym == SDLK_ESCAPE )//quit if 'ESC'
          quit = true;
        else if ( key[0] == 'q'  )       //quit if 'q'  pressed
          quit = true;
    }
    SDL_Delay ( 100 );                  //give up some CPU time
  }

  SDL_WaitThread ( consumer, NULL );    //wait for child threads
  SDL_WaitThread ( producer, NULL );
  printf("Video play successful!\n");
  //no need to free buf[i] because SQL_Quit does the job.
  return 0;
}
```

Program **tplayer.cpp** of Listing 10-8 uses hard-coded parameters and plays video data that are not compressed. Figure 10-3 shows a sample output frame of the program. If we need to play compressed data, we only have to change the **decoder()** thread, which decodes data and acts as the producer. Instead of using hard-coded parameters, we can read in the parameters from the file containing the encoded data. Also, in order to test or perform experiments on our encoder and decoder, we need to download some video files which are saved in a standard video format. We shall address these problems in the next chapter.

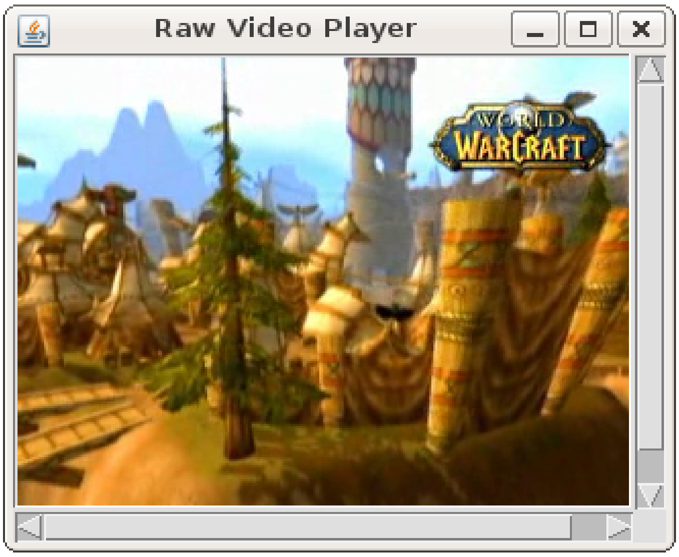

Figure 10-3 A Frame of Sample Raw Video