Lazy Foo' Productions

Semaphores

Last Updated 6/02/14

The only multithreading we've done had the main thread and a second thread each do their own thing. In most cases two threads will have to share data and with semaphores you can prevent two threads from accidentally accessing the same piece of data at once.

```
//Our worker thread function
int worker( void* data );
```

Here is our worker thread function. We will spawn two threads that will each execute their copy of this code.

```
//Data access semaphore
SDL_sem* gDataLock = NULL;

//The "data buffer"
int gData = -1;
```

The object gDataLock is our semaphore which will lock our gData buffer. A single integer is not much of a data buffer to protect, but since there are going to be two threads that

are going to be reading and writing to it we need to make sure it is only being accessed by one thread at a time.

```
bool loadMedia()
{
   //Initialize semaphore
   gDataLock = SDL_CreateSemaphore( 1 );

   //Loading success flag
   bool success = true;

   //Load splash texture
   if( !gSplashTexture.loadFromFile( "47_semaphores/splash.png" ) )
   {
      printf( "Failed to load splash texture!\n" );
      success = false;
   }

   return success;
}
```

To create a semaphore we call SDL_CreateSemaphore with an initial value for the semaphore. The initial value controls how many times code can pass through a semaphore before it locks.

For example, say you only want 4 threads to run at a time because you're on hardware with 4 cores. You'd give the semaphore a value of 4 to start with to make sure no more than 4 threads run at the same time. In this demo we only want 1 thread accessing the data buffer at once so the mutex starts with a value of one.

```
void close()
{
   //Free loaded images
   gSplashTexture.free();

   //Free semaphore
   SDL_DestroySemaphore( gDataLock );
   gDataLock = NULL;

   //Destroy window
   SDL_DestroyRenderer( gRenderer );
   SDL_DestroyWindow( gWindow );
   gWindow = NULL;
   gRenderer = NULL;

   //Quit SDL subsystems
   IMG_Quit();
   SDL_Quit();
}
```

When we're done with a semaphore we call SDL_DestroySemaphore.

```
int worker( void* data )
```

```
{
    printf( "%s starting...\n", data );

    //Pre thread random seeding
    srand( SDL_GetTicks() );
```

Here we are starting our worker thread. An important thing to know is that seeding your random value is done per thread, so make sure you seed your random values for each thread you run.

```
    //Work 5 times
    for( int i = 0; i < 5; ++i )
    {
        //Wait randomly
        SDL_Delay( 16 + rand() % 32 );

        //Lock
        SDL_SemWait( gDataLock );

        //Print pre work data
        printf( "%s gets %d\n", data, gData );

        //"Work"
        gData = rand() % 256;

        //Print post work data
        printf( "%s sets %d\n\n", data, gData );

        //Unlock
        SDL_SemPost( gDataLock );

        //Wait randomly
        SDL_Delay( 16 + rand() % 640 );
    }

    printf( "%s finished!\n\n", data );

    return 0;
}
```

What each worker thread does is delay for a semi random amount, print the data that is there when it started working, assign a random number to it, print the number assigned to the data buffer, and delay for a bit more before working again. The reason we need to lock data is because we do not want two threads reading or writing our shared data at the same time.

Notice the calls to SDL_SemWait and SDL_SemPost. What's in between them is the critical section or the code we only want one thread to access at once. SDL_SemWait decrements the semaphore count and since the initial value is one, it will lock. After the critical section executes, we call SDL_SemPost to increment the semaphore and unlock it.

If we have a situation where thread A locks and then thread B tries to lock, thread B will wait until thread A finishes the critical section and unlocks the semaphore. With the critical section protected by a semaphore lock/unlock pair, only one thread can execute the critical section at once.

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;

//Run the threads
srand( SDL_GetTicks() );
SDL_Thread* threadA = SDL_CreateThread( worker, "Thread A", (void*)"Thread A" );
SDL_Delay( 16 + rand() % 32 );
SDL_Thread* threadB = SDL_CreateThread( worker, "Thread B", (void*)"Thread B" );
```

In the main function before we enter the main loop we launch two worker threads with a bit of random delay in between them. There no guarantee thread A or B will work first but since the data they share is protected, we know they won't try to execute the same piece of code at once.

```
//While application is running
while( !quit )
{
    //Handle events on queue
    while( SDL_PollEvent( &e ) != 0 )
    {
        //User requests quit
        if( e.type == SDL_QUIT )
        {
            quit = true;
        }
    }

    //Clear screen
    SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
    SDL_RenderClear( gRenderer );

    //Render splash
    gSplashTexture.render( 0, 0 );

    //Update screen
    SDL_RenderPresent( gRenderer );
}

//Wait for threads to finish
SDL_WaitThread( threadA, NULL );
SDL_WaitThread( threadB, NULL );
```

Here the main thread runs while the threads to their work. If the main loop end before the threads finish working, we wait on them to finish with SDL_WaitThread.

Download the media and source code for this tutorial here.

Back to SDL Tutorials