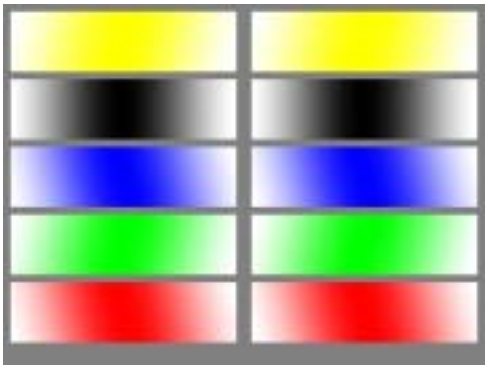
Lazy Foo' Productions

Mutexes and Conditions



Last Updated 11/15/07

Here we're going to do some more advanced thread synchronization using mutexes and conditions.

In case you missed the semaphores tutorial let me say this once again:
In this tutorial we have video functions running in seperate threads. You should never do this in a real application. It's just bad software design and in some cases can cause your OS to become unstable. The only reason we're doing it here is because it's a small program and nothing's going to go wrong. We're doing it here just as a simple demonstration of mutexes/conditions in action. Now on with the tutorial.

In this tutorial we'll have a "producer" thread which will pick one of 5 surfaces and store it in a buffer, then show the "generated" surface on the left side of the screen.

Then we'll have a "consumer" thread which show the surface in the buffer on the right side of the screen then empty the buffer out.

Here's the catch: unlike in the previous tutorial where there was 5 blits in order every 1/5 of a second, in this program we're going to have the producer produce 5 times at random and the consumer consume 5 times at random.

In the last tutorial we used semaphores to prevent the two threads from trying to manipulate the screen at the same time. Here we're going to use a mutex. A mutex is just a binary semaphore or one that will only let one thread pass through it at a time. In fact the semaphores tutorial could be redone with mutexes instead. All you'd have to do is swap the semaphore with a mutex and swap the lock/unlock functions.

Because the threads are doing things at random and they're dependent on each other, using just a mutex isn't enough. What if the consumer tries to consume and the buffer is empty? Or producer tries to produce but the buffer is full? This is where conditions come into play.

```
SDL_Surface *images[ 5 ] = { NULL, NULL, NULL, NULL, NULL };
SDL_Surface *buffer = NULL;
```

The buffer is just points to the surface "produced" (which in fact is just randomly chosen) by the producer. It points 1 of 5 surfaces which are loaded in the beginning of the program.

I just want to prevent any confusion on what the buffer is and what it holds.

```
//The threads that will be used
SDL_Thread *producerThread = NULL;
SDL_Thread *consumerThread = NULL;

//The protective mutex
SDL_mutex *bufferLock = NULL;

//The conditions
SDL_cond *canProduce = NULL;
SDL_cond *canConsume = NULL;
```

Here we have our threads along with our mutex. The mutex will prevent the threads from manipulating the buffer and/or screen at the same time.

Then we have the conditons which will tell when the producer can producer and the consumer can consume.

```
bool init()
{
    //Initialize all SDL subsystems
    if( SDL_Init( SDL_INIT_EVERYTHING ) == -1 )
    {
        return false;
    }

    //Set up the screen
    screen = SDL_SetVideoMode( SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_BPP, SDL_SWSURFACE

    //If there was an error in setting up the screen
    if( screen == NULL )
    {
        return false;
    }

    //Create the mutex
    bufferLock = SDL_CreateMutex();

    //Create Conditions
    canProduce = SDL_CreateCond();
    canConsume = SDL_CreateCond();

    //Set the window caption
    SDL_WM_SetCaption( "Producer / Consumer Test", NULL );

    //If everything initialized fine
    return true;
}
```

Before we can use a mutex or condition we have to create them. We do so by calling SDL_CreateMutex() and SDL_CreateCond() in our init() function.

```
int producer( void *data )
{
    //The offset of the blit.
    int y = 10;

    //Seed random
    srand( SDL_GetTicks() );

    //Produce
    for( int p = 0; p < 5; p++ )
    {
        //Wait
        SDL_Delay( rand() % 1000 );

        //Produce
        produce( 10, y );

        //Move down
```

```
        y += 90;
    }

    return 0;
}

int consumer( void *data )
{
    //The offset of the blit.
    int y = 10;

    for( int p = 0; p < 5; p++ )
    {
        //Wait
        SDL_Delay( rand() % 1000 );

        //Consume
        consume( 330, y );

        //Move down
        y += 90;
    }

    return 0;
}
```

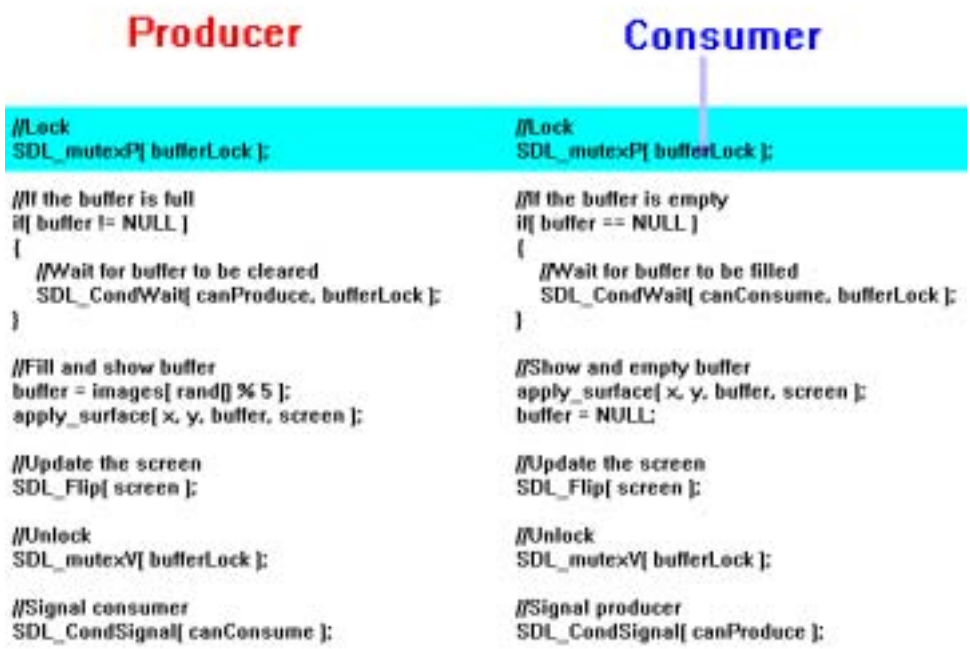
Here we have our producer/consumer thread functions. They produce/consume 5 times at random time intervals.

```
void produce( int x, int y )
{
    //Lock
    SDL_mutexP( bufferLock );

    //If the buffer is full
    if( buffer != NULL )
    {
        //Wait for buffer to be cleared
        SDL_CondWait( canProduce, bufferLock );
    }

    //Fill and show buffer
    buffer = images[ rand() % 5 ];
    apply_surface( x, y, buffer, screen );

    //Update the screen
    SDL_Flip( screen );

    //Unlock
    SDL_mutexV( bufferLock );

    //Signal consumer
    SDL_CondSignal( canConsume );
}

void consume( int x, int y )
{
    //Lock
    SDL_mutexP( bufferLock );

    //If the buffer is empty
    if( buffer == NULL )
    {
        //Wait for buffer to be filled
        SDL_CondWait( canConsume, bufferLock );
    }

    //Show and empty buffer
    apply_surface( x, y, buffer, screen );
    buffer = NULL;
```

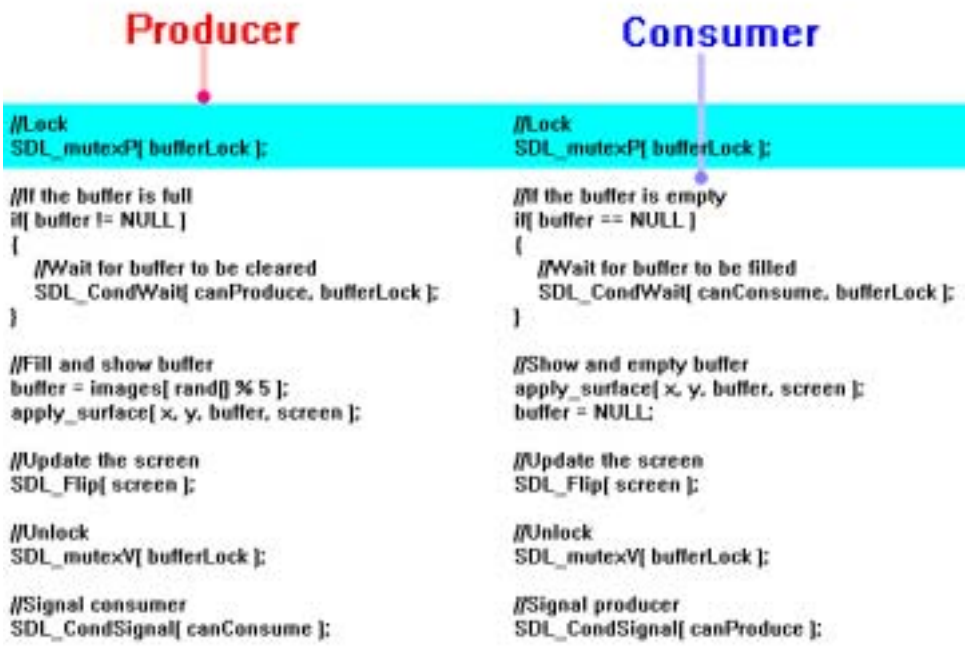
```
    //Update the screen
    SDL_Flip( screen );

    //Unlock
    SDL_mutexV( bufferLock );

    //Signal producer
    SDL_CondSignal( canProduce );
}
```

Here are our producer/consumer functions which are each called 5 times at random. How do they work? Well let's take this example situation:

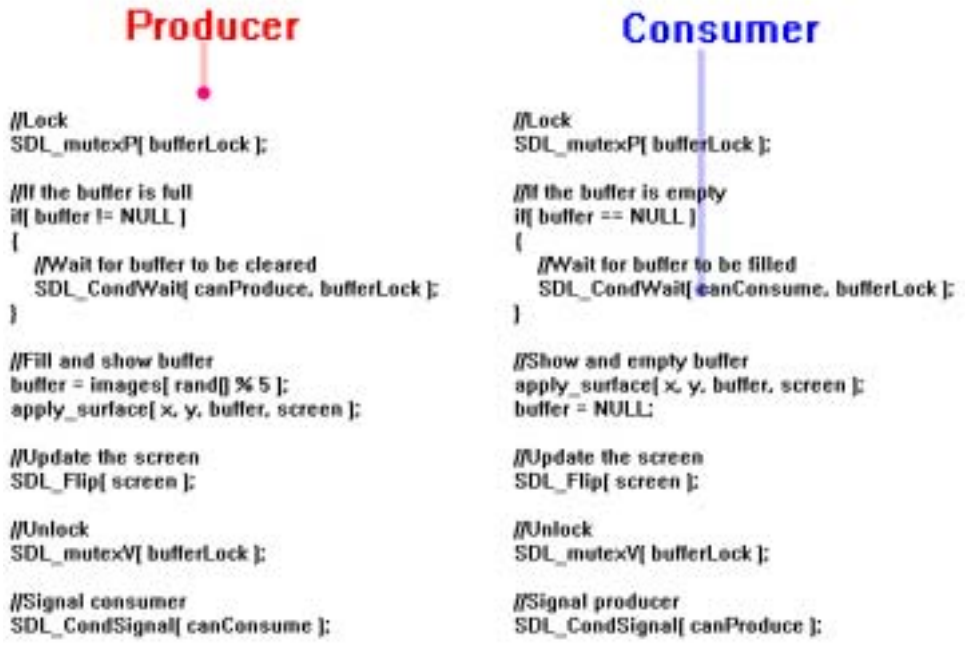
Let's say the consumer function is called first. It goes in and calls SDL_mutexP() to lock the mutex:



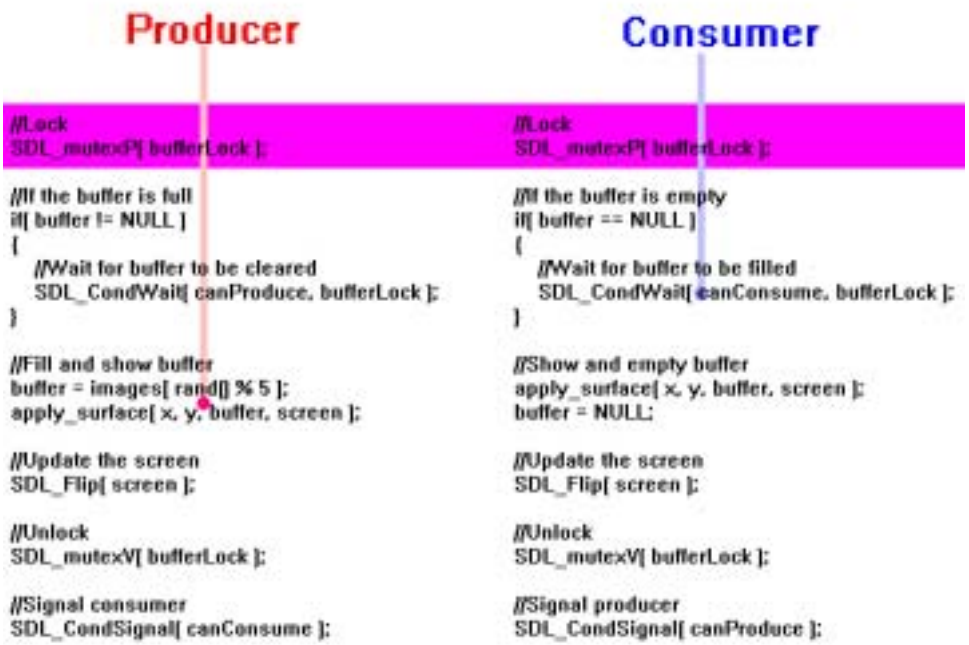
Then the producer tries to go in but can't because the mutex is locked. The mutex makes sure that the buffer and/or screen aren't manipulated by two threads at once.



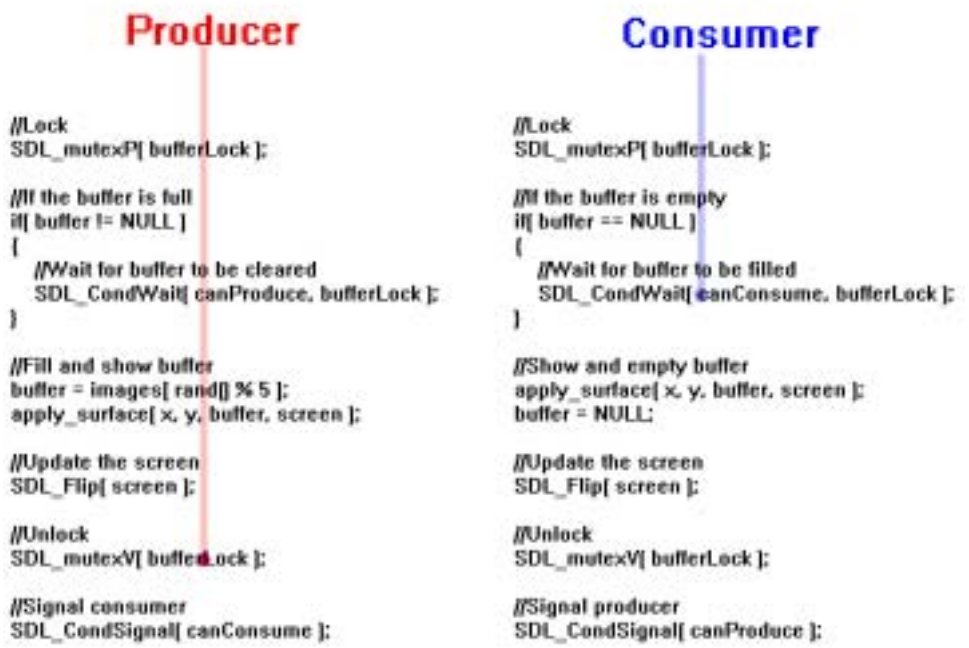
But let's say the buffer is empty. Now the consumer calls SDL_CondWait() which makes the thread wait on the "canConsume" condition. It also unlocks the mutex.



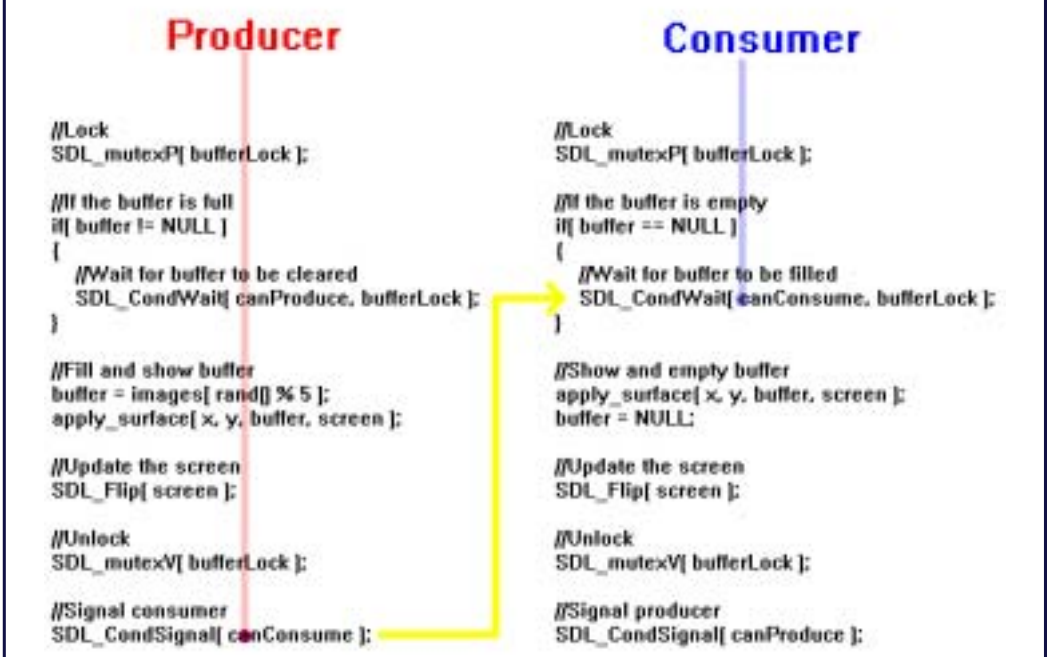
Now the producer can go through and produce.



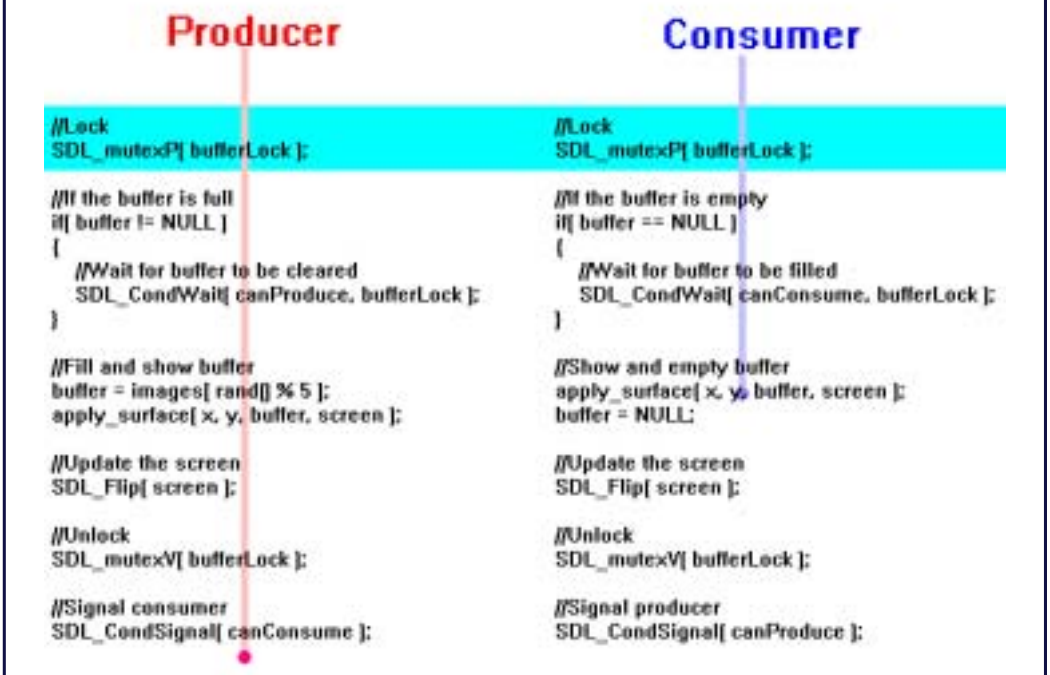
When the producer is done it calls SDL_mutexV() to unlock the mutex. But the consumer thread is still sleeping.



That's why we call SDL_CondSignal() to signal the consumer waiting on the "canConsume" condition.



Now that SDL_CondWait() has been signaled, the consumer wakes up, the mutex is relocked and now the consumer does its thing.



The threads not only stay out of each other's way, they also wait on and signal on each other thanks to mutexes/conditions.

```
void clean_up()
{
    //Destroy mutex
    SDL_DestroyMutex( bufferLock );

    //Destroy condition
    SDL_DestroyCond( canProduce );
    SDL_DestroyCond( canConsume );

    //Free the surfaces
```

```
    for( int i = 0; i < 5; i++ )
    {
        SDL_FreeSurface( images[ i ] );
    }

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

As always, don't forget to free anything dynamically allocated. Here we free our mutex and conditions using SDL_DestroyMutex() and SDL_DestroyCond().

Download the media and source code for this tutorial here.