Explanation Of My Thread Based Consumer

Production Program

Source: http://jlmedina123.wordpress.com/2014/04/08/255/

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define SIZE 5

#define NUMB\_THREADS 6

#define PRODUCER\_LOOPS 2

typedef int buffer\_t;

buffer\_t buffer[SIZE];

int buffer\_index;

pthread\_mutex\_t buffer\_mutex;

/\* initially buffer will be empty.  full\_sem

   will be initialized to buffer SIZE, which means

   SIZE number of producer threads can write to it.

   And empty\_sem will be initialized to 0, so no

   consumer can read from buffer until a producer

   thread posts to empty\_sem \*/

sem\_t full\_sem;  /\* when 0, buffer is full \*/

sem\_t empty\_sem; /\* when 0, buffer is empty. Kind of

                    like an index for the buffer \*/

/\* sem\_post algorithm:

    mutex\_lock  sem\_t->mutex

    sem\_t->value++

    mutex\_unlock sem\_t->mutex

   sem\_wait algorithn:

    mutex\_lock sem\_t->mutex

    while (sem\_t->value > 0) {

        mutex\_unlock sem\_t->mutex

        sleep... wake up

        mutex\_lock sem\_t->mutex

    }

    sem\_t->value--

    mutex\_unlock sem\_t->mutex

\*/

void insertbuffer(buffer\_t value) {

    if (buffer\_index < SIZE) {

        buffer[buffer\_index++] = value;

    } else {

        printf("Buffer overflow\n");

    }

}

buffer\_t dequeuebuffer() {

    if (buffer\_index > 0) {

        return buffer[--buffer\_index]; // buffer\_index-- would be error!

    } else {

        printf("Buffer underflow\n");

    }

    return 0;

}

void \*producer(void \*thread\_n) {

    int thread\_numb = \*(int \*)thread\_n;

    buffer\_t value;

    int i=0;

    while (i++ < PRODUCER\_LOOPS) {

        sleep(rand() % 10);

        value = rand() % 100;

        sem\_wait(&full\_sem); // sem=0: wait. sem>0: go and decrement it

        /\* possible race condition here. After this thread wakes up,

           another thread could aqcuire mutex before this one, and add to list.

           Then the list would be full again

           and when this thread tried to insert to buffer there would be

           a buffer overflow error \*/

        pthread\_mutex\_lock(&buffer\_mutex); /\* protecting critical section \*/

        insertbuffer(value);

        pthread\_mutex\_unlock(&buffer\_mutex);

        sem\_post(&empty\_sem); // post (increment) emptybuffer semaphore

        printf("Producer %d added %d to buffer\n", thread\_numb, value);

    }

    pthread\_exit(0);

}

void \*consumer(void \*thread\_n) {

    int thread\_numb = \*(int \*)thread\_n;

    buffer\_t value;

    int i=0;

    while (i++ < PRODUCER\_LOOPS) {

        sem\_wait(&empty\_sem);

        /\* there could be race condition here, that could cause

           buffer underflow error \*/

        pthread\_mutex\_lock(&buffer\_mutex);

        value = dequeuebuffer(value);

        pthread\_mutex\_unlock(&buffer\_mutex);

        sem\_post(&full\_sem); // post (increment) fullbuffer semaphore

        printf("Consumer %d dequeue %d from buffer\n", thread\_numb, value);

   }

    pthread\_exit(0);

}

int main(int argc, int \*\*argv) {

    buffer\_index = 0;

    pthread\_mutex\_init(&buffer\_mutex, NULL);

    sem\_init(&full\_sem, // sem\_t \*sem

             0, // int pshared. 0 = shared between threads of process,  1 = shared between processes

             SIZE); // unsigned int value. Initial value

    sem\_init(&empty\_sem,

             0,

             0);

    /\* full\_sem is initialized to buffer size because SIZE number of

       producers can add one element to buffer each. They will wait

       semaphore each time, which will decrement semaphore value.

       empty\_sem is initialized to 0, because buffer starts empty and

       consumer cannot take any element from it. They will have to wait

       until producer posts to that semaphore (increments semaphore

       value) \*/

    pthread\_t thread[NUMB\_THREADS];

    int thread\_numb[NUMB\_THREADS];

    int i;

    for (i = 0; i < NUMB\_THREADS; ) {

        thread\_numb[i] = i;

        pthread\_create(thread + i, // pthread\_t \*t

                       NULL, // const pthread\_attr\_t \*attr

                       producer, // void \*(\*start\_routine) (void \*)

                       thread\_numb + i);  // void \*arg

        i++;

        thread\_numb[i] = i;

        // playing a bit with thread and thread\_numb pointers...

        pthread\_create(&thread[i], // pthread\_t \*t

                       NULL, // const pthread\_attr\_t \*attr

                       consumer, // void \*(\*start\_routine) (void \*)

                       &thread\_numb[i]);  // void \*arg

        i++;

    }

    for (i = 0; i < NUMB\_THREADS; i++)

        pthread\_join(thread[i], NULL);

    pthread\_mutex\_destroy(&buffer\_mutex);

    sem\_destroy(&full\_sem);

    sem\_destroy(&empty\_sem);

    return 0;

}

Lines 1 – 8: Imports the libraries required for input output, threads and semaphores and sets symbolic names for SIZE to 5, NUMB\_THREADS to 6 and PRODUCER\_LOOPS to 2

Line 10: Declares integer variable representing the buffer

Line 11: Initializes buffer array of type buffer to 10 elements

Line14: Declares mutex thread and sets it to buffer

Line 21: Declares the full semaphore thread

Line 22: Declares the empty semaphore thread

Line 42: The function that gets called when inserting values into the buffer, two arguments get passed in; the buffer and the current value to be entered into said buffer.

Line 43: Checks if the current index of the buffer is less than the size of the buffer, i.e. the last element in the buffer array

Line 44: If the above is the case, the buffer index is incremented and the current value is put into the buffer at this buffer index in the buffer array

Line 46: If the current buffer index is greater than or equal to the last element in the buffer array, No more can be added to the buffer array so “Buffer Overflow” gets printed.

Line 50: This function is used for decrementing the buffer index. It returns integer variables of type buffer\_t

Line 51: Checks if the current buffer index is greater than 0

Line 52: If the above is the case the program will return from the function while decrementing the current buffer index

Line 54: If the current buffer index is less than or equal to 0, No more can be taken away from the buffer so “Buffer Overflow” gets printed

Line 60: The producer function, the current thread being used get passed in

Line 61: The value of the current thread is casted into an int and assigned to thread\_numb. This variable tells us the number of the current thread

Line 62: The variable value of type buffer is declared

Line 63: An integer I is initialized and set to 0

Line 64: i is incremented at the start of each running of a while loop which check if I is less than the symbolic name PRODUCER\_LOOPS. i.e. while i < 2, I gets incremented

Line 65: Sets a random time limit to wait of between 1 and 10 seconds

Line 66: Sets value to a random number between 1 and 100

Line 67: Locks the semaphore referenced by *sem* by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread will not return from the call to *sem\_wait()* until it either locks the semaphore or the call is interrupted by a signal.

Line 73: Locks the buffer array down so that only the current thread can access it, protecting critical section.

Line 74: Insertbuffer function is called and the current value is passed in with it

Line 75: The buffer array is unlocked, allowing other threads to access it

Line 76: The empty buffer semaphore is incremented

Line 77: Outputs the current thread number and the value it has placed in the buffer

Line 79: Ends the current thread

Line 82: The consumer function, the current thread being used gets passed in

Line 83: The value of the current thread is casted into an int and assigned to thread\_numb. This variable tells us the number of the current thread

Line 84: The variable value of type buffer is declared

Line 85: The integer i is initialized and set to 0

Line 86: i is incremented at the start of each running of a while loop which check if I is less than the symbolic name PRODUCER\_LOOPS. i.e. while i < 2, I gets incremented

Line 87: Locks the semaphore referenced by *sem* by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread will not return from the call to *sem\_wait()* until it either locks the semaphore or the call is interrupted by a signal.

Line 90: Locks the buffer array down so that only the current thread can access it, protecting critical section.

Line 91: dequeebuffer function is called and value is passed into it, where the the value of the buffer array at the current index will be removed. The return of dequeebuffer is passed into value.

Line 92: Unlocks the buffer array so it can be accessed by other threads

Line 93: Increments the full buffer semaphore

Line 94: Outputs the current thread number and the value it has removed from the buffer

Line 96: Exits the process

Line 99: The main function, passes in two arguments, an integer and a pointer to a pointer. Returns an integer

Line 100: Initializes the buffer index to 0

Line 102: Initializes the mutex thread, (used to lock down buffer array when a thread is directly accessing it, stopping other threads from being able to access the array)

Line 103: Initializes the full semaphore, it’s setup with a value of 0 rather than 1 to allow thread sharing of processes and is given an initial size of SIZE (which is 5), because SIZE number of producers can add one element to buffer each. They will wait semaphore each time, which will decrement semaphore value.

Line 106: Initializes the empty semaphore, it’s setup with a value of 0 rather than 1 to allow thread sharing of processes, and is given an initial size of 0 because buffer starts empty and consumer cannot take any element from it. They will have to wait until producer posts to that semaphore (increments semaphore value)

Line 116: An array called threads of type pthread is initialized to have NUMB\_THREADS (6) elements. i.e. 6 threads are initialized

Line 117: An integer array thread\_number is initialized to have NUMB\_THREADS (6) elements

Line 118: An integer I is initialized and given a value of 0

Line 119: A for loop which loops until it is equal to NUMB\_THREADS – 1 (5), NOTE, I is not incremented here

Line 120: The array thread\_numb at element I is made equal to i. i.e. if I = 0, thread\_numb[0] = 0

Line 121: A thread is created and sent to the producer function, given the value of i

Line 125: I is now incremented

Line 126: The array thread\_numb at element I is made equal to i.

Line 127: A thread is created and sent to the consumer function, given the value of i

Line 132: I is incremented

Line 135: Another for loop, I is made equal to 0 and is incremented while I is less than NUMB\_THREADS (which is 6)

Line 136: Suspends execution of threads[i]

Line 137: Locks down the current thread and destroys it

Line 138: The full semaphore is destroyed

Line 139: The empty semaphore is destroyed