PRODUCER-CONSUMER PROBLEM A.K.A. BOUNDED BUFFERS

Module Number 2. Section 11
COP4600 – Operating Systems
Richard Newman

CLASSIC SYNCHRONIZATION PROBLEMS

Critical Section Problem: Mutual exclusion – only one process can be in Critical Region at a time.

Producer-Consumer Problem: Producers produce items and write them into buffers; Consumers remove items from buffers and consume them. A buffer can only be accessed by one process at a time.

Bounded Buffer Problem: P-C with finite # buffers.

Dining Philosophers: Philosophers arranged in a circle share a fork between each pair of neighbors. Both forks are needed for a philosopher to eat, and only one philosopher can use a fork at a time.

Readers-Writers: Any number of readers can read from a DB simultaneously, but writers must access it alone

BOUNDED BUFFER PROBLEM

- Problem naturally arises in many situations
- Example 1 – I/O device writing to a kernel buffer in RAM at relatively slow speed, while device driver quickly performs a transfer to user memory after the buffer is full
- Example 2 – High-speed internet router (input demodulator – input buffer – forwarding logic – output buffer – output modulator)
- Example 3 – Internet video streaming; packets arrive intermittently (producer) and fill display buffer; MPEG4 decoder reads packets and produces frames for display at constant rate

PRODUCER-CONSUMER PROBLEM

Two types of process: Producers and consumers

There may be many of each type, and they may all run in parallel.

Producers produce items and write them into buffers; there must be an empty buffer available.

Consumers remove items from buffers (empty them for reuse by the producers) and consume them; the buffer must contain an item for the consumer to remove it.

Concurrency: (**C1**) Producers produce items in parallel with each other and with consumers who are consuming items in parallel. (**C2**) For maximum concurrency, multiple producers must be able to write their items into buffers at the same time and at the same time as multiple consumers are removing items from other buffers; (**C3**) No assumptions may be made about the speed of any process.

PRODUCER-CONSUMER PROBLEM

Safety: It is NOT allowed for (**S1**) Two producers to write to the same buffer at the same time; or (**S2**) Two consumers to read from the same buffer at the same time; or (**S3**) A consumer to read from a buffer at the same time that a producer is writing to it.

Liveness: (**L1**) No process that is not currently writing to a buffer or reading from a buffer must prevent another process from locating a suitable buffer for their access, if one exists; (**L2**) No process must wait indefinitely to obtain a suitable buffer for its access as long as suitable buffers are available;

Correctness: (**R1**) No full buffer shall be overwritten with a new item before the previous item is removed; (**R2**) No empty buffer shall be read by a consumer.

SLEEP AND WAKEUP
THE PRODUCER-CONSUMER PROBLEM (1)

```
#define N 100                              /* number of slots in the buffer */
int count = 0;                             /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                         /* repeat forever */
        item = produce_item();             /* generate next item */
        if (count == N) sleep();           /* if buffer is full, go to sleep */
        insert_item(item);                 /* put item in buffer */
        count = count + 1;                 /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);  /* was buffer empty? */
    }
}


void consumer(void)
{
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

SLEEP AND WAKEUP
THE PRODUCER-CONSUMER PROBLEM (2)

```
        if (count == 1) wakeup(consumer);          /* was buffer empty? */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                                  /* repeat forever */
        if (count == 0) sleep( );                   /* if buffer is empty, got to sleep */
        item = remove_item( );                      /* take item out of buffer */
        count = count − 1;                          /* decrement count of items in buffer */
        if (count == N − 1) wakeup(producer);       /* was buffer full? */
        consume_item(item);                         /* print item */
    }
}
```

So what is (are) the problem(s)?

Figure 2-27. The producer-consumer problem with a fatal race condition.

FLAWS IN P-C SOLUTION

Several ways things can go wrong:

1 – Corrupt shared variables indicating how many empty buffers and/or how many full buffers there are

2 – Two producers try to fill same buffer

3 – Two consumers try to empty same buffer

4 – Producer and consumer try to access same buffer

SEMAPHORES

Semaphores are a *special type* of shared memory:

1 – A semaphore S can only be declared and initialized, or accessed by Up(S) or Down(S);

NO direct access

2 – When Down(S) is called, if S > 0, S-- and continue; otherwise block the caller and put on S's queue Q

3 – When Up(S) is called, if Q empty, S++ and continue; otherwise remove a process P from Q and unblock P

4 – Up() never blocks

5 – A process blocked on Down() does not use the CPU

SEMAPHORES (1)

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
      int item;

      while (TRUE) {                    /* TRUE is the constant 1 */
            item = produce_item( );     /* generate something to put in buffer */
            down(&empty);               /* decrement empty count */
            down(&mutex);               /* enter critical region */
            insert_item(item);          /* put new item in buffer */
            up(&mutex);                 /* leave critical region */
            up(&full);                  /* increment count of full slots */
      }
}


void consumer(void)
```

Figure 2-28. The producer-consumer problem using semaphores.

SEMAPHORES (2)

```
        up(&full);                          /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                          /* infinite loop */
        down(&full);                        /* decrement full count */
        down(&mutex);                       /* enter critical region */
        item = remove_item( );              /* take item from buffer */
        up(&mutex);                         /* leave critical region */
        up(&empty);                         /* increment count of empty slots */
        consume_item(item);                 /* do something with the item */
    }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

COMMENTS ON P-C SOLUTION

Concurrency issues solution must handle:

1 – Corrupt shared variables indicating how many empty buffers and/or how many full buffers there are

2 – Two producers try to fill same buffer

3 – Two consumers try to empty same buffer

4 – Producer and consumer try to access same buffer

Semaphore solution given solves (1) by ...

counting semaphores,

solves (2)-(4) by

using a mutual exclusion semaphore (mutex)

MUTEXES IN PTHREADS (1)

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure 2-30. Some of the Pthreads calls relating to mutexes.

MUTEXES IN PTHREADS (2)

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure 2-31. Some of the Pthreads calls relating to condition variables.

MUTEXES IN PTHREADS (3)

Wait releases mutex, reacquires it when signaled

Single buffer

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* used for signaling */
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                         /* put item in buffer */
                pthread_cond_signal(&condc);        /* wake up consumer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}
```

Figure 2-32. Using threads to solve the producer-consumer problem.

MUTEXES IN PTHREADS (4)

Wait releases mutex, reacquires it when signaled

```
        pthread_exit(0);
}

void *consumer(void *ptr)                          /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                        /* take item out of buffer */
                pthread_cond_signal(&condp);       /* wake up producer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
```

Figure 2-32. Using threads to solve the producer-consumer problem.

MUTEXES IN PTHREADS (5)

```
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

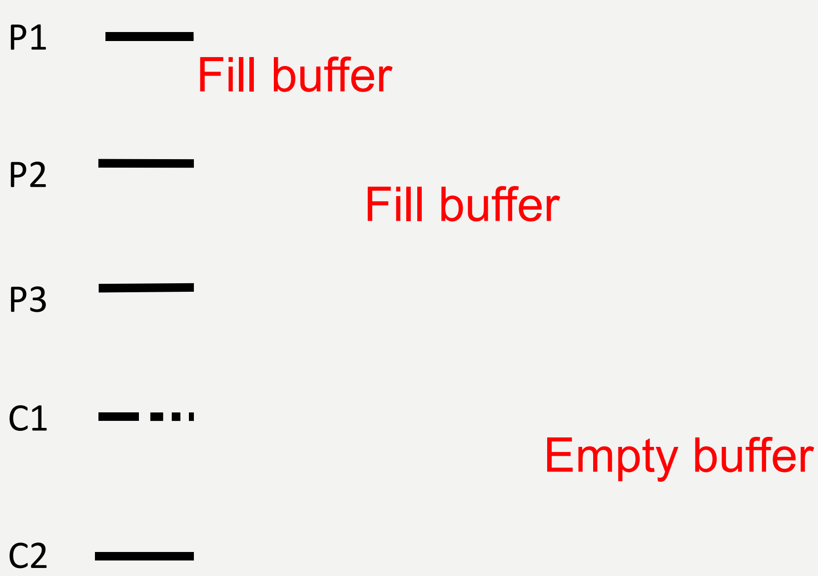
Initialization

Threads created and run

Clean up

Figure 2-32. Using threads to solve the producer-consumer problem.

TIMING DIAGRAM OF P-C SOLN



C1 must wait for a full buffer

P2 (et al.) must wait for P1 to finish copying item into buffer1

P3 must wait until there is an empty buffer, etc.

COMMENTS ON P-C SOLUTION (2)

Solution does not permit maximum concurrency:

only one process can insert or remove item at a time

Why is this potentially an issue?

What if items are very large, and take a long time to insert or remove?

So, what can we do about it????

BETTER P-C SOLUTION

```
Producer() {
    while (TRUE) {
    Produce(item)                   /* arbitrarily long time */
    Get_Empty(&buffer)              /* find an empty buffer */
        CopyTo(item,&buffer)                /* could take a while */
    Alert_Full(&buffer)             /* tell consumers */
    }
}
Consumer() {
    while (TRUE) {
    Get_Full(&buffer)               /* find a full buffer */
        CopyFrom(item,&buffer)              /* could take a while */
    Alert_Empty(&buffer)            /* tell consumers */
    Consume(item)                   /* arbitrarily long time */
    }
}
```

BETTER P-C SOLUTION (2)

```
Get_Empty(&buffer) {
    down(&empty)       /* wait for empty buffer to be available */
        down(&mutex)   /* only one process touches control vars */
        buffer=find_empty_buffer()   /* find a buffer marked empty*/
         Mark_Busy(&buffer)  // buffer is neither full nor empty
      up(&mutex)
    }
```

BETTER P-C SOLUTION (2)

```
Alert_Full(&buffer) {
    down(&mutex)           /* only one process touches control vars */
        Mark_Full(&buffer)      /* this buffer is full */
    up(&mutex)
    up(&full)              /* let consumers know a full buffer is available */
}
```

BETTER P-C SOLUTION (2)

```
Get_Full(&buffer) {
    down(&full)            /* wait for an item to be available */
     down(&mutex)          /* only one process touches control vars */
        buffer=find_full_buffer()       /* find a buffer marked full */
        Mark_Busy(&buffer)              /* buffer is neither full nor empty */
    up(&mutex)
    }
```

BETTER P-C SOLUTION (2)

```
Alert_Empty(&buffer) {
    down(&mutex)          /* only one process touches control vars */
        Mark_Empty(&buffer)    /* this buffer is empty */
    up(&mutex)
    up(&empty)            /* let producers know an empty buffer is
                          available */
}
```

TIMING DIAGRAM BETTER P-C SOLN

Full:
Empty:
Busy:

P1

P2

P3

C1

C2

Starting with N=5
Empty buffers

C1 must wait for a full buffer; P2 (et al.) must wait for P1 to find buffer1, but can find buffer2 while P1 fills buffer1, etc.

Running/ready outside CS

Running/ready buffer access

Blocked

waiting for CS

Running/ready inside CS

COMMENTS ON BETTER P-C SOLUTION

- Semaphore solution only allows one process in the buffer store at a time (uses counting semaphores to make sure that they will find what they are after)
- Better P-C solution has copy operations OUTSIDE critical regions – allows multiple processes to access different buffers concurrently
- The two “Get” procedures must
  - Find an idle buffer in the right state (full or empty)
  - Mark that buffer as BUSY so nobody else uses it
  - Return location to caller
- The two “Alert” procedures must
  - Mark buffer as FULL or EMPTY
  - Signal other processes by up on counting semaphore
- All four Get and Alert procedures are themselves CS's
  - But the Copy procedures are NOT exclusive!

MONITORS AND CONDITION VARIABLES

Monitors are a *language construct*

– Monitor acts like an Abstract Data Type or Object – only access internal state through public methods, called entry procedures

– Monitor guarantees that at most one process at a time is executing in any entry procedure of that monitor

Like java "synchronized" methods

– Condition "variables" local to monitor allow a process to block until another process signals that condition

– Unlike semaphores, condition vars have NO memory!

– If no process is waiting on the condition, signaling the condition variable *has NO effect at all*!

– A process that waits on a condition variable can be awakened and resume execution in entry procedure

MONITOR SOLUTION

```
monitor ProducerConsumer
      condition full, empty;
      integer count;

      procedure insert(item: integer);
      begin
            if count = N then wait(full);
            insert_item(item);
            count := count + 1;
            if count = 1 then signal(empty)
      end;

      function remove: integer;
      begin
            if count = 0 then wait(empty);
            remove = remove_item;
            count := count − 1;
            if count = N − 1 then signal(full)
      end;

      count := 0;
end monitor;
```

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots

MONITOR SOLUTION

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

```
procedure producer;
begin
      while true do
      begin
            item = produce_item;
            ProducerConsumer.insert(item)
      end
end;

procedure consumer;
begin
      while true do
      begin
            item = ProducerConsumer.remove;
            consume_item(item)
      end
end;
```

JAVA SOLUTION

```
public class ProducerConsumer {
      static final int N = 100;      // constant giving the buffer size
      static producer p = new producer( );    // instantiate a new producer thread
      static consumer c = new consumer( ); // instantiate a new consumer thread
      static our_monitor mon = new our_monitor( );      // instantiate a new monitor

      public static void main(String args[ ]) {
         p.start( );      // start the producer thread
         c.start( );      // start the consumer thread
      }

      static class producer extends Thread {
         public void run( ) {// run method contains the thread code
            int item;
            while (true) {      // producer loop
               item = produce_item( );
               mon.insert(item);
            }
         }
         private int produce_item( ) { ... }      // actually produce
      }

      static class consumer extends Thread {
```

Figure 2-35. A solution to the producer-consumer problem in Java.

JAVA SOLUTION

```
    private int produce_item( ) { ... }       // actually produce
}

static class consumer extends Thread {
    public void run( ) {run method contains the thread code
        int item;
        while (true) {       // consumer loop
            item = mon.remove( );
            consume_item (item);
        }
    }
    private void consume_item(int item) { ... }// actually consume
}

static class our_monitor {   // this is a monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0;   // counters and indices

    public synchronized void insert(int val) {
```

Figure 2-35. A solution to the producer-consumer problem in Java.

JAVA SOLUTION

```
        if (count == N) go_to_sleep();     // if the buffer is full, go to sleep
        buffer [hi] = val; // insert an item into the buffer
        hi = (hi + 1) % N;       // slot to place next item in
        count = count + 1;       // one more item in the buffer now
        if (count == 1) notify();       // if consumer was sleeping, wake it up
    }

    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep();     // if the buffer is empty, go to sleep
        val = buffer [lo]; // fetch an item from the buffer
        lo = (lo + 1) % N;       // slot to fetch next item from
        count = count – 1;       // one few items in the buffer
        if (count == N – 1) notify(); // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};}
}
```

Figure 2-35. A solution to the producer-consumer problem in Java.

MESSAGE PASSING SOLUTION

```
#define N 100                              /* number of slots in the buffer */

void producer(void)
{
      int item;
      message m;                           /* message buffer */

      while (TRUE) {
            item = produce_item( );        /* generate something to put in buffer */
            receive(consumer, &m);         /* wait for an empty to arrive */
            build_message(&m, item);       /* construct a message to send */
            send(consumer, &m);            /* send item to consumer */
      }
}

void consumer(void)
```

Figure 2-36. The producer-consumer problem with *N* messages.

MESSAGE PASSING SOLUTION

```
            send(consumer, &m);                  /* send item to consumer */
        }
}

void consumer(void)
{
        int item, i;
        message m;

        for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
        while (TRUE) {
            receive(producer, &m);               /* get message containing item */
            item = extract_item(&m);             /* extract item from message */
            send(producer, &m);                  /* send back empty reply */
            consume_item(item);                  /* do something with the item */
        }
}
```

Initialize system

Figure 2-36. The producer-consumer problem with *N* messages.

SUMMARY

- Statement of Producer-Consumer problem
  - Safety, Liveness, correctness, concurrency
- Real examples/motivation
- Analyzed flawed solution
  - Race conditions
- Semaphore solutions
- Better solution that maximizes concurrency
- Monitor solution
- Java solution
- Message-passing solution