CS 4500 - Operating Systems
Module 4: The Producer-Consumer Problem and Solution Methods

Stanley Wileman

Department of Computer Science
University of Nebraska at Omaha
Omaha, NE 68182-0500, USA

June 3, 2017
In This Module...

- The producer-Consumder Problem
- A “Solution” using Sleep and Wakeup
- Semaphores
- Event Counters
- Monitors
- Message Passing
- Equivalence of the Methods
The Producer-Consumer Problem

Two types of processes, producers and consumers, wish to communicate.

Producers have information to send to the consumers. They place the information items to be communicated in a bounded (that is, limited size) buffer, as long as there is space available. If there is insufficient space, the producer must wait.

Consumers can extract an item from the buffer, but only if the buffer isn't empty. If the buffer is empty, the consumer must wait.

The buffer is a shared resource; use of the buffer must be synchronized.
Producer-Consumer Synchronization

- Since the buffer is a shared resource, each access to it must be done in a critical section.
- When the buffer is full, producers cannot place any more items in it. They therefore go to sleep, to be awakened when a consumer removes an item—which makes an empty “slot” in the buffer.
- Similarly, consumers must go to sleep when the buffer is empty. Eventually they will be awakened when a producer places an item in a previously empty “slot”.

A “Solution” to the Producer-Consumer Problem

- The next two slides show a proposed solution to the producer-consumer problem using the sleep and wakeup functions we looked at earlier.
- Recall that the sleep function causes the invoking process to become blocked (asleep, waiting), and the wakeup function explicitly causes a process to be awakened.
- In general, we may have multiple producers and multiple consumers, each with appropriate synchronized access to the buffer.
- We assume the existence of the functions produce_item, consume_item, enter_item, and remove_item. These functions perform as their names imply, but have no effect on process synchronization—that we must provide ourselves.
- The items being produced and consumed are integers.
- The global count variable, initially 0, indicates the number of items in the buffer (which has N items when full).
- In the proposed solution, we will consider only a single producer process and a single consumer process.
void producer {
    int item; /* data item */
    while (TRUE) {
        produce_item(&item);
        if (count == N) /* buffer full? */
            sleep(); /* wait */
        enter_item(item);
        count = count + 1; /* increase buffer full count */
        if (count == 1) /* empty -> not empty transition? */
            wakeup(consumer);
    }
}
void consumer {
    int item; /* data item */
    while (TRUE) {
        if (count == 0) /* buffer empty? */
            sleep(); /* wait */
        remove_item(&item);
        count = count - 1; /* reduce buffer full count */
        if (count == N - 1) /* not empty -> empty transition? */
            wakeup(producer);
        consume_item(item);
    }
}
The solution, as presented, has a race condition that can result in both processes (producer and consumer) going to sleep forever. What is it?

Hint. Think about places in the producer’s code where the processor could be interrupted and do a context switch to execute in the consumer’s code, and vice versa.

Considering the equivalent assembler language code for some parts of the functions may help.
A1. The variable **count** is shared by the producer and consumer, but access to it was not synchronized.

Recall our illustration of race conditions in the program `race.c`. Even though the code “looked” correct, we observed that in the real world, on a real processor, there could be cases when operations on variables are **not** performed atomically by default.

A2. Even if access to **count** was synchronized, there is still a flaw in the solution.

The flaws should be apparent when you realize that it is possible for the processor to be interrupted **after** the test in an **if** statement, but **before** the **then** part of the statement is executed.
Event Sequence Illustrating a Race Condition

- Assume the buffer is empty, and the consumer tests count and finds that it is zero.
- Immediately after that (before the consumer executes the call to the sleep function) assume a context switch occurs and the producer runs.
- The producer enters and item in the buffer, sets count to 1, and sends a wakeup to the consumer. Since the consumer is not yet sleeping, the wakeup is ignored.
- Another context switch occurs, and the consumer runs. Since it has already completed the “count == 0” test, it calls the sleep function, and goes to sleep!
- Another context switch occurs, and the producer runs until it fills the buffer and goes to sleep.

Note that a similar failure scenario could be prepared showing the producer testing count, the consumer running—reducing count and sending an ignored wakeup to the producer, and the producer going to sleep because the value of count it read was N.
One way to eliminate the race condition is to add a “wakeup waiting” bit. This bit would be set when a wakeup is sent to a process that isn’t sleeping. When the process next decides to sleep, it first examines the wakeup waiting bit, and if is set, it clears the bit and does not go to sleep.

Unfortunately, this doesn’t work with more than one producer and consumer. Even adding multiple “wakeup waiting” bits still presents a limit that could be exceeded.
In 1965, Edsger Dijkstra (pronounced “dikes truh”) suggested using an integer to record the number of waiting wakeups.

A new data structure called the semaphore was created to record the wakeup count and the set containing the identities of the sleeping processes.

Two operations on a semaphore, down and up (historically called P and V) were introduced.
Dijkstra was Dutch, and the main operations on semaphores are often identified using the capitalized first letters of Dutch words related to the actions. *Probeer* means try, and *Verhoog* means increment.

In other instances, different names are used. In the Linux kernel, the operations are called *down* and *up*. These are the names we’ll use (since the textbook uses them).

Elsewhere you’ll see these names:

- *wait* and *signal*
- *acquire* and *release*
- *pend* and *post*

There are likely others!
A semaphore is a data structure that has two members:

- a non-negative integer count that indicates the number of units of a resource available for allocation (that is, the number of waiting “wakesups”)
- a set containing the identification of each process waiting for a unit of the resource controlled by the semaphore

Although called a set here, the collection of processes may be ordered by a particular operating system (e.g. first in first out or priority).
The down operation does the following:

- Is count greater than zero?
  - Yes: decrement count and continue.
  - No: place current process identification in the set and sleep. When awakened, repeat the test on count.

The down operation is atomic. This means the operation will complete before any other process is allowed to manipulate the semaphore with a down or up operation.
The up operation does the following:

- Increment count.
- If the set of waiting processes is not empty, awaken one of them, causing it to move from the blocked state to the ready state.

The up operation is atomic.

No process will ever block as a result of performing an up operation.

As noted earlier, the selection of the process to be awakened may vary depending on the operating system implementation.
Assume there are N slots in the buffer.

Three semaphores are used in the solution:

- **empty**: count (initially N) represents the number of unused slots in the buffer; set contains the identity of producers waiting to add to the buffer.
- **full**: count (initially 0) contains the number of filled slots in the buffer; set contains the identity of consumers waiting to remove from the buffer.
- **mutex**: count (initially 1), controls access to buffer.
void producer {
    int item;    /* data item */
    while (TRUE);
    produce_item(&item);
    down(&empty);  /* wait for an empty */
    down(&mutex);  /* wait for buffer use */
    enter_item(item);
    up(&mutex);   /* release buffer */
    up(&full);    /* increase full count */
}

void consumer {
    int item; /* data item */
    while (TRUE);
    down(&full); /* wait for a full */
    down(&mutex); /* wait for buffer use */
    remove_item(item);
    up(&mutex); /* release buffer */
    up(&empty); /* increase empty count */
    consume_item(&item);
}

Typical Uses of Semaphores

- **Mutual Exclusion.** This use is illustrated by the **mutex** semaphore in the producer-consumer solution. Some systems even provide a semaphore type with count limited to 0 or 1 and call them a mutex (e.g. Windows).

- **Synchronization.** This use is illustrated by the **empty** and **full** semaphores in the producer-consumer solution.
An event counter is another data structure that can be used for process synchronization. Like a semaphore, it has an integer count and a set of waiting process identifications.

Unlike semaphores, the count variable only increases monotonically. It is similar to the “next customer” number used in systems where each customer takes a sequentially numbered ticket and waits for that number to be called.
Operations on Event Counters

- **read(E):** return the count associated with event counter $E$.
- **advance(E):** atomically increment the count associated with event counter $E$.
- **await(E,v):** if $E.count \geq v$, then continue. Otherwise, block until $E.count \geq v$. 
Two event counters are used, in and out, each of which has an initial count of zero.

Each process includes a private sequence variable which indicates the sequence number of the last item it has produced or will consume. Items are sequentially produced and consumed.

Each item has a unique location in the buffer based on its sequence number, so mutually exclusive access to the buffer is not required.

In our sample solution, we assume N is the number of buffer slots.
Mapping Sequence Numbers to Buffer Locations

Note that items with sequence numbers 0 and 5 cannot both be in the buffer at the same time, as this would imply that there are six items in the buffer.
void producer {
    int item; /* data item */
    int sequence = 0; /* # of last item produced */

    while (TRUE);
    produce_item(&item);
    sequence = sequence + 1;
    await(out,sequence - N); /* wait for space */
    enter_item(item, /* put in right spot */
               buffer[(sequence-1)%N]);
    advance(&in); /* update event counter */
}
}
Consumer, with Event Counters

```c
void consumer {
    int item;       /* data item */
    int sequence = 0;  /* # of item to consume */

    while (TRUE);
    sequence = sequence + 1;
    await(in, sequence);  /* wait for item */
    enter_item(&item,  /* get from right spot */
               buffer[(sequence-1)%N]);
    advance(&out);      /* update event counter */
    consume_item(item);
}
```
Problems with Semaphores

- Semaphores do not completely eliminate race conditions or other problems like starvation, indefinite postponement, and deadlock.
- Recall that semaphores have two purposes: mutual exclusion and synchronization.
- This can make it difficult to produce a correct formulation of a solution to a problem using semaphores, often resulting in obscure problems that are difficult to identify and correct.
- For example, consider reversing the order of the down operations in the producer’s code. Can you identify a condition in which this would cause the solution to fail?
- Semaphores also have state information that is not normally visible to processes using them. Some systems may provide methods to examine the semaphore’s count, but it can change transparently after it is examined.
Monitors

- Tony Hoare (in 1974) and Per Brinch Hansen (in 1975) proposed a new synchronization structure called a monitor.
- Monitors were features to be included in high-level programming languages. The best-known monitor implementation was provided by Concurrent Pascal.
- A monitor is a construction that groups a collection of functions, declarations, and initialization statements.
- Only one process at a time is allowed inside the monitor. Compilers for languages that support monitors generate code that guarantees this restriction.
Monitors can also be provided as a set of programming conventions that you can use when doing concurrent programming in languages like C or C++. Basically all that’s required is to provide the appropriate explicit calls to functions or methods that implement locks and condition variables.
Blocking Inside a Monitor

A process, after invoking a function that is part of a monitor, may need to block. For example, suppose a monitor for the producer-consumer problem has two functions, enter_item and remove_item. When a producer calls enter_item, there may be no empty slots in the buffer, and the producer must block. It cannot leave the function until the item has been entered, so it must effectively block, inside the monitor.

Unfortunately, blocking inside the monitor will prevent access to remove_item by a consumer, so some other solution must be found.
The solution is to conceptually provide a region outside the monitor where processes can wait until the resource they need is possibly available.

In reality, this is provided by a method that allows processes to atomically release the lock preventing others from entering the monitor and putting the process to sleep.

A process can wait—moving itself to the waiting “room” by executing `wait(cv)`, where `cv` is a condition variable declared inside the monitor.

A waiting process is awakened by another process, currently inside the monitor, by executing `signal(cv)`.

Note that this use of signal has nothing to do with the UNIX signal facility.
- A condition variable is essentially just a set including the identification of each process that is waiting outside the monitor on that condition variable.

- Note that there is no explicit “count” associated with a condition variable as there is with a semaphore.

- Executing signal on a condition variable with no waiting processes is ignored; no “wakeup” is stored as it would if we did an up operation on a semaphore with a count of zero.
The implementation of signal presents an interesting problem. When should a signaled process be allowed to continue execution inside the monitor?

The awakened process cannot be allowed to execute inside the monitor as long as the signaling process is still inside the monitor, as that would violate the monitor requirements.
Proposed Implementations of Signal

- Per Brinch Hansen proposed requiring that a signaling process immediately exit the monitor; that is, signal must be the last action taken by the signaling process inside the monitor. This leaves the way clear for the awakened process to re-enter the monitor and continue execution.

- Tony Hoare proposed moving the signaling process to still another “holding pen” outside the monitor, and resuming its execution automatically at a later time when the signaling process finally exits the monitor.

- Brinch Hansen’s proposal has been the most popular technique.
The next few slides show a solution to the producer-consumer problem using a monitor. The code is written in Concurrent Pascal.

The monitor (on the next slide) has two condition variables, and an integer (count) that indicates the number of objects in the buffer. Procedure insert adds an item, and function remove returns an item. Initialization consists of just setting count to zero. As in our other solutions, N is the size of the buffer.

The second following slide shows a pair of simple processes: a producer and a consumer, each using the monitor.
monitor ProducerConsumer
    condition full, empty;
    int 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;
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;
Semaphores, monitors and event counters are all designed to function within a single system (that is, a system with a single primary memory, not a networked system).

They do not support synchronization of processes running on separate machines connected to a network.

Messages, which can be sent across a network, can be used to provide synchronization.
Typical Message-Passing Functions

The following functions and data items are characteristic of what you might find in a typical message-passing environment:

- **source** and **destination** addresses must identify the machine and the process being addressed.

- **message** is a generic data item to be delivered.

- **send (destination, &message);** – the calling process sends message to destination without blocking; the senders address is usually included.

- **receive (source, &message);** – the calling process receives the next sequential message from source, blocking until it is available.
In this solution, each message has two components: an empty/full flag, and a data component being passed from the producer to the consumer.

Initially, the consumer sends N messages marked as empty to the producer.

The producer receives an empty message, blocking until one is available, fills it, and sends it to the consumer.

The consumer receives a filled message, blocking if necessary, processes the data it contains, and returns the empty message to the producer.
void producer(void) {
    int item;
    message m; /* message buffer */

    while (TRUE) {
        item = produce(item); /* produce something */
        receive(consumer, &m); /* wait for an empty */
        build_message(&m, item); /* construct message */
        send(consumer, &m); /* send item to consumer */
    }
}
void consumer(void) {
    int item, i;
    message m;

    for(i=0;i<N;i++) /* send N empties */
        send(producer, &m);

    while (TRUE) {
        receive(producer, &m); /* get message with item */
        item = extract_item(&m); /* extract item */
        send(producer, &m); /* send empty reply */
        consume_item(item); /* do something with item */
    }
}
Message Passing Problems

- Our simplified message passing functions hide many important implementation details. For example:
  - How can we guarantee the correct delivery of messages?
  - How are processes identified?
  - How are fraudulent messages avoided?
  - How is efficiency affected?

These problems are considered in a course on data communication and networking.
Message Passing Variants

There are numerous variations on the message passing idea, many of which are used in modern systems and languages. Some examples:

- Mailboxes
- Rendezvous
- MPI (common in parallel systems)
Mailboxes

- A mailbox is a fixed size buffer with an address.
- Addresses in messages are those of mailboxes, not processes.
- A process trying to send to a full mailbox will block, as will a process trying to receive from an empty mailbox.
- A UNIX pipe is very similar to a mailbox, except message boundaries are not preserved in a pipe.
Rendezvous

- The rendezvous mechanism eliminates all buffering.
- A process must do a send, and another process must do a receive before the message is transferred.
- The first process to send or receive must wait for the other.
- Rendezvous is easier to implement than mailboxes, but processes must run in lockstep.
- MINIX 3 OS processes use the rendezvous method with fixed-size messages to communicate among themselves.
- The ADA programming language also provides facilities for processes to rendezvous.
MPI – Message Passing Interface

- MPI is an open-source library frequently used in the implementation of parallel processing systems.
- With this library, it is relatively easy to take a networked collection of processors and build parallel solutions to solve problems.
- The “bottleneck” in many parallel system is the network bandwidth (and the algorithms).
Other Synchronization Methods

- There are numerous variations on the synchronization methods we have studied.
- Most modern operating systems implement some of these in various forms. For example, Windows includes functions to wait for one or all of a set of objects used for synchronization.
- An advanced course on operating systems will likely consider interprocess communication and synchronization in greater detail.
Since all the primitives we have studied serve to provide synchronization and mutually exclusive access to resources, can they all be considered equivalent?

Yes, at least in terms of their capabilities. Since each primitive (semaphores, event counters, etc.) can be written in terms of any of the others, we can prove they are equivalent.
In the next module we will look at several classic problems in interprocess communication and their solutions.

This module will give you additional confidence that the methods we have just examined are sufficient to solve a wide range of problems.

Even so, some additional problems will be discovered.