Producer–consumer problem

From Wikipedia, the free encyclopedia

In computing, the **producer–consumer problem**[1][2] (also known as the **bounded-buffer problem**) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.

Contents

- 1 Inadequate implementation
- 2 Using semaphores
- 3 Using monitors
- 4 Without semaphores or monitors
- 5 See also
- 6 References
- 7 Further reading
- 8 External links

Inadequate implementation

To solve the problem, a less experienced programmer might come up with a solution shown below. In the solution two library routines are used, `sleep` and `wakeup`. When sleep is called, the caller is blocked until another process wakes it up by using the wakeup routine. The global variable `itemCount` holds the number of items in the buffer.

```
int itemCount = 0;

procedure producer() {
```

```
    while (true) {
        item = produceItem();

        if (itemCount == BUFFER_SIZE) {
            sleep();
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1) {
            wakeup(consumer);
        }
    }
}

procedure consumer() {
    while (true) {

        if (itemCount == 0) {
            sleep();
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

The problem with this solution is that it contains a race condition that can lead to a deadlock. Consider the following scenario:

1. The consumer has just read the variable `itemCount`, noticed it's zero and is just about to move inside the `if` block.
2. Just before calling sleep, the consumer is interrupted and the producer is resumed.
3. The producer creates an item, puts it into the buffer, and increases `itemCount`.
4. Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
5. Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when `itemCount` is equal to 1.
6. The producer will loop until the buffer is full, after which it will also go to sleep.

Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory.

An alternative analysis is that if the programming language does not define the semantics of concurrent accesses to shared variables (in this case `itemCount`) without use of synchronization, then the solution is unsatisfactory for that reason, without needing to explicitly demonstrate a race condition.

Using semaphores

Semaphores solve the problem of lost wakeup calls. In the solution below we use two semaphores, `fillCount` and `emptyCount`, to solve the problem. `fillCount` is the number of items already in the buffer and available to be read, while `emptyCount` is the number of available spaces in the buffer where items could be written. `fillCount` is incremented and `emptyCount` decremented when a new item is put into the buffer. If the producer tries to decrement `emptyCount` when its value is zero, the producer is put to sleep. The next time an item is consumed, `emptyCount` is incremented and the producer wakes up. The consumer works analogously.

```
semaphore fillCount = 0; // items produced
semaphore emptyCount = BUFFER_SIZE; // remaining space

procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
        putItemIntoBuffer(item);
        up(fillCount);
    }
}

procedure consumer() {
    while (true) {
        down(fillCount);
        item = removeItemFromBuffer();
        up(emptyCount);
        consumeItem(item);
    }
}
```

The solution above works fine when there is only one producer and consumer. With multiple producers sharing the same memory space for the item buffer, or multiple consumers sharing the same memory space, this solution contains a serious race condition that could result in two or more processes reading or writing into the same slot at the same time. To understand how this is possible, imagine how the procedure `putItemIntoBuffer()` can be implemented. It could contain two actions, one determining the next available slot and the other writing into it. If the procedure can be executed concurrently by multiple producers, then the following scenario is possible:

1. Two producers decrement `emptyCount`
2. One of the producers determines the next empty slot in the buffer
3. Second producer determines the next empty slot and gets the same result as the first producer
4. Both producers write into the same slot

To overcome this problem, we need a way to make sure that only one producer is executing `putItemIntoBuffer()` at a time. In other words, we need a way to execute a critical section with mutual exclusion. The solution for multiple producers and consumers is shown below.

```
semaphore mutex = 1;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
            down(mutex);
                putItemIntoBuffer(item);
            up(mutex);
        up(fillCount);
    }
}

procedure consumer() {
    while (true) {
        down(fillCount);
            down(mutex);
                item = removeItemFromBuffer();
            up(mutex);
```

```
        up(emptyCount);
        consumeItem(item);
    }
}
```

Notice that the order in which different semaphores are incremented or decremented is essential: changing the order might result in a deadlock.

Using monitors

The following pseudo code shows a solution to the producer–consumer problem using monitors. Since mutual exclusion is implicit with monitors, no extra effort is necessary to protect the critical section. In other words, the solution shown below works with any number of producers and consumers without any modifications. It is also noteworthy that using monitors makes race conditions much less likely than when using semaphores.

```
monitor ProducerConsumer {
    int itemCount;
    condition full;
    condition empty;

    procedure add(item) {
        while (itemCount == BUFFER_SIZE) {
            wait(full);
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1) {
            notify(empty);
        }
    }
    procedure remove() {
        while (itemCount == 0) {
            wait(empty);
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1) {
            notify(full);
        }


        return item;
    }
}

procedure producer() {
    while (true) {
        item = produceItem();
        ProducerConsumer.add(item);
    }
}

procedure consumer() {
    while (true) {
        item = ProducerConsumer.remove();
        consumeItem(item);
    }
}
```

Note the use of `while` statements in the above code, both when testing if the buffer is full or empty. With multiple consumers, there is a race condition where one consumer gets notified that an item has been put into the buffer but another consumer is already waiting on the monitor so removes it from the buffer instead. If the `while` was instead an `if`, too many items might be put into the buffer or a remove might be attempted on an empty buffer.

Without semaphores or monitors

The producer–consumer problem, particularly in the case of a single producer and single consumer, strongly relates to implementing a FIFO or a channel. The producer–consumer pattern can provide highly efficient data communication without relying on semaphores, mutexes, or monitors *for data transfer*. Use of those primitives can give performance issues as they are expensive to implement. Channels and FIFOs are popular just because they avoid the need for end-to-end atomic synchronization. A basic example coded in C is shown below. Note that:

- Atomic read-modify-write access to shared variables is avoided, as each of the two `Count` variables is updated only by a single thread. Also, these variables stay incremented all the time; the relation remains correct when their values wrap around on an integer overflow.
- This compact example should be refined for an actual implementation by adding a memory barrier between the line that accesses the buffer and the line that updates the `Count` variable.
- This example does not put threads to sleep, which may be acceptable depending on the system context. The `sched_yield()` is there just to behave nicely and could be removed. Thread libraries typically require semaphores or condition variables to control the sleep/wakeup of threads. In a multi-processor environment, thread sleep/wakeup would occur much less frequently than passing of data tokens, so avoiding atomic operations on data passing is beneficial.
- This example does not work for multiple producers and/or consumers because there is a race condition when checking the state. For example, if only one token is in the storage buffer and two consumers find the buffer non-empty, then both will consume the same token and possibly increase the count of consumed tokens over produced counter.
- This example, as written, requires that `UINT_MAX + 1` is evenly divisible by `BUFFER_SIZE`; if it is not evenly divisible, `[Count % BUFFER_SIZE]` produces the wrong buffer index after `Count` wraps past `UINT_MAX` back to zero. An alternate solution without this restriction would employ two additional `Idx` variables to track the current buffer index for the head (producer) and tail (consumer). These `Idx` variables would be used in place of `[Count % BUFFER_SIZE]`, and each of them would have to be incremented at the same time as the respective `Count` variable is incremented, as follows: `Idx = (Idx + 1) % BUFFER_SIZE`.

```
volatile unsigned int produceCount, consumeCount;
TokenType buffer[BUFFER_SIZE];

void producer(void) {
    while (1) {
        while (produceCount - consumeCount == BUFFER_SIZE)
            sched_yield(); // buffer is full

        buffer[produceCount % BUFFER_SIZE] = produceToken();
        // a memory_barrier should go here, see the explanation above
        ++produceCount;
    }
}

void consumer(void) {
    while (1) {
        while (produceCount - consumeCount == 0)
           sched_yield(); // buffer is empty

        consumeToken(buffer[consumeCount % BUFFER_SIZE]);
        // a memory_barrier should go here, the explanation above still applies
```

```
        ++consumeCount;
    }
}
```

See also

- Atomic operation
- Design pattern
- FIFO
- Pipeline

References

1. Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C. (2014), *Operating Systems: Three Easy Pieces [Chapter: Condition Variables]* (PDF), Arpaci-Dusseau Books
2. Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C. (2014), *Operating Systems: Three Easy Pieces [Chapter: Semaphores]* (PDF), Arpaci-Dusseau Books

Further reading

- Mark Grand *Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML (http://www.mindspring.com/~mgrand/pattern_synopses.htm)*
- C/C++ Users Journal (Dr.Dobb's) January 2004, "A C++ Producer-Consumer Concurrency Template Library" (http://www.ddj.com/showArticle.jhtml?articleID=184401751), by Ted Yuan, is a ready-to-use C++ template library. The small template library source code (http://www.bayimage.com/code/ProCon.h) and examples can be found here (http://www.bayimage.com/code/archive.zip)

External links

- Producer Consumer Thread Race Conditions (http://256.com/gray/docs/misc/producer_consumer_race_conditions/), with line by line walk through of the race and sample Java code demonstrating the issues, by Gray Watson

Categories: Concurrency (computer science)
