Concurrent Computing 3: Synchronization

1. Introduction
2. Terminology
3. **Synchronization**
   - **Linux synchronization mechanisms**
   - **semaphores and mutexes**
   - **the producer-consumer problem**
4. C and Concurrency
5. More Asynchrony
6. Event-Driven Programming

Linux Synchronization Mechanisms

Linux/UNIX provides several **synchronization mechanisms**, some of which are designed to synchronize *processes*, some *Pthreads*, and some that can work with both.

The synchronization mechanims that work among *processes* are:

- signals
- semaphores
- file locks
- pipes/FIFOs
- wait for child termination

Linux Synchronization Mechanisms (contd.)

The synchronization mechanims that work among *Pthreads* are:

- mutexes (Pthreads only)
- condition variables (Pthreads only)
- barriers (Pthreads only)
- semaphores
- file locks
- wait for thread termination (Pthreads only)

C11 adds thread support to C, and includes mutexes and condition variables as supported synchronization mechanisms.

Semaphores and Mutexes

Two of the most important synchronization-related concepts are the **semaphore** and the **mutex**.

The semaphore was invented by Edsger Dijkstra in 1965.

The name "semaphore" is borrowed from signaling systems such as those involving flags for use between ships.

A *semaphore* can be used to control access to a (finite size) *shared resource pool*, to eliminate *race conditions* in a concurrent program accessing the shared resource.

A *mutex* is intended to enforce *mutually-exclusive access* to *critical sections* in different threads of a concurrent program, to avoid race conditions that could arise if the critical sections were executed concurrently.

Semaphores and Mutexes (contd.)

A semaphore is a special-purpose *shared non-negative integer*:

- a **general semaphore** typically corresponds to the number of currently available objects in a shared resource pool
- a **binary semaphore** can take on only the values 0 or 1
- a semaphore gets initialized when created, typically to its maximum value or its minimum (which is zero)

The current value of a semaphore indicates how many threads could currently gain access to the resource.

When a thread obtains access to the resource, the semaphore is *decremented*.

When access is relinquished, the semaphore is *incremented*.

Semaphores and Mutexes (contd.)

The key to a semaphore is that the semaphore's value can be modified only by *two special operators*:

- an *increment* operator and a *decrement* operator
- these operators must be **atomic** so that they can be used concurrently without the possibility of race conditions
- the *decrement operator* **blocks** (i.e., causes process to *sleep*) if the integer value is zero when it is called (it cannot become negative)

Semaphores and Mutexes (contd.)

Dijkstra originally named the *decrement* operator **P()** and the *increment* operator **V()**, from Dutch words.

Nowdays, decrement/increment are typically referred to using one of the following combinations of names:

- down/up (e.g., ALGOL 68 and the Linux kernel)
- wait/post (e.g., POSIX semaphores)
- wait/signal (e.g., POSIX condition variables)
- lock/unlock (e.g., POSIX mutex)
- acquire/release (e.g., Java)
- procure/vacate (to be similar to original p and v)

Semaphores and Mutexes (contd.)

A *mutex* is effectively identical to a *binary semaphore*.

Some sources claim that they are conceptually somewhat different, but this was not Dijkstra's view:
"The semaphores are essentially non-negative integers; when only used to solve the mutual exclusion problem, the range of their values will even be restricted to '0' and '1'."

In practice, an OS might specialize its "mutex" and "semaphore" models so this equivalence does not hold.

In Linux/UNIX, mutexes can be used among only Pthreads, while semaphores can be used among processes.

The Producer-Consumer Problem

One of the classic examples used to demonstrate issues and solutions in concurrent programming is the **producer-consumer problem** (**bounded buffer** version):

- the program consists of two threads: (1) the **producer** and (2) the **consumer**
- the *producer* creates data to be processed by the *consumer*
- data is passed from the producer to the consumer by being written into a *fixed-size buffer*
- the *consumer* should read data from the buffer only when there is unread data in the buffer (otherwise it should *sleep* until there is data)
- the *producer* should *not* write data to the buffer if the buffer is *full* (it should instead *sleep* until there is free space again)

The Producer-Consumer Problem (contd.)

For simplicity, we assume the data consists of *fixed-length records*, and the buffer can hold a fixed, integer number of records.

There are several common variations on this basic problem:

- there are *multiple* producers and/or multiple consumers rather than just one of each
- an unlimited number of records can be stored (e.g., by making the buffer a **linked list**)
- the data units are of variable size

The Producer-Consumer Problem (contd.)

An example "solution" without using special mechanisms:

```
//Shared variables:
struct record buff[N];              //array of records to be used for queue
int count = 0;                      //number of records currently in buff

//Producer code:
struct record next_rec;
while (1) {
  next_rec = get_next_record();  //obtain next record--i.e., from TTY
  while (count == N);            //wait for buff to have free space
  enqueue(buff,next_rec);        //store new record in buff queue
  count++;                       //record that additional record available
}

//Consumer code:
struct record next_rec;
while (1) {
  while (count == 0);            //wait for buff to contain record(s)
  next_rec = dequeue(buff);      //get oldest record from buff queue
  count--;                       //record that a record was removed
  process_next_record(next_rec); //process the record appropriately
}
```

The Producer-Consumer Problem (contd.)

Unforunately, there are a few problems with this "solution:"

- the `enqueue()` and `dequeue()` calls are obviously *not atomic*, and if they are interleaved the `buff`-based queue can be corrupted (a *race condition*)
- the `while` loops checking `count` represent *busy waits*, so will waste CPU cycles and can lead to resource starvation issues
- even the increment/decrement operations are not guaranteed to be *atomic*, potentially leading to invalid values for `count` (also a *race condition*)

The Producer-Consumer Problem (contd.)

Fixing the problems requires that:

- the `enqueue()` and `dequeue()` calls be prevented from occuring concurrently—i.e., we must impose *mutual exclsion*
- the producer must be able to be *suspended* (put to *sleep*) if the queue is full and then awakened when this changes
- the consumer must be be *suspended* (put to *sleep*) if the queue is empty and then awakened when this changes
- `count` must be incremented/decremented *atomically* or have its value modified *mutually exclusively*

The Producer-Consumer Problem (contd.)

The standard solution is to use one *binary semaphore* (or *mutex*) and two *general semaphores*.

This requires that the operating system and programming language provide/support these mechanisms.

We will represent semaphores as follows:

- a semaphore will be a variable of *type* `semaphore`
- the *decrement* operation will be called `semwait()`
- the *increment* operation will be called `semsignal()`

The Producer-Consumer Problem (contd.)

An example solution using semaphores:

```
//Shared variables:
struct record buff[N];              //array of records to be used for queue
semaphore queue_op = 1;             //can the buff queue be modified
semaphore queue_avail = 0;          //number of available records in queue
semaphore queue_free = N;           //number of free records in queue

//Producer code:
struct record next_rec;
while (1) {
  next_rec = get_next_record();     //obtain next record--i.e., from TTY
  semwait(queue_free);              //wait for buff to have free space & --
  semwait(queue_op);                //wait to be able to operate on queue
  enqueue(buff,next_rec);           //store new record in buff queue
  semsignal(queue_op);              //signal can operate on queue
  semsignal(queue_avail); }         //signal record(s) available & ++

//Consumer code:
struct record next_rec;
while (1) {
  semwait(queue_avail);             //wait for buff to contain record(s) & --
  semwait(queue_op);                //wait to be able to operate on queue
  next_rec = dequeue(buff);         //get oldest record from buff queue
  semsignal(queue_op);              //signal can operate on queue
  semsignal(queue_free);            //signal queue has free record(s) & ++
  process_next_record(next_rec); } //process the record appropriately
```