

1 Synchronization Constructs

The raw mutex protocol is tedious and error-prone. Even with hardware support, programmers are likely to make errors, and concurrency errors are notoriously difficult to debug. Additionally, spinning on a mutex variable is inefficient (if the thread spins for many hundreds of cycles), plus saturating the memory bus with lots of interlock instructions (most of which aren't necessary) will slow down the system. Ages ago (in the 1970s), Dijkstra realized that abstractions were useful and formulated the first classic synchronization construct, the semaphore. Since then, more constructs have been created: namely monitors, message queues, and transactional memory.

2 Semaphores

A Semaphore is a simple mutex, and is an abstraction of a counter. Strictly speaking, a semaphore is a software object containing a non-negative integer (0 or more), which we'll call S . A semaphore has two operations:

Wait/DOWN: delay/block while $S = 0$, else decrement S atomically.

Signal/UP: increments S atomically. This will release blocked threads

At its core, the semaphore isn't much different from the flag-style mutex we investigated earlier. If the semaphore is initialized to 1, then it exhibits the same behavior as the flag mutex. **Question:** what happens when the semaphore is initialized to a value larger than 1? Using an extension of Java that has a CAS-style atomic update operator a semaphore might look like:

```
class Semaphore{
    private int S;

    public Semaphore(int init){ S = init; }

    public void down(){
        int i = S;
        while((S == 0) || (!CAS(S, i, i-1))){
            wait();
            i = S;
        }
    }

    public int up(){
        int i = S;
        while(!CAS(S, i, i+1)){
            i = S;
        }
    }
}
```

```
}  
}
```

In both the code for up and down, we read the value of S into a local integer first. Then we use this integer to calculate values for the CAS. What this means is that if the CAS fails, that implies that another thread changed the value of S out from under us. If that happens, we need to refresh our view of the world and then retry. down leverages the short-circuited logical or construct in java so that the CAS is only attempted if the semaphore is not zero. An alternate approach would be to have the semaphore be guarded by a locking flag and just use the flag-style mutex primitives from previous lectures to prevent concurrent modification of the semaphore.

In practice, semaphores are usually implemented in two ways. First, using something like a CAS, semaphores are created as abstract data types in a library for a programming language. The programmer then just uses semaphores like any other library data structure. Second, semaphores are implemented as a primitive in the OS and are accessed via system calls (e.g. down and up are syscalls). Linux has support for OS semaphores, for example.

2.1 example usage

For semaphores (and following examples) I'll be using the example of the bounded buffer. Recall that a bounded buffer is abstractly a queue with a fixed maximum size (a fixed-size array, basically), and that readers/consumers should block on an empty queue and that writers/producers should block on a full queue.

If we assume a pseudo-C definition of a bounded buffer like so:

```
typedef struct{  
    void* buf[BUF_SIZE];  
    int start;  
    int len;  
    semaphore buf_sem;  
    semaphore empty(BUF_SIZE);  
    semaphore full(0);  
}BoundedBuffer;
```

Then buf_sem is the semaphore guarding the buffer proper. empty is a semaphore used for signalling an empty queue (initialized to BUF_SIZE). full is a semaphore for signalling a full queue (initialized to 0).

Then a function to add something to the buffer would look like:

```
void add(BoundedBuffer* buf, void* value){  
    down(buf->empty); //blocks if queue is full  
    down(buf->buf_sem);  
    //critical section  
    buf->buf[(buf->start + buf->len) % BUF_SIZE] = value;  
    buf->len++;
```

```

    up(buf->buf_sem);
    up(buf->full);
}

```

Already some of the complexities of using semaphores is becoming apparent. A producer decrements the empty semaphore and increments the full semaphore. Note too, that the semaphore guarding the bounded buffer data is acquired AFTER the condition is checked. This is absolutely necessary. If the thread grabbed `buf_sem` before checking `empty`, the thread would be unable to release `buf_sem`. This would prevent any other threads from being able to modify the buffer. This is a situation known as deadlock, and what it means is that the program is stuck, and cannot proceed. And yes, deadlock is a very serious bug.

Now consider the code for the consumer function, remove:

```

void* remove(BoundedBuffer* buf) {
    down(buf->full);
    down(buf->buf_sem);
    //critical section
    void* val = buf->buf[buf->start];
    buf->start = (buf->start + 1) % BUF_SIZE;
    buf->len--;
    up(buf->buf_sem);
    up(buf->empty);
    return val;
}

```

In this case, on an empty queue (`full == 0`), a consumer would block until an element was added. And if a queue is full (`empty == 0`), a producer would block until an element is removed.

As you can see, semaphores can be complicated. Semaphores attempt to unify mutual exclusion (binary semaphores) and conditional signalling (semaphores > 1). This means that the semaphore primitive functions (`up`, `down`) end up doing double duty as locking and signalling functions. This can obscure the semantics of the threaded program somewhat. In larger programs semaphore calls can get scattered throughout the code, making it harder to identify critical sections. In practice, semaphores are used in conjunction with other synchronization constructs rather than as the only game in town.

3 Monitors

Monitors are a synchronization construct associated with a collection of data and methods. The monitor itself ensures that only one thread can execute a monitor method at a time. The mutual exclusion is enforced by the language runtime or the compiler. Monitor data is only accessible within a monitor method, and this ensures that that data is safe from data races. This coupling of data and functionality is reminiscent of OOP and, in fact, the default Java mutual exclusion mechanism (`synchronized`) is a variant of monitors. Monitors support synchronization through condition variables.

A monitor, in its simplest form, supports two methods:

wait(c): Suspend execution/block the calling thread on condition *c*. The monitor now becomes available to another thread.

signal(c): Resume execution of a thread blocked on condition *c*. If there are several such threads, pick one. If there are no waiting threads, do nothing.

With simple monitors (also called Hoare monitors, after the computer scientist who invented them), if there are no waiting threads to receive a signal, the signal is lost. Because the monitor itself ensures that only one thread is executing a monitor method at any given time, monitor code can dispense with the tedious lock/unlock style code surrounding critical sections. Note too that because signal activates a waiting thread, the currently executing thread must relinquish the monitor to the signalled thread.

3.1 example usage

Java does use monitors, however they are somewhat different. In standard Java monitors, there are no condition variables (which can make things tedious). So, for this example we're going to continue with the bounded buffer example in psuedo-C. Assume that there's a new keyword, `monitor` that "monitorizes" a struct and methods on that struct. In this case the definition of the bounded buffer may look like:

```
monitor BoundedBuffer{
    void* buf[BUF_SIZE];
    int start;
    int len;
    condition notfull;
    condition notempty;
};
```

`notfull` and `notempty` are just condition variables of this monitor. Now add would look like:

```
monitor void add(BoundedBuffer* buf, void* val) {
    if(buf->len >= BUF_SIZE) {
        wait(buf->notfull);
    }

    buf->buf[(buf->start + buf->len) % BUF_SIZE] = value;
    buf->len++;

    signal(buf->notempty);
}
```

As you can see, monitors can clarify the code quite a bit over semaphores. If the queue is full, a thread invoking `add` will wait on the `notfull` condition. When the thread is finished adding an element, it will signal any waiting threads that the queue is no longer empty. Because non-recieved signals are discarded, you can always signal `notempty`, even if the queue wasn't empty to begin with. This may be a tad inefficient, but it is much clearer. Now, lets look at `remove`:

```
monitor void* remove(BoundedBuffer* buf){
    if(buf->len == 0){
        wait(buf->notempty);
    }

    void* val = buf->buf[buf->start];
    buf->start = (buf->start + 1) % BUF_SIZE;
    buf->len--;

    signal(buf->notfull);
    return val;
}
```

Again, clearer than the semaphore version (at least to me, that is). Of course, there are some drawbacks. If the thread calling `signal` hasn't finished execution, then it can be very inconvenient to have to immediately switch to another thread (which **MUST** be done to avoid races on the monitor itself). This requirement imposes an implementation requirement, namely that signalling and thread scheduling be perfectly reliable. This is required in order to correctly arbitrate between multiple threads that may be trying to enter the monitor while multiple threads may be signalling. In practice, ensuring this kind of reliable scheduling on a multi-processor system is often quite painful. In fact, it may require halting all threads (which can be difficult), and this can be grossly inefficient. For this reason, a variant of Hoare monitors, called mesa monitors were developed. Instead of `signal`, mesa monitors have a function `notify` that sends a signal to a waiting thread, but doesn't necessarily switch out the currently running thread. What this means is that the signalled thread may be run after some other threads have got to the monitor. So this thread may need to recheck the condition that caused it to wait in the first place. What this means in the bounded buffer case is that the `if` statements are replaced with `while` loops (because the thread may have to wait multiple times).

3.2 java Monitors

Java uses a version of Mesa monitors. Java has a `wait` and `notify` function. Java introduces a `notifyAll` function that wakes up all waiting threads on a monitor. A class in Java is declared to be a monitor at the function level, where functions declared as `synchronized` are monitor methods. Unfortunately, Java monitors **DON'T** have condition variables! signals are sent to all threads waiting on a monitor. So, the bounded buffer example in Java would look like:

```
class BoundedBuffer{
```

```

private Object[] buf;
private int len;
private int start;
public static final int BUF_SIZE = 32;

public BoundedBuffer(){
    buf = new Object[BUF_SIZE];
    len = 0;
    start = 0;
}

public synchronized void add(Object o){
    while(len >= BUF_SIZE){
        wait();
    }

    buf[(start + len) % BUF_SIZE] = o;
    len++;

    notifyAll();
}

public synchronized Object remove(){
    while(len == 0){
        wait();
    }

    Object obj = buf[start];
    start = (start + 1) % BUF_SIZE;
    len--;

    notifyAll();
}
}

```

Because Java lacks condition variables, both add and remove have to signal ALL waiting threads. So all the producers waiting for an empty queue will be awakened by a completing producer. This is fairly inefficient, and one of the many reasons why Java concurrency was overhauled in Java 1.5 and Java 1.6.

3.3 Semaphores in Monitors

Semaphores can be implemented in terms of monitors trivially. The following code illustrates how to do it in old-school Java.

```
class Semaphore{
    private int S;

    public Semaphore(int init){ S = init; }

    public synchronized up(){
        S++;
        notify();
    }

    public synchronized down(){
        while(S == 0){
            wait();
        }

        S--;
    }
}
```

Because the monitor ensures that only one thread may execute monitor code at a time, the updates to `S` are atomic. Also, because `up` only increments `S` by 1, the calling thread need only wake one waiting thread (as only one additional thread may enter). Hence the call to `notify` rather than `notifyAll`. `notifyAll` would also be correct, just inefficient. Consider the case where there are many waiting threads. A single call to `up` will only be able to allow a single one of those threads through. Therefore if all the waiting thread were activated (with a call to `notifyAll`), then most of them would simply go back to sleep without accomplishing any useful work (this is sometimes called the “thundering herd” problem, because it potentially activates many threads that all start racing for a single resource).

4 Message Passing

Another synchronization tool is message passing. In a message passing system threads communicate by sending and receiving messages. Messages can be used to enforce both conditional synchronization as well as mutual exclusion. In the abstract, the two primitives look like:

```
send(destination, message)

message = receive(source)
```

First, let's consider send. There are two possibilities, either the sending thread is blocked until the message is received, or it is not. Similarly, for receive if there is a message waiting, it is received. If there is no message either the receiver blocks awaiting a message or not. In general, non-blocking send, blocking receive is the normal way to do things. The sender doesn't have to wait for a recipient to show up, and a recipient will wait around for a message to arrive. There are three basic cases:

Blocking send, Blocking receive AKA "the rendezvous", this ensures that both the sender and receiver are attached to the message queue at the same time. This allows for tight synchronization.

Non-blocking send, blocking receive The normal state of affairs. This is how networking code usually works. The sender sends their data/message off to a recipient, and then continues working. Whereas a receiver usually has to get the message before any useful work can be done.

Non-blocking send, non-blocking receive Asynchronous communication. Both sender and receiver are very loosely coupled. This is the case in many high-throughput server applications where the server (receiver) often has other requests to process and so cannot block.

Another issue is one of addressing. In a **direct** addressing scheme, the threads explicitly send messages to one another (e.g. thread 1 sends a message to thread 2). In an **indirect** scheme, the messages are sent to a message queue (or buffer). This indirection means that senders and receivers do not need to have information about one another. A sender can send a message to a queue and continue executing, while any receiver can pluck a waiting message from the queue. In general, indirect messaging is often used (information hiding usually simplifies design). Another consideration is the mapping from senders to receivers.

1 to 1 A message can only be received by one receiver.

1 to many A message from one sender will be sent to multiple receivers (broadcasting)

many to 1 Many senders can send to a single receiver (client(sender)-server(receiver))

many to many The most general case, many senders may send messages that will be received by multiple receivers

Yet another issue is message formatting. The messages themselves can get arbitrarily complex. For a system that needs to emulate mutual exclusion and condition variables, the message really just needs to be an identifier that specifies the condition in question. But, in general, you could have very sophisticated message structures. Again, this is not often done in practice simply for performance reasons.

4.1 Mutual Exclusion

With message passing, mutual exclusion can be thought of as a kind of relay race. There is a magic token (the relay wand) associated with a critical section, and the thread that holds this token can execute the critical section. When the thread is done, it passes the token on to another thread. This token is simply a message. For each critical section, one can create a message queue (initialized with a single message). Before a thread may enter the critical section, it will try to get the message (receive). Once the thread has the message, it may enter the critical section. On exiting, the message is sent back to the queue. For example:

```
MessageQueue lock;

void foo(){
    message msg;
    receive(&lock, &msg); //lock
    //CRITICAL SECTION
    send(&lock, NULL);    //unlock
}
```

For the example, assume that this psuedo-C dialect has a message queue construct called `msgQ`, and two functions: `send(msgQ*, message*)` and `receive(msgQ*, message*)`.

```
typedef struct{
    void* buf[BUF_SIZE];
    int start;
    int len;
    msgQ* may_produce;
    msgQ* may_consume;
} BoundedBuffer;

BoundedBuffer* createNewBuffer(){
    BoundedBuffer* buf = malloc(sizeof(BoundedBuffer));
    buf->start = 0;
    buf->len = 0;
    buf->may_produce = createNewMsgQ();
    buf->may_consume = createNewMsgQ();

    int i = 0;
    for(i = 0 ; i < BUF_SIZE ; i++){
        send(buf->may_produce, NULL);
    }
}

void add(BoundedBuffer* buf, void* val){
```

```

message msg;
receive(buf->may_produce, &msg);

buf->buf[(buf->start + buf->len) % BUF_SIZE] = value;
buf->len++;

send(buf->may_consume, NULL);
}

void* remove(BoundedBuffer* buf) {
message msg;
receive(buf->may_consume, &msg);

void* val = buf->buf[buf->start];
buf->start = (buf->start + 1) % BUF_SIZE;
buf->len--;

send(buf->may_produce, NULL);
return val;
}

```

In this example, there are two message queues. One for each of the conditions. The `may_produce` queue has to be prepopulated with `BUF_SIZE` messages so that producers (if so inclined) may fill up the buffer.

Message passing is a very general mechanism and is used for much more than just synchronization. For example, smalltalk-derived languages use message passing instead of method calls. And in a networking context, abstractions like sockets use a message-passing abstraction to hide the complexities of network communication. In general, if a language/system already has message passing features, then message-passing can be used to synchronize. Unfortunately, message-passing can be tricky to make efficient, therefore many systems adopt one of the other, faster schemes. In some cases, systems with message passing may elect to also support more synchronization-specific schemes to allow programmers to choose speed over generality.

Message passing for multithreading has experienced some traction in the functional language community, however. Both concurrent ML and concurrent Haskell use message queues to synchronize multiple threads.

4.2 Message Queues with Monitors

5 pthreads locks

6 Transactions/Atomic Sections