Semaphores

A Binary (or Mutex) Semaphore: serves as a lock. It can be used to guarantee mutually exclusive access to a resource (only one process is in the critical section at a time). It is initialized to free (value = 1).

A Counting Semaphore: represents a resource with many units available. The initial count to which the semaphore is initialized is usually the number of resources. A counting semaphore lets a process continue as long as more instances are available.

A queue of waiting processes/threads is associated with each semaphore.

Semaphores

A semaphore supports two atomic operations, Semaphore->Wait() and Semaphore->Signal().

```
S->Wait()              // wait until semaphore S
                       // is available
<critical section>

S->Signal()            // signal to other processes
                       // that semaphore S is free
```

- Each semaphore supports a queue of processes that are waiting to access the critical section (e.g., to buy milk) — or to do something else!
- If a process executes **S->Wait()** and semaphore S is free, it continues executing. If semaphore S is not free, the OS puts the process on the wait queue for semaphore S.
- A **S->Signal()** unblocks one process on semaphore S's wait queue.

Simple Example

Mutual exclusion using semaphores:

```
Thread A                    Thread B

Semaphore->Wait();          Semaphore->Wait();
if (noMilk){                if (noMilk){
   buy milk;                   buy milk;
}                           }
Semaphore->Signal();        Semaphore->Signal();
```

Counting Semaphores

A Counting Semaphore: represents a resource with many units available. The initial count to which the semaphore is initialized is usually the number of resources (value=n). A counting semaphore lets processes continue as long as more instances are available.

```
S->Wait()              // Decrement value.
                       // If value < 0, wait on queue.
<critical section>

S->Signal()            // Increment value.
                       // If there is a waiting process on the
                       // queue, wake it up.
```

Example: For each process, the OS keeps track of the open files. In order to have a PCB of fixed size, the OS only allows each process to open a maximum of 16 files. If the OS wants to support a file protocol that forces the thread to wait when the maximum is exceeded, we can write:

```
Semaphore->value = 16;   // initialization

Kernel::OpenFile(...) {
  Semaphore->Wait();
  ...
}
Kernel::CloseFile(...) {
  ...
  Semaphore->Signal();
}
```

Implementing Signal and Wait

```
class Semaphore {
  public:
    void Wait(Process P);
    void Signal();
  private:
   int value;
   Queue Q;    // queue of processes;
}
Semaphore::Semaphore(int val) {
   value = val;  // Avail-
able resources...
   Q = empty;
}
Semaphore::Wait(Process P) {
   value = value - 1;
   if (value < 0) {
      add P to Q;
      P->block();
}  }
Semaphore::Signal() {
   value = value + 1;
   if (value <= 0){
      remove P from Q;
      wakeup(P);
}  }
```

$\Longrightarrow$ Signal and Wait of course must be atomic!

Signal and Wait

Example:

```
P1:   S->Wait();
      S->Wait();
      S->Signal();
      S->Signal();
```

```
P2:   S->Wait();
      S->Signal();
```

			process state: execute or block	
	value	Queue	P1	P2
	2	empty	execute	execute
P1: S->Wait();				
P2: S->Wait();				
P1: S->Wait();				
P1: S->Signal();				
P2: S->Signal();				
P1: S->Signal();				

	value	Queue	P1	P2
	1	empty	execute	execute
P1: S->Wait();				
P2: S->Wait();				
P1: S->Wait();				
P1: S->Signal();				
P2: S->Signal();				
P1: S->Signal();				

Using Semaphores

1. **Mutual Exclusion** - the semaphore has an initial value of 1, S->Wait() is called before the critical section, and S->Signal is called after the critical section.
2. **Scheduling Constraints** - Semaphores can be used to express general scheduling constraints where threads must wait for some circumstance. The initial value of the semaphore is usually 0 in this case.

   Example: You can implement thread *join* (or the Unix system call waitpid(PID)) with semaphores:

   ```
   Semaphore S;

   S->value = 0; \\ semaphore initialization

   Thread::Join
       S->Wait();

   Thread::Finish
       S->Signal();
   ```

Multiple Consumers and Producers

```
class BoundedBuffer {
   public:
     void Producer();  void Consumer();
   private:
     Items *buffer;
     Semaphore mutex; // control access to shared buffer
     Semaphore empty; // count of empty slots
     Semaphore full;  // count of full slots
}
BoundedBuffer::BoundedBuffer(int N) {
     mutex->value = 1;
     empty->value = N;
     full->value  = 0;
     new buffer[N];
}
BoundedBuffer::Producer() {
   <produce item>
   empty->Wait();     // one fewer buffer, or wait
   mutex->Wait();     // get access to buffer pointers
   <add item to buffer>
   mutex->Signal();   // release pointers
   full->Signal();    // note one more item in buffer
}
BoundedBuffer::Consumer() {
   full->Wait();      // wait until there’s an item
   mutex->Wait()      // get access to buffer pointers
   <remove item from buffer>
   mutex->Signal();   // release pointers
   empty->Signal();   // count down
   <use item> }
```

Multiple Consumers and Producers Problem

	empty	full
initially	● ● ● ●	○ ○ ○ ○
Producer 1 empty->wait();	● ● ● ○	
... full->signal();		● ○ ○ ○
Producer 2 empty->wait();	● ● ○ ○	
... full->signal();		● ● ○ ○
Consumer full->wait();		● ○ ○ ○
... empty->signal();	● ● ● ○	

Recap

Semaphores can be used for three purposes:

- To ensure mutually exclusive execution of a critical section (as locks do).
- To control access to a shared pool of resources (using a counting semaphore).
- To cause one thread to wait for a specific action to be signalled from another thread.

More Examples Using Semaphores

Readers/Writers Problem

- An object is shared among several threads, some that only read it and some that write it.
- To get correct results, we allow *multiple readers* at a time, but *only one writer* at a time.
- How do we control access to the object to permit this protocol?

Correctness criteria:

- Each read or write of the shared data must be inside a critical section.
- Guarantee mutual exclusion for writers.
- Allow multiple readers to execute in the critical section at once.

Readers/Writers Problem

```
class ReadWrite {
   public:
     void Read();
     void Write();
   private:
     int       readers; // counts readers
     Semaphore mutex;   // controls access to readers
     Semaphore wrt;     // controls entry to first
}                       // writer or reader
ReadWrite::ReadWrite {
   readers      = 0;
   mutex->value = 1;
   wrt->value   = 1;
}
ReadWrite::Write(){
   wrt->Wait();         // any writers or readers?
   <perform write>
   wrt->Signal();       // enable others
}
ReadWrite::Read(){
   mutex->Wait();       // ensure mutual exclusion
      readers += 1;     // another reader
      if (readers == 1)
         wrt->Wait();   // block writers
   mutex->Signal();
   <perform read>
   mutex->Wait();       // ensure mutual exclusion
      readers -= 1;     // reader done
      if (readers == 0)
         wrt->Signal();// enable writers
   mutex->Signal(); }
```

Readers/Writers Solution Discussion

Implementation notes:

1. The first reader blocks if there is a writer; any other readers who try to enter block on `mutex`.
2. The last reader to exit signals a waiting writer.
3. When a writer exits, if there is both a reader and writer waiting, which goes next depends on the scheduler.
4. If a writer exits and a reader goes next, then all readers that are waiting will fall through (at least one is waiting on `wrt` and zero or more can be waiting on `mutex`).
5. Does this solution guarantee all threads will make progress?

Alternative desirable semantics:

- Let a writer enter its critical section as soon as possible.

Readers/Writers Solution Favoring Writers

```
ReadWrite::Write(){
   write_mutex->Wait();  // ensure mutual exclusion
      writers += 1;      // another pending writer
      if (writers == 1)  // block readers
         read_block->Wait();
   write_mutex->Signal();
   write_block->Wait();  // ensure mutual exclusion
   <perform write>
   write_block->Signal();
   write_mutex->Wait();  // ensure mutual exclusion
      writers -= 1;      // writer done
      if (writers == 0)  // enable readers
         read_block->Signal();
   write_mutex->Signal(); }
}
ReadWrite::Read(){
   write_pending->Wait();  // ensures at most one reader will go
                           // before a pending write
   read_block->Wait();
   read_mutex->Wait();       // ensure mutual exclusion
      readers += 1;          // another reader
      if (readers == 1)      // synchronize with writers
         write_block->Wait();
   read_mutex->Signal();
   read_block->Signal();
   write_pending->Signal();
   <perform read>
   read_mutex->Wait();       // ensure mutual exclusion
      readers -= 1;          // reader done
      if (readers == 0)      // enable writers
         write_block->Signal();
   read_mutex->Signal(); }
```

Summary

Semaphores, binary and counting, are powerful primitives that allow one to enforce mutual exclusion and other synchronization constraints.

However, they are not structured: semaphore as control mechanism is divorced from the shared data it protects (in the mutual exclusion scenario).