Semaphores

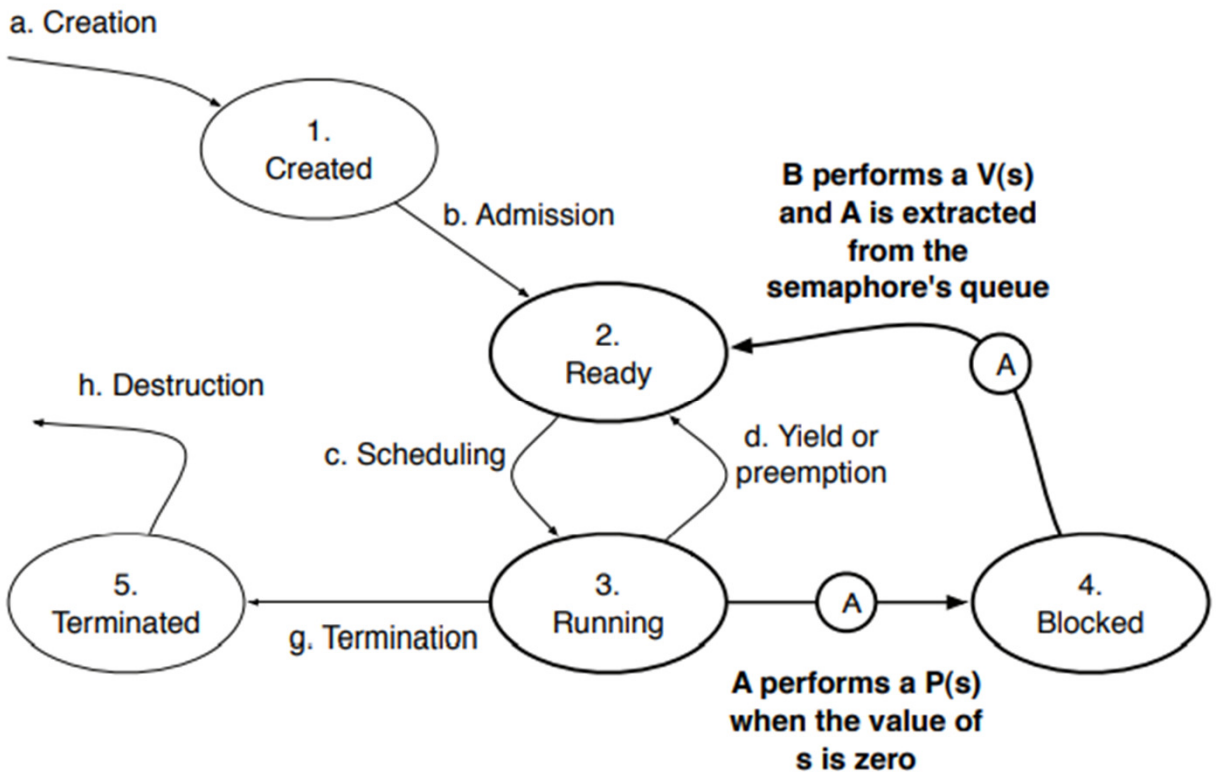
- A semaphore is an object that comprises two abstract items of information:
  - 1. a nonnegative, integer value;
  - 2. a queue of processes passively waiting on the semaphore.

- Upon initialization, a semaphore acquires an initial value specified by the programmer, and its queue is initially empty.

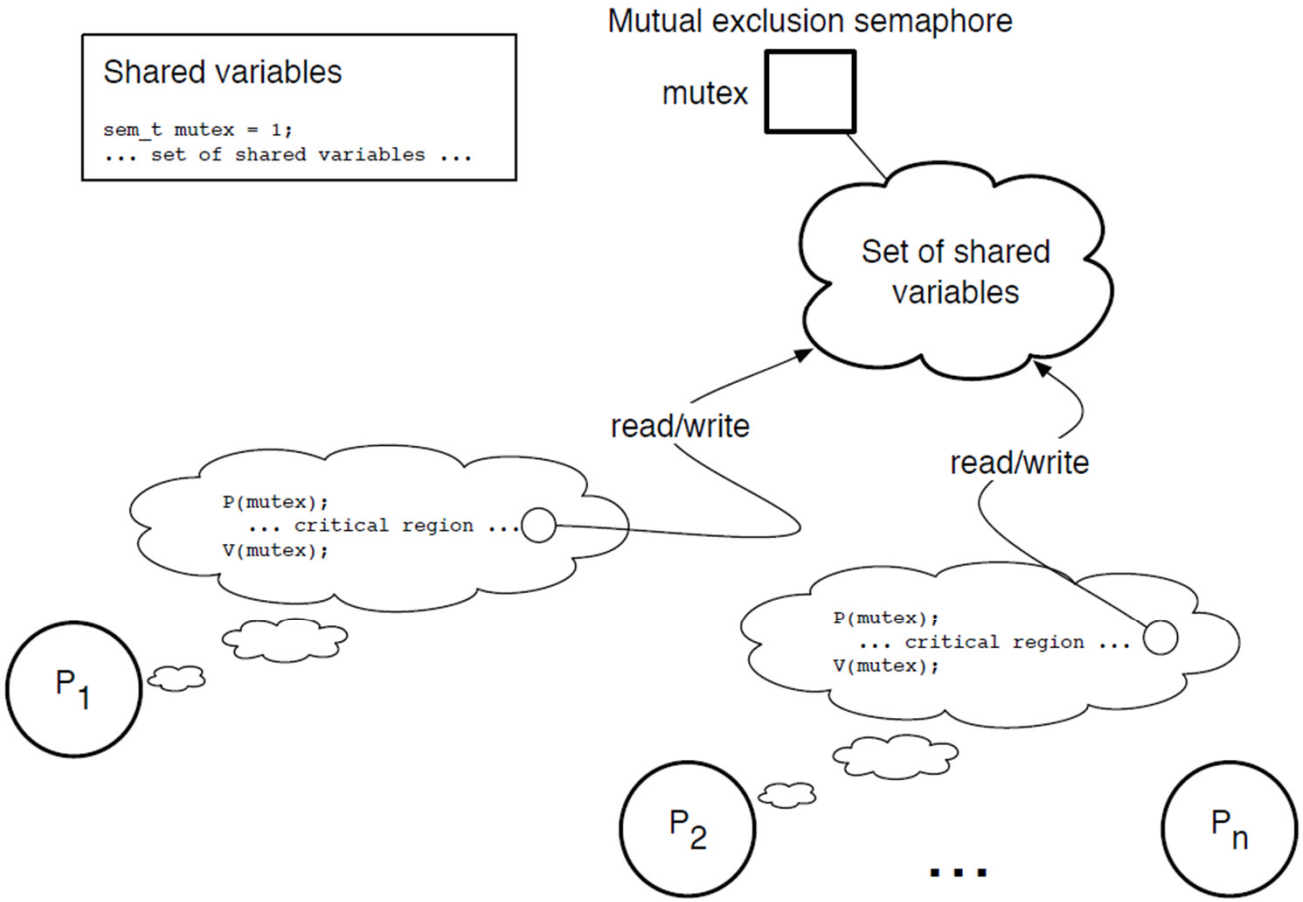
- Neither the value nor the queue associated with a semaphore can be read or written directly after initialization.

- On the contrary, the only way to interact with a semaphore is through the following two primitives that are assumed to be executed atomically:
  - 1. P(s), when invoked on semaphore s, checks whether the value of the semaphore is (strictly) greater than zero. If this is the case, it decrements the value by one and returns to the caller without blocking. Otherwise, it puts the calling process into the queue associated with the semaphore and blocks it by moving it into the Wait (Blocked) state of the process state diagram.
  - 2. V(s), when invoked on semaphore s, checks whether the queue associated with that semaphore is empty or not. If the queue is empty, it increments the value of the semaphore by one. Otherwise, it picks one of the blocked processes found in the queue and makes it ready for execution again by moving it into the Ready state of the process state diagram.

Semaphores and Process states



Using Semaphores to protect a critical section



Using Semaphores for condition synchronization

- Besides mutual exclusion, semaphores are also useful for condition synchronization, that is, when we want to block a process until a certain event occurs or a certain condition is fulfilled.
- For example, considering the producers–consumers problem, it may be desirable to block any consumer that wants to get a data item from the shared buffer when the buffer is completely empty, instead of raising an error indication.
    - A blocked consumer must be unblocked as soon as a producer puts a new data item into the buffer.
- Symmetrically, we might also want to block a consumer when it tries to put more data into a buffer that is already completely full.
- In order to do this, we need one semaphore for each synchronization condition that the concurrent program must respect. In this case, we have two conditions, and hence we need two semaphores:
    - 1. The semaphore empty counts how many empty elements there are in the buffer. Its initial value is N because the buffer is completely empty at the beginning. Producers perform a P(empty) before putting more data into the buffer to update the count and possibly block themselves, if there is no empty space in the buffer.After removing one data item from the buffer, consumers perform a V(empty) to either unblock one waiting producer or increment the count of empty elements.
    - 2. Symmetrically, the semaphore full counts how many full elements there are in the buffer. Its initial value is 0 because there is no data in the buffer at the beginning. Consumers perform P(full) before removing a data item from the buffer, and producers perform a V(full) after storing an additional data item into the buffer.

The producer - consumer example

```
void prod(int d) {
    P(empty);
    P(mutex);

    buf[in] = d;
    in = (in+1) % N;

    V(mutex);
    V(full);
}
```

```
#define N 8
int buf[N];
int in=0, out=0;
sem_t mutex=1;
sem_t empty=N;
sem_t full=0;
```

```
int cons(void) {
    int c;

    P(full);
    P(mutex);

    c = buf[out];
    out = (out+1) % N;

    V(mutex);
    V(empty);

    return c;
}
```

Semaphores in Linux

- **`sem_t *sem_open(const char *`*`name`*`, int `*`oflag`*`, mode_t `*`mode`*`, unsigned int `*`value`*`)`**
  Open a named semaphore

- **`int sem_close(sem_t *`*`sem`*`);`**
  Close a unnamed semaphore

- **`int sem_init(sem_t *`*`sem`*`, int `*`pshared`*`, unsigned int `*`value`*`);`**
  Initialize an unnamed semaphore

- **`int sem_post(sem_t *`*`sem`*`);`**
  V Operation

- **`int sem_wait(sem_t *`*`sem`*`);`**
- **`int sem_trywait(sem_t *`*`sem`*`);`**
- **`int sem_timedwait(sem_t *restrict `*`sem`*`, const struct timespec *restrict `*`abs_timeout`*`);`**
  P Operation

Monitors

- Semaphore represent a rather ‘low level’ approach for synchronization and lead to complicated solutions with increased risk of bugs.

- To address these issues, a higher-level and more structured interprocess communication mechanism, called monitor, was proposed by Brinch Hansen and Hoare.
  - It is interesting to note that, even if these proposals date back to the early ’70s, they were already based on concepts that are common nowadays and known as object-oriented programming.

- In its most basic form,a monitor is a composite object and contains
  - a set of shared data;
  - a set of methods that operate on them.

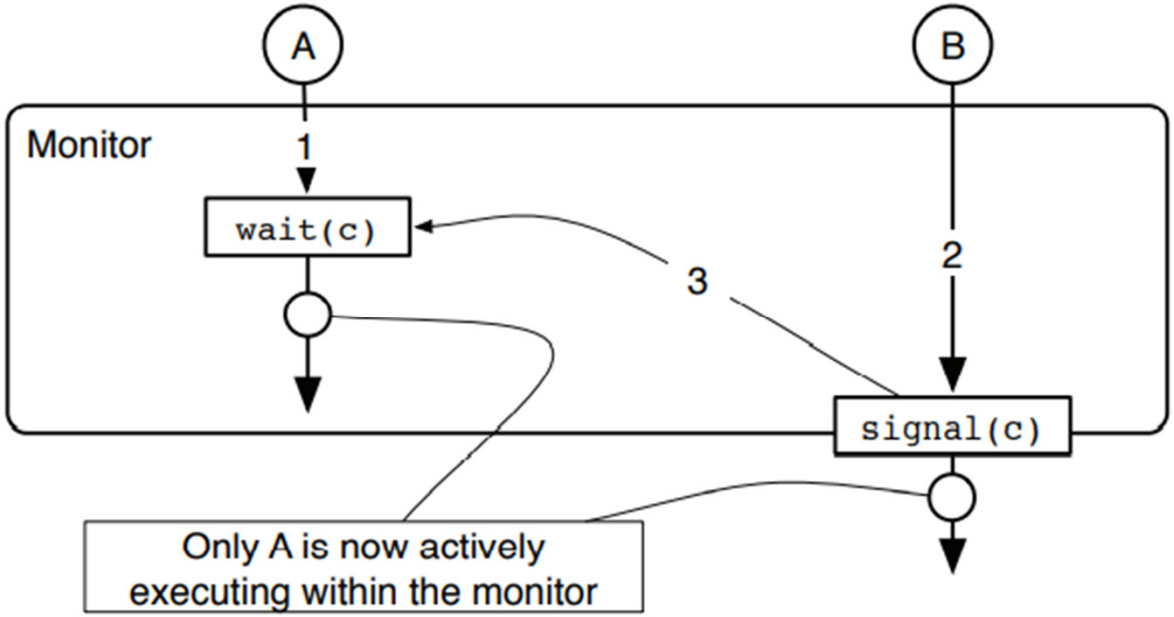
- With respect to its components, a monitor guarantees the following two main properties:
  - Information hiding, because the set of shared data defined in the monitor is accessible only through the monitor methods and cannot be manipulated directly from the outside. Monitor methods are not hidden and can be freely invoked from outside the monitor.
  - Mutual exclusion among monitor methods, that is, the monitor implementation, must guarantee that only one process will be actively executing within any monitor method at any given instant.

- No direct support in C++, but the java synchronized method closely reflect the monitor approach

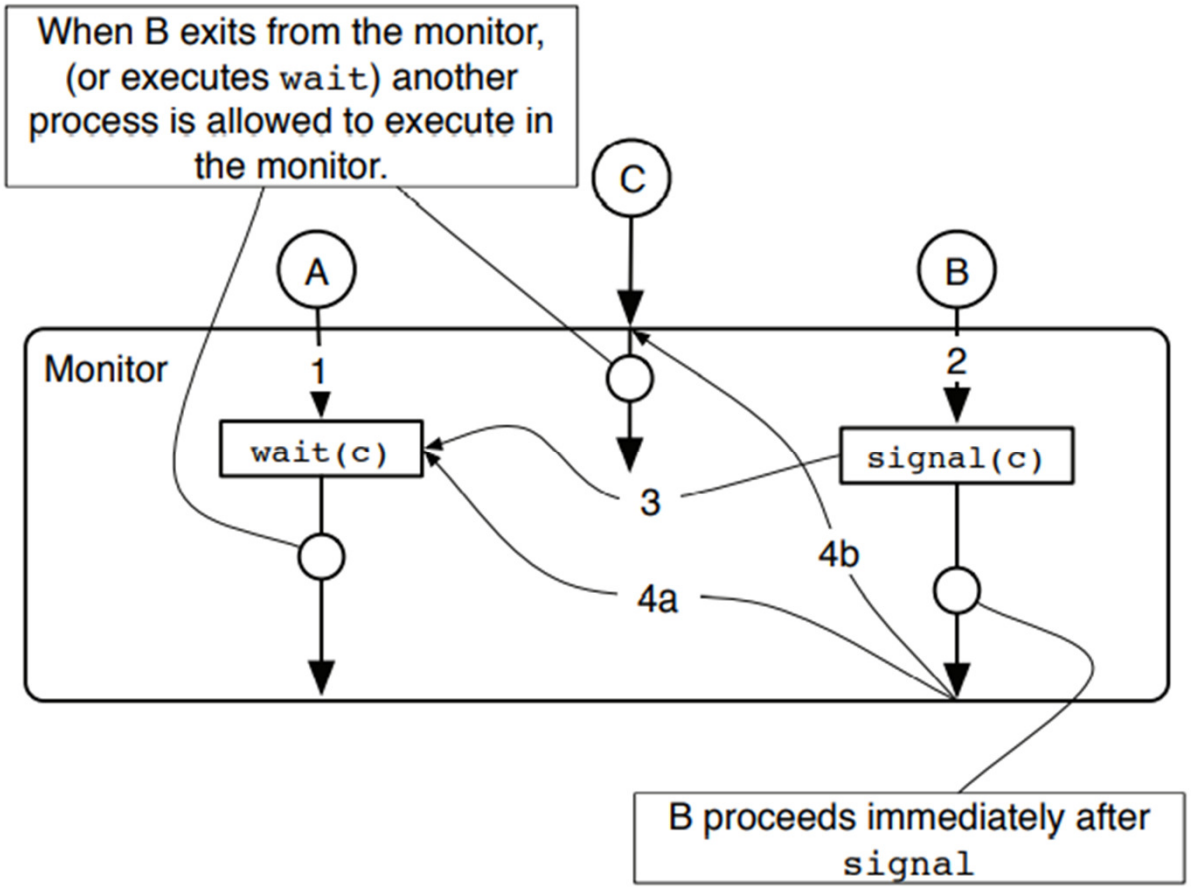
Synchronization in monitors

- We have seen that semaphores can be also used for synchronization
- Synchronization is achieved also in monitors by means of the ***condition variables***.
- Condition variables can be used only by the methods of the monitor they belong to, and cannot be referenced in any way from outside the monitor boundary. They are therefore hidden exactly like the monitor's shared data.
- The following two primitives are defined on a condition variable c:
  - **`wait(c)`** blocks the invoking process and releases the monitor in a single, atomic action.
  - **`signal(c)`** wakes up one of the processes blocked on c; it has no effect if no processes are blocked on c.
- The informal reasoning behind the primitives is that, if a process starts executing a monitor method and then discovers that it cannot finish its work immediately, it invokes wait on a certain condition variable. In this way, it blocks and allows other processes to enter the monitor and perform their job. When one of those processes, usually by inspecting the monitor's shared data, detects that the first process can eventually continue, it calls signal on the same condition variable.
- **`wait()`** and **`signal()`** are implemented as java object methods and in this case the condition variable is the object instance itself
- Condition variables are also provided by POSIX for thread synchronization, but the Monitor abstraction is not pushed further, and mutexes will be defined to handle critical sections

Two possible synchronization implementations (1)



Two possible synchronization implementations (2)



Producer-Consumer implementation using a Monitor

```
#define N 8

monitor ProducersConsumers
{
    int buf[N];
    int in = 0, out = 0;
    condition full , empty;
    int count = 0;

    void produce (int v)
    {
        if(count == N) wait( empty );
        buf[in] = v;
        in = (in + 1) % N;
        count = count + 1;
        if(count == 1) signal (full );
    }
```

```
int consume (void)
  {
        int v;

        if(count == 0) wait(full );

        v = buf[out];
        out = (out + 1) % N;
        count = count - 1;

        if(count == N -1) signal (empty );

        return v;
  }

}
```

Mutexes and Conditions in POSIX

- Creation of a mutex variable (pthread_mutex_t)

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *
attr);
```

- Mutex lock/unlock/trylock

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Creation of a condition variable

```
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t
*restrict attr);
```

- Condition wait

```
int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t *mutex);
```

- Condition signal

```
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```