**Sync. Primitives**

- **Busy-waiting**
  - **Software solutions** (Dekker’, Bakery, etc.)
  - **Hardware-assisted solutions** (based on atomic read-modify-write instructions)

  - **Semaphore**: it contains an internal counter indicating the number of resources.
    - **Binary Semaphore** is a special semaphore, whose counter value can only be 0 or 1; it can be used as a *mutex*

- **Blocking**
  - **Monitor**: language-assisted synchronization; data + procedures + lock
  - **Condition variable**: guarded by a lock to wait on some event
Monitor: a language-assisted construct that encapsulates data, procedures, and synchronization. A monitor contains a mutex lock and a queue.

A process has to acquire the lock before invoking any of the procedures; if the lock is not available, the process sleeps at the queue. When a monitor procedure returns, the lock is released automatically.

Compiler *automatically* inserts lock/unlock operations upon entry/exit of each monitor procedure.
Condition Variable (CV)

• Note that a monitor contains a queue (waiting for the lock), while a CV has another queue (waiting to be waken up)

• Three operations for CVs
  – **Wait(c):** It does two things
    • The calling process is put into the wait queue of the CV
    • Release the monitor lock (so somebody else can get in)
  – **Signal(c):** it also does two things
    • Move the calling process out of the CV’s queue
    • Move the calling process into the monitor’s queue
    • Note that: once it acquires the monitor lock, it resumes from where it was suspended (after the wait() call)
  – **Broadcast(c)**
    • Wake up all processes in the wait queue of the CV, and put them into the monitor’s queue
How to use a Condition Variable

1. Always perform CV operations within critical sections protected by a lock
2. Always put the wait operation in a loop (Mesa type)
   - In order to check whether the condition is true
   - Note the the `signal(c)` only hints the conditions changes; it does not imply that the condition is definitely true

```c
Strange_withdraw(x) {
    ... 
    pthread_mutex_lock(&m);
    while(account < x) // “if” will not work
        pthread_cond_wait(&c, &m);
    account -= x;
    pthread_mutex_unlock(&m);
}

Deposit(y){
    pthread_mutex_lock(&m);
    account += y;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}
```

```c
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```
“A Condition Variable has to be used with a lock.” Is this true?

Yes. A CV is used either with an implicit lock (e.g., the lock of a Monitor) or an explicit lock.
Outline

• Restroom problem
• Bar problem
• Enforcing execution order
Using Semaphores:
The restroom problem and the bar problem
Binary Semaphore for the restroom problem: mutual exclusion

- Given a single-spot restroom, write the function for using the restroom
- Hint: Binary Semaphore, a special semaphore, whose counter value can only be 0 or 1
- Here, Binary Semaphore is used as Mutex, a blocking lock for enforcing MUTual EXclusion

```c
S = 1; // shared among customers (processes)

use_restroom() {
    down(S); // try to enter the restroom; = lock()
    Use the restroom //critical section
    up(S); // leave the restroom;  = unlock()
}
```
Semaphore for the bar problem

- Capacity = 100
- Many customers try to enter the bar concurrently
- Please write code to make sure customers $\leq$ 100
- Caution: a Mutex will not work well; why?

\[
S = 100; \quad // \text{shared among processes}
\]

// each process does the following
down(S); \quad // \text{try to enter the bar}
Have fun;
up(S); \quad // \text{leave the bar}
Using Semaphores: Enforcing orders
Semaphore for enforcing order

• Process 0 and Process 1
• How to make sure statement A in Process 0 gets executed before statement B in Process 1
• Hint: use a semaphore and initialize it as 0

```
S = 0; // shared

// Process 0
A;
up(S);

// Process 1
down(S);
B;
```
Semaphore for enforcing order

- Two processes: p0 and p1
- Hint: use two semaphores

S1 = 0; // shared
S2 = 0; // shared

// Process 0
A1;
up(S1);
down(S2);
A2;

// Process 1
down(S1);
B1;
B2;
up(S2);
Summary

- Restroom problem
- Bar problem
- Enforcing execution order