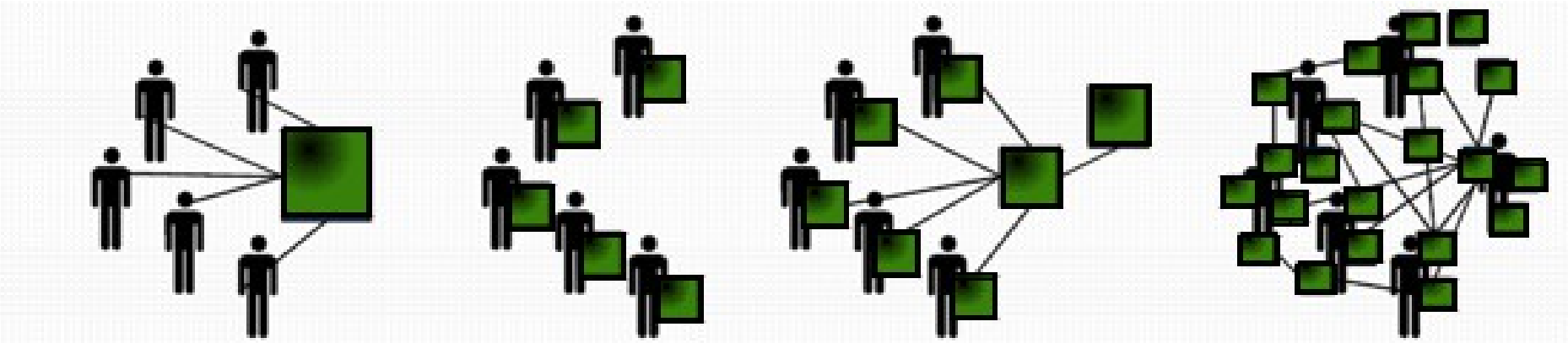
Disciplina
Sistemas de Computação

Aula 13



Semaphores

- What are semaphores?
  - Semaphores are basically generalized locks.
  - Like locks, semaphores are a special type of variable that supports two atomic operations and offers elegant solutions to synchronization problems.
  - They were invented by Dijkstra in 1965.
  - hardware support for synchronization
    - Use interrupts or test&set to ensure atomicity

Semaphores

- Semaphore: an integer variable that can be updated only using two special atomic instructions.
- Binary (or Mutex) Semaphore: (same as a lock)
  - Guarantees mutually exclusive access to a resource (only one process is in the critical section at a time).
  - Can vary from 0 to 1
  - It is initialized to free (value = 1)
- Counting Semaphore:
  - Useful when multiple units of a resource are available
  - The initial count to which the semaphore is initialized is usually the number of resources.
  - A process can acquire access so long as at least one unit of the resource is available

Semaphores: Key Concepts

- Like locks, 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).
- If a process executes S.Wait() and semaphore S is free (non-zero), 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.

Binary Semaphores: Example

Too Much Milk using locks:

Thread A

```
Lock.Acquire();
if (noMilk){
  buy milk;
}
Lock.Release();
```

Thread B

```
Lock.Acquire();
if (noMilk){
     buy milk;
}
Lock.Release();
```

Too Much Milk using semaphores:

Thread A

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

Thread B

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

Signal and Wait: Example

```
P1:  S.Wait();
     S.Wait();
     S.Signal();
     S.Signal();
```

```
P2:  S.Wait();
     S.Signal();
```

```
P1:   S->Wait();
P2:   S->Wait();
P1:   S->Wait();
P2:   S->Signal();
P1:   S->Signal();
P1:   S->Signal();
```

value	Queue	process state: execute or block P1	P2
2	empty	execute	execute
1	empty	execute	execute
0	empty	execute	execute
-1	P1	block	execute
0	empty	execute	execute
1	empty	execute	execute
2	empty	execute	execute

Using Semaphores

- Mutual Exclusion: used to guard critical sections
  - 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.
- Scheduling Constraints: 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:

```
Semaphore S;

S.value = 0; // semaphore initialization

Thread.Join          Thread.Finish
  S.Wait();            S.Signal();
```

Summary

- Locks can be implemented by disabling interrupts or busy waiting
- Semaphores are a generalization of locks
- 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 signaled from another thread.

What's wrong with Semaphores?

- Semaphores are a huge step up from the equivalent load/store implementation, but have the following drawbacks.
  - They are essentially shared global variables.
  - There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
  - Access to semaphores can come from anywhere in a program.
  - They serve two purposes, mutual exclusion and scheduling constraints.
  - There is no control or guarantee of proper usage.
- Solution: use a higher level primitive called monitors

What is a Monitor?

- A monitor is similar to a class that ties the data, operations, and in particular, the synchronization operations all together,
- Unlike classes,
  - monitors guarantee mutual exclusion, i.e., only one thread may execute a given monitor method at a time.
  - monitors require all data to be private.

Monitors: A Formal Definition

- A Monitor defines a lock and zero or more condition variables for managing concurrent access to shared data.
  - The monitor uses the lock to insure that only a single thread is active in the monitor at any instance.
  - The lock also provides mutual exclusion for shared data.
  - Condition variables enable threads to go to sleep inside of critical sections, by releasing their lock at the same time it puts the thread to sleep.

Monitors: A Formal Definition

- Monitor operations:
  - Encapsulates the shared data you want to protect.
  - Acquires the mutex at the start.
  - Operates on the shared data.
  - Temporarily releases the mutex if it can't complete.
  - Reacquires the mutex when it can continue.
  - Releases the mutex at the end.

Condition Variables

- It is a queue of threads waiting for something inside a critical section.
- It enable a thread to sleep inside a critical section
- Any lock held by the thread is atomically released when the thread is put to sleep