Concurrency: a crash course
Concurrent computing

Applications designed as a collection of computational units that may execute in parallel

- logical vs. physical parallelism
- parallel vs. distributed

What’s concurrency good for?

- improved user experience
  - applications carry out several tasks at once
- better usage of resources
  - interactive computing
- performance
  - clusters and multi-core CPUs
Processes and threads

Concurrency can have two levels of granularity, according to what is the unit of parallel computation

- **Processes**
  - the abstraction of a running program
    - includes program counter, registers, variables, ...
  - different processes have independent address spaces

- **Threads**
  - an independent thread of execution within a process
    - a “lightweight process”
  - threads within the same process share the address space

This brief introduction refers to threads, but the same notions apply to processes as well
Coordination of threads

Threads need to **coordinate** when accessing the shared memory to avoid **race conditions**

- inconsistent access to shared resources

```plaintext
-- shared memory
s: shared INTEGER
invariant s ≥ 0 end
```

```plaintext
-- thread A
if s > 0 then
  s := s - 1
end
```

```plaintext
-- thread B
s := 0
```

What happens if B executes just after A has tested the **if** condition (before the decrement)?
Coordination of threads

Coordination must guarantee **mutual exclusion** when accessing shared resources

- a section of code that accesses some shared resource is called **critical region**
- at any given time, **no more than one** thread should be in the critical region

```plaintext
-- A’s crit. reg.        -- B’s crit. reg.
if s > 0 then
    s := s - 1
s := 0
end
```
Coordination mechanisms for shared memory

A few coordination mechanisms, roughly in increasing level of abstraction

We won’t specifically discuss how to use synchronization mechanisms to avoid problems such as deadlocks, starvation, livelocks, etc.

Locks

- a lock is a variable (or an object) that is owned by no more than one thread at a time
- locks can be acquired and released
- guarding with locks the access to critical regions is a way to ensure mutual exclusion
Coordination mechanisms for shared memory

Mutexes

- a way to implement locks
- a mutex is a binary variable accessed with primitives `lock` and `unlock`
  - `lock`: if the mutex is unlocked acquire the lock, otherwise suspend execution
  - `unlock`: release the lock and resume all suspended executions
- the `lock` and `unlock` operations are guaranteed to be non-interruptible
Mutex: example

-- shared memory
s: shared INTEGER
invariant s ≥ 0 end

-- mutex
m: MUTEX

-- thread A
m.lock
if s > 0 then
  s := s - 1
end
m.unlock

-- thread B
m.lock
s := 0
m.unlock
Coordination mechanisms for shared memory

Semaphores

- generalization of mutexes
  - an integer variable that can be atomically incremented (up) and decremented, if its value is positive (down)
- invented by Dijkstra (1965)
- support more complex waiting conditions than mutexes, for example involving multiple resources
Coordination mechanisms for shared memory

Monitors

- a collection of routines (methods) that are guaranteed mutually exclusive access to shared resources
  - no more than one routine in the monitor is active at once
  - in other words: only one thread can be active in a monitor at any instant
- threads within the same monitor coordinate with signals
  - a thread may not be able to proceed because it needs some other thread’s work. Then it can `wait` and yield control to other threads.
  - when a thread performs an action that some other threads may be waiting for it can `signal` it and wake them up (interrupting their waiting)
- invented by Brinch Hansen (1973) and Hoare (1974)
Monitors: example

```
monitor mon

s:  INTEGER
invariant s ≥ 0 end

decrement do
  if s > 0 then s := s - 1 end
end

set_zero do
  s := 0
end
end -- monitor
```