Even in uni-processor system scheduler and interrupts produce concurrent behaviour of processes.

Example:
Consider disk buffer
Used by application program (via system call) and disk driver (via interrupts)
Following scenario possible:

- System call reads data from buffer
- Disk driver writes into buffer

System call and disk driver code may be executed in arbitrary order
⇒ buffer consistency not guaranteed

Next example: Process handling

Have one central list of process control blocks

Following scenario possible:
Scheduler pre-empts process and adds it to ready-queue

Process finishes I/O and is added to ready-queue

⇒ two processes access list of PCB's possibly at the same time

Only one process may modify shared data structures at one time

⇒ Need mutual exclusion:

A piece of code satisfies mutual exclusion if whenever two processes attempt to execute this code, one of them will execute all of it before the other executes any code at all.

Such a piece of code is called a critical section.

Standard means of ensuring mutual exclusion:

- Acquiring a token for access to critical section
- Executing critical section
- Releasing token for access to critical section

Such a token is called a semaphore, which is global variable treated specially. Initial value usually 1 (one process allowed in the critical section at the same time)

Have two operations on semaphores:

- Acquire: test whether semaphore is > 0, and if so, decrease value by 1. If not, block process
- Release: increase semaphore by 1, and wakeup some process blocked in the corresponding acquire-operation.

Important: Test-and-set in acquire must be indivisible

$\Rightarrow$ hardware must help by providing such an operation.

Example: Access to shared buffer

Distinction between readers and writers Common situation: have common buffer with several processes reading or writing it

Various subcases depending on whether readers or writers get priority

Consider here the case where writer waits for all readers to finish

Introduce two semaphores:

- writers: initially set to 1, decreased whenever a writing is in progress
- readers: initially set to 0, increased whenever a reading is in progress

Have different procedures for reader and writer:

- readAcquire: test whether writers $= 1$. If not, block process. If yes, increase readers.
- writeAcquire: test whether writers $= 1$ and readers $= 0$. If one of these conditions is false, block process. If both conditions are true, decrease writers.
- readRelease: decrease readers.
- writeRelease: increase writers.

Again, test-and-set in acquire must be indivisible.

Deadlocks

Need to be careful that two process don't wait for each other

Classical example of those problems:

Dining philosophers:

Philosophers sit around a round table, with a fork between each of them

Require both forks to eat

Even if we model forks as semaphores, can have deadlock:

Each philosopher picks left fork.

Now no philosopher can pick right fork.

$\Rightarrow$ no-one can eat.

Two possible solutions:

First one: introduce randomness:

Each philosopher picks first fork at random. If other fork not available, drop fork and try again after a random interval.

Second one: introduce priorities:

Assign number to each fork. Each philosopher obtains fork with lowest number first. and waits for the other one to become available.

First solution treats all philosophers equally, whereas second solution gives priority to philosopher with forks 1 and n.