Mutual exclusion

Fundamental problem in concurrent programming:

- Ensure co-ordinated access to shared resources
- Ensure that a process is not interrupted by its peers during a critical section.

Examples:

- Avoid data races (two processes reading and incrementing a value)
- Logical reasons (e.g., device can only execute one job at a time)
- Time-critical operations (communication with hardware)
- Complex transactions (multiple changes must be made at once to maintain consistency)

Critical section

Abstract model:

A collection of processes, each with a critical section that is accessed from time to time.

Requirement: No more than one process may be in a critical section at any given moment.

Variant: No more than k processes may enter a critical section.

Processes are distinguished by some identifier.

Environment

Mutual exclusion happens in different environments:

- inter-thread/process communication
- single-core vs multi-core
- processes running on different computers

Many algorithms and mechanisms and mechanisms exists to deal with different assumptions:

- Interleaved vs true concurrency
- Which actions are atomic (uninterruptible)?
- Means of communication: synchronous, asynchronous, bounded delay

Mechanisms

Some means of implementing mutual exclusion *on a single machine*:

Shared memory (synchronous). Problem: atomicity, data races

Inside the CPU: interrupt masks (but only kernel may use it)

Software solutions: Semaphores (or: Flags/Monitors/Locks)

Semaphores

A semaphore is a data structure with the following atomic operations:

Init(n), where n = number of allowable concurrent accesses;
inits counter to n

Wait: if counter positive, decrease it and return;
otherwise wait until it becomes positive

Post: increase the counter

Example

Typical use case for critical sections with semaphore:

```
                Init(1);

while (1) {              while (1) {
    ...;                     ...;
    Wait();                  Wait();
    Critical1();             Critical2();
    Signal();                Post();
    ...;                     ...;
}                        }
```

Every access to a critical section is surrounded by Wait and Post.

Semaphores in Unix

Semaphore support at the OS level, see `sem_overview`(7):

SystemV semaphores: `semget` etc, older interface, no longer recommended

Posix locks (demo)

Unnamed semaphores (between related threads/processes):

`sem_init`, `sem_wait`, `sem_post`

Named semaphores (system-wide):

`sem_open`, `sem_unlink`

Implementing a Semaphore

Naïve:

```
Init(n) { ctr = n; }

Wait() { while (ctr == 0); ctr = ctr-1; }

Signal() { ctr = ctr+1; }
```

Two problems:

Atomicity: no interruption must occur between reading and updating the value of `ctr`!

Waiting: active waiting or block/wakeup?

For POSIX semaphores, atomicity is ensured by blocking interrupts during Wait and Post (not available to normal user code!).

Passive waiting: block process until counter reaches non-zero state

- requires OS-level support (for re-activating the process)
- liberates CPU for other tasks (including doing nothing and saving energy)
- but: necessitates at least two (costly) context switches
- solution of choice for long waits, or on single-core CPU

Spinlocks

So-called spinlocks use active waiting: periodically (or continuously) query counter state

- burns CPU time; blocks other processes from executing

- effective in true concurrency setting (e.g. multi-core CPU) if wait is guaranteed to be (very) short.

See also: `pthread_spin_lock`

Producer-Consumer Problem

Two processes, a producer (left) and a consumer (right).

The consumer uses up what the producer creates (data, requests, . . .).

`put` and `get` are used to insert objects into a shared buffer of size N.

Solution I: Active waiting

Init: counter = 0;

```
while (1) {
      produce(&object);
      while (counter == N);
      put(object);
      counter = counter+1;
   }
```

```
while (1) {
      while (counter == 0);
      counter = counter-1;
      get(&object);
      consume(object);
   }
```

Note that this solution also requires atomic increment/decrement on counter.

Producer/consumer using semaphors

Init(empty,N); Init(full,0);

```
while (1) {
    produce(&object);
    Wait(empty);
    put(object);
    Signal(full);
}
```

```
while (1) {
    Wait(full);
    get(&object);
    Signal(empty);
    consume(object);
}
```

Two sempaphores necessary to deal with both ends of the range $0..N$.