## Systems programming

A thorough knowledge of OS fundamentals is needed for Systems Programming, i.e. programming focused on:

- Communication between processes (and between processes and the OS) known as Interprocess Communication (**IPC**). This includes special topics such as threads.
- File input/output.
- Network communication (a form of **IPC** using network protocols.
- Communication with slow devices.

# Interprocess Communication

The most important part of Systems Programming is **IPC**.

Three aspects of it are relevant:

1. Information exchange between processes.
2. Sharing without the possibility of (accidental/malicious) damage.
3. Control the sequence of actions performed by several processes.

The last two aspects form one topic sometimes called **process synchronisation**.

## Information exchange

Processes share information by using:

**Files:** the oldest medium for **IPC**.

**Shared memory:** a block of main memory is part of the address spaces of two (or more) processes. Information put into this block by one process is visible by the other.

**Message passing:** a sequence of messages flow from one process to another directly (as a pipe) or via a mailbox (makes message passing by 3 or more processes possible).

# Synchronisation

When two entities coexist (live concurrently), they may be controlled by the same timing mechanism or they may have separate timing mechanisms that are independent of each other.

If they share the same timing mechanism, they are **synchronous** because they perform everything according to the same timing order.

If they have separate timing mechanisms, they are **asynchronous** because their timing is not coordinated.

Note that asynchronous entities may choose to coordinate their actions with other asynchronous entities; this act is called **synchronisation** (only asynchronous entities need to synchronise; synchronous entities are always synchronised by definition).

Consider two types of cars:

- Freight cars forming a train (railroad). These cars are synchronous with the timing provided by the train engine (whenever it moves, they all move).
- Motor cars (highway). Each car moves separately and (to some extent) independently of other cars. Cars are synchronised from time to time by external events such as red lights. Asynchronous cars may choose to adopt long–term synchronisation schemes, as in a truck convoy.

# Processes are asynchronous

Processes run concurrently and are independent of other processes; they are asynchronous entities.

They cooperate, however, and that requires that they synchronise their actions with the actions of other processes.

There are two basic kinds of synchronisation:

**Temporal** synchronisation when two actions must be performed in a preset order (first then second).

**Spatial** synchronisation when two actions cannot be performed on the same resource ("space") at the same time, although these actions can be performed simultaneously on different resources.

Parking in a single–booth prepaid parking lot requires both types of synchronisation: a car must pay before parking (temporal). As there is a single booth, two cars cannot pay at the same time (spatial and not temporal because two cars can pay in any order), nor can they drive into the same stall at the same time (also spatial).

## When to synchronise

Two processes must coordinate their activities when they are about to access shared resources (e.g. cars + intersection = lights).

In the programming domain, synchronisation is needed when a process needs to alter some common data or when a process wants to inspect some common data:

**Alter:** no other process should be allowed to access the shared data when it is being altered in a non–atomic way (write+write is obvious; read+write can be harmful, too).

**Inspect:** two processes can inspect the same shared data concurrently. However, they can only be allowed to inspect existing shared data.

# Race condition

When two (or more) processes fail to follow a correct protocol for accessing shared data, they may cause a number of problems:

**Deadlock:** they may all end up waiting for something.

**Race:** they may all execute to the end but the result depends on their relative speed.

**Occasional deadlock:** is a variant of a race in which the processes sometimes end up waiting or finish, again, depending on their relative speed.

These are serious errors and must be prevented. The worst situation is when the outcome of the actions of the processes depends on timing (**race condition**); in this case, the result may be correct sometimes (even almost always) but incorrect in some circumstances. Careless testing may leave a race condition undetected.

## Critical section

When a process executes code that leads to a race condition, the process is said to be in a **critical section** (also called critical region). Synchronisation with other processes is required before entering a critical section.

The basic idea is to make sure that when a process is in a critical section related to a shared variable $\mathcal{V}$, all other processes must be prevented from entering their critical sections related to the same variable $\mathcal{V}$. This is called **mutual exclusion**.

Note that other processes can still enter their critical sections related to other shared variables.

## Mutual exclusion

Mutual exclusion has several different forms depending on what the cooperating processes do inside their critical sections.

The standard scenario is the readers–writers cooperation when a number of processes create shared data (write) while another set of processes inspect the data read). Each writer must enforce strict mutual exclusion but the readers can, in some situations, be allowed to access the shared data concurrently, provided no writer is trying to alter it.

## Date and time

A system utility called date returns the current date and time. It can also be used to change the current time and date.

Any number of users can execute date concurrently to check the current time. However, they will all have to be blocked if another process is changing the current time.

Conversely, if a process is reading the current time, any process that wants to change it must wait.

## Mutual exclusion

Many bad implementations of mutual exclusion are in use. In order to provide a quality standard the following 4 properties of mutual exclusion are **required**:

1. No two processes may be concurrently inside their critical sections.
2. No assumptions may be made about the speeds or the number of CPUs.
3. No process running outside its critical section may prevent other processes from entering their critical sections.
4. No process can be forced to wait forever trying to enter its critical section.

Requirement 1 relates to mutual exclusion protecting one shared variable.

## Busy wait

While the mutual exclusion rules say nothing about efficiency, the programmer's code of honour adds another requirement:

**Never enter a busy wait!**

This implies that your implementation of mutual exclusion must give the CPU away whenever you have to wait.

There is a notable exception to the ban on busy waiting: on multi–core machines it is useful to enter a busy wait waiting for a lock (spinning for a spinlock) hoping that another processor will release that lock very soon (thus avoiding a context switch).

## Implementing mutual exclusion

Mutual exclusion turns a critical section into an atomic statement which cannot be interrupted before completion.

On single–processor machines, this can be enforced by disabling interrupts for the execution of a critical section. This approach is commonly used inside the kernel but never at the level of user processes.

At the user process level, locks and flags are used. They have a fundamental weakness: obeying them is voluntary and a misbehaving (or malicious) process can destroy any user–level synchronisation scheme. Mandatory locks solve nothing: a malicious process can put a mandatory lock on a shared resource and get into an infinite loop, eventually blocking all the other processes.

## Moral

Correct synchronisation requires that all the processes are well–behaved and are cooperating, i.e. following the rules.

Correct synchronisation requires careful design; it is the most difficult part of software development.

## Avoid the pitfalls

While the main priority of **IPC** programming is correctness, there are many potential problems that must be avoided:

**Deadlock:** a group of at least two processes that are waiting for resources held by other processes belonging to this group. Example: $P_1$ holds $r_1$ and waits for $r_2$; $P_2$ holds $r_2$ and waits for $r_3$; $P_3$ holds $r_3$ and waits for $r_1$.

**Livelock:** A process does something all the time but makes no headway (remain equally far from completion).

**Starvation:** a process is denied the right to use a resource, even though that resource is not continuously held.

**Unfairness:** the synchronisation protocol gives an advantage to some processes at the expense of others. It depends on how fairness is defined.

Unfairness may be tolerated in some applications, the other problems must be avoid.

## Files and IPC

A file has two characteristic attributes:

- It has contents that can be written and read by two (or more) processes.
- Its name is a unique entry in the file system. This can be used: the presence or absence of a file can act as a flag indicating that a process is in its critical section.

No special functions are needed, although numerous utilities are available (especially for locking parts of a file).

Many systems offer the option of opening a file for exclusive access, which provides mutual exclusion for the file as a shared variable.

## Shared memory

The most common method to communicate with another process is through a shared block of memory. This is particularly convenient for threads which share memory by definition.

Shared memory requires a synchronisation tool and semaphores are the tool used most often.

# Semaphores

A semaphore is a protected variable that acts as a lock. It is protected in the sense that it can only be accessed through trusted system calls.

The original concept, due to E.W. Dijkstra, consisted of 2 system calls:

**V:**
```
void V( sem S ) // also called signal
{
   S++ ;
}
```

**P:**
```
void P( sem S ) // also called wait
{
   while( S <= 0 ) ;   // spin
   S-- ;
}
```

The semaphore is assumed to be initialised to 1 (binary semaphore).

The $\mathcal{P}$ operation provides mutual exclusion as only one process can successfully execute it before a $\mathcal{V}$ is executed.

A simple use of a semaphore to provide mutual exclusion for a critical section (accessing shared memory):

```
P( 0 ) ;
shmptr⟶confirmation = 2024561111 ;
shmptr⟶code = 2 ;
V( 0 ) ;
```

Here, 0 is the semaphore number in the cluster acquired by semget; its starting value must be 1.

## Counting semaphores

When several units of the shared resource are available, it is convenient to use **counting semaphores**. They can take any value (positive or negative). When the value of a counting semaphore equals **s**, the interpretation is:

$s > 0$**:** $s$ units of the shared resource are available.

$s < 0$**:** no units are available and $-s$ processes are waiting for the resource.

$0$**:** no units are available and no process is waiting.

## Producer–Consumer

One producer process continuously fills empty buffers while one consumer process continuously empties full buffers.

The total number of buffers is *n* and they all start empty.

A classic solution uses one binary semaphore (mutex) and two counting semaphores (full and empty).

## Producer

```
shared semaphore mutex , full , empty ;
shared buffer B[n] ;
private buffer T ;
full = 0 ;
empty = n ;
mutex = 1 ;

while( 1 ) {
    fill buffer T ;
    P( empty ) ;
    P( mutex ) ;

    ... copy T to an empty buffer ...

    V( mutex ) ;
    V( full ) ;
}
```

## Consumer

```
shared semaphore mutex , full , empty ;
shared buffer B[n] ;
private buffer T ;
while( 1 ) {
    P( full ) ;
    P( mutex ) ;

    ... copy a full buffer to T ...

    V( mutex ) ;
    V( empty ) ;

    ... process T ...
}
```