- **Multiprogramming:** The management of multiple processes within a uniprocessor system
- **Multiprocessing:** The management of multiple processes within a multiprocessor
- **Distributed processing:** The management of multiple processes executing on multiple, distributed computer systems.
    - The recent proliferation of clusters is a prime example of this type of system.

Issues:

- Communication among processes
- Sharing of and competing for resources
- Synchronization activities of multiple processes
- Allocation of processor time to processes

Contexts:

1. Multiple applications:
    - Multiprogramming to allow processing time to be dynamically shared among active applications
2. Structured applications: extension for modular design and structured programming —> concurrent processes
3. Operating system structure: same structuring advantages apply to systems programs (os implemented as set of processes and threads)

Atomic operation	- Sequence of instructions guaranteed to execute as a group or not at all. - guarantees isolation from concurrent processes
Critical section	Code section within a process that requires access to shared resources (must not be executed while others in a corresponding section)
Deadlock	Two or more processes waiting on another; unable to proceed
Livelock	Two or more processes changing states in response to one another; no useful work done
Mutual exclusion	The requirement of only one process being in a critical section that accesses shared resources
Race condition	Multiple threads or processes read/write a shared data item, where the final output depend on their relative timing of execution
Starvation	A runnable process that can proceed is being overlooked by the scheduler

Mutual Exclusion: Software Approaches

Can be implemented for concurrent processes that execute on:

- a single processor
- multiprocessor with shared main memory

Dekker Algorithm

First Attempt

Global memory location -> Turn

Check turn if it's equal the process value: Process can enter critical section

Process 0	Process 1
`. while (Turn !=0): //nothing //Critical section . . Turn = 1;`	`. while (Turn !=1): //nothing //Critical section . . Turn = 0;`

Two Drawbacks:

1. Strict alternation in the use of critical section between processes (pace of execution dictated by the slower of them)
2. If one of them fails in any part of the code -> the other is permanently blocked

Second Attempt

- we need state information about both processes so if one fails the other can access the critical section

Boolean vector: Flag (false: not in the critical section), where:

- Flag[0] -> process 0
- Flag[1] -> process 1

Each process can examine the other's flag but can't alter it

Process 0	Process 1
`. while (flag[1] is true): //nothing flag[0] = true; //Critical section flag[0] = false; . .`	`. while (flag[0] is true): //nothing flag[1] = true; //Critical section flag[1] = false; . .`

Deadlock: If process fails

- after setting flag true just before entering critical section
- inside its critical section the other is permanently blocked

No mutual exclusion

- P0 finds flag[1] is false
- P1 finds flag[0] is false
- P0 sets flag[0] to true, enters critical section
- P1 sets flag[1] to true, enters critical section

The solution isn't independent of relative process execution speeds

Third Attempt

Process 0	Process 1
`. flag[0] = true; while (flag[1] is true): //nothing //Critical section flag[0] = false; . .`	`. flag[1] = true; while (flag[0] is true): //nothing //Critical section flag[1] = false; . .`

- **Deadlock:** Failing before or during critical section permanently blocks the other
- Guarantees mutual exclusion
- **New Problem of Deadlock:** if flag[0] and flag[1] both got set to true before any of the processes executes the while statement both will think the other is in the critical section

Fourth Attempt

```
.
flag[0] = true;
while (flag[1] is true):
       flag[0] = false;
       //delay
       flag[0] = true;
//Critical section
flag[0] = false;
.
.
```

```
.
flag[1] = true;
while (flag[0] is true):
       flag[1] = false;
       //delay
       flag[1] = true;
//Critical section
flag[1] = false;
.
.
```

Livelock problem

Process 0	Process 1
-set flag[0] = true -is flag[1] true? Yes -set flag[0] = false -set flag[0] = true . .	-flag[1] = true -is flag[0] true? Yes -flag[1] = false -set flag[0] = true . .

Dekker Algorithm: Correct Solution

```
boolean flag[2];
int turn;
```

Process 0 while true:

```
flag[0] = true;
      while (flag[1])
            if (turn == 1)
                  flag[0] = false;
                  while (turn == 1)
                        //do
                        nothing
                  flag[0] = true;
//critical section
turn = 1;
flag[0] = false;
.
.
```

Process 1 while true:

```
flag[1] = true;
while (flag[0])
      If(turn == 0)
            flag[1] = false;
            while(turn == 0)
                  //do nothing
            flag[1] = true;
//critical section
turn = 0;
flag[1] = false;
.
.
```

Main

```
flag[0] = false;
flag[1] = false;
turn = 1;
parbegin(P0, P1);
```

Peterson Algorithm

```
boolean flag[2];
int turn;
```

Process 0 while true:

```
flag[0] = true;
turn = 1;
while(flag[1] && turn ==1)
      //do nothing
//critical section
flag[0] = false;
.
.
```

Process 1 while true:

```
flag[1] = true;
turn = 0;
while(flag[0] && turn ==0)
      //do nothing
//critical section
flag[1] = false;
.
.
```

Main

```
flag[0] = false;
flag[1] = false;
turn = 1;
parbegin(P0, P1);
```

Principles of Concurrency

- Interleaving and overlapping
  - Can be viewed as examples of concurrent processing
  - Both present the same problems
- Uniprocessor – the relative speed of execution of processes cannot be predicted, depends on:
  - Activities of other processes
  - The way the OS handles interrupts
  - Scheduling policies of the OS

Difficulties of Concurrency

- Sharing of global resources
- Managing allocation of resources optimally
- Locating programming errors (results aren't deterministic nor reproducible)

Problems when Sharing

In the case of a **uniprocessor** system

- an interrupt can stop instruction execution anywhere in a process.

In the case of a **multiprocessor** system

- The same condition
- Two processes may be executing simultaneously and both trying to access the same global variable.

The solution to both types of problem is the same: control access to the shared resource.

(Data) Race Condition

Occurs when multiple processes or threads read/write data items

- The final result depends on the order of execution
- The “loser” of the race -> the process that updates last
  - It determines the final value of the variable
- Nondeterministic output.

Resource Race Condition

When concurrent processes are competing for use of the same resource

- Ex: I/O devices, memory, processor time, clock

Problems must be faced:

1. Need for mutual exclusion
2. Deadlock
3. Starvation

Degree of Awareness	Relationship	Influence of one Process on another	Potential Control Problems
Processes unaware of each other	Competition	- Results of one process are independent of the action of others - Timing of process may be affected	- Mutual exclusion - Deadlock (renewable resource) - Starvation
Processes indirectly aware of each other (ex: shared object)	Cooperate by sharing	- Results of one process may depend on information obtained from others - Timing of process may be affected	- Mutual exclusion - Deadlock (renewable resource) - Starvation - Data coherence: mutual exclusion -> only for writing operations
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	- Results of one process may depend on information obtained from others - Timing of process may be affected	- Deadlock (renewable resource) - Starvation (No mutual exclusion requirement)

Requirements for Mutual Exclusion

1. **Only one process at a time in a critical section** among all alike critical sections
2. **No interferring with others if a process halts**
3. **No deadlock or Starvation:** not be possible for a process requiring access to a critical section to be delayed indefinitely
4. **No delay for entering a critical section** if there is no other process in a similar critical section
5. **No assumptions about relative process speeds or number of processes**
6. **Finite time inside critical section**

Mutual Exclusion: Hardware Approaches

Interrupt Disabling

In a uniprocessor system, concurrent processes cannot have overlapped execution; they can only be interleaved

To guarantee mutual exclusion: **prevent a process from being interrupted** is sufficient

- This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts
- Disadvantages:
  - The efficiency of execution could be noticeably degraded -> limited ability to interleave processes
  - This approach will not work in a multiprocessor architecture

```
while (true) {
        /* disable interrupts */;
        /* critical section */;
        /* enable interrupts */;
        /* remainder */;
}
```

Special Machine Instructions

In a multiprocessor configuration: there is no interrupt mechanism between processors on which mutual exclusion can be based.

Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory

- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable

Disadvantages

- **Busy-waiting is employed**
  - While a process is waiting for access to a critical section it continues to consume processor time
- **Starvation is possible**
  - When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary; some process could indefinitely be denied access
- **Deadlock is possible**

Compare & Swap(Exchange) Instruction

```
int compare_and_swap (int *word, int testval, int newval) {
    int oldval;
    oldval = *word
    if (oldval == testval)  *word = newval;
    return oldval;
}
```

Program

```
const int n = /* number of processes */;
int bolt;
void P (int i) {
    while (true) {
          while (compare_and_swap(bolt, 0, 1) == 1)
              /* do nothing */;
               /* critical section */;
               bolt = 0;
               /* remainder */;
     }
}
void main() {
    bolt = 0;
    parbegin (P(1), P(2), ... ,P(n));
}
```

Exchange Instruction

```
void exchange (int *register, int *memory) {
    int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
}
```

Program

```
int const n = /* number of processes */;
int bolt;
void P (int i) {
     while (true) {
          int keyi = 1;
          do exchange (&keyi, &bolt)
          while (keyi != 0);
          /* critical section */;
          bolt = 0;
          /* remainder */;
     }
}
void main() {
     bolt = 0;
     parbegin (P(1), P(2), ..., P(n));
}
```

- Each process has a local variable *key* —> initialized to 1.
- Only process enter its critical section: that finds keyi = 0 after exchanging with *bolt* = 0.
- It excludes all other processes from the critical section by setting *bolt* = 1.
- When a process leaves its critical section, it resets *bolt* to 0, allowing another process to gain access to its critical section.