

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
```

```
{
    int freq[MAXPAROLA]; /* vettore di contatori
    delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE * f;
```

```
    for(i=0; i<MAXPAROLA; i++)
        freq[i]=0;
```

```
    if(argc != 2)
```

```
    {
        fprintf(stderr, "ERRORE: serve un parametro con il nome del file\n");
        exit(1);
    }
```

```
    f = fopen(argv[1], "rt");
    if(f==NULL)
```

```
    {
        fprintf(stderr, "ERRORE: impossibile aprire il file %s\n", argv[1]);
        exit(1);
    }
```

```
    while( fgets( riga, MAXRIGA, f ) != NULL )
```

## Processes

# Processes and concurrency

Stefano Quer and Pietro Laface

Dipartimento di Automatica e Informatica

Politecnico di Torino

## Program

- ❖ **Algorithm:** a logical procedure that in a finite number of steps solves a problem
- ❖ **Program:** formal expression of an algorithm by means of a programming language
  - Static entity
  - Sequence of code lines

## Process

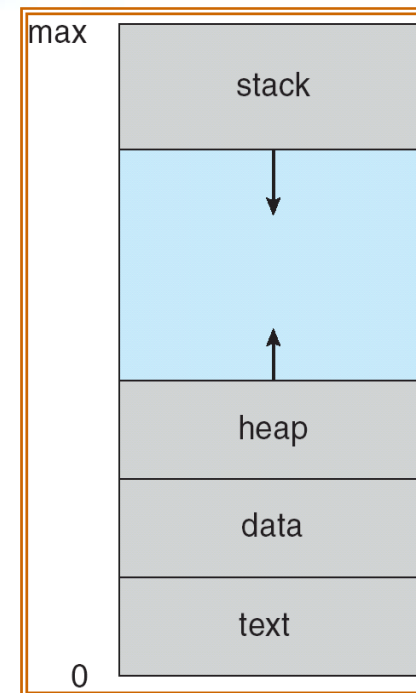
- ❖ **Process**: a sequence of operations performed by a program in execution on a given set of input data.
- ❖ The temporal behavior of a process can be analyzed through its **trace**
  - Program in execution
  - Dynamic entity

trace execute like debug

Program counter	Stack pointer	Register A	Register B	Variable X	Variable Y
0	0x1234	0	0	0	0
4	0x1234	-10	0	0	0
8	0x1234	-10	0	-10	0

# Process

- Text area (source code) **executable code**
- Data area (global variables)
- Stack (function parameters and local variables)
- Heap (dynamic variables allocated during the process execution)
- Registers (Program counter, stack pointer, etc.)

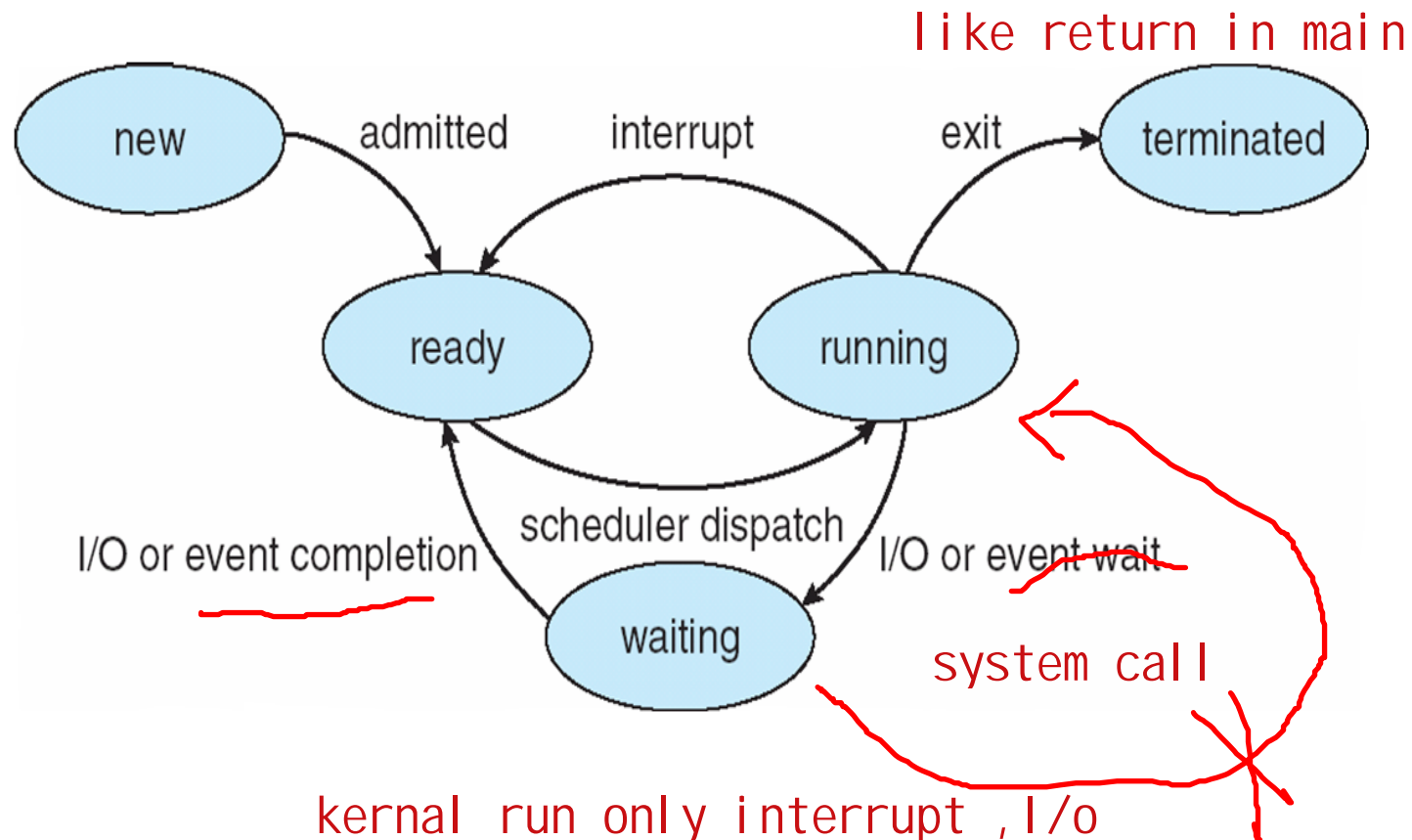
**local var****malloc****total memory**

## Process state

- ❖ During its execution a process change its state
  - **New:** process is created and submitted to the OS ask for res
  - **Running:** a CPU is allocated to the process
  - **Ready:** logically ready to run, waiting that a CPU is available
  - **Waiting:** for an event or for a resource
  - **Terminated:** releases the resource it is using

## State diagram

- ❖ The possible state evolution of a process is described by a state diagram



## Process Control Block (PCB)

- ❖ The kernel stores for each process a set of data, e.g.,
- The process state
    - New, Ready, Running, Waiting, Terminated
  - Copy of the CPU registers
    - Their number and type is hardware-dependent
  - The program counter
    - Address of the next instruction to be executed

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	

where it can access

snapshot of hardware situation of trace  
for restart the process

## Process Control Block (PCB)

- Data useful for CPU scheduling
  - Priority, pointers to queues, etc.
- Data useful for memory management
  - Segment and paging registers, segment and page tables, etc.
- File table
  - open files
- Signal table
  - signal handlers
- Etc.

link to some state list

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	



## Process Control Block (PCB)

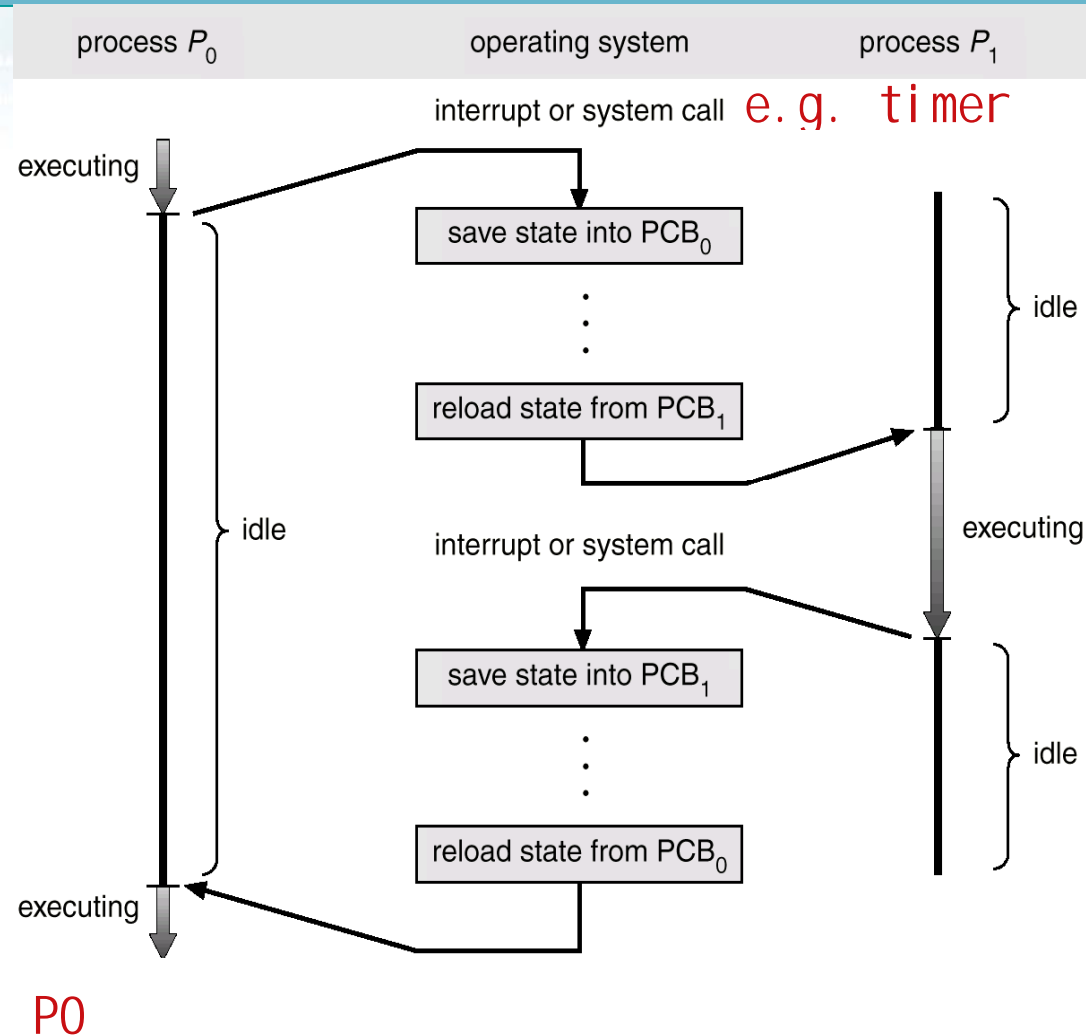
- Various administration data
  - CPU usage, limits, etc.
- I/O status information
  - I/O device list, etc.

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	

## Context switching

- ❖ When the CPU is assigned to another process, the kernel
  - Save the state of the running process **snapshot**
  - Load the state previously saved for the new process
- ❖ The time devoted to this **context switching** is overhead, i.e., time not directly useful for any process
- ❖ The amount of time for context switching is hardware-dependent **application independent**

# Context switching



## Process scheduling

- ❖ Multiprogramming aims at maximizing the CPU usage by processes
- ❖ Processes can be classified as
  - I/O-bound
    - Spend more time for I/O than for computation
    - Require short CPU service times
  - CPU-bound
    - Spend more time for computation than for I/O
    - Require long CPU service times

## Process scheduling

- ❖ The kernel manages the sharing of the CPU among processes by means of a **scheduler**
  - A scheduler **selects the next process to run**, among the ready ones, according to a strategy that tries to maximize the CPU usage and to satisfy the response time for users

## Process scheduling

### ❖ Different types of schedulers

#### ➤ Short-term scheduler

- Selects the next process to run (context-switching)
- Run frequently
- Rescheduling performed every 1 to 10 milliseconds
- Must be extremely fast

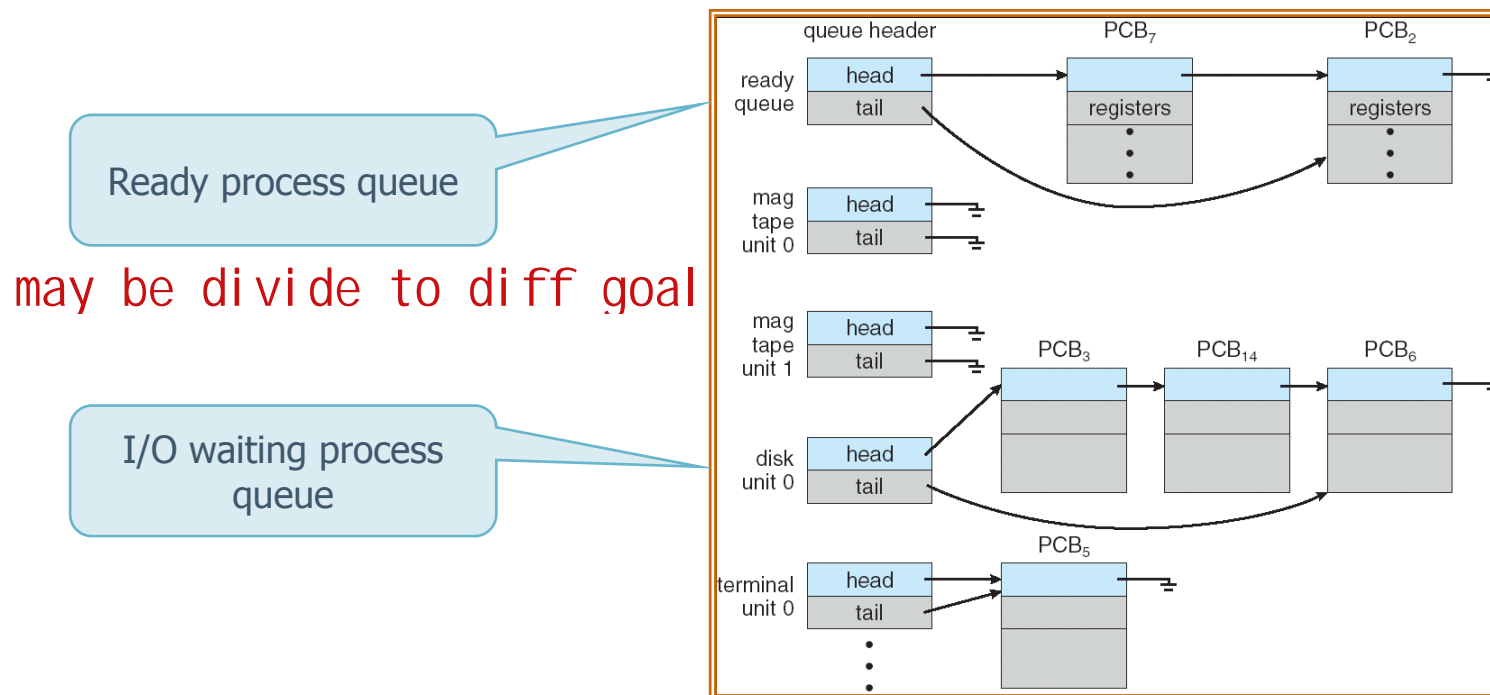
## Process scheduling

### ➤ Long-term scheduler

- Run less frequently
- Rescheduling time in the order of seconds/minutes
- Selects which process image can be loaded in main memory (swapper) **high priority**
- Controls multiprogramming to avoid trashing
  - Too many processes that conflicts for the use of limited resources

# Process scheduling

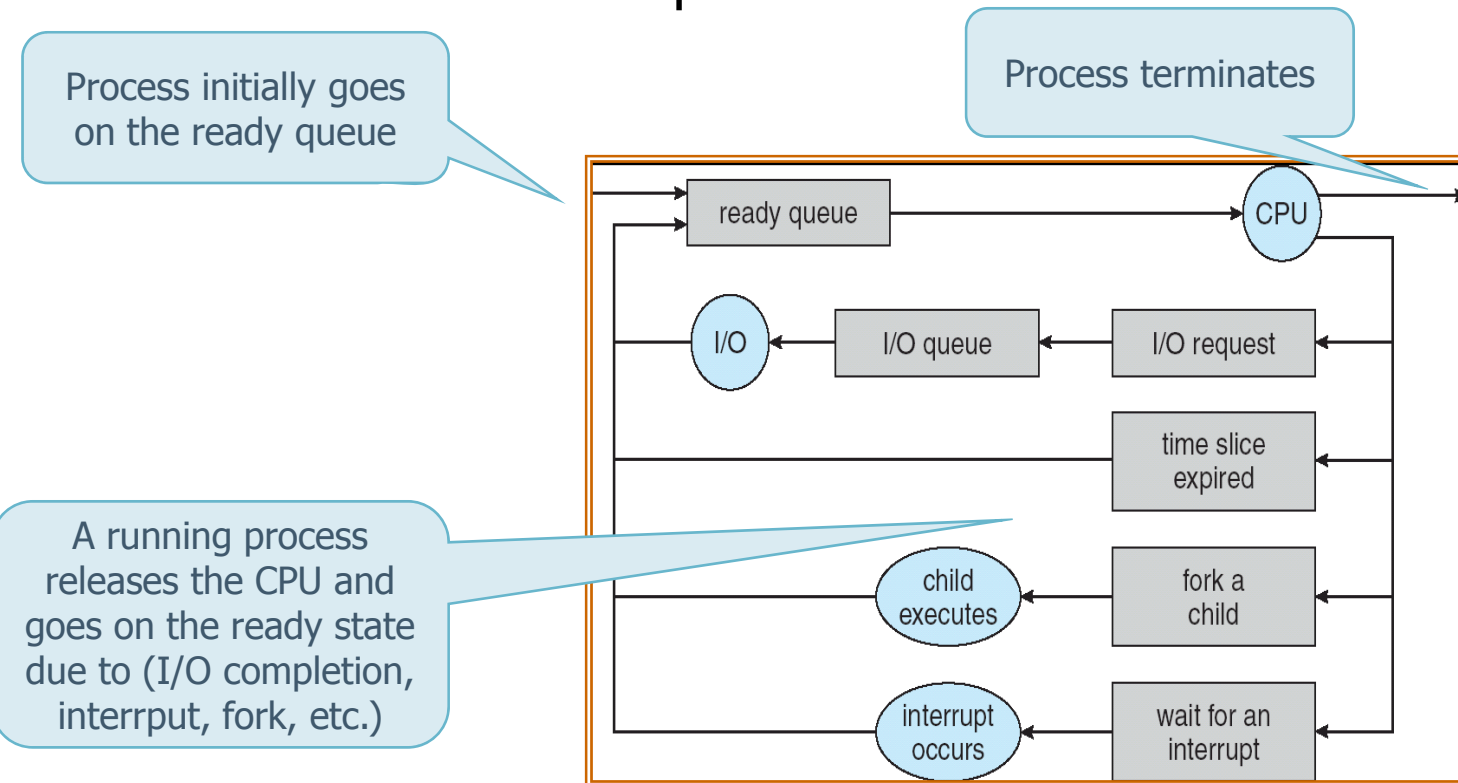
- ❖ A scheduler manages **process queues**





## Queuing diagram

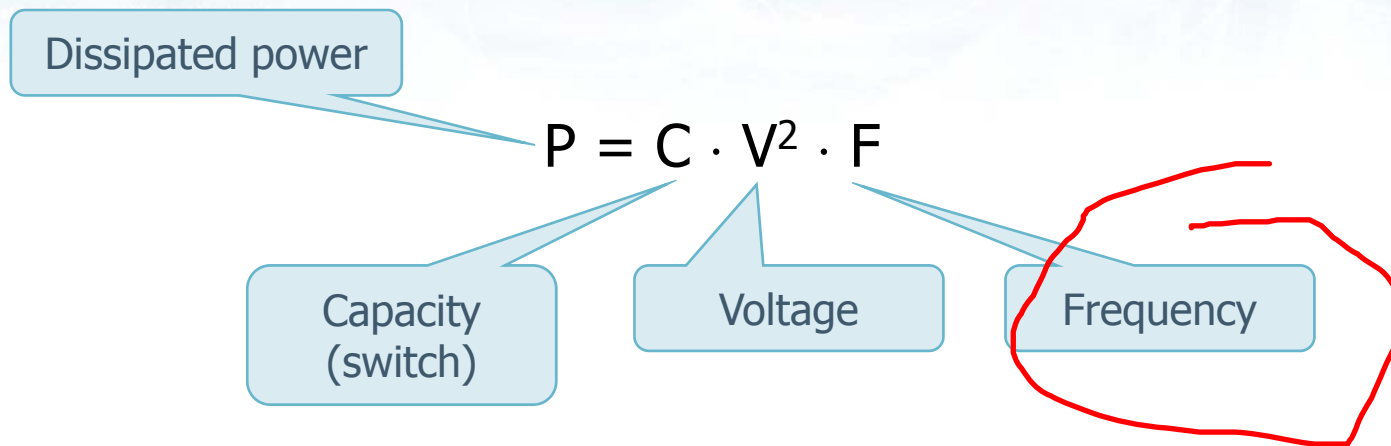
- ❖ The queuing diagram shows the possible process transitions from one queue to another one



## Concurrency

- ❖ **Parallel computing** is a type of computation in which many tasks are carried out simultaneously.
  - Large problems can often be divided into smaller ones, which can then be solved at the same time
- ❖ **Concurrency**
  - Has been used since long time
  - Large development in the last years due to upper limits on the CPU frequency, and power consumption

# Concurrency



- ❖ Due to the limits of frequency scaling
  - Parallel computing is now one of the main programming paradigms
  - Concurrent programming has introduced new challenges and pitfalls (bugs)

## Concurrency

### ❖ Parallel computation levels

#### ➤ Bit-level

- Word length determines the efficiency of an instruction (e.g., 8 bit versus 16 bit adder)

#### ➤ Instruction-level

- Use of multi-stage pipelines for the execution of an instruction flow (e.g., fetch, decode, execute)

#### ➤ Task-level

- Different computations are executed in parallel (e.g., sorting and matrix product)

## Speed-up

- ❖ Concurrency aims at increasing the usage of the CPUs
  - Only multi-processor or multi-core systems allow obtaining real concurrency.
- ❖ Different levels of concurrency
  - Computer-cluster
  - Multi-processor
  - Multi-core

## Speed-up

❖ **In theory** concurrency should allow linear speed-up

➤ Doubling the number of computation units, the execution time should be halved

❖ This behavior is obtained

- Rarely

- If a process is intrinsically sequential, augmenting the processing units does not change its execution time.

- For a limited number of processors/cores

- The speed-up curve grows linearly initially, but then goes towards an horizontal asymptote

difficult

# Speed-up

## ❖ Amdahl law [1967]

- Small program segments intrinsically sequential limit the total speed-up that can be obtained

$$\lim_{P \rightarrow \infty} \frac{1}{\frac{1-\alpha}{P} + \alpha} = \frac{1}{\alpha}$$

#Processors

% sequential  
run-time

$\alpha = 10\% = 0.1$

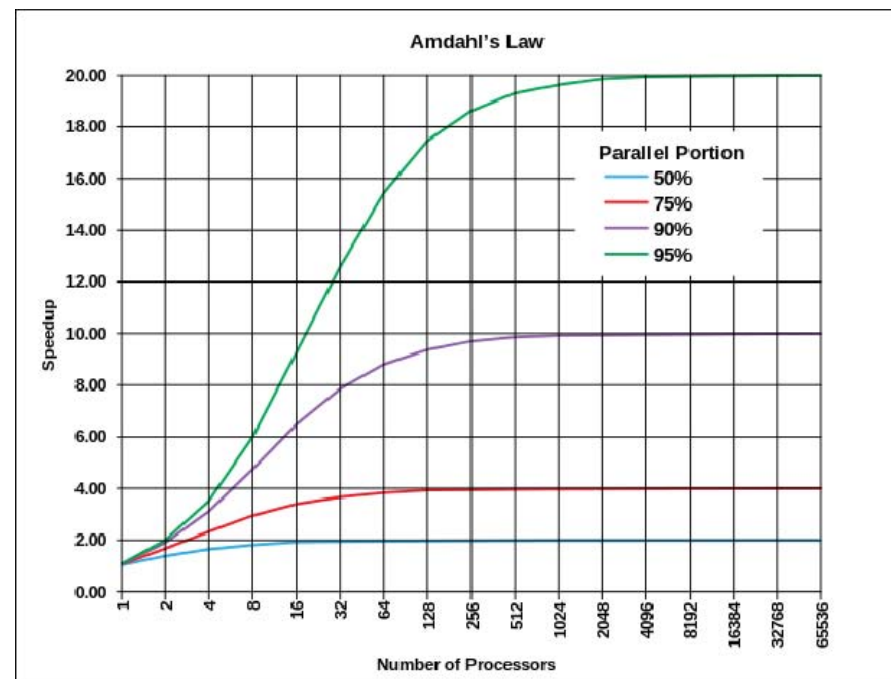
→

10x speed-up max

# Speed-up

## ❖ Amdahl law [1967]

- Small program segments intrinsically sequential limit the total speed-up that can be obtained



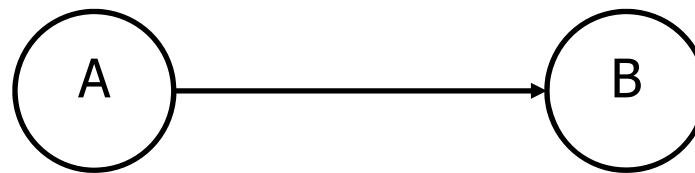


## Parallel implementation

- ❖ An algorithm can be parallelized only respecting all its dependencies
  - Precedence constraints among operations (i.e., instructions, instruction blocks, processes)
  - A program cannot be executed faster than its slower sequence of operations (**critical path**)
- ❖ Precedence constraints can be represented by means of **precedence graphs**
  - Relation with Control Flow Graph and Process generation trees

## Precedence graphs

- ❖ A precedence graph is an acyclic direct graph with
  - Nodes corresponding to instructions, instruction blocks, processes
  - Arcs represent precedence conditions
    - An arc from node A to node B means that B can be executed only when A is completed



## Sequential and concurrent processes

### ❖ Sequential execution

#### ➤ Actions are executed one **after** the other

- A new action begins only after the termination of the previous one

#### ➤ Deterministic behavior

- Given the same input, the output produced is always the same, it does not depend on
  - The time of execution
  - The speed of execution
  - The number of active processes on the same system

Sequential actions

Input,  
Processing  
Output

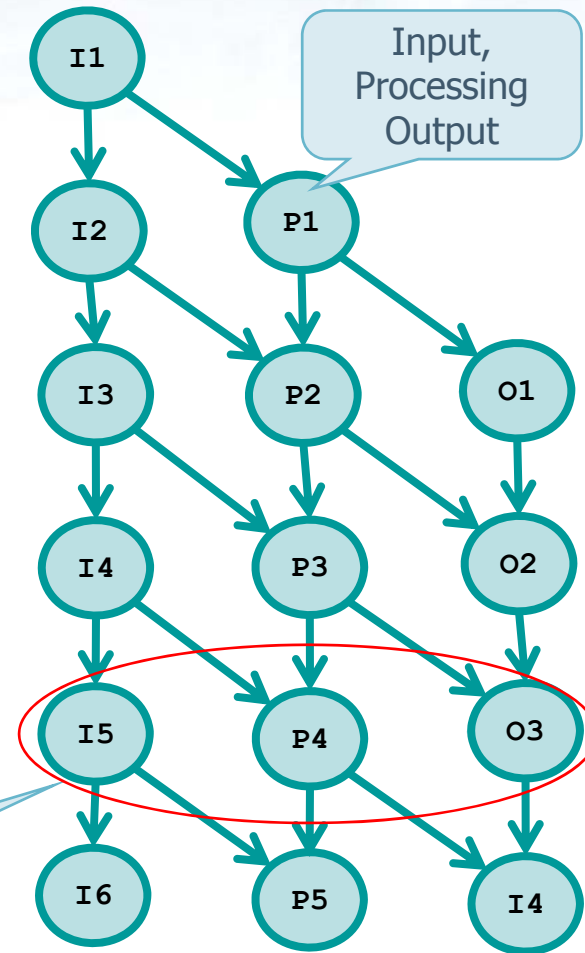


easily to debug

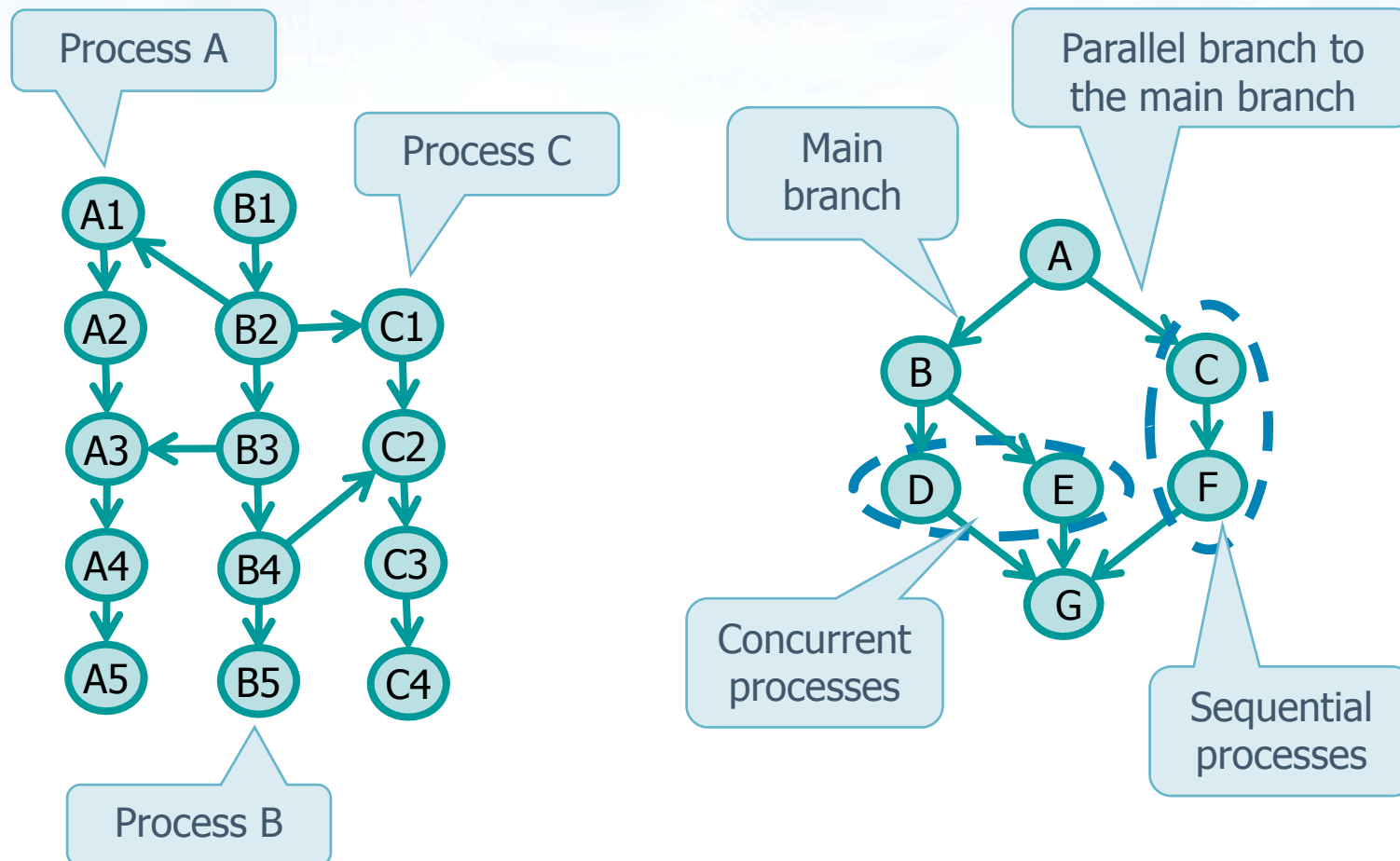
## Sequential and concurrent processes

### ❖ Concurrent execution

- More than one action can be executed at the same time
  - There is not order relation
  - Non deterministic behavior
- Real concurrency
  - on multi-processor or multi-core systems
- Pseudo-concurrency
  - on mono-processor systems



## Precedence graphs



## Conditions for concurrency

hard to debug

- ❖ When two processes can be executed in concurrency?
- ❖ Given a process **P**, let's define
  - **R (P)**
    - Read set of P: the set of variables read by P
  - **W (P)**
    - Write set of P: the set of variables modified by P

## Conditions for concurrency

### ❖ Bernstein conditions [1966]

- Two processes  $P_i$  and  $P_j$  can be executed in concurrency iff

- $R(P_i) \cap W(P_j) = \emptyset$
- $W(P_i) \cap R(P_j) = \emptyset$
- $W(P_i) \cap W(P_j) = \emptyset$

eg, global var  
service process

R R is ok

- Otherwise time-dependent errors, or the programmer must impose regions of **Mutual Exclusion** among processes

## Example

```
S1. a = x + y  
S2. b = z + 1  
S3. c = a - b  
S4. w = c + 1
```

❖ The sequential flow of instructions

➤  $S1 \rightarrow S2 \rightarrow S3 \rightarrow S4$

can be optimized because

- instruction 3 must be executed after 1 and 2
- instruction 4 must be executed after 3
- but, instructions 1 and 2 can be executed in parallel



## Example

```
S1. a = x + y
S2. b = z + 1
S3. c = a - b
S4. w = c + 1
```

```
R(S1)={x, y}, W(S1)={a}
R(S2)={z},    W(S2)={b}
R(S3)={a, b}, W(S3)={c}
R(S4)={c},    W(S4)={w}
```

```
R(S1) ∩ W(S2) = 0
R(S2) ∩ W(S1) = 0
W(S1) ∩ W(S2) = 0
...
```

