The Structure of Memory for Process & Threads

martes, 26 de abril de 2022 06:51 a. m.

The structure of a shared mem program:

↳ what is a program

Set of instructions

A task → solved by a processor

↳ PROCESS: is a program that executes a program

P → T

it has

↳ THREAD: it is a subtask of a program that executes in parallel with the program } inside of a process

↓

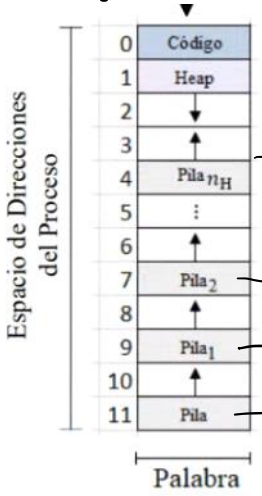
→ creation of process (main thread) can create threads to run with it

→ a thread generates its own stack

↑

function variables

A process has its address structure



all threads

all threads (including process) use the same heap, etc

from here on, it is shared space

stack for thread 2

stack for thread one

process stack

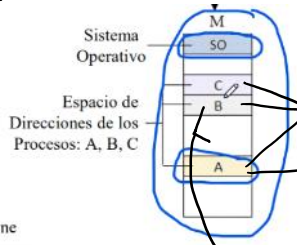
its thread

each thread uses its stack

[stack is used for local variables of a function where the program is]

This still has to be mapped to RAM:

Main memory structure



the programs executing

each has this inside here, theoretically (since there is RAM fragments and so on)

a thread inside process A cannot access process B threads.

→ a process can create ∞ threads

↳ a thread does not have shared space, a process does.

"address space" { ↳ they just share Heap and code space. (2)

↳ they do not share the process stack(s)

Shared Memory

→ we have one SO executing that executes a program using

a process (which can create ∞ threads inside it)

↳ All processors 'share' one SO.

→ communication: through memory spaces

→ has a) Static threads: all threads are created and destroyed at the same time (return to main thread)

b) Dynamic threads: threads are created and destroyed in different times

→ process and threads can create threads.

problem

→ problem: critical sections (requires synchronization) (with threads)

↳ code piece that 1) accesses a shared resource

↓

give origin to Race Condition } many processors / threads modify the same memory space / locality.

↓ outcome

Inconsistency

→ has to have: Mutual Exclusion

↳ guarantees that 1 process / thread accesses the critical section at a time (blocks)

↓

increases serialization

Objective: minimize the number of critical sections

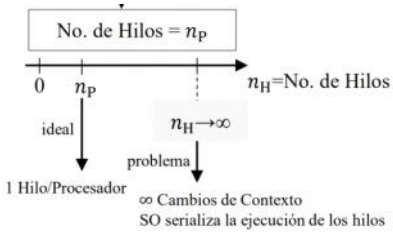
Thus, critical section → 0 : less serialization

In a shared mem system:

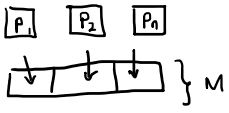
↳ the ideal number of threads

threads = number of processors (n_p)

because when a process is executing, its threads are spread accross the n_p processors, and so we can only generate n_p threads that run in real parallel. From then on, they are serialized and have context changes



→ a shared mem can be seen as a distrib mem system if: the memory is sliced for each processor



→ In distrib mem systems

↳ we have ∞ processes since each processor runs one SO each, and thus for each execution we have one process

↳ communication is messages

↳ can be seen as a shared mem system if all processors use a distributed SO (the same SO for all machines)

C++ 11 : multithread

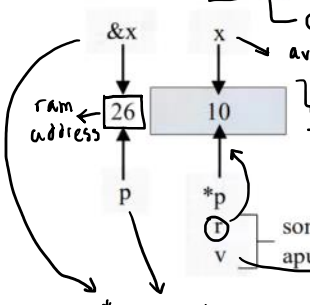
Apuntador y Referencia

int x; → a variable

int *p = &x; → a pointer : *p when defined

int &r = x; ↔ a reference : $\underbrace{\&r}_{\text{reference}}$ = x ; } basically r and x are the 'names' for the same mem address

int &&v = $\underbrace{\text{move(x)}}$ — C++ 11: 'reference to type r'



x → avariable : each x reference points to its value (10) } variable value

↓

&x = 26 (the address)

int &r = x ; // r is an alias to x

↳ r points to 10 as well as x

son como apuntadores

int && v = move (x)

*p : each reference as 'p' is the address

as '*p' is value reference = x .

a reference to type

a constant that is the value of x

all constants in c are type r values

v also points to $\underbrace{\text{x value}}$

a reference to the mem space where 10 is

C++11 introduced two definitions:

- l-value: is an object that has an identifiable mem space, i.e., x variable, pointer p, ret r. → they have a mem address
- r-value: everything that does not have an identiable mem space.

identifiable mem address: it means that somehow the programmer can get the address

Since C++11: a reference is to an l-value (&r)

&& v is a reference to an r-value