LABORATORIO SISTEMI OPERATIVI: PROCESS VARS

- Un **processo è un'istanza in esecuzione di un programma**.
- Quando eseguo un programma, viene creata **un'immagine nella memoria** che contiene una rappresentazione dei contenuti del codice binario e **quest'istanza** è quella che chiamiamo **processo**.
- All'interno del sistema, **ogni processo** è caratterizzato da una serie di attributi, in particolare **contiene**:
  - **CODICE**: Il **codice che viene eseguito** (codice che dipende dall'architettura, CPU e dall'ambiente d'esecuzione in cui ci troviamo).
  - **STATO**:
    - Composto da una serie di elementi:
      - **Program Counter**: cursore di posizione che specifica, all'interno dell'istanza (che è costituita da tante istruzioni) il punto in cui ci troviamo in un determinato momento (fa da segnalibro per l'istruzione da eseguire).
      - **Stack**: utile per registrare il punto di ritorno in caso di chiamata di funzione.
  - **DATI**: Dove **sono salvati i valori di alcune variabili** settate durante l'esecuzione.

- Programmi e processi sono due cose differenti:
  - **PROGRAMMA**:
    - **Entità passiva** costituita tipicamente da uno o più file, rappresentanti del codice di programmazione da eseguire.
  - **PROCESSO**:
    - È un'istanza di un programma, **un'entità attiva ed in esecuzione**.

- Un **processo è identificato** da:
  - UID:
    - **User ID**
    - ID che **identifica l'user** che ha generato il processo.
  - PID:
    - **Process ID**
    - **Identificatore unico che rappresenta il processo** in memoria (unico durante l'esecuzione).
    - In ogni istante temporale, **c'è un unico processo associato a quell'identificativo**.
  - PPID:
    - **Parent Process ID**
    - È un **identificativo** di processo ma fa **riferimento al "genitore"**, cioè al processo che ha creato il processo.
    - Tendenzialmente questo ID **non cambia durante l'esecuzione**, ma può capitare che il padre termini la sua esecuzione prima del suo processo figlio:
      - In questo caso il **processo figlio diventa un processo orfano**.
      - Visto che tutti i processi figli devono avere un padre, questo **processo orfano viene "adottato" dal processo init** (processo principale).
  - PGID:
    - **Process Group ID**
    - **Identificatore di gruppo**, ci permette di identificare e gestire un insieme di processi.

```
root@LABSO:/# ( ( echo | sleep 2s ) & ) ; ps -o uid,pid,ppid,pgid,sid,ruid,comma
nd
  UID   PID  PPID  PGID   SID  RUID COMMAND
    0     1     0     1     1     0 /bin/bash
    0   297     1   296     1     0 /bin/bash
    0   299     1   299     1     0 ps -o uid,pid,ppid,pgid,sid,ruid,command
    0   300   297   296     1     0 sleep 2s
root@LABSO:/#
```

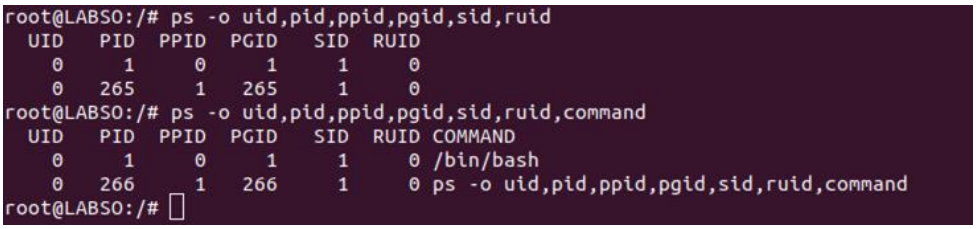
      - I comandi **echo** e **sleep** vengono eseguiti in **un altro processo bash in background**.
      - Questi due comandi generano **due processi differenti** in memoria, ma hanno stesso PGID.
      - Il comando **ps** viene eseguito sulla **bash** attualmente in esecuzione e mostra lo stato dei processi.

- SID:
  - **Process Session ID**
  - Una **sessione è un insieme di gruppi di processi**, in particolare **ogni sessione è collegata al massimo con un terminale** (può non averlo) e viceversa, **ogni terminale è collegato ad una sessione**.
  - Se chiudo il terminale, il sistema chiude tutti i processi che erano collegati a quel terminale usando il SID associato.

- RUID:
  - **Real User ID**
  - Può succedere che un **processo venga invocato da un utente ma eseguito come un altro**.
  - Ad esempio, quando si esegue un comando ma questo ha bisogno di privilegi root perché modifica qualche file di sistema, in quel caso il **RUID differisce dal UID**.

- **COMANDO PS**:
  - Il comando **ps** mostra **informazioni relative ai processi attualmente in memoria** nel sistema.
  - Invocando **ps** con il **flag -o [option]** è possibile **visualizzare** sul terminale **alcune particolari etichette** dei processi:
    - **ps -o uid, pid, ppid, pgid, sid, ruid**:
      - vengono **visualizzati i processi principali attivi**, mostrando a video le loro etichette (in questo caso sono i **vari ID che li caratterizzano**).

    - **ps -o uid, pid, ppid, pgid, sid, ruid, command**:
      - esecuzione analoga a quella descritta precedentemente, solo che oltre ai vari ID, viene **stampato** anche il **nome del processo** (il comando eseguito).



- **DAEMON**:
  - I **processi daemon** (demoni) sono una categoria specifica di **processi particolari**.
  - Solitamente si usa questo termine per riferirsi a **processi in esecuzione in background** tipicamente **senza un terminale collegato**.
  - **Creazione** di un demone:
    - **Processo P** (parent) **crea processo** figlio **C** (children).
    - **Processo P termina** la sua esecuzione **prima** del figlio C.
    - **Processo C** rimane in **esecuzione** ed esegue la funzione **setsid()**, funzione che avvia una nuova sessione senza un terminale collegato ad essa.
    - **Processo C** crea un **nuovo processo** figlio **G** (grandchild).
    - **Processo C termina** la sua esecuzione **prima** del figlio G.
    - Il **processo G** è un **daemon**.

- **MEMORIA**:
  - La memoria ha una sua organizzazione (può variare da sistema a sistema):
    - **OS KERNEL**: il sistema riserva gli indirizzi di memoria più bassi per il Kernel.
    - **TEXT SEGMENT**: riservato per le **istruzioni eseguite dalla CPU**.
    - **DATA SEGMENT**: riservato per le **inizializzazioni** (int a = 5).
    - **BSS SEGMENT**: riservato per le **variabili non inizializzate** (int b).
    - **HEAP**: spazio di memoria con **gestione dinamica** personalizzata dell'allocazione.
    - **STACK**: spazio di memoria con **gestione automatica dell'allocazione**, come ad esempio il punto di ritorno delle chiamate di funzione.

  - **Stack** e **Heap** si **contendono lo spazio** (se Heap cresce lo Stack diminuisce e viceversa).

  - In C la gestione della memoria avviene in **modo diretto**, non ci sono funzioni ad alto livello (salvo l'uso di librerie), ma di fatto **è demandato al programmatore la gestione** ottimale delle richieste e rilasci della memoria.

  - Non c'è un **GARBAGE COLLECTOR**, per cui **se alloco memoria devo anche liberarla**, altrimenti le zone di memoria allocate ma non usate verranno mantenute per tutta la durata dell'esecuzione del mio processo (spreco di memoria).

  - Le variabili possono essere:
    - **STATICHE**:
      - Sono **variabili che vengono inizializzate al momento della dichiarazione** (salvate nel **Data Segment**) **o non inizializzate**, viene dichiarato solo il tipo (salvate nel **BSS Segment**).

    - **DINAMICHE**:
      - Sono variabili che **contengono un riferimento ad una cella di memoria** (puntatori).
      - Sono **settate a run-time** e se non sono inizializzate assumo un valore casuale.
      - Devono essere **gestite mediante** apposite funzioni:
        - **malloc():**
          - Utilizzando la funzione malloc() **posso allocare e riservare una certa porzione di memoria**.
          - Questa zona di memoria verrà **referenziata mediante una variabile di tipo puntatore**.
          - Non è detto che utilizzando la malloc(), mi venga data una zona di memoria vuota, può darsi che **contenga dati vecchi**.

        - **free():**
          - Una volta utilizzata la **zona di memoria** appena allocata, **devo liberarla**.
          - La funzione **free() dealloca zone di memoria**, in modo da non sprecare risorse.

    - **AUTOMATICHE**:
      - **Variabili create localmente all'interno di funzioni** (se non inizializzate, assumono valori casuali).

- **LINGUAGGIO C:**
  - Nel linguaggio C, si hanno pochi e semplici tipi di dato (per strutture più complicate, dobbiamo usare costrutti e crearle da noi):
    - **INTEGERS:**
      - **Char**:
        - **Range**: 0-255 (sottoinsieme di numeri interi, ha un range limitato)
        - **Segnaposto**:
          - %i: stampata come numero intero
          - %c: stampata secondo la rappresentazione ASCII (65 = 'A')

```
root@LABSO:/tmp/printf# cat prinf.c
#include <stdio.h>

char x=65;

int main(){
  printf("Il valore di X e': %d mentre secondo la codifica ASCII vale: %c\n", x, x);
  return 0;
}

root@LABSO:/tmp/printf# ./printf
Il valore di X e': 65 mentre secondo la codifica ASCII vale: A
root@LABSO:/tmp/printf#
```

      - **Int**:
        - **Range**: (-32767 : +32767)
        - **Segnaposto**: %i o %d
      - **Long Int**:
        - **Range**: (-2147483647 : +2147483647)
        - **Segnaposto**: %li

    - **BOOLEAN**:
      - **Non esistono** tipi di dato booleani in C, vengono usati valori interi:
        - **0: falso**
        - **1: vero**
      - Usando la direttiva #define, si definiscono le macro:
        #define BOOL char
        #define FALSE 0
        #define TRUE 1

    - **STRING**:

```
#include <stdio.h>

char *testo="Ciao mondo"; //i doppi apici " " rappresentano un vettore
//la variabile testo e' un puntatore che punta ad un area di memoria che contiene una
//lista di byte
//Con i due apici, implicitamente, viene creato un vettore di char e alla fine di questo
//vettore viene aggiungo in automatico il terminatore di stringa: '\0'

char testo1[11] = "Ciao mondo"; //di dimensione 11, perche' alla fine ho '\0'

char testo2[11] = {'C', 'i', 'a', 'o', ' ', 'm', 'o', 'n', 'd', 'o', '\0'};
char testo3[11] = {'C', 'i', 'a', 'o', ' ', 'm', 'o', 'n', 'd', 'o', 0};
//in questo caso devo aggiungere il terminatore di stringa

int main(){
  printf("Ciao mondo\n");
  printf("%s \n", testo);
  printf("%s \n", testo1);
  printf("%s \n", testo2);
  printf("%s \n", testo3);
  return 0;
}
root@LABSO:/tmp/strings# ./stringexample
Ciao mondo
Ciao mondo
Ciao mondo
Ciao mondo
Ciao mondo
root@LABSO:/tmp/strings#
```

      - Le stringhe non sono un tipo di dato primitivo, sono infatti **vettori di char** (vettori di interi con range 0-255) e come **ultimo elemento dell'array**, hanno un **terminatore** di **stringa** (obbligatorio):
        - **0**: numero
        - **'\0'**: carattere
      - Sono **simili a puntatori** a **celle di memoria** che **contengono char**.

- **POINTERS**:
  - Sono variabili che **fanno riferimento a delle aree di memoria**.
  - **Operatori**:
    - **&var**:
      - Per accedere alla locazione di memoria assegnata alla variabile var, come **indizio di memoria**.
    - ***pointer**:
      - Definizione esplicita di un puntatore
      - Permette di **accedere al contenuto** dell'area di memoria puntata dalla variabile pointer.

  - **Dichiarazione**:
    - **int *pointer** o **int* pointer** (il primo modo è più concettualmente corretto, il puntatore si lega alla variabile, non al tipo).

```
root@LABSO:/tmp/pointers# cat pointer.c
#include <stdio.h>

int a=7;

int *m; //long m;
int b;

int main(){
  m = &a;
  printf("a = %d, memoria=%li\n", a, (long) m);
  b = *m;
  printf("b = %d, memoria=%li\n", b, (long) m);
  return 0;
}
root@LABSO:/tmp/pointers# ./pointer
a = 7, memoria=93834218655760
b = 7, memoria=93834218655760
root@LABSO:/tmp/pointers#
```

- **UNION**:
  - Permettono di **costruire delle alternative alle variabili**.
  - Sono strutturalmente simili alle struct, **creano un'aggregazione di variabili**, ma differiscono per l'accesso a queste variabili.
  - I **campi contenuti** all'interno della **union sono** da vedere come delle **alternative** (non si possono usare insieme né contemporaneamente), in quanto **condividono la stessa zona di memoria**.

```c
#include <stdio.h>
#include <string.h>

//crea un gruppo con 3 alternative possibili, una variabile Data puo' assumere solo un tipo dei 3 possibili
//questi 3 elementi sono da utilizzare in forma alternativa perche' condividono la stessa zona di memoria
union Data{
  int i;
  float f;
  char str[20];
};

int main(){
  union Data data;
  data.i = 10; //area di memoria viene valorizzata con il numero 10
  data.f = 220.5; //stessa area di memoria viene valorizzata con il numero 220.5 (10 viene sovrascritto)
  strcpy(data.str, "Esempio stringa"); //idem per la stringa (220.5 viene sovrascritto)
  printf("data.i = %d\n", data.i);
  printf("data.f = %f\n", data.f);
  printf("data.str = %s\n", data.str);
  return 0;
}
```

- **STRUCT**:
  - Permette di costruire degli **aggregati di variabili** (definite come campi).
  - I **campi delle struct** sono **accessibili** mediante la **dot notation**.
  - Tipicamente struct e typedef vengono usati per maggiore **leggibilità** del codice.

- **TYPEDEF**:
  - È un costrutto **utilizzato per rinominare** dei tipi di dato (è una macro dedicata per i tipi)
  - Viene usata per **creare degli alias**, i tipi di dato complessi o complicati, vengono rinominati per avere delle diciture più semplici ed auto-esplicative.

- **PUNTATORI/ARRAY:**
  - **Puntatore**: <type> *<var>
  - **Array**:
    - Sono dei **puntatori ad aree di memoria** che contengono dati omogenei.
    - <type> <var> [<size>]

  - ***(a+i) = a[i] = i[a]**