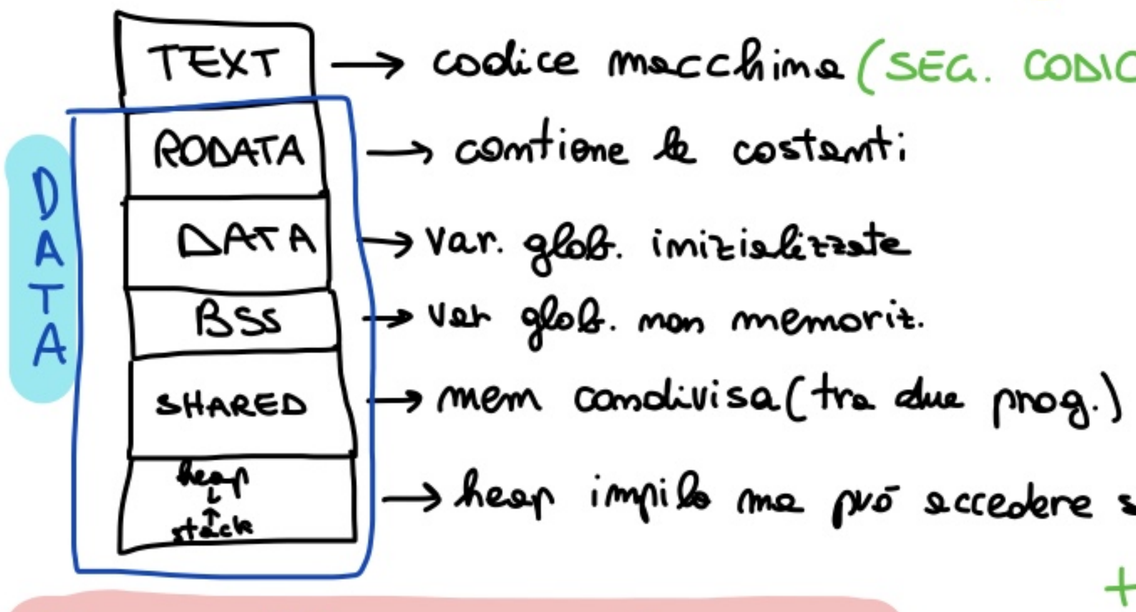
MEMORIA SEGMENTATA

→ suddd. della memoria fisica in blocchi, in modo da suddividere dati da processi e non danneggiare la mem intera in caso di problemi;

→ un **indirizzo di memoria** è formato da una parte per il **segmento** e una parte di **offset** entro il segmento indicato. Può riferirsi a mem fisica o virtuale;

↳ ad ogni segmento viene anche associata una comb. di permessi che consente di capire gli accessi consentiti: es. se si tratta di seg. di programma, di dati e di stack



→ codice macchina (SEG. CODICE)

→ contiene le costanti

→ var. glob. inizializzate (SEG. DATA)

→ var glob. non memorizz.

→ mem condivisa (tra due prog.)

→ heap impilo ma può accedere sempre a tutti mentre stack è un FSA

MEM. SEG. DATI DI UN PROC.

+ SEG. DI SISTEMA → contiene dati non gestiti esplicit. dal prog., ma da S.O. (es. tabella file aperti);

PROGRAMMAZIONE PARALLELA

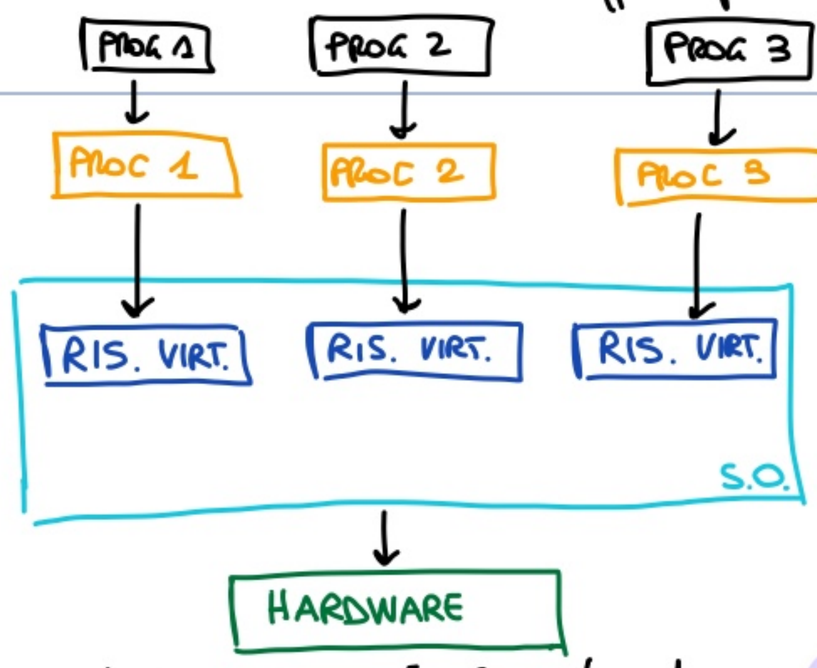
↳ avvio due programmi contemporaneamente, risparmiando tempo;

→ si creano conflitti tra gestione di risorse, determinismo, ordine esecuzione ....

→ problemi gestiti da **SISTEMA OPERATIVO** tramite scheduling, mem virtuale, perif. virtuali, ecc...

↳ insieme (sys-calls) di prog che svolgono funz. di servizio (mettere a disp. macc. virtuale e

PROCESSO → (parte dell' O.S.), insieme di info che legano i programmi alle risorse hw o in generale dell' o.s. Ogni processo ha un Address Space, un IP/PC, stack e heap;



CONCORRENTE: su uno stesso μP, concorrono per eseguire

Vs

PARALLELA: 2 μP, esegue contemporaneamente;

→ ogni processo è generato da un **padre** a parte il primo processo **INIT**, e ha una tabella associata detta **Process Descriptor** contenente PID, info su utente e mem, riferimenti a tab e

→ da init discende **LOGIN n°** e ad ognuno la propria shell (altre info (es. su time-execution)

creato da PROGRAMMA → eseguito da PROCESSO

→ il **KERNEL** si occupa tram. assembly di creare il primo processo

→ i processi non agiscono sulla mem. fisica ma sulla **memoria virtuale**, il s.o. virtualizza quindi le risorse;

OPERAZIONI SU PROCESSI

PID → process IDentifier, seq. 16 bit assegnato da Linux

↳ ovvero le **system call** (servizi di sistema) che consentono di operare sui processi

FORK() → creazione di processi figli (richiesti da un programma al s.o.), quindi uguali al padre a differenza del valore restituito: padre → PID figlio; figlio → ∅

LAZY LOADING: copio subito lo stack, aspetto azione per mem. map del PID

→ la funzione standard **vfork()** evita di copiare i seg. dati da padre a figlio inutilmente qualora si voglia modificare il figlio e creare un altro processo

EXIT() → terminazione, non cancella nulla e invia al padre **SIGCHILD** per dire che un figlio ha terminato e non ritorna valori;

WAIT() o **waitpid** → sospende esecuzione del processo che la esegue fino alla terminazione dei proc. figli:
↳ se termina un figlio specifico

pid_t wait(int*) restituisce pid del figlio terminato e il puntatore assume valore cod. di terminazione del figlio;

CASI PARTICOLARI → 1, se proc. figlio prima che il padre chiami la wait diventa **ZOMBIE**, ovvero il suo spazio non è stato ripulito;

2, se il padre muore prima del figlio, diventa **ORPHANED** ed è adottato da INIT

3, se non si chiama la exit si entra in un loop ∞ creando quindi un **DEMONE**;

EXEC() → sostituisce codice e dati (segmentati) del proc. corrente con quelli di un altro prog., MA il system data non è sostituito (il processo non cambia, ma le risorse del prog. si):

exec"x"(path prog, arg. prog. terminati da "NULL")

↳ da qui esistono varianti per la sostituzione del codice, per es. execl, execlp....

[se X=p non passo il "path" | se x=v accetta punt a stringhe | X=l accetta passaggio params con "varargs"]

↳ **getpid()** e **getppid()** restituiscono rispettivamente PID di un processo o PID del padre

→ per **monitorare i processi** si usa: **top** e **htop** (comando t per albero gerarchia)

KILL() → termina programmi e prende come parametri PID e flag-KILL del processo da eliminare e invia ai proc. segnali da shell. → termine anomalo di proc. da segnali

es. SIGBUS, SIGINT, SIGTERM...

es. creazione di un processo

```
int main(int argc, char** argv) {
    pid_t pid;
    pid = fork();
    if(pid == 0) {
        printf("\n Child pid %d \n", (int) getpid());
    }
    else{
        printf("\n Parent pid %d \n", (int) getpid());
    }
```

→ il segnale SIGKILL impone la terminazione immediata di un processo che non può gestire il segnale in questione

→ se un proc. riceve un segnale può: 1→ default, esegue handler di default, uso SIG_DFL;

2→ ignorandolo, con SIG_IGN passato a sigaction; non può ignorare SIGKILL e SIGSTOP

3→ gestito, se ha un handler specifico che ne modifica l'uso; specif. da SIGNAL()

HANDLER: funzione che specifica azioni di esec. per un segnale;

→ altri segnali per comunicare errore: SIGBUS, SIGSEGV, SIGFPE terminano prog. e generano codice errore;

COSTRUZIONE DI UN PROGRAMMA

→ dal punto di vista dell'utente il programma inizia con il main()

→ in realtà un programma inizia:

→ la funzione abort() termina tutto, anche exit handler prima che vengano finito o anche chiamati, tramite segnale SIGABRT

→ un segnale è un 'software interrupt', gestisce eventi asincroni e sincronizza processi.
- rappr. da 'SIG<nome>' o da numeri;
- generati da tasti o da 'kill()', da hw (kernel o proc), eccezioni su pipe chiusa

→ la funzione KILL() : pid>0 → invia segnale a proc indicato;
pid≤0 → comport complex;
SIG=0 → viene ignorato;

STANDARD POSIX definisce questa funzione

→ la funzione RAISE(SIGNAL): invia segnale al processo stesso ed è def da STANDARD ANSI C

→ la funzione sighandler_t signal(...) → registra la funz come sig-hand. Meglio arg viene passato un 'signum' intero assegnato al param e un param. handler che rappresenta azione processo (SIG_IGN, SIG_DFL, funz. 'void handler(int)');
IGNORA DEFAULT GEST. PARTICOLARI

→ la funzione unsigned int alarm(unsigned int secs) → imposta un 'software timer' di S=secs secondi per mandare un allarme al processo
es. interruzione di un processo lungo per continuarne altri

→ la funz. pause(void) → sospende esecuzione finché il processo non riceve un segnale

COMBINAZIONI
→ ALARM+HANDLER = mecc. TIMEOUT / seq. periodiche eventi
→ ALARM+HANDLER+PAUSE = mecc. ATTESA