

Differenza tra macro e funzione:

- 1) La macro sostituisce testo per testo
- 2) Una macro viene espansa a tempo di compilazione e una genericamente chiamata in fase di esecuzione
- 3) Un parametro di una macro viene valutato più volte nel suo corpo (una per ogni occorrenza)

Differenza tra struttura e union

Struttura: Collezione di una o più var. che possono avere tipi diversi.

Unione: Variabile che può contenere ad infinito diversi oggetti di tipo e dimensione diversi

Funzione Ricorsiva

Una funzione è ricorsiva se la sua esecuzione può essere interrotta in un qualunque punto e rieseguita senza che questo produca effetti indesiderati

Non sono ricorsive le funzioni che modificano variabili statiche o globali e che chiamano funzioni non-ricorsive

Librerie: Gruppi di file oggetto di uso comune



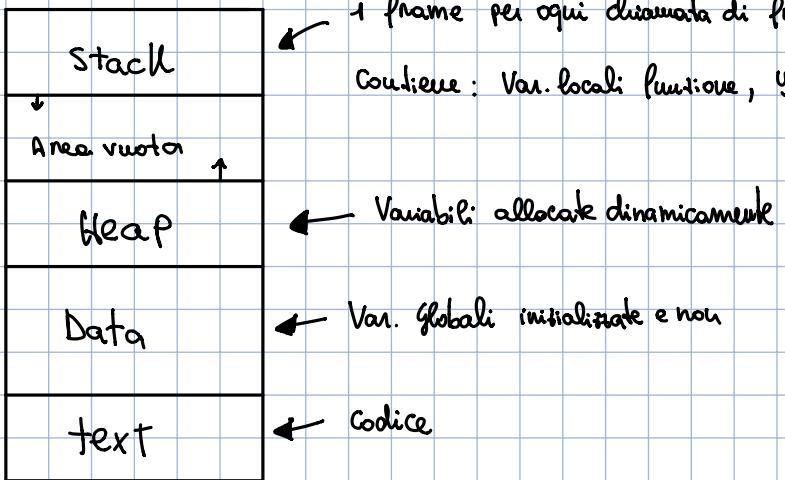
Vengono usate in fase di linking per generare l'eseguibile finale

Statiche: Agganciate al programma all'atto della compilazione

Condivise: Agganciate al programma in 2 fasi: 1) A tempo di compilazione il linker verifica che tutti i simboli siano definiti

2) Nella fase di caricamento il loader carica le librerie necessarie agganciate al programma

Come vede la memoria un programma C in esecuzione



1 frame per ogni chiamata di funzione da cui non abbiamo fatto ritorno

Contiene: Var. locali funzione, Ind. di ritorno, Copia value parametri

Variabili allocate dinamicamente

Var. globali initializzate e non

Codice

Ottenere un file eseguibile

- 1) Preprocessing:
- Espansione #include
- Sostituzione #Macro
- Compilazione condizionale

In Linux ha formato ELF (Executable and Linking Format)

- 2) Compilazione: Produce un **file oggetto .o** che contiene:

1) Magic Number: Numero che contraddistingue il file

2) Tabella dei simboli: lista dei simboli non conosciuti nel programma (main, printf...)

3) Tabella di rilocazione: lista degli elementi dei cui indirizzi questo file avrà bisogno

4) Data/BSS: Var. globali initializzate a 0

(Esterni al file main:
funzioni e var globali ecc...)

5) TEXT: Codice

- 3) Linking: Mette insieme file oggetti, librerie, stdlib ecc...

1) Unisce segmenti stessi tipo file oggetto + sistemazione indirizzi (tramite tab. di rilocazione)

2) Utilizzo delle Tabelle dei simboli per creare una tab. dei simboli globali e risolve i simboli

Makefile: è un file dependency system in Unix che permette:

- 1) Esprimere dipendenze tra file
- 2) Esprime cosa deve fare il sistema per aggiornare il target nel caso viene aggiornato un file da cui dipende
- 3) Mantiene consistenza il sistema

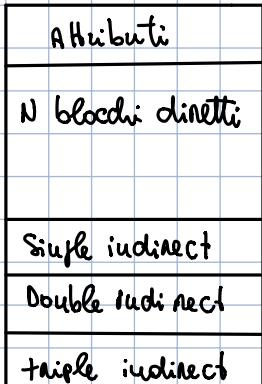
le regole di dipendenza formano un albero. Se il tempo di modifica di un nodo figlio è più recente della modifica del nodo padre, esegue la command list che ha come target il padre
Quindi si ha una vista bottom-up.

Phony targets: target che hanno lo scopo di eseguire una seq. di azioni `make namephony target`



Ogni file è rappresentato da un inode che contiene tutte le info riguardanti (- tipo
- protezione
- size
:)

Un inode contiene anche i puntatori a blocchi di dati in memoria o puntatori ad altri i-node



Superblocco: contiene info riguardanti il file system

- Dimensione
- # blocchi libri, lista e prox
- dim. lista inode, lista inode ...
- :

Differenza tra System Call e procedure di libreria:

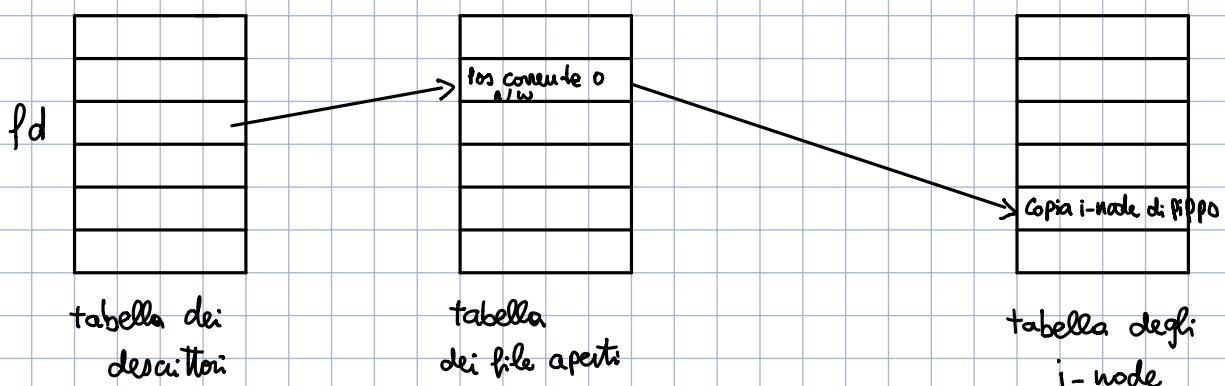
SC \Rightarrow non sono bufferizzate ma richiedono context switch

libstdc++ \Rightarrow sono bufferizzate

\Rightarrow conviene usare le stdlib func quando si lavora su pochi dati alla volta perché evita il context switch. Viceversa convengono le sc perché non hanno bufferizzazione

Implementazione della open

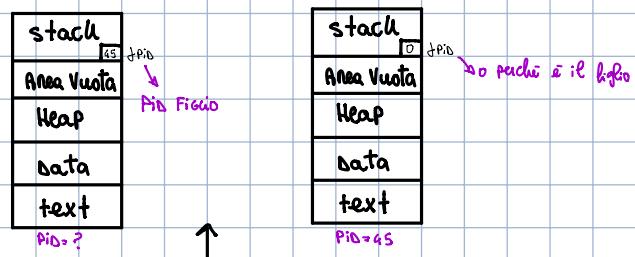
Ogni processo ha sua tabella di file descriptors che puntano alla file table che è la tabella che contiene i puntatori agli i-node dei file aperti



Nella tabella dei file aperti ho sempre i file stdcout/stdin/stderr

Foto

- 1) Crea un nuovo processo
- 2) Crea lo spazio di ind. del nuovo processo con una copia di quello del padre
- 3) Il figlio ha una copia della tabella dei descrittori di file del padre, non nuova la stessa
- 4) Condividono la tabella dei file aperti



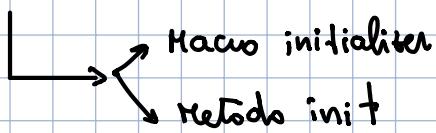
Attenzione: Si fa la copy-on-write, cioè si copia solo la tabella delle

pagine in sola lettura e se il figlio volere modificarle, il SO fa una copia di esse al volo.

Questo perché se il figlio fa l'exec, butta via tutto lo spazio di indirizzamento

tmead: lightweight - process che condividono la memoria

lock: chiave di accesso a una risorsa condivisa



Pipe: File speciali utilizzati per connettere due processi con un canale di comunicazione

1) Senza nome: File senza nome visibile, che viene utilizzato per comunic.

↓
array pfd[1] (^{composta}
_{da 2 elementi}) tra processi che condividono puntatori alla tabella dei dei file aperti (tabella dei descrittori)

Tipico uso: 1) Il padre crea la pipe

2) Fa la fork()

3) Il processo scrittore chiude pfd[0], mentre il processo lettore chiude pfd[1]

4) I processi comunicano col **read/write**

non fallisce se non ci sono scrittori → automatica se scrivo meno byte della dim. della pipe

→ ignorare segnale perché fallire se non ci sono lettori

5) Quando la comunicazione è finita, chiudono le loro estremità

Per implementare il piping e ridistribuzione si utilizzano le pipe + le

SVC **dup()** e **dup2()** per la duplicazione dei descrittori.

↓
duplico un descrittore
in una posizione libera
della tab. dei descrittori

Potrei dire in quale FD effettuare la duplicazione. Es: **dup(3,1)**

duplico il file descriptor 3 e lo metto su 1stout → Ridistribuzione

Cioè faccio una **printf(...)**; scriverebbe sul fd 3

2) Con nome (FIFO): File speciale con nome visibile a tutti gli altri processi sulla stessa macchina

↓
mkfifo(path, permissions)

tipico uso: Client - **server** → Deve ignorare SIGPIPE per non essere terminato da una write (perché invia SIGPIPE)
FA TERMINARE IL PROCESSO

- 1) Il server apre la pipe sia in lettura che scrittura
(perché se non alla chiusura dell'ultimo client il server leggerebbe un EOF ed uscirebbe → Noi vogliamo che il server rimanga attivo)
- 2) I client aprono in scrittura la stessa pipe
- 3) Le risposte dei client vengono generalmente date su pipe private (una per client)

Sockets: File usati per connettere due o più processi con un canale di comunicazione

SERVER

- 1) Creare socket e allocare un FD socket()
- 2) Assegnare socket con un indirizzo bind(socket, "my socket", ...)
- 3) Mettersi in listen listen(socket, ...)
- 4) Bloccarsi in attesa di richieste da altri processi accept(socket, ...)

CLIENT

- 1) Creare socket e allocare un FD socket()
- 2) Collegarsi con il socket del server usando il nome associato connect(client, "my socket", ...)
↓
Solitamente meno in un loop di tentativi
- 3) Se c'è un match fra accept e connect, viene creato un nuovo socket

Attenzione: Dopo aver accettato una connessione, il server deve rimettersi in listening e completare le richieste dei client comuni perché un read mi potrebbe occupare tanto tempo

- ↓
- 1) Server multithread
 - ↳ un thread dispatcher che fa l'accept
 - ↳ un thread per ogni client che gestisce le richieste
 - 2) Usare un server sequenziale e la SC select() che permette di capire quale FD è pronto
 - Gli passo fd-numberi (numero descrittivo)
 - Mi evita di fare una read/write insieme

Si blocca fin quando:

atteso per la lettura (readset)

Qualcuno è
pronto per
comunicare

- 1) lettura su descrittore fd^v non si blocca cioè quando ci sono i dati o è chiuso
- 2) write su descrittore fd atteso per la scrittura (writset) non si blocca
- 3) viene notificata un'eccezione su descrittore fd atteso per eccezioni (errset)

All'uscita dalla select() i fd sono modificati per indicare chi è pronto



Verifico chi è pronto facendo FD_ISSET sui bit della maschera.

AF_INET: Famiglia di indirizzi che permette di far comunicare i socket su internet (indirizzo IP + porta)

Quando avviene la comunicazione tra due macchine diverse, potrebbe succedere che lavorano su ENDIAN differenti: - bigEndian

- littleEndian

Per evitare questo i dati sono ordinati in un modo neutro detto: **network byte order**

- SEGNALI:** Sono interruzioni software \Rightarrow
- 1) Comunicano al process un evento
 - 2) Ad ogni evento corrisponde un signal number
 - 3) Azioni possibili:
 - Ignorarlo
 - Farlo gestire al S.O.
 - Specificare signal handler

Strutture dati del kernel relative ai segnali

- 1) Signal handler array: Decide cosa fare quando arriva un segnale
 - per processo condiviso tra i thread
- 2) Pending signal map: Un bit per ogni tipo di segnale \Rightarrow bit x a + se c'è un segnale pendente
- 3) Signal Mask: Un bit per ogni tipo di segnale \Rightarrow bit x a + se non voglio ricevere quel segnale
 - per thread

Entrando che lavora con maschere \Rightarrow Se arrivano più segnali, ne viene gestito uno solo

- Cosa accade quando arriva un segnale X ad un processo, con un solo thread?
 - I segnali vengono inseriti nella pending signal bitmap e controllati al ritorno da SC
 - Se la signal mask non blocca X il processo viene interrotto
 - il kernel stabilisce quale comportamento adottare controllando il contenuto del signal handler array
 - default, SIG_IGN, signal_handler
 - se deve essere eseguito un signal handler safun:
 - Si crea un frame "fittizio" sullo stack e si salva l'indirizzo di ritorno (quello dalla istruzione successiva a quella interrotta)
 - si esegue **safun** e il processo riprende l'esecuzione dalla istruzione successiva a quella interrotta
- Cosa accade quando arriva un segnale X ad un processo, con più thread?
 - Se il segnale è destinato ad un thread particolare T nel processo:
 - Se la signal mask di T non blocca il segnale X il thread viene interrotto
 - il kernel stabilisce quale comportamento adottare controllando il contenuto del signal handler array (globale nel processo)
 - se deve essere eseguito un signal handler si procede come già discusso
- Cosa accade quando arriva un segnale X ad un processo, con più thread? (segue)
 - Se il segnale è destinato al processo: viene scelto un thread T a caso e si procede come prima

Cioè ne viene scelto uno dal S.O.

Create un gestore

Struct sigaction s; \longrightarrow

1. sa_handler = funzione gestore;

```
} void *sa_handler (int); // gestore
} sigset sa-mask; // Maschera che dice quale
                   // segnali bloccare mentre
                   // installa il gestore
```

sigaction() Indico il gestore e quale segnale gestisce

Pthread_sigmask(): Indice come settare la maschera \Rightarrow

Quando è utile mascherare i segnali?

- Per indirizzare i segnali verso un thread specifico
- Per non essere interrotti durante l'esecuzione di un gestore
 - il segnale per cui è registrato il gestore è automaticamente mascherato durante la gestione (se usiamo sigaction) ma gli altri no
- Nello startup del programma quando ancora non abbiamo registrato tutte le gestione con sigaction
- Quando non vogliamo ricevere un segnale per tutta la durata del processo (in questo caso equivale a **SIG_IGN**)

Caratteristiche del gestore:

- 1) Deve essere breve e veloce
- 2) Deve utilizzare funzioni safe \Rightarrow Printf, scanf non sono safe!
- 3) Accesso safe a variabili globali (Definite volatile __atomic_t)

Esempio:

```
foo = 0;  
while (foo != 255)  
    ↪ non viene ottimizzata  
    in  
    while (true)
```

Sottolinea che la variabile può essere mod. da altri thread in un contesto multithread

Dice al compilatore di non fare ottimizzazioni su quella variabile

è un tipo

dipende dall'architettura

Ammira che gli accessi alla variabile sono atomici (+ ciclo macchina)

Forme più di + ciclo potrebbe essere interrotta l'operazione da parte del gestore

stati inconsistenti

Fork, exec e pthread_create:

Gestore:

- 1) Con la fork() il figlio eredita la gestione dei segnali
- 2) dopo la exec() le gestioni ritrovano quelle di default perché se lasciavo inviato, parerei al nuovo processo un indirizzo di mem. che non è il suo.

Signal Mask:

- 1) Con la fork() il figlio eredita la signal mask
- 2) Dopo la exec rimane la stessa

Pending Signal Mask:

- 1) Con la fork viene messa a 0
- 2) Rimane la stessa del thread che ha invocato la exec()

Inviare un segnale: viene messo a 1 il bit corrispondente della bit mask pending del processo che riceve il segnale

Si usa il metodo kill(pid, signum) o pthread_kill(tid, signum)
↓
segnale inviato

Il segnale viene inviato se : 1) i due processi hanno lo stesso owner
2) il processo che invia il segnale è root

Se manda una kill a un processo \Rightarrow uccide tutti i thread

Attendere un segnale

Pause();

Sigwait(maschera, signum) \Rightarrow permette attesa formata
↓ ↓
segnali che segnale
aspetta arrivato