DOMANDE/RISPOSTE GRUPPO B

1. Descrive il processo di avvio di un sistema Linux.
Il processo di avvio di un sistema Linux può essere suddiviso in diverse fasi:

BIOS: La prima fase dell'avvio di un sistema Linux inizia quando si accende il computer. In questa fase, il BIOS (Basic Input/Output System) del computer effettua alcuni controlli hardware, inizializza i dispositivi di input/output e cerca il bootloader del sistema operativo.

Bootloader: Una volta trovato il bootloader, viene caricato in memoria. Il bootloader di solito è GRUB (Grand Unified Bootloader) in Linux, che permette di scegliere il kernel del sistema operativo da caricare.

Kernel: Il kernel del sistema operativo viene caricato in memoria. Questo è il cuore del sistema operativo Linux, che gestisce l'accesso alle risorse hardware, i processi del sistema e le operazioni di input/output.

Initramfs: Dopo il caricamento del kernel, viene caricato l'initramfs (Initial RAM Filesystem). L'initramfs contiene un sistema operativo minimale che viene caricato in memoria e utilizzato per avviare il sistema operativo Linux completo.

Init: Dopo il caricamento dell'initramfs, viene eseguito l'init. L'init è il primo processo del sistema operativo Linux e viene utilizzato per avviare tutti gli altri processi del sistema. L'init avvia anche i servizi di sistema, come il demone del display manager o il demone di rete.

Runlevel: Una volta che tutti i processi sono stati avviati correttamente, il sistema operativo Linux passa al runlevel. Il runlevel è uno stato di esecuzione del sistema che determina i servizi che devono essere eseguiti. Ci sono diversi runlevel in Linux, a seconda della distribuzione, ma di solito c'è un runlevel per l'avvio normale del sistema, uno per la modalità di ripristino e uno per il riavvio del sistema.

Login: Infine, quando il sistema operativo Linux è completamente avviato, il login manager viene mostrato all'utente. L'utente può inserire il proprio nome utente e la password per accedere al sistema.

2. Descrivere il contenuto del file /etc/passwd e del file /etc/shadow. Inoltre, descrivere come si può ottenere, tramite shell, la riga del file /etc/passwd corrispondente a un determinato utente. Descrivere inoltre come si può aggiungere un utente al sistema, e verificare il suo inserimento nei file sopra citati.

Il file /etc/passwd è un file di testo che contiene le informazioni sugli account degli utenti del sistema operativo Linux. Ogni riga del file rappresenta un account utente e contiene i seguenti campi separati da due punti:
Nome utente

Password criptata o "x" per indicare che la password è memorizzata in un file separato (/etc/shadow)

ID utente numerico (UID)

ID di gruppo numerico (GID)

Descrizione dell'account utente (campo commento)

Directory home dell'utente

Shell di login dell'utente

Il file /etc/shadow è un file di testo crittografato che contiene le password degli account utente del sistema operativo Linux. Le password sono crittografate in modo da proteggere le informazioni dell'utente. Ogni riga del file rappresenta un account utente e contiene i seguenti campi separati da due punti:

Nome utente

Password crittografata

Ultima data di modifica della password

Numero di giorni prima che sia consentito modificare la password

Numero di giorni prima che la password scada

Numero di giorni dopo la scadenza della password prima che l'account sia disabilitato

Data di disabilitazione dell'account (in formato di giorni dall'epoca di Unix)

Un campo riservato per future modifiche

Per ottenere la riga del file /etc/passwd corrispondente a un determinato utente, è possibile utilizzare il comando grep dalla shell. Ad esempio, per ottenere la riga del file /etc/passwd corrispondente all'utente "bob", eseguire il seguente comando:

```
grep bob /etc/passwd
```

Per aggiungere un nuovo utente al sistema, è possibile utilizzare il comando useradd dalla shell. Ad esempio, per aggiungere un nuovo utente chiamato "jane", eseguire il seguente comando:

```
sudo useradd jane
```

Una volta creato l'account utente, è possibile verificare la sua presenza nei file /etc/passwd e /etc/shadow eseguendo i seguenti comandi:

```
grep jane /etc/passwd
```

grep jane /etc/shadow

Se l'utente fosse stato creato correttamente, dovrebbe apparire una riga corrispondente in entrambi i file.

3. Descrivere la gestione dei permessi in Linux, e la modifica dei permessi tramite shell.

Fornire un esempio di modifica dei permessi tramite shell, utilizzando la notazione ottale

per la definizione dei permessi.

In Linux, i permessi di file e directory sono gestiti dal kernel del sistema operativo, che utilizza un sistema di controllo degli accessi basato su utenti e gruppi. Ogni file e directory ha tre tipi di permessi: lettura, scrittura ed esecuzione, che possono essere impostati separatamente per il proprietario del file, il gruppo proprietario del file e gli altri utenti.

La modifica dei permessi di un file o di una directory può essere effettuata tramite la shell di Linux, utilizzando il comando chmod. Il comando chmod utilizza una notazione numerica ottale per definire i permessi, in cui ogni tipo di permesso (lettura, scrittura, esecuzione) viene rappresentato da un valore numerico:

4 rappresenta il permesso di lettura

2 rappresenta il permesso di scrittura

1 rappresenta il permesso di esecuzione

La combinazione di questi valori numerici viene utilizzata per definire i permessi per il proprietario del file, il gruppo proprietario del file e gli altri utenti.

Ecco un esempio di modifica dei permessi di un file tramite la shell di Linux, utilizzando la notazione numerica ottale:

Supponiamo che il file si chiami "mio_file.txt" e attualmente ha i seguenti permessi:

proprietario: lettura e scrittura (rw-)

gruppo proprietario: lettura (r--)

altri utenti: nessun permesso (--)

Per impostare i permessi in modo che il proprietario, il gruppo proprietario e gli altri utenti abbiano tutti i permessi di lettura e scrittura, utilizziamo il seguente comando:

chmod 666 mio_file.txt

In questo comando, il primo numero "6" rappresenta la combinazione dei permessi per il proprietario del file (lettura e scrittura), il secondo numero "6" rappresenta la combinazione dei permessi per il gruppo proprietario del file (lettura e scrittura) e il terzo numero "6" rappresenta la combinazione dei permessi per gli altri utenti (lettura e scrittura).

4. Cosa sono i gruppi, come si può verificare i gruppi a cui appartiene il nostro nome utente, e come si può aggiungere un utente a un gruppo?

In Linux, i gruppi sono una forma di organizzazione degli utenti in modo che possano avere determinati privilegi e permessi condivisi. Un gruppo può contenere uno o più utenti, e ogni utente può appartenere a uno o più gruppi.

Per verificare i gruppi a cui appartiene il nostro nome utente in Linux, possiamo utilizzare il comando "groups" seguito dal nostro nome utente:

groups nomeutente

Questo comando elencherà tutti i gruppi a cui l'utente appartiene.

Per aggiungere un utente a un gruppo in Linux, possiamo utilizzare il comando "usermod" seguito dal nome dell'utente e dal nome del gruppo a cui vogliamo aggiungerlo:

sudo usermod -a -G nomegruppo nomeutente

In questo comando, l'opzione "-a" indica di aggiungere l'utente al gruppo senza rimuovere l'appartenenza a eventuali altri gruppi, mentre l'opzione "-G" indica il nome del gruppo a cui vogliamo aggiungere l'utente.

Dopo aver eseguito questo comando, l'utente dovrà effettuare nuovamente il login affinché le modifiche ai gruppi diventino effettive. In alternativa, possiamo riavviare il sistema.

5. Commentare le seguenti righe di output del comando ls -l. Descrivere cosa indicano i

vari campi.

drwxrwxr-x 13 uA uB 4096 Nov 27 17:32 toolbox

-r-xr-xr-x 1 uA UB 4584 Feb 29 2020 note

lrwxrwxrwx 1 root root 8 Jan 31 2018 ypdomainname -> hostname

Il comando ls -l mostra i dettagli dei file e delle cartelle nella directory corrente. Ogni riga di output rappresenta un file o una cartella e contiene diverse informazioni sui permessi, il proprietario, il gruppo, la dimensione, la data di creazione e il nome del file.

La prima colonna indica il tipo di file e i permessi associati. La lettera iniziale indica se si tratta di una cartella (d), di un file (-) o di un link simbolico (l). Le tre triplette di permessi (rwx) indicano i permessi di lettura, scrittura ed esecuzione per il proprietario (u), il gruppo (g) e gli altri (o) utenti.

La seconda colonna indica il numero di hard link al file. In genere, questo valore è 1 per i file normali, ma può essere diverso per le cartelle e i file speciali.

Le colonne successive indicano il nome del proprietario del file, il nome del gruppo associato, la dimensione del file in byte e la data di ultima modifica.

Infine, l'ultima colonna indica il nome del file o della cartella.

Analizzando le righe di output fornite come esempio:

La prima riga indica una cartella chiamata "toolbox" con permessi di lettura, scrittura ed esecuzione per il proprietario e il gruppo, ma solo permessi di lettura ed esecuzione per gli altri utenti. La cartella è stata creata il 27 novembre alle 17:32 e appartiene al proprietario "uA" e al gruppo "uB".

La seconda riga indica un file chiamato "note" con permessi di lettura ed esecuzione per tutti gli utenti (proprietario, gruppo e altri). Il file è stato creato il 29 febbraio 2020 e appartiene al proprietario "uA" e al gruppo "uB". La dimensione del file è di 4584 byte.

La terza riga indica un link simbolico chiamato "ypdomainname" che punta a un altro file chiamato "hostname". Il link simbolico ha permessi di lettura, scrittura ed esecuzione per tutti gli utenti ed è stato creato il 31 gennaio 2018. Il link simbolico è di dimensioni 8 byte. Il proprietario e il gruppo del link simbolico sono entrambi "root".

6. Come si redirige da shell lo standard input, standard output e standard error? Come si può redirezionare lo standard output in *append*? Fornire un esempio di redirezione, in cui si utilizzano i dispositivi virtuali *null* e *random*.

In Unix e Unix-like shell, si possono redirigere lo standard input, lo standard output e lo standard error utilizzando il simbolo "<" per lo standard input, il simbolo ">" per lo standard output e il simbolo "2>" per lo standard error.

Ecco alcuni esempi di redirezione:

Per redirigere lo standard output in un file, si utilizza il simbolo ">", ad esempio:

ls > elenco_file.txt

Questo comando elenca i file nella directory corrente e li redireziona nello standard output nel file "elenco_file.txt".

Per redirigere lo standard output in append a un file esistente, si utilizza il doppio simbolo ">>", ad esempio:

echo "Aggiungo testo a fine file" >> elenco_file.txt

Questo comando aggiunge la stringa "Aggiungo testo a fine file" alla fine del file "elenco_file.txt" senza sovrascrivere il contenuto esistente.

Per redirigere lo standard input da un file, si utilizza il simbolo "<", ad esempio:

sort < elenco_file.txt

Questo comando ordina le righe nel file "elenco_file.txt" e le utilizza come input.

Per redirigere lo standard error in un file, si utilizza il simbolo "2>", ad esempio:

ls /file/inventato 2> errori.txt

Questo comando tenta di elencare un file inesistente e redirige l'errore nello standard error nel file "errori.txt".

Per redirigere lo standard output e lo standard error in un file, si utilizza il simbolo "&>", ad esempio:

ls /file/inventato &> errori.txt

Questo comando tenta di elencare un file inesistente e redirige l'output e l'errore nel file "errori.txt".

Per redirigere lo standard output e lo standard error in un dispositivo virtuale "null", si utilizza il simbolo "/dev/null", ad esempio:

ls /file/inventato &> /dev/null

Questo comando tenta di elencare un file inesistente e redirige l'output e l'errore nel dispositivo virtuale "null", che ignora tutto l'input.

Per redirigere lo standard input da un dispositivo virtuale "random", si utilizza il simbolo "/dev/random", ad esempio:

cat /dev/random

Questo comando mostra i byte casuali generati dal dispositivo virtuale "random".

7. Cosa è e come si utilizza il meccanismo di pipe su shell? Fornire un esempio di utilizzo.

Il meccanismo di pipe su shell è un modo per collegare l'output di un comando alla successiva operazione di input di un altro comando, consentendo così di concatenare comandi e creare pipeline di elaborazione dati.

In pratica, l'output di un comando viene utilizzato come input per un altro comando, senza la necessità di salvare i dati in un file intermedio.

Il simbolo "|" viene utilizzato per creare una pipe tra due comandi nella shell. Ad esempio, il seguente comando crea una pipe tra i comandi "ls" e "grep":

ls | grep foo

In questo caso, l'elenco dei file nella directory corrente prodotto dal comando "ls" viene passato come input al comando "grep", che cerca le linee che contengono la stringa "foo".

Un altro esempio di utilizzo della pipe può essere il seguente:

cat file.txt | grep "parola" | wc -l

In questo caso, il contenuto del file "file.txt" viene passato come input al comando "grep", che cerca le linee che contengono la stringa "parola", e l'output di "grep" viene passato come input al comando "wc -l", che conta il numero di linee che corrispondono alla stringa di ricerca.

Il meccanismo di pipe su shell è molto potente e consente di combinare comandi in modi complessi per ottenere il risultato desiderato.

8. Come si concatenano i comandi su shell, e come si creano sequenze condizionali di comandi? Fornire un esempio di utilizzo per ciascun operatore di concatenazione.

Su shell, si possono concatenare i comandi utilizzando gli operatori di concatenazione ";" e "&&".

Con l'operatore ";" si eseguono due o più comandi in sequenza, indipendentemente dal risultato del precedente comando. Ad esempio:

cd /cartella; ls -l; echo "Fine lista file"

In questo caso, viene prima cambiata la directory nella cartella desiderata, poi viene eseguito l'elenco dei file nella cartella corrente, e infine viene visualizzato un messaggio di conferma.

Con l'operatore "&&" si esegue il secondo comando solo se il primo comando ha avuto successo. Ad esempio:

cd /cartella && ls -l

In questo caso, viene cambiata la directory nella cartella desiderata solo se l'operazione ha avuto successo, poi viene eseguito l'elenco dei file nella cartella corrente.

Con l'operatore "||" si esegue il secondo comando solo se il primo comando ha fallito. Ad esempio:

cd /cartella || echo "Errore: cartella inesistente"

In questo caso, viene cambiata la directory nella cartella desiderata solo se l'operazione ha avuto successo, altrimenti viene visualizzato un messaggio di errore.

Inoltre, si possono creare sequenze condizionali di comandi utilizzando le parentesi graffe "{}" e l'operatore "&&" o "||". Ad esempio:

{ cd /cartella && ls -l; } || { echo "Errore: cartella inesistente"; }

In questo caso, viene cambiata la directory nella cartella desiderata solo se l'operazione ha avuto successo, poi viene eseguito l'elenco dei file nella cartella corrente. Se l'operazione fallisce, viene visualizzato un messaggio di errore.

In generale, gli operatori di concatenazione e le sequenze condizionali sono molto utili per automatizzare le operazioni di shell e gestire gli errori in modo efficace.

8. Come si concatenano i comandi su shell, e come si creano sequenze condizionali di comandi? Fornire un esempio di utilizzo per ciascun operatore di concatenazione.

Per concatenare comandi sulla shell, si possono utilizzare gli operatori di concatenazione ;, && e ||.

L'operatore ; esegue i comandi uno dopo l'altro, indipendentemente dal successo o dal fallimento dell'operazione precedente. Ad esempio:

mkdir example; cd example; ls

crea una directory chiamata "example", si sposta all'interno di essa e infine elenca il contenuto della directory.

L'operatore && esegue il comando successivo solo se quello precedente ha avuto successo (exit code 0). Ad esempio:

make && make install

compila il codice sorgente utilizzando il comando make, e se la compilazione ha successo, esegue il comando make install.

L'operatore || esegue il comando successivo solo se quello precedente ha fallito (exit code diverso da 0). Ad esempio:

grep foo file.txt || echo "non trovato"

cerca la stringa "foo" nel file "file.txt", e se non la trova, esegue il comando echo "non trovato".

Per creare sequenze condizionali di comandi, si possono utilizzare gli operatori if, then, else, fi.

Ad esempio:

if [-f file.txt]; then echo "Il file esiste"; else echo "Il file non esiste"; fi

controlla se il file "file.txt" esiste. Se esiste, il comando echo "Il file esiste" viene eseguito, altrimenti viene eseguito il comando echo "Il file non esiste".

9. Fornire un esempio di utilizzo (scrivendo la sintassi corrispondente) dei seguenti comandi

di shell.

a. find combinata con grep

b. find combinata con ok

c. find con redirezione degli errori di accesso in /dev/null

d. find combinata con l'azione di rimozione dei file

e. find con il flag per ricercare i file di un determinato tipo

f. find con il flag per specificare il nome del file

g. find combinata con exec, e con l'azione print

Ecco degli esempi di utilizzo dei comandi di shell richiesti:

a. Trova tutti i file nella directory corrente e nelle sottodirectory che contengono la stringa "example" nel loro nome:

find . -name "*example*" | grep -i "example"

b. Trova tutti i file nella directory corrente e nelle sottodirectory che contengono la stringa "example" nel loro nome e stampa il loro percorso assoluto:

find . -name "*example*" -ok echo {} \;

c. Trova tutti i file nella directory corrente e nelle sottodirectory escludendo quelli ai quali non si ha accesso, e invia gli eventuali errori in /dev/null:

find . -readable 2>/dev/null

d. Trova tutti i file nella directory corrente e nelle sottodirectory con estensione ".log" e li elimina:

find . -name "*.log" -exec rm {} \;

e. Trova tutti i file nella directory corrente e nelle sottodirectory con estensione ".txt":

find . -name "*.txt"

f. Trova tutti i file nella directory corrente e nelle sottodirectory che contengono la stringa "example" nel loro nome, ignorando la differenza tra maiuscole e minuscole:

find . -iname "*example*"

g. Trova tutti i file nella directory corrente e nelle sottodirectory che contengono la stringa "example" nel loro nome e stampa il loro percorso assoluto:

find . -name "*example*" -exec echo {} \; -print

10. Da shell, come si può trovare la lista dei processi attivi, e come si può terminare un processo?

Per trovare la lista dei processi attivi sulla shell, si può utilizzare il comando ps, che visualizza una lista dei processi in esecuzione sulla macchina.

Ecco alcuni esempi di utilizzo del comando ps:

Per visualizzare la lista dei processi attivi di tutti gli utenti:

ps aux

Per visualizzare la lista dei processi attivi del proprio utente:

ps u

Per visualizzare la lista dei processi attivi in formato albero:

ps axjf

Per terminare un processo, si può utilizzare il comando kill, seguito dall'ID del processo o dal PID (Process ID).

Ecco un esempio di utilizzo del comando kill:

Per terminare un processo con un determinato PID:

kill PID

È anche possibile utilizzare il comando killall per terminare un processo tramite il nome del processo stesso.

Ecco un esempio di utilizzo del comando killall:

Per terminare tutti i processi con un determinato nome:

killall nome_processo

In entrambi i casi, se il processo è in esecuzione, verrà terminato immediatamente. È possibile utilizzare il flag -9 per forzare la terminazione del processo.

11. Da shell, come si può visualizzare e modificare una variabile di ambiente? Come si può definire una nuova variabile, assegnandole un valore? Per visualizzare le variabili di ambiente attualmente impostate nella shell, si può utilizzare il comando env.

Ecco un esempio di utilizzo del comando env:

env

Questo comando visualizzerà tutte le variabili di ambiente impostate nella shell corrente.

Per modificare il valore di una variabile di ambiente esistente, è possibile utilizzare il comando export, seguito dal nome della variabile e dal nuovo valore.

Ecco un esempio di utilizzo del comando export per modificare il valore della variabile di ambiente PATH:

export PATH=/usr/local/bin:$PATH

In questo esempio, la variabile di ambiente PATH viene impostata su /usr/local/bin, con $PATH che indica che il valore esistente della variabile deve essere conservato.

Per definire una nuova variabile di ambiente e assegnarle un valore, è possibile utilizzare il comando export, seguito dal nome della variabile e dal valore.

Ecco un esempio di utilizzo del comando export per definire una nuova variabile di ambiente MYVAR:

export MYVAR=myvalue

In questo esempio, la variabile di ambiente MYVAR viene impostata su myvalue.

12. Cosa è e come può essere utilizzato il file bashrc?

Il file .bashrc è un file di configurazione per la shell Bash, che viene eseguito ogni volta che viene avviata una nuova shell interattiva. È un file di script Bash, che contiene comandi che vengono eseguiti automaticamente quando la shell viene avviata.

Il file .bashrc viene utilizzato per personalizzare l'ambiente della shell Bash per uno specifico utente, e contiene di solito le seguenti informazioni:

Definizione di variabili d'ambiente

Definizione di alias per comandi frequentemente utilizzati

Definizione di funzioni personalizzate

Impostazione delle variabili di prompt per personalizzare l'aspetto della riga di comando

Esecuzione di script o programmi personalizzati all'avvio della shell

Il file .bashrc si trova nella directory home dell'utente (tipicamente /home/username per gli utenti Linux), e può essere modificato con qualsiasi editor di testo.

Ecco un esempio di come può essere utilizzato il file .bashrc:

```
# Definizione di variabili d'ambiente
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64/
export PATH=$PATH:$JAVA_HOME/bin

# Definizione di alias
alias ll='ls -alF'

# Definizione di funzioni personalizzate
function myfunc() {
  echo "Hello, World!"
}

# Impostazione delle variabili di prompt
PS1='\[\e[32m\]\u@\h:\w\$\[\e[0m\] '

# Esecuzione di script personalizzato all'avvio della shell
source /path/to/my/script.sh
```

In questo esempio, le variabili d'ambiente JAVA_HOME e PATH vengono impostate per includere il percorso della versione di Java installata sul sistema. Viene definito anche l'alias ll per eseguire il comando ls -alF. Viene anche definita una funzione personalizzata myfunc() che stampa un messaggio di saluto. La variabile di prompt PS1 viene impostata per personalizzare l'aspetto della riga di comando. Infine, viene eseguito uno script personalizzato all'avvio della shell mediante il comando source.

In questo modo, il file .bashrc può essere utilizzato per personalizzare l'ambiente della shell Bash in modo da soddisfare le esigenze specifiche dell'utente.

13. Scrivere un helloworld in C, e le istruzioni per la sua compilazione da shell.

Ecco un esempio di codice per un programma "Hello, World!" in linguaggio C:

```
#include <stdio.h>

int main() {
   printf("Hello, World!\n");
   return 0;
}
```

Questo programma utilizza la funzione printf per stampare il messaggio "Hello, World!" sulla console.

Per compilare il programma, è possibile utilizzare il compilatore C gcc da linea di comando. Assumendo che il file sorgente si chiami helloworld.c, ecco le istruzioni per la compilazione da shell:

```
gcc -o helloworld helloworld.c
```

In questo esempio, il comando gcc viene utilizzato per compilare il file sorgente helloworld.c e generare un file eseguibile chiamato helloworld. L'opzione -o viene utilizzata per specificare il nome del file di output. Se la compilazione avviene senza errori, verrà creato un file eseguibile chiamato helloworld.

Per eseguire il programma, è possibile digitare il nome del file eseguibile:

```
./helloworld
```

In questo modo, il messaggio "Hello, World!" verrà stampato sulla console.

14. Come si acquisisce l'input da tastiera in C? Scrivere un esempio

In C, è possibile acquisire l'input da tastiera utilizzando la funzione scanf(), che permette di leggere input formattati dalla console.

Ecco un esempio di codice che utilizza la funzione scanf() per acquisire un intero da tastiera:

```
#include <stdio.h>

int main() {
   int num;
   printf("Inserisci un numero: ");
   scanf("%d", &num);
   printf("Hai inserito il numero %d\n", num);
   return 0;
}
```

In questo esempio, la funzione scanf() viene utilizzata per leggere un intero dalla console e memorizzarlo nella variabile num. L'argomento %d specifica che l'input è un intero, e l'operatore & viene utilizzato per passare l'indirizzo di memoria della variabile num alla funzione scanf().

La funzione printf() viene poi utilizzata per stampare il valore inserito dall'utente sulla console.

Quando si esegue il programma, verrà visualizzato il messaggio "Inserisci un numero: " sulla console. L'utente potrà quindi inserire un numero e premere il tasto Invio. Il programma leggerà l'input dalla console utilizzando la funzione scanf(), e stamperà il valore inserito dall'utente sulla console utilizzando la funzione printf().

15. Descrivere l'operatore di indirizzamento e dereferenziazione di un puntatore e fornire un esempio.

In C, un puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile. L'operatore di indirizzamento (&) permette di ottenere l'indirizzo di memoria di una variabile, mentre l'operatore di dereferenziazione (*) permette di accedere al valore di una variabile puntata da un puntatore.

Ecco un esempio che mostra l'utilizzo degli operatori di indirizzamento e dereferenziazione:

```
#include <stdio.h>

int main() {
   int num = 42;
   int *ptr = &num; // dichiarazione e inizializzazione di un puntatore
   printf("Il valore di num è %d\n", num);
```

```
    printf("L'indirizzo di num è %p\n", &num);
    printf("Il valore puntato da ptr è %d\n", *ptr);
    printf("L'indirizzo memorizzato in ptr è %p\n", ptr);
    return 0;
}
```

In questo esempio, viene dichiarata una variabile num di tipo int e viene assegnato il valore 42. Viene poi dichiarato un puntatore ptr di tipo int *, che viene inizializzato con l'indirizzo di memoria della variabile num utilizzando l'operatore di indirizzamento (&).

La funzione printf() viene poi utilizzata per stampare il valore di num sulla console utilizzando la variabile num e l'operatore di indirizzamento &. Viene poi stampato il valore puntato da ptr utilizzando l'operatore di dereferenziazione (*), e l'indirizzo memorizzato in ptr utilizzando la variabile ptr.

Quando si esegue il programma, verranno stampati i valori di num e ptr, insieme ai relativi indirizzi di memoria. In particolare, la stampa del valore puntato da ptr utilizzando l'operatore di dereferenziazione darà come risultato il valore di num, ovvero 42.

16. Data la dichiarazione

int a[10], *p

dire quali delle seguenti affermazioni sono corrette, motivando la risposta.

a = p

p = &a[0]

a[3] equivale a &(a + 3)

p = a + 3 equivale a p = &a[3]

a[3] equivale a *(a + 3)

L'affermazione a = p è scorretta. La variabile a è un array di interi, mentre p è un puntatore a interi. Un array non può essere assegnato a un puntatore.

L'affermazione p = &a[0] è corretta. &a[0] restituisce l'indirizzo di memoria del primo elemento dell'array a, che è equivalente a a. Quindi p viene inizializzato con l'indirizzo di memoria di a[0] tramite l'operatore di indirizzamento &.

L'affermazione a[3] equivale a &(a + 3) è scorretta. a[3] rappresenta il valore del quarto elemento dell'array a, mentre &(a + 3) rappresenta l'indirizzo di memoria del quarto elemento dell'array che sarebbe fuori dai limiti dell'array.

L'affermazione p = a + 3 equivale a p = &a[3] è corretta. a + 3 rappresenta l'indirizzo di memoria del quarto elemento dell'array a. Questo indirizzo viene assegnato a p, che diventa quindi un puntatore al quarto elemento dell'array.

L'affermazione a[3] equivale a *(a + 3) è corretta. a[3] rappresenta il valore del quarto elemento dell'array a, mentre *(a + 3) rappresenta il valore del quarto elemento dell'array a tramite l'operatore di dereferenziazione *.

17. Cosa è il passaggio dei parametri per riferimento, e come si può realizzare in C? Fornire un piccolo esempio.

Il passaggio dei parametri per riferimento è una tecnica utilizzata in programmazione per passare gli indirizzi di memoria delle variabili ai parametri di una funzione, invece di passare le copie dei valori stessi. In questo modo, la funzione può accedere e modificare direttamente i dati della variabile originale, anziché operare su una copia temporanea.

In C, è possibile realizzare il passaggio dei parametri per riferimento utilizzando i puntatori come parametri della funzione. La sintassi consiste nel dichiarare il parametro come un puntatore al tipo della variabile che si vuole passare per riferimento, e poi utilizzare l'operatore di dereferenziazione * all'interno della funzione per accedere al valore della variabile originale.

Esempio:

```
#include <stdio.h>

void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int main() {
   int x = 10, y = 20;
   printf("Prima dello scambio: x = %d, y = %d\n", x, y);
   swap(&x, &y);
```

```
    printf("Dopo lo scambio: x = %d, y = %d\n", x, y);
    return 0;
}
```

In questo esempio, la funzione swap prende come parametri due puntatori a interi a e b, che rappresentano gli indirizzi di memoria delle variabili x e y dichiarate nel main. All'interno della funzione, i valori delle variabili vengono scambiati utilizzando l'operatore di dereferenziazione *. Per passare gli indirizzi di memoria delle variabili x e y alla funzione swap, vengono utilizzati gli operatori di indirizzamento &x e &y.

18. Descrivere i parametri della funzione main.

La funzione main è il punto di ingresso di un programma C, che viene eseguito automaticamente all'avvio del programma. La sua firma standard è:

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

dove int indica il tipo di dato restituito dalla funzione, argc è un intero che rappresenta il numero di argomenti passati alla riga di comando, e argv è un puntatore ad un array di stringhe che rappresenta gli argomenti passati alla riga di comando.

Il primo elemento dell'array argv (argv[0]) è il nome del programma stesso, mentre gli argomenti effettivi iniziano dall'elemento argv[1].

Ad esempio, se il programma viene eseguito con il comando ./program arg1 arg2 arg3, allora argc sarà uguale a 4 e argv conterrà i seguenti valori:

argv[0] = "./program"

argv[1] = "arg1"

argv[2] = "arg2"

argv[3] = "arg3"

È importante notare che argv è un array di puntatori a stringhe, ovvero ogni elemento dell'array è un puntatore ad un'area di memoria contenente una stringa di caratteri (terminata da un carattere nullo). Questo significa che ogni elemento di argv può essere manipolato come una stringa di caratteri.

Inoltre, è possibile definire un terzo parametro opzionale nella firma della funzione main, chiamato envp, che rappresenta l'array delle variabili d'ambiente. La firma completa in questo caso sarebbe:

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

dove envp è un puntatore ad un array di stringhe che rappresenta le variabili d'ambiente del programma.

19. Cosa è un tipo enumerativo in C? Fornire un esempio.

Un tipo enumerativo, o enum, in C è un tipo di dato che permette di definire un insieme di costanti intere. Le costanti di un tipo enumerativo possono essere utilizzate come variabili di tipo int, e sono elencate come una lista di identificatori, separati da virgole, all'interno delle parentesi graffe dell'enum.

Ecco un esempio di definizione di un tipo enumerativo in C:

```
enum giorni_settimana {
   LUNEDI,
   MARTEDI,
   MERCOLEDI,
   GIOVEDI,
   VENERDI,
   SABATO,
   DOMENICA
};
```

In questo esempio, abbiamo definito un tipo enumerativo chiamato "giorni_settimana" che contiene sette costanti intere elencate come identificatori. Questo enum può essere utilizzato come un tipo di variabile per rappresentare i giorni della settimana. Ad esempio, possiamo dichiarare una variabile di tipo "giorni_settimana" come segue:

```
enum giorni_settimana oggi = MARTEDI;
```

In questo caso, abbiamo dichiarato una variabile di tipo "giorni_settimana" chiamata "oggi" e l'abbiamo inizializzata con la costante intera "MARTEDI".

20. Cosa è una struttura, e quali sono le più evidenti differenze di utilizzo rispetto ad un array? Fornire un esempio di struttura e un esempio di array.

In programmazione, una struttura è un tipo di dato composto che permette di raggruppare variabili di diversi tipi in un'unica entità. Queste variabili, chiamate membri, possono essere di tipo primitivo come interi, booleani o caratteri, oppure di tipo complesso come altre strutture.

Le principali differenze tra una struttura e un array sono le seguenti:

Un array è una collezione di elementi dello stesso tipo, mentre una struttura può contenere membri di diversi tipi.

In un array, gli elementi sono acceduti attraverso un indice numerico intero, mentre in una struttura, i membri sono acceduti attraverso il loro nome.

Gli elementi di un array sono allocati in modo contiguo in memoria, mentre i membri di una struttura possono essere allocati in posizioni diverse della memoria.

Ecco un esempio di definizione di una struttura in linguaggio C:

```
struct Persona {
   string nome;
   int eta;
   bool genere;
};
```

In questo esempio, abbiamo definito una struttura chiamata "Persona" che ha tre membri: "nome" di tipo stringa, "eta" di tipo intero e "genere" di tipo booleano. Questa struttura può essere utilizzata per rappresentare le informazioni di una persona, come il suo nome, la sua età e il suo genere.

Ecco invece un esempio di definizione di un array:

```
int numeri[5] = {1, 2, 3, 4, 5};
```

In questo esempio, abbiamo definito un array chiamato "numeri" che contiene cinque elementi di tipo intero. Gli elementi dell'array sono stati inizializzati con i valori 1, 2, 3, 4 e 5. Questo array può essere utilizzato per memorizzare una sequenza di numeri.

21. Cosa è una struttura anonima e come si utilizza? Fornire un esempio di definizione e utilizzo.

In programmazione, una struttura anonima è una struttura che non ha un nome definito. Invece, viene definita direttamente all'interno del codice, senza specificare un nome per la struttura stessa. Le strutture anonime sono utilizzate per raggruppare variabili correlate in modo da poter accedere ai loro membri in modo più semplice.

Ecco un esempio di definizione di una struttura anonima in linguaggio C:

```
struct {
   int x;
   int y;
} punto;
```

In questo esempio, abbiamo definito una struttura anonima che ha due membri: "x" e "y", entrambi di tipo intero. La struttura anonima è stata assegnata alla variabile "punto".

Possiamo utilizzare la struttura anonima per creare un oggetto "punto" con i valori "x" e "y" specificati:

punto.x = 10;

punto.y = 20;

In questo modo, abbiamo assegnato il valore 10 alla variabile "x" dell'oggetto "punto" e il valore 20 alla variabile "y" dell'oggetto "punto".

Le strutture anonime sono utili quando si desidera raggruppare variabili correlate in modo da poter accedere ai loro membri in modo più semplice senza dover definire una struttura con un nome separato. Tuttavia, poiché le strutture anonime non hanno un nome definito, non possono essere utilizzate in modo indipendente e sono disponibili solo all'interno del contesto in cui sono state definite.

22. Come si può definire un nuovo tipo di dato? Fornire un esempio.

In molti linguaggi di programmazione, è possibile definire un nuovo tipo di dato utilizzando la parola chiave "typedef". La sintassi generale per definire un nuovo tipo di dato è la seguente:

typedef vecchio_tipo nuovo_tipo;

dove "vecchio_tipo" è il tipo di dato esistente che si desidera utilizzare come base per il nuovo tipo di dato e "nuovo_tipo" è il nome del nuovo tipo di dato.

Ad esempio, in linguaggio C, possiamo definire un nuovo tipo di dato "booleano" che rappresenta un valore booleano (vero o falso) utilizzando la seguente definizione di typedef:

typedef enum {falso, vero} booleano;

In questo esempio, abbiamo definito il nuovo tipo di dato "booleano" utilizzando un enum (enumerazione) che ha due costanti: "falso" e "vero". In questo modo, possiamo utilizzare il nuovo tipo di dato "booleano" al posto del tipo di dato esistente "enum" per rappresentare valori booleani in modo più leggibile e semplice.

Ad esempio, possiamo utilizzare il nuovo tipo di dato "booleano" per definire una variabile booleana come segue:

booleano condizione = vero;

In questo modo, abbiamo definito la variabile "condizione" come una variabile booleana utilizzando il nuovo tipo di dato "booleano". Possiamo assegnare alla variabile "condizione" il valore "vero" utilizzando la costante definita nella definizione di typedef.

La definizione di un nuovo tipo di dato può semplificare la lettura e la scrittura del codice, poiché consente di utilizzare nomi di tipo più significativi e leggibili per rappresentare i dati nel programma.

23. Dato un puntatore a una struttura, come si accede ai membri della struttura stessa?

Per accedere ai membri di una struttura attraverso un puntatore, in genere si utilizza l'operatore "->" (punto-e-freccia). L'operatore "->" viene utilizzato per dereferenziare il puntatore e accedere ai membri della struttura.

Ad esempio, supponiamo di avere la seguente definizione di una struttura in linguaggio C:

```
struct Persona {
   char nome[50];
   int eta;
   float altezza;
};
```

Possiamo definire un puntatore a questa struttura come segue:

```
struct Persona *puntatore_persona;
```

Per accedere ai membri della struttura utilizzando il puntatore, possiamo utilizzare l'operatore "->" seguito dal nome del membro della struttura. Ad esempio, per accedere al membro "nome" della struttura attraverso il puntatore "puntatore_persona", possiamo scrivere:

```
printf("Nome: %s\n", puntatore_persona->nome);
```

In questo modo, l'operatore "->" dereferenzia il puntatore e accede al membro "nome" della struttura puntata.

Analogamente, per accedere agli altri membri della struttura utilizzando il puntatore, possiamo utilizzare l'operatore "->" seguito dal nome del membro. Ad esempio, per accedere al membro "eta" della struttura attraverso il puntatore "puntatore_persona", possiamo scrivere:

```
printf("Età: %d\n", puntatore_persona->eta);
```

E così via per tutti i membri della struttura

24. Descrivere le funzioni malloc, calloc e free, inclusa la definizione del prototipo della funzione.

In linguaggio C, le funzioni malloc, calloc e free sono utilizzate per gestire la memoria dinamica del programma.

La funzione malloc (memory allocation) viene utilizzata per allocare un blocco di memoria di dimensione specifica durante l'esecuzione del programma. La sintassi della funzione malloc è la seguente:

void *malloc(size_t dimensione);

La funzione restituisce un puntatore a un blocco di memoria di dimensione "dimensione" in byte. Il tipo restituito dalla funzione malloc è un puntatore generico void, che può essere assegnato a un puntatore di qualsiasi tipo di dato.

Ad esempio, per allocare un blocco di memoria di 100 byte, possiamo scrivere:

char *puntatore = (char *)malloc(100);

In questo esempio, abbiamo allocato un blocco di memoria di 100 byte utilizzando la funzione malloc e abbiamo assegnato il puntatore restituito dalla funzione a un puntatore di tipo char.

La funzione calloc (memory allocation and clear) è simile alla funzione malloc, ma a differenza di malloc, calloc inizializza tutti i byte del blocco di memoria allocato a 0. La sintassi della funzione calloc è la seguente:

void *calloc(size_t numero_elementi, size_t dimensione_elemento);

La funzione restituisce un puntatore a un blocco di memoria che può contenere "numero_elementi" elementi, ognuno di "dimensione_elemento" byte. Il tipo restituito dalla funzione calloc è un puntatore generico void, che può essere assegnato a un puntatore di qualsiasi tipo di dato.

Ad esempio, per allocare un blocco di memoria di 10 interi inizializzati a 0, possiamo scrivere:

int *puntatore = (int *)calloc(10, sizeof(int));

In questo esempio, abbiamo allocato un blocco di memoria che può contenere 10 interi, ognuno di 4 byte (la dimensione di un intero in byte è fornita dalla funzione sizeof), utilizzando la funzione calloc e abbiamo assegnato il puntatore restituito dalla funzione a un puntatore di tipo int.

La funzione free viene utilizzata per liberare la memoria precedentemente allocata con malloc o calloc. La sintassi della funzione free è la seguente:

void free(void *puntatore);

La funzione riceve come argomento il puntatore alla memoria precedentemente allocata con malloc o calloc e libera la memoria associata a quel puntatore.

Ad esempio, per liberare il blocco di memoria allocato in precedenza con la funzione malloc, possiamo scrivere:

```
free(puntatore);
```

In questo esempio, abbiamo liberato la memoria precedentemente allocata con la funzione malloc utilizzando la funzione free e il puntatore "puntatore".

25. Come si può visualizzare a schermo l'indirizzo in memoria di i) una variabile nello stack; ii) una variabile globale inizializzata; ii) una variabile nello heap?

i) La visualizzazione dell'indirizzo in memoria di una variabile nello stack dipende dal linguaggio di programmazione utilizzato e dall'ambiente di sviluppo. Tuttavia, in generale, è possibile ottenere l'indirizzo di una variabile utilizzando l'operatore "&" in molti linguaggi di programmazione.

Ad esempio, in linguaggio C, per visualizzare l'indirizzo in memoria di una variabile nello stack, è possibile utilizzare il seguente codice:

```
#include <stdio.h>

int main() {
  int var = 5;
  printf("L'indirizzo di memoria della variabile var è: %p\n", &var);
  return 0;
}
```

In questo esempio, viene utilizzata la funzione printf() per stampare l'indirizzo in memoria della variabile var utilizzando l'operatore "&". L'opzione di formato %p viene utilizzata per visualizzare l'indirizzo in memoria in formato esadecimale.

ii) L'opzione di formato %p viene utilizzata per visualizzare l'indirizzo in memoria in formato esadecimale.

iii) Tuttavia, il concetto di base è lo stesso della variabile nello stack o della variabile globale. In molti linguaggi di programmazione, come ad esempio C e C++, per allocare una variabile nello heap, è necessario utilizzare una funzione come malloc() o new, che restituisce un puntatore all'indirizzo di memoria della variabile allocata.

Per visualizzare l'indirizzo in memoria di una variabile nello heap allocata, quindi, è possibile stampare il valore del puntatore restituito dalla funzione di allocazione utilizzando l'operatore "&".

26. Nel caso della domanda precedente, qual è l'effetto della ASLR, e come si può disabilitare?

La ASLR (Address Space Layout Randomization) è una tecnica di sicurezza utilizzata nei sistemi operativi moderni per rendere più difficile agli attaccanti sfruttare vulnerabilità di sicurezza. L'ASLR agisce randomizzando l'indirizzo di base dello spazio degli indirizzi di un processo in modo che sia diverso ad ogni avvio del processo, rendendo più difficile ai malintenzionati prevedere l'indirizzo esatto della memoria in cui si trovano le variabili, le librerie e il codice del programma.

Quindi, se si utilizza la ASLR, il risultato dell'operazione per visualizzare l'indirizzo in memoria di una variabile sarà differente ad ogni avvio del processo.

Per disabilitare la ASLR su un sistema operativo Linux, è possibile utilizzare il comando sysctl con l'opzione kernel.randomize_va_space. Ecco un esempio di come disabilitare l'ASLR su un sistema Ubuntu:

Aprire una finestra del terminale e digitare il seguente comando per disabilitare l'ASLR:

sudo sysctl -w kernel.randomize_va_space=0

Digitare la password di amministratore quando richiesto.

Dopo aver disabilitato l'ASLR, il processo di visualizzazione dell'indirizzo in memoria di una variabile nello heap dovrebbe fornire risultati costanti ad ogni avvio del processo.

Tuttavia, disabilitare la ASLR può ridurre la sicurezza del sistema e potrebbe essere controindicato in alcuni ambienti. Pertanto, è importante utilizzare questa opzione solo se strettamente necessario e dopo aver compreso i rischi potenziali. Inoltre, l'abilitazione o la disabilitazione dell'ASLR potrebbe richiedere i privilegi di amministratore del sistema.

27. Descrivere, tramite un esempio, come si può costruire una lista utilizzando le strutture e l'allocazione di memoria.

Ecco un esempio di come costruire una lista concatenata utilizzando le strutture e l'allocazione di memoria in linguaggio C:

```
#include <stdio.h>
#include <stdlib.h>

// Definizione della struttura Nodo
struct Nodo {
   int valore;
```

```
    struct Nodo* prossimo;
};

// Funzione per creare un nuovo Nodo
struct Nodo* creaNodo(int valore) {
    struct Nodo* nuovoNodo = (struct Nodo*)malloc(sizeof(struct Nodo));
    nuovoNodo->valore = valore;
    nuovoNodo->prossimo = NULL;
    return nuovoNodo;
}

int main() {
    // Creazione della lista
    struct Nodo* testa = creaNodo(1);
    struct Nodo* secondo = creaNodo(2);
    struct Nodo* terzo = creaNodo(3);

    testa->prossimo = secondo;
    secondo->prossimo = terzo;

    // Stampa della lista
    struct Nodo* corrente = testa;
    while (corrente != NULL) {
        printf("%d ", corrente->valore);
        corrente = corrente->prossimo;
    }
    printf("\n");

    // Deallocazione della memoria
    free(testa);
    free(secondo);
    free(terzo);
```

```
    return 0;
}
```

In questo esempio, viene definita la struttura Nodo, che rappresenta un nodo della lista concatenata e contiene un intero valore e un puntatore prossimo al nodo successivo nella lista.

Viene quindi definita la funzione creaNodo che alloca dinamicamente un nuovo nodo e restituisce un puntatore ad esso.

Nel main, vengono creati tre nodi utilizzando la funzione creaNodo e vengono collegati tra loro impostando i puntatori prossimo.

Viene quindi eseguita una iterazione della lista per stampare i valori dei nodi.

Infine, viene deallocata la memoria utilizzata dalla lista utilizzando la funzione free per ogni nodo.

In questo modo, viene creata una lista concatenata utilizzando le strutture e l'allocazione dinamica della memoria in C, consentendo di manipolare la lista in modo dinamico, ad esempio, aggiungendo o rimuovendo nodi in qualsiasi momento.

28. Quali sono le sei modalità di apertura di un file?

In informatica, ci sono sei modalità di apertura di un file che possono essere utilizzate durante l'accesso ai file. Queste modalità sono:

"r" - Modalità di sola lettura: il file viene aperto in modalità di sola lettura e non può essere modificato. Se il file non esiste, il tentativo di apertura fallirà.

"w" - Modalità di scrittura: il file viene aperto per la scrittura. Se il file non esiste, viene creato. Se il file esiste già, il suo contenuto precedente viene eliminato e il nuovo contenuto viene scritto al suo posto.

"a" - Modalità di append: il file viene aperto per la scrittura in append, ovvero i dati scritti vengono aggiunti alla fine del file esistente senza eliminare il contenuto precedente.

"r+" - Modalità di lettura e scrittura: il file viene aperto per la lettura e la scrittura, consentendo di modificare il contenuto esistente.

"w+" - Modalità di scrittura e lettura: il file viene aperto per la scrittura e la lettura. Se il file esiste già, il contenuto precedente viene eliminato e il nuovo contenuto viene scritto al suo posto.

"a+" - Modalità di append e lettura: il file viene aperto per la lettura e la scrittura in append. I dati scritti vengono aggiunti alla fine del file esistente senza eliminare il contenuto precedente.

Queste modalità possono essere utilizzate per controllare l'accesso ai file e le operazioni eseguibili sui file, come la lettura, la scrittura e l'aggiunta di dati.

29. Qual è il ruolo della funzione fflush?

La funzione fflush() in C viene utilizzata per "svuotare" il buffer di output associato a uno stream. In altre parole, quando si scrive qualcosa su uno stream, come ad esempio su stdout (lo standard output), le informazioni possono essere mantenute temporaneamente in un buffer di output invece di essere inviate immediatamente al dispositivo di output. Questo è fatto per migliorare le prestazioni, in quanto inviare dati al dispositivo di output può essere un'operazione costosa in termini di tempo.

Tuttavia, in alcuni casi può essere necessario che le informazioni siano immediatamente inviate al dispositivo di output. In questi casi, la funzione fflush() può essere chiamata per svuotare il buffer e inviare i dati al dispositivo di output. Ciò è particolarmente utile quando si scrive su file, in quanto garantisce che i dati scritti siano salvati sul file prima di chiuderlo.

Ad esempio, se si desidera scrivere qualcosa su stdout e assicurarsi che venga immediatamente visualizzato a video, si potrebbe utilizzare la funzione fflush() per forzare la scrittura del buffer su stdout. Ecco un esempio di utilizzo:

```
printf("Scrivo qualcosa su stdout...");
fflush(stdout);  // forza l'invio del buffer su stdout
```

In generale, la funzione fflush() viene utilizzata quando si desidera controllare esplicitamente la scrittura dei dati sul dispositivo di output.

30. Descrivere il comportamento delle funzioni ftell ed fseek, possibilmente aiutandosi con un esempio.

Le funzioni ftell() e fseek() sono utilizzate in C per spostarsi in un file e leggere o scrivere in una posizione specifica.

La funzione ftell() viene utilizzata per ottenere la posizione corrente del cursore di lettura/scrittura all'interno di un file, ovvero per determinare l'offset dalla posizione iniziale del file. Questa funzione restituisce un valore di tipo long int che rappresenta la posizione corrente nel file. Ad esempio:

```
FILE* fp;
long int pos;

fp = fopen("file.txt", "r");
if (fp == NULL) {
  printf("Impossibile aprire il file.");
  exit(1);
}

// Sposta il cursore di lettura a 10 byte dalla posizione iniziale del file
fseek(fp, 10, SEEK_SET);

// Ottiene la posizione corrente del cursore di lettura
pos = ftell(fp);

printf("La posizione corrente del cursore di lettura è %ld\n", pos);

fclose(fp);
```

In questo esempio, viene aperto il file "file.txt" in modalità di sola lettura e viene spostato il cursore di lettura a 10 byte dalla posizione iniziale del file utilizzando la funzione fseek(). Successivamente, la posizione corrente del cursore di lettura viene ottenuta utilizzando la funzione ftell() e viene stampata a schermo.

La funzione fseek() invece viene utilizzata per spostare il cursore di lettura/scrittura all'interno di un file in una posizione specifica. Questa funzione ha tre argomenti: il puntatore al file, l'offset e la posizione da cui iniziare lo spostamento all'interno del file. L'argomento offset rappresenta il numero di byte da spostarsi rispetto alla posizione specificata dalla costante SEEK_SET (inizio del file), SEEK_CUR (posizione corrente del cursore di lettura/scrittura) o SEEK_END (fine del file). Ad esempio:

```
FILE* fp;
char buffer[100];

fp = fopen("file.txt", "r+");
if (fp == NULL) {
  printf("Impossibile aprire il file.");
  exit(1);
}

// Sposta il cursore di lettura/scrittura a 10 byte dalla fine del file
fseek(fp, -10, SEEK_END);

// Scrive "nuovi dati" nella posizione corrente del cursore di lettura/scrittura
fwrite("nuovi dati", sizeof(char), 10, fp);

// Sposta il cursore di lettura/scrittura all'inizio del file
fseek(fp, 0, SEEK_SET);

// Legge i primi 100 byte del file
fread(buffer, sizeof(char), 100, fp);

printf("Contenuto del file: %s\n", buffer);

fclose(fp);
```

In questo esempio, viene aperto il file "file.txt" in modalità di lettura/scrittura utilizzando fopen(). Successivamente, il cursore di lettura/scrittura viene spostato di 10 byte dalla fine del file utilizzando la funzione fseek().

31. Descrivere gli effetti e le differenze delle opzioni di compilazione -E e -c , per il compilatore GNU GCC

Le opzioni di compilazione -E e -c sono due delle opzioni supportate dal compilatore GNU GCC.

L'opzione -E fa sì che il compilatore si comporti come un preprocessore di codice, ovvero genera l'output del preprocessore C e lo scrive sulla standard output o su un file specificato dall'utente. L'output generato include il codice sorgente originale con le direttive di preprocessing risolte, inclusi i file di header, le macro sostituite, le condizioni di compilazione, ecc. L'opzione -E è spesso utilizzata per verificare come il preprocessore elabora il codice sorgente o per generare una versione pre-elaborata del codice sorgente per la diagnostica.

Esempio di utilizzo dell'opzione -E:

gcc -E main.c -o main.i

In questo esempio, il file main.c viene preprocessato e l'output del preprocessore viene scritto nel file main.i.

L'opzione -c, invece, fa sì che il compilatore produca un file oggetto compilato dal codice sorgente, ma non esegue la fase di linking. Questo significa che il codice sorgente viene compilato in un file oggetto che contiene il codice macchina e le informazioni sulle funzioni e le variabili definite nel file sorgente. Tuttavia, non viene creato un file eseguibile finale, poiché mancano ancora molte funzioni e variabili definite in altri file di codice sorgente o librerie esterne. L'opzione -c è spesso utilizzata per compilare il codice sorgente in modo da creare un file oggetto, che può essere successivamente unito ad altri file oggetto per creare un file eseguibile finale tramite la fase di linking.

Esempio di utilizzo dell'opzione -c:

gcc -c main.c -o main.o

In questo esempio, il file main.c viene compilato e il risultato della compilazione viene scritto nel file oggetto main.o.

In sintesi, l'opzione -E è utilizzata per generare l'output del preprocessore C, mentre l'opzione -c è utilizzata per generare un file oggetto compilato dal codice sorgente. La principale differenza tra le due opzioni è che -E produce un file di testo, mentre -c produce un file binario. Inoltre, -E non genera un file oggetto compilato, mentre -c lo fa, ma non esegue la fase di linking.

32. Che funzionalità realizza l’utility objdump?

L'utility "objdump" è un programma utilizzato per analizzare i file oggetto generati da un compilatore durante la compilazione di un programma. Questo strumento offre diverse funzionalità, tra cui:

Disassemblaggio: objdump può disassemblare il contenuto di un file oggetto e mostrarlo come codice assembly, inclusi i simboli e le sezioni.

Visualizzazione delle informazioni sul file: objdump può mostrare le informazioni sul file oggetto, tra cui la tabella dei simboli, le sezioni, le intestazioni di sezione e le informazioni di debug.

Analisi del file eseguibile: objdump può essere utilizzato per analizzare un file eseguibile e mostrare le informazioni sulle sezioni del programma, come la tabella dei simboli, il codice di avvio e l'offset delle sezioni.

Verifica dell'uso delle librerie condivise: objdump può essere utilizzato per verificare quali librerie condivise sono utilizzate da un file eseguibile o da una libreria condivisa.

In sintesi, objdump è uno strumento utile per l'analisi e il debugging di file oggetto e file eseguibili.

33. Cosa sono le direttive del preprocessore, e come si includono nel nostro codice? Riportare un esempio di direttiva.

Le direttive del preprocessore sono istruzioni che vengono processate dal preprocessore durante la fase di precompilazione di un programma. Le direttive del preprocessore sono utilizzate per controllare il comportamento del preprocessore stesso, per definire costanti simboliche, per includere file di intestazione e per condizionare l'inclusione di parti di codice in base a espressioni booleane.

Le direttive del preprocessore iniziano con il carattere '#' e sono scritte prima del codice sorgente del programma. Ad esempio, la direttiva "#include" viene utilizzata per includere un file di intestazione nel codice sorgente. L'esempio seguente mostra come includere il file di intestazione "stdio.h" nel codice sorgente:

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

Un'altra direttiva comune è "#define", che viene utilizzata per definire costanti simboliche. Ad esempio, il seguente codice definisce una costante simbolica chiamata "PI" con valore 3.14159:

```
#define PI 3.14159
```

Le direttive del preprocessore sono una potente funzionalità del linguaggio di programmazione C e sono utilizzate comunemente per semplificare il codice sorgente e migliorare la leggibilità del programma.

34. Cosa è un file di intestazione, e come si può distinguere tra file di intestazione in percorsi definiti dall'utente o dalla configurazione del compilatore?

Un file di intestazione (header file) è un file di testo che contiene dichiarazioni di funzioni, macro, costanti e tipi di dati utilizzati in un programma. Il suo scopo è quello di fornire informazioni al compilatore sulle funzioni e le variabili che il programma utilizzerà, consentendo così al compilatore di risolvere i riferimenti durante la compilazione.

I file di intestazione possono essere forniti dall'utente o dalla configurazione del compilatore. I file di intestazione forniti dall'utente si trovano in percorsi specificati dall'utente stesso, mentre i file di intestazione forniti dal compilatore si trovano in percorsi specificati dalla configurazione del compilatore.

Per distinguere tra file di intestazione in percorsi definiti dall'utente o dalla configurazione del compilatore, è possibile utilizzare il prefisso del percorso. Ad esempio, i file di intestazione forniti dal compilatore di solito hanno un prefisso come <nome_libreria>/header.h, mentre i file di intestazione dell'utente potrebbero essere in una directory personalizzata, ad esempio ./include/header.h. Inoltre, alcuni compilatori forniscono una lista di directory di ricerca per i file di intestazione, il che consente di distinguere facilmente tra i due tipi di file.

35. Descrivere la direttiva define, e fornire almeno un esempio di utilizzo.

La direttiva define è una funzionalità del preprocessore del linguaggio di programmazione C e C++ che consente di definire macro di sostituzione. Una macro di sostituzione è un'istruzione che il preprocessore sostituisce con un valore o un'espressione specificati prima della compilazione del programma.

La sintassi della direttiva define è la seguente:

#define nome_macro valore_macro

dove nome_macro è il nome della macro di sostituzione e valore_macro è il valore o l'espressione che verrà sostituita.

Ecco un esempio di utilizzo della direttiva define:

```
#include <stdio.h>

#define PI 3.14159

int main()
{
   double raggio = 2.5;
   double area = PI * raggio * raggio;

   printf("L'area del cerchio di raggio %f è %f\n", raggio, area);
   return 0;
}
```

In questo esempio, la direttiva define viene utilizzata per definire la macro PI come 3.14159. Successivamente, la macro PI viene utilizzata nell'espressione per calcolare l'area di un cerchio di raggio

raggio. Durante la compilazione, il preprocessore sostituirà la macro PI con il valore 3.14159, generando il codice equivalente a area = 3.14159 * raggio * raggio;.

36. Descrivere le direttive #if, #else, #elif e #endif

Le direttive #if, #else, #elif e #endif sono utilizzate nel preprocessore del linguaggio di programmazione C e C++ per creare blocchi condizionali di codice. Questi blocchi di codice consentono di includere o escludere parti del codice durante la compilazione, in base al valore di una o più espressioni booleane.

La sintassi delle direttive condizionali è la seguente:

```
#if espressione_booleana
   /* Codice che viene incluso se l'espressione è vera */
#elif espressione_booleana2
   /* Codice che viene incluso se l'espressione è falsa, ma espressione_booleana2 è vera */
#else
   /* Codice che viene incluso se nessuna delle espressioni è vera */
#endif
```

Dove espressione_booleana è un'istruzione booleana che può contenere operatori di confronto, booleani e macro definite in precedenza. La direttiva #elif viene utilizzata per specificare una seconda condizione, che viene valutata solo se la condizione precedente è falsa. La direttiva #else viene utilizzata per specificare un blocco di codice da eseguire se nessuna delle condizioni precedenti è vera.

Ecco un esempio di utilizzo delle direttive condizionali:

```
#include <stdio.h>

#define DEBUG 1

int main()
{
   int x = 5;

   #if DEBUG
      printf("x = %d\n", x);
   #else
```

```
    printf("Programma compilato senza opzione di debug.\n");
  #endif

  return 0;
}
```

In questo esempio, la macro DEBUG è definita come 1. Durante la compilazione, il preprocessore valuterà la condizione #if DEBUG e sostituirà il codice all'interno del blocco se la macro DEBUG è definita e ha un valore diverso da 0. Se la macro non è definita o ha un valore 0, il codice all'interno del blocco non verrà incluso nella compilazione e verrà invece incluso il codice all'interno del blocco #else.

37. Descrivere la direttiva #ifdef, e presentare il suo possibile utilizzo per abilitare o disabilitare codice di debug.

La direttiva #ifdef è una funzionalità del preprocessore del linguaggio di programmazione C e C++ che consente di verificare se una macro di sostituzione è stata definita con #define. La direttiva #ifdef è simile alla direttiva #if, ma valuta solo la presenza o l'assenza di una macro, piuttosto che il valore di una espressione booleana.

La sintassi della direttiva #ifdef è la seguente:

```
#ifdef nome_macro
  /* Codice che viene incluso se la macro è definita */
#else
  /* Codice che viene incluso se la macro non è definita */
#endif
```

Dove nome_macro è il nome della macro che si vuole verificare se è stata definita con la direttiva #define. Il blocco di codice all'interno della direttiva #ifdef verrà incluso nella compilazione solo se la macro è stata definita, altrimenti verrà incluso il blocco di codice all'interno della direttiva #else.

Ecco un esempio di utilizzo della direttiva #ifdef per abilitare o disabilitare il codice di debug:

```
#include <stdio.h>

#define DEBUG

int main()
{
```

```
    int x = 5;

    #ifdef DEBUG
        printf("Il valore di x è %d\n", x);
    #endif

    return 0;
}
```

In questo esempio, viene definita la macro DEBUG con la direttiva #define. Durante la compilazione, il preprocessore verificherà se la macro DEBUG è stata definita con la direttiva #ifdef. Se la macro è stata definita, il blocco di codice all'interno della direttiva #ifdef verrà incluso nella compilazione, consentendo la stampa del valore di x. Se la macro non è stata definita, il blocco di codice verrà escluso dalla compilazione. In questo modo, è possibile abilitare o disabilitare facilmente il codice di debug senza dover commentare o eliminare manualmente parti del codice.

38. Quali sono i passi, in assenza di Makefile, per compilare un programma suddiviso in più file sorgenti?

Se non si ha un Makefile e si desidera compilare un programma suddiviso in più file sorgenti, è possibile seguire questi passaggi:

Assicurarsi di avere tutti i file sorgenti necessari per il programma. Ad esempio, se il programma utilizza un file header (.h) e un file di implementazione (.c) per ciascun modulo, assicurarsi di avere tutti i file necessari.

Compilare ciascun file sorgente in un file oggetto (.o). Ad esempio, se si ha un file di implementazione denominato "module.c", eseguire il comando "gcc -c module.c" per generare un file oggetto denominato "module.o". Assicurarsi di specificare eventuali opzioni di compilazione necessarie.

Linkare i file oggetto in un eseguibile. Ad esempio, se si ha due file oggetto denominati "module1.o" e "module2.o", eseguire il comando "gcc -o programma module1.o module2.o" per creare un eseguibile denominato "programma". Assicurarsi di specificare eventuali librerie esterne o opzioni di linkaggio necessarie.

Eseguire il programma appena creato. Ad esempio, eseguire il comando "./programma" per avviare il programma.

Questi passaggi possono diventare molto complessi se il programma ha molte dipendenze e richiede molte librerie esterne o opzioni di compilazione. In questo caso, potrebbe essere utile creare un Makefile per automatizzare il processo di compilazione.

39. Descrivere l’effetto o scopo delle opzioni di compilazione per GNU GCC -c, -o,-l,-v,-

D,-I

Ecco una descrizione dell'effetto o dello scopo delle opzioni di compilazione per GNU GCC:

"-c": questa opzione indica a GCC di compilare solo i file sorgente in file oggetto e non eseguire la fase di linking. Viene utilizzata per creare i file oggetto di ciascun modulo del programma.

"-o": questa opzione specifica il nome del file oggetto o dell'eseguibile generato dal compilatore. Ad esempio, se si vuole generare un file oggetto chiamato "modulo.o", si può usare l'opzione "-o modulo.o". Se si vuole generare un eseguibile chiamato "programma", si può usare l'opzione "-o programma".

"-l": questa opzione specifica le librerie esterne necessarie per il linking del programma. Ad esempio, se il programma utilizza la libreria math, si può usare l'opzione "-lm" per linkare la libreria math.

"-v": questa opzione indica a GCC di stampare il progresso della compilazione sulla console. Viene utilizzata per diagnosticare eventuali problemi durante la compilazione.

"-D": questa opzione definisce una macro durante la compilazione. Ad esempio, se si vuole definire la macro "DEBUG" durante la compilazione, si può usare l'opzione "-DDEBUG".

"-I": questa opzione specifica il percorso di inclusione delle directory di ricerca per i file di intestazione. Ad esempio, se si ha un file di intestazione chiamato "header.h" in una directory chiamata "include", si può usare l'opzione "-I./include" per indicare a GCC di cercare il file di intestazione nella directory "include".

In sintesi, queste opzioni sono utilizzate per controllare il comportamento del compilatore durante il processo di compilazione e per specificare informazioni aggiuntive come librerie esterne e percorsi di ricerca dei file di intestazione.

40. Descrivere la struttura tipica di un Makefile.

Un Makefile è un file di testo utilizzato per automatizzare il processo di compilazione di un programma. Una struttura tipica di un Makefile comprende i seguenti elementi:

Dichiarazione delle variabili: la sezione iniziale del Makefile di solito definisce le variabili che saranno utilizzate durante il processo di compilazione. Le variabili possono contenere informazioni come il compilatore da utilizzare, le opzioni di compilazione, i percorsi dei file di inclusione, i nomi dei file sorgente, i nomi dei file oggetto e l'eseguibile finale. Ad esempio, si può definire la variabile CC (compilatore) come "gcc", e la variabile CFLAGS (opzioni di compilazione) come "-Wall -g".

Dichiarazione delle dipendenze: questa sezione indica le dipendenze tra i file sorgente, i file oggetto e l'eseguibile. Ad esempio, se il programma è composto da due file sorgente "main.c" e "module.c", la sezione delle dipendenze indicherà che "main.c" dipende da entrambi i file oggetto "main.o" e "module.o".

Regole di compilazione: questa sezione specifica le regole per compilare i file sorgente in file oggetto e linkare i file oggetto in un eseguibile finale. Ad esempio, si può definire una regola per compilare il file sorgente "main.c" in un file oggetto "main.o" usando il compilatore GCC e le opzioni di compilazione definite nella variabile CFLAGS.

Regole di pulizia: questa sezione indica come eliminare i file oggetto e l'eseguibile finale dopo la compilazione. Questo può essere utile per rimuovere i file temporanei creati durante il processo di compilazione. Ad esempio, si può definire una regola per eliminare tutti i file oggetto e l'eseguibile chiamato "clean".

In sintesi, la struttura tipica di un Makefile comprende la definizione delle variabili, le dipendenze tra i file, le regole di compilazione e le regole di pulizia. Questa struttura permette di automatizzare il processo di compilazione e semplificare la gestione del codice sorgente.

41. Cosa rappresentano tipicamente gli obiettivi all, clean, install in un Makefile?

Gli obiettivi "all", "clean" e "install" sono comuni in un Makefile e rappresentano azioni specifiche che possono essere eseguite durante il processo di compilazione e installazione del software.

"all": questo obiettivo è utilizzato per compilare l'intero progetto. Quando si esegue il comando "make all" o semplicemente "make", il Makefile esegue tutte le regole di compilazione necessarie per creare l'eseguibile finale.

"clean": questo obiettivo è utilizzato per rimuovere tutti i file temporanei generati durante il processo di compilazione. Quando si esegue il comando "make clean", il Makefile elimina tutti i file oggetto, i file di dipendenza e l'eseguibile finale. Questa operazione è utile per ripulire il progetto in preparazione per una nuova compilazione.

"install": questo obiettivo è utilizzato per installare l'eseguibile finale in una posizione predefinita del sistema. Quando si esegue il comando "make install", il Makefile copia l'eseguibile finale in una directory specificata dal sistema, ad esempio "/usr/local/bin". Questo consente agli utenti del sistema di utilizzare l'applicazione senza dover accedere alla directory di compilazione del progetto.

In sintesi, gli obiettivi "all", "clean" e "install" sono utilizzati per automatizzare il processo di compilazione, ripulire il progetto e installare l'applicazione finale in una posizione comune del sistema. Questi obiettivi semplificano la gestione del codice sorgente e facilitano la distribuzione dell'applicazione agli utenti finali.

42. Qual è lo scopo del comando di shell ulimit? Fornire un esempio di utilizzo.

Il comando di shell "ulimit" è utilizzato per visualizzare o impostare i limiti delle risorse di sistema per un utente o un processo. I limiti delle risorse di sistema includono la dimensione massima del file che può essere creato, la quantità massima di memoria che può essere utilizzata da un processo, il numero massimo di file che possono essere aperti contemporaneamente, il tempo massimo di CPU per processo, e così via.

Ecco un esempio di utilizzo del comando "ulimit" per impostare il limite di memoria massimo per un processo:

ulimit -v 500000 # imposta il limite di memoria virtuale a 500000 KB (o 500 MB)

Questo comando imposta il limite di memoria virtuale massimo per il processo corrente a 500000 KB. Se il processo tenta di utilizzare più di 500000 KB di memoria virtuale, verrà interrotto e genererà un errore.

È importante notare che il comando "ulimit" è un comando di shell e il limite impostato con esso si applica solo alla sessione corrente della shell. Quando si chiude la shell, il limite impostato con "ulimit" viene ripristinato ai valori predefiniti di sistema.

Il comando "ulimit" è particolarmente utile per evitare che i processi utilizzino troppe risorse di sistema e causino rallentamenti o malfunzionamenti del sistema. Inoltre, il comando "ulimit" può essere utilizzato per verificare i limiti delle risorse di sistema impostati per un utente o un processo specifico.

43. Cosa è un descrittore di file, e a cosa corrispondono i descrittori 0, 1, 2?

In un sistema operativo, un descrittore di file è un numero intero che identifica un file aperto dal sistema. Quando un processo apre un file, il sistema operativo assegna un numero univoco, chiamato descrittore di file, per rappresentare quel file all'interno del processo.

I descrittori di file sono utilizzati per accedere ai file aperti dal processo e per effettuare operazioni di lettura e scrittura sui file. I descrittori di file possono anche rappresentare altre entità, come pipe, socket e dispositivi di rete.

Nei sistemi Unix e Linux, i descrittori di file 0, 1 e 2 sono predefiniti per rappresentare rispettivamente lo standard input (stdin), lo standard output (stdout) e lo standard error (stderr) del processo. Questi descrittori di file sono creati automaticamente quando viene avviato un nuovo processo e sono disponibili per il processo per effettuare operazioni di lettura e scrittura.

Il descrittore di file 0 (stdin) rappresenta lo standard input del processo. Questo descrittore di file è utilizzato per leggere i dati di input da un utente o da un altro processo.

Il descrittore di file 1 (stdout) rappresenta lo standard output del processo. Questo descrittore di file è utilizzato per scrivere i dati di output dal processo a un terminale o a un altro processo.

Il descrittore di file 2 (stderr) rappresenta lo standard error del processo. Questo descrittore di file è utilizzato per scrivere i messaggi di errore dal processo a un terminale o a un altro processo.

I descrittori di file standard sono utilizzati ampiamente nei programmi Unix e Linux per interagire con l'ambiente del sistema e comunicare con l'utente o altri processi.

44. Elencare le principali proprietà di un descrittore di file.

Di seguito sono elencate le principali proprietà di un descrittore di file:

Un descrittore di file è un numero intero che identifica un file aperto dal sistema operativo.

Un descrittore di file può rappresentare diversi tipi di file, tra cui file regolari, dispositivi di blocco e di carattere, pipe e socket.

I descrittori di file sono gestiti dal kernel del sistema operativo.

I descrittori di file vengono assegnati a un processo quando un file viene aperto o creato all'interno del processo.

I descrittori di file possono essere passati da un processo a un altro tramite la comunicazione interprocesso.

I descrittori di file possono essere utilizzati per effettuare operazioni di lettura e scrittura sui file rappresentati dal descrittore.

I descrittori di file hanno un flag di stato che indica se il file è aperto in modalità di lettura, scrittura o lettura-scrittura.

I descrittori di file possono essere utilizzati per la gestione dei processi, come la creazione di processi figli o l'individuazione dei processi attualmente in esecuzione.

I descrittori di file standard (stdin, stdout e stderr) sono predefiniti e disponibili per tutti i processi.

45. Quali sono i principali flag della system call open?

La system call open è utilizzata per aprire un file e restituire un descrittore di file che rappresenta il file aperto. I principali flag utilizzati con la system call open sono:

O_RDONLY: apre il file in modalità di sola lettura.

O_WRONLY: apre il file in modalità di sola scrittura.

O_RDWR: apre il file in modalità di lettura e scrittura.

O_CREAT: se il file non esiste, lo crea.

O_TRUNC: se il file esiste e viene aperto in modalità di scrittura, tronca il file a zero byte.

O_APPEND: scrive i dati alla fine del file.

O_EXCL: se O_CREAT è specificato e il file esiste già, la chiamata a open fallisce.

O_DIRECTORY: apre solo una directory.

O_NOFOLLOW: se il file specificato è un link simbolico, la chiamata a open fallisce.

O_CLOEXEC: chiude il file automaticamente quando il processo viene sostituito da un altro processo.

Questi flag possono essere combinati tra loro usando l'operatore bitwise OR (|) per specificare la modalità di apertura del file. Ad esempio, O_CREAT | O_RDWR apre il file in modalità di lettura e scrittura e lo crea se non esiste.

46. Descrivere il comportamento della system call umask. Fornire un esempio in cui si applica il valore di default.

La system call umask è utilizzata per impostare le autorizzazioni di default per i file creati dal processo chiamante. Quando un nuovo file viene creato, le autorizzazioni predefinite vengono applicate a meno che non vengano specificate autorizzazioni esplicite.

Il valore di default della umask è 022, che significa che vengono esclusi i permessi di scrittura e di esecuzione per il gruppo e gli altri utenti. Questo valore è espresso in notazione ottale, dove il primo

numero rappresenta i permessi del proprietario del file, il secondo il gruppo e il terzo gli altri utenti. Ad esempio, un valore di umask di 022 corrisponde a un valore in notazione ottale di 0022.

Per esempio, se un processo crea un nuovo file con le autorizzazioni predefinite e il valore della umask è 022, il file avrà le seguenti autorizzazioni: -rw-r--r-- (permessi di lettura e scrittura per il proprietario del file, permessi di sola lettura per il gruppo e gli altri utenti).

Il valore della umask può essere modificato durante l'esecuzione del processo utilizzando la system call umask(). Ad esempio, se si desidera impostare la umask su 077 per impedire l'accesso al file da parte del gruppo e degli altri utenti, si può utilizzare il seguente comando nella shell:

umask 077

In questo modo, i file creati dal processo chiamante avranno le autorizzazioni 600 (-rw-------).

47. Descrivere le system call read, write, lseek.

Le system call read, write e lseek sono utilizzate per leggere, scrivere e modificare la posizione corrente all'interno di un file aperto tramite un descrittore di file.

La system call read ha la seguente firma:

ssize_t read(int fd, void *buf, size_t count);

Questa chiamata legge fino a count byte dal file aperto identificato dal descrittore di file fd e li memorizza nel buffer buf. Restituisce il numero di byte effettivamente letti dal file e memorizzati nel buffer. Se la chiamata a read restituisce 0, significa che è stata raggiunta la fine del file. Se la chiamata a read restituisce un valore negativo, si è verificato un errore.

La system call write ha la seguente firma:

ssize_t write(int fd, const void *buf, size_t count);

Questa chiamata scrive count byte dal buffer buf nel file aperto identificato dal descrittore di file fd. Restituisce il numero di byte effettivamente scritti nel file. Se la chiamata a write restituisce un valore negativo, si è verificato un errore.

La system call lseek ha la seguente firma:

off_t lseek(int fd, off_t offset, int whence);

Questa chiamata modifica la posizione corrente all'interno del file aperto identificato dal descrittore di file fd. L'argomento offset indica la nuova posizione all'interno del file, mentre whence indica la posizione di partenza da cui il file deve essere posizionato. Ci sono tre possibili valori per whence:

SEEK_SET: la nuova posizione corrente è offset byte dal principio del file.

SEEK_CUR: la nuova posizione corrente è offset byte dalla posizione corrente.

SEEK_END: la nuova posizione corrente è offset byte dalla fine del file.

La chiamata a lseek restituisce la nuova posizione corrente all'interno del file, misurata in byte dall'inizio del file. Se la chiamata a lseek restituisce un valore negativo, si è verificato un errore.

In sintesi, la system call read viene utilizzata per leggere dati da un file, write per scrivere dati in un file, e lseek per modificare la posizione corrente all'interno del file.

48. Descrivere il comportamento della funzione perror.

La funzione perror è una funzione standard della libreria C che viene utilizzata per stampare un messaggio di errore su stderr, corrispondente all'ultimo errore che si è verificato nel programma.

La firma della funzione perror è la seguente:

void perror(const char *s);

L'argomento s è una stringa che viene utilizzata per pre-pendere il messaggio di errore. Il messaggio di errore effettivo viene stampato a video tramite stderr e dipende dal valore della variabile errno, che viene impostata dalle system call e dalle altre funzioni della libreria C in caso di errore.

In pratica, la funzione perror viene utilizzata per stampare un messaggio di errore significativo quando una system call o una funzione della libreria C restituisce un valore di errore. Il messaggio di errore stampato include il valore di errno e una descrizione dell'errore corrispondente.

Ad esempio, supponiamo di avere il seguente codice:

```
#include <stdio.h>

#include <errno.h>

#include <string.h>

int main() {

   FILE *f = fopen("non_esiste.txt", "r");
```

```
    if (f == NULL) {
        perror("Errore apertura file");
        fprintf(stderr, "Errore %d: %s\n", errno, strerror(errno));
        return 1;
    }
    fclose(f);
    return 0;
}
```

In questo esempio, il programma cerca di aprire un file che non esiste, e quindi la chiamata a fopen restituisce un valore nullo e imposta il valore di errno a un valore specifico che indica l'errore. La funzione perror viene utilizzata per stampare un messaggio di errore significativo, mentre la funzione strerror viene utilizzata per ottenere una descrizione dell'errore corrispondente.

49. Descrivere cosa sono gli hard link, incluso come è possibile visualizzare gli hard link di un file.

Gli hard link sono un tipo di collegamento simbolico (o link) che si può creare in un sistema di file. A differenza dei cosiddetti "soft link", o "link simbolici", gli hard link sono delle vere e proprie copie di un file esistente, e condividono con esso la stessa area di memorizzazione sul disco.

In pratica, quando si crea un hard link, si crea un nuovo nome per un file esistente. La creazione di un hard link non comporta la creazione di una nuova copia del file, ma piuttosto la creazione di un collegamento diretto ad un file esistente. In questo modo, è possibile accedere allo stesso contenuto di un file utilizzando diversi nomi di file.

Per visualizzare gli hard link di un file in un sistema Unix-like, si può utilizzare il comando ls -l seguito dal nome del file. Il numero di hard link di un file viene visualizzato nella seconda colonna della lista di file. Ad esempio, se si desidera visualizzare il numero di hard link del file myfile.txt, si può utilizzare il seguente comando:

```
$ ls -l myfile.txt
-rw-r--r-- 2 user user 1024 Mar 30 13:47 myfile.txt
```

In questo esempio, il file myfile.txt ha due hard link, come indicato dal valore "2" nella seconda colonna della lista di file. È importante notare che il numero di hard link di un file non è lo stesso del numero di nomi di file che si riferiscono allo stesso file, in quanto ogni file può avere un solo nome di file (o pathname) primario, ma può avere molti nomi di file alternativi tramite gli hard link.

50. Descrivere come si può creare e rimuovere un hard link, sia tramite chiamata di sistema che tramite shell.

In Unix e sistemi operativi simili, un hard link è un collegamento fisico tra un file e un altro nome di file, che fa riferimento agli stessi dati del file. In altre parole, un hard link è una copia di un file che non occupa spazio aggiuntivo su disco.

Ecco come creare e rimuovere un hard link tramite chiamata di sistema e tramite la shell:

Creare un hard link tramite chiamata di sistema:

La chiamata di sistema per creare un hard link è link(). Il formato della funzione è il seguente:

int link(const char *oldpath, const char *newpath);

dove oldpath è il percorso del file esistente che si desidera collegare, e newpath è il percorso e il nome del nuovo link.

Esempio di creazione di un hard link tramite chiamata di sistema in C:

```
#include <unistd.h>

int main() {
   const char* oldpath = "/path/to/oldfile.txt";
   const char* newpath = "/path/to/newfile.txt";
   int result = link(oldpath, newpath);
   if (result == -1) {
      perror("link");
      return 1;
   }
   return 0;
}
```

Creare un hard link tramite shell:

Per creare un hard link tramite shell, è possibile utilizzare il comando ln. Il formato del comando è il seguente:

ln [OPZIONI] NOME_FILE_ORIGINE NOME_LINK

dove NOME_FILE_ORIGINE è il nome del file esistente che si desidera collegare e NOME_LINK è il nome del nuovo link.

Esempio di creazione di un hard link tramite shell:

ln /path/to/oldfile.txt /path/to/newfile.txt

Rimuovere un hard link tramite chiamata di sistema:

La chiamata di sistema per rimuovere un hard link è unlink(). Il formato della funzione è il seguente:

int unlink(const char *path);

dove path è il percorso del file link da rimuovere.

Esempio di rimozione di un hard link tramite chiamata di sistema in C:

```
#include <unistd.h>

int main() {
   const char* linkpath = "/path/to/link.txt";
   int result = unlink(linkpath);
   if (result == -1) {
      perror("unlink");
      return 1;
   }
   return 0;
}
```

Rimuovere un hard link tramite shell:

Per rimuovere un hard link tramite shell, è possibile utilizzare il comando rm. Il formato del comando è il seguente:

rm NOME_LINK

dove NOME_LINK è il nome del link da rimuovere.

Esempio di rimozione di un hard link tramite shell:

rm /path/to/newfile.txt

51. Descrivere la differenza tra hard link e soft link.

In un sistema di file, un "link" è un riferimento a un file o a una directory. Ci sono due tipi di link: hard link e soft link (o symbolic link).

Hard link: un hard link è un secondo nome per lo stesso file sul sistema di file. In altre parole, è un riferimento diretto ai dati del file, piuttosto che a una copia del file. Tutti gli hard link condividono gli stessi dati del file e quindi sono equivalenti. Quando si crea un hard link, il nuovo link viene trattato come un file completamente separato, ma in realtà si riferisce agli stessi dati del file originale. In pratica, questo significa che se si elimina il file originale, gli hard link continuano ad esistere e i dati del file sono ancora accessibili tramite questi link. Tuttavia, gli hard link non possono essere creati tra file system diversi.

Soft link: un soft link (o symbolic link) è un tipo di file speciale che contiene un riferimento a un altro file o directory nel sistema di file. In pratica, è un'istanza separata del file o della directory a cui fa riferimento. Quando si fa clic sul link simbolico, il sistema operativo segue il percorso indicato dal link per trovare il file o la directory effettiva a cui fa riferimento. A differenza degli hard link, i soft link possono essere creati tra file system diversi.

In sintesi, la differenza principale tra gli hard link e i soft link è che gli hard link sono riferimenti diretti ai dati del file, mentre i soft link sono riferimenti indiretti ai file o alle directory tramite un percorso di file simbolico. Inoltre, gli hard link appaiono come file separati sul sistema di file, mentre i soft link appaiono come file speciali che puntano ad altri file o directory.

52. Descrivere le principali chiamate di sistema per operare sulle directory.

Le chiamate di sistema per operare sulle directory sono un insieme di funzioni offerte dal sistema operativo per gestire le directory, ovvero delle strutture che permettono di organizzare i file in modo gerarchico. Le principali chiamate di sistema per operare sulle directory sono le seguenti:

opendir(): questa chiamata di sistema apre una directory specificata dal percorso e restituisce un puntatore a una struttura che rappresenta la directory.

readdir(): questa chiamata di sistema legge il contenuto della directory aperta tramite opendir() e restituisce il nome del file successivo nella directory.

closedir(): questa chiamata di sistema chiude la directory aperta tramite opendir() e rilascia le risorse allocate per essa.

chdir(): questa chiamata di sistema cambia la directory corrente del processo in quella specificata dal percorso.

mkdir(): questa chiamata di sistema crea una nuova directory con il nome specificato dal percorso.

rmdir(): questa chiamata di sistema rimuove la directory specificata dal percorso se essa è vuota.

stat(): questa chiamata di sistema restituisce le informazioni sul file specificato dal percorso, come ad esempio la dimensione, i permessi, la data di creazione, la data di accesso e la data di modifica.

rename(): questa chiamata di sistema rinomina il file specificato dal percorso con il nuovo nome specificato.

unlink(): questa chiamata di sistema elimina il file specificato dal percorso.

In sintesi, le chiamate di sistema per operare sulle directory permettono di gestire la struttura gerarchica dei file del sistema operativo, permettendo di creare, aprire, leggere, scrivere, rinominare e cancellare i file e le directory.

53. Descrivere come cambiare il proprietario e il gruppo di un file, fornendo un esempio

tramite comando di shell.

In un sistema operativo Unix-like, è possibile cambiare il proprietario e il gruppo di un file utilizzando il comando "chown".

Per cambiare il proprietario di un file, è necessario utilizzare la sintassi seguente:

chown nuovo_proprietario nome_file

Dove "nuovo_proprietario" è il nuovo proprietario a cui si vuole assegnare il file, e "nome_file" è il nome del file di cui si vuole cambiare il proprietario.

Per cambiare il gruppo di un file, invece, è necessario utilizzare la sintassi seguente:

chown :nuovo_gruppo nome_file

Dove "nuovo_gruppo" è il nuovo gruppo a cui si vuole assegnare il file, e "nome_file" è il nome del file di cui si vuole cambiare il gruppo.

Per cambiare sia il proprietario che il gruppo di un file, è necessario utilizzare la sintassi seguente:

chown nuovo_proprietario:nuovo_gruppo nome_file

Dove "nuovo_proprietario" è il nuovo proprietario a cui si vuole assegnare il file, "nuovo_gruppo" è il nuovo gruppo a cui si vuole assegnare il file, e "nome_file" è il nome del file di cui si vuole cambiare il proprietario e il gruppo.

Ad esempio, per assegnare il file "miofile.txt" all'utente "pippo" e al gruppo "users", il comando da utilizzare sarebbe:

sudo chown pippo:users miofile.txt

In questo modo, il file "miofile.txt" sarebbe assegnato all'utente "pippo" e al gruppo "users".

54. Descrivere il comportamento di dup e dup2.

Le funzioni "dup" e "dup2" sono chiamate di sistema in ambiente Unix-like che permettono di duplicare un file descriptor (fd), ovvero un riferimento numerico ad un file o ad una risorsa di sistema che un processo può utilizzare per accedere ad essa.

La funzione "dup" crea una copia del file descriptor passato come argomento, restituendo un nuovo file descriptor che fa riferimento alla stessa risorsa di sistema. Il nuovo file descriptor viene assegnato al primo file descriptor disponibile, quindi potrebbe non avere lo stesso valore numerico del file descriptor originale. Ad esempio, se il file descriptor passato come argomento è 3, il nuovo file descriptor potrebbe essere 4.

La sintassi della funzione "dup" è la seguente:

int dup(int oldfd);

dove "oldfd" è il file descriptor da duplicare.

La funzione "dup2" invece, crea una copia del file descriptor passato come primo argomento, utilizzando il file descriptor specificato come secondo argomento. In altre parole, "dup2" permette di specificare esplicitamente il valore numerico del nuovo file descriptor. Se il file descriptor specificato come secondo argomento è già aperto, verrà prima chiuso e sostituito con il nuovo file descriptor creato dalla funzione "dup2".

La sintassi della funzione "dup2" è la seguente:

int dup2(int oldfd, int newfd);

dove "oldfd" è il file descriptor da duplicare e "newfd" è il nuovo file descriptor da utilizzare.

Entrambe le funzioni sono utilizzate principalmente per gestire l'input/output dei processi. Ad esempio, è possibile utilizzare la funzione "dup" per ridirigere lo standard output di un processo su un file specifico, come nel seguente esempio di codice:

```
#include <unistd.h>
#include <fcntl.h>

int main() {
  int fd = open("output.txt", O_WRONLY | O_CREAT, 0666);
  dup2(fd, 1); // il file descriptor 1 (standard output) viene sostituito con il nuovo file descriptor "fd"
  close(fd); // il file descriptor "fd" non è più necessario, quindi viene chiuso
  printf("Hello, world!"); // questa stringa viene scritta nel file "output.txt"
  return 0;
}
```

In questo esempio, la funzione "open" viene utilizzata per creare un nuovo file "output.txt" con i permessi di scrittura. Successivamente, la funzione "dup2" viene utilizzata per sostituire il file descriptor 1 (standard output) con il file descriptor appena creato. Infine, la funzione "printf" viene utilizzata per scrivere la stringa "Hello, world!" nel file "output.txt", invece che sulla console.

55. Descrivere il comportamento di fcntl.

La funzione fcntl() (abbreviazione di file control) è una chiamata di sistema presente in ambiente Unix-like che permette di eseguire diverse operazioni di controllo su un file descriptor aperto da un processo.

La funzione fcntl() può essere utilizzata per eseguire diverse operazioni, a seconda del valore del secondo parametro "cmd" passato come argomento. Alcune delle operazioni comuni che possono essere eseguite utilizzando fcntl() includono:

Cambiare il flag di accesso al file descriptor (F_SETFL): è possibile utilizzare questa opzione per modificare i flag di accesso a un file descriptor aperto. Ad esempio, è possibile impostare la modalità non bloccante per la lettura o la scrittura di un file descriptor utilizzando l'opzione O_NONBLOCK.

Ottenere i flag di accesso del file descriptor (F_GETFL): è possibile utilizzare questa opzione per ottenere i flag di accesso del file descriptor aperto.

Sbloccare una regione del file (F_UNLCK): è possibile utilizzare questa opzione per sbloccare una regione del file che è stata precedentemente bloccata utilizzando la funzione fcntl() stessa o la funzione lockf().

Bloccare una regione del file (F_SETLK, F_SETLKW): è possibile utilizzare queste opzioni per bloccare una regione del file. L'opzione F_SETLK esegue un blocco non bloccante, ovvero se la regione è già bloccata, la

funzione fcntl() restituisce immediatamente con un errore. L'opzione F_SETLKW invece esegue un blocco bloccante, ovvero se la regione è già bloccata, la funzione fcntl() attende fino a quando non è disponibile.

Ottenere informazioni sul file descriptor (F_GETFD, F_SETFD): è possibile utilizzare queste opzioni per ottenere o impostare il flag di chiusura del file descriptor. Se il flag di chiusura è impostato, il file descriptor viene automaticamente chiuso quando il processo esegue una delle funzioni exec().

La sintassi della funzione fcntl() è la seguente:

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

dove "fd" è il file descriptor su cui eseguire l'operazione, "cmd" è il comando da eseguire e "arg" è un argomento opzionale richiesto per alcune operazioni.

In generale, la funzione fcntl() è utilizzata principalmente per il controllo di accesso al file e per la gestione delle regioni bloccate all'interno di un file. La funzione è particolarmente utile in ambienti multithread o multiprocessore, dove è necessario gestire l'accesso concorrente a un file o a una risorsa di sistema.

56. Descrivere il comportamento di ioctl.

La funzione ioctl() (abbreviazione di input/output control) è una chiamata di sistema presente in ambiente Unix-like che permette di eseguire diverse operazioni di controllo su un dispositivo di input/output.

La funzione ioctl() può essere utilizzata per eseguire diverse operazioni, a seconda del valore del secondo parametro "request" passato come argomento. Alcune delle operazioni comuni che possono essere eseguite utilizzando ioctl() includono:

Ottenere informazioni sul dispositivo (ad esempio, il nome, il numero di versione, il tipo di dispositivo) tramite il comando "IOCTL_GETINFO".

Impostare un parametro specifico del dispositivo tramite il comando "IOCTL_SET_PARAM".

Leggere o scrivere dati dal dispositivo, utilizzando i comandi "IOCTL_READ_DATA" e "IOCTL_WRITE_DATA", rispettivamente.

Ottenere o impostare lo stato del dispositivo tramite il comando "IOCTL_GET_STATE" e "IOCTL_SET_STATE".

La sintassi della funzione ioctl() è la seguente:

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ... /* arg */ );
```

dove "fd" è il file descriptor del dispositivo su cui eseguire l'operazione, "request" è il comando da eseguire e "arg" è un argomento opzionale richiesto per alcune operazioni.

Il comportamento esatto della funzione ioctl() dipende dal dispositivo specifico e dal driver del dispositivo che lo supporta. In genere, la funzione ioctl() è utilizzata per eseguire operazioni su dispositivi che non sono facilmente gestiti utilizzando le funzioni standard di input/output come read() e write().

La funzione ioctl() è stata utilizzata in passato per la gestione di dispositivi hardware come le schede di rete, le schede audio e le periferiche di archiviazione. Oggi, molti di questi dispositivi sono gestiti utilizzando driver di sistema specifici e interfacce di programmazione di più alto livello, come ad esempio l'API di sockets per la comunicazione in rete. Tuttavia, la funzione ioctl() è ancora utilizzata in alcune situazioni in cui è necessario controllare direttamente il funzionamento di un dispositivo.

57. Descrivere l'esecuzione di una fork. In particolare, descrivere quali risorse sono duplicate, e le azioni per distinguere tra processo padre e processo figlio.

La funzione fork() in C consente di creare un nuovo processo identico al processo chiamante (chiamato processo padre) che esegue il programma. Il nuovo processo creato è chiamato processo figlio, e inizia l'esecuzione dal punto in cui è stata chiamata la funzione fork().

Durante l'esecuzione della fork(), il sistema operativo crea una copia identica del processo padre, compresi tutti i suoi dati e la sua memoria. La copia del processo figlio viene creata come un processo separato, con un PID (Process ID) univoco e una tabella dei descrittori dei file separata.

La funzione fork() restituisce un valore diverso a seconda del processo in cui viene eseguita. Nel processo padre, restituisce l'ID del processo figlio appena creato, mentre nel processo figlio restituisce il valore zero. In questo modo, il programma può distinguere tra il processo padre e il processo figlio e adottare il comportamento appropriato in base alla condizione del processo.

In sintesi, la fork() è una funzione di sistema in C che consente di creare un nuovo processo identico al processo chiamante, consentendo di eseguire più istanze dello stesso programma in parallelo o di creare processi figli isolati dal processo padre.

58. Descrivere il comportamento della exit e della wait, in particolar modo spiegando il loro utilizzo combinato in caso di un processo padre (che esegue la wait) e un processo figlio (che esegue la exit).

La funzione exit() viene utilizzata da un processo figlio per terminare la sua esecuzione e restituire un valore di stato al processo padre. Quando un processo figlio esegue exit(), il sistema operativo dealloca tutte le risorse utilizzate dal processo figlio e notifica il processo padre dell'avvenuta terminazione del figlio, fornendo al padre il valore di stato restituito dal figlio.

D'altra parte, la funzione wait() viene utilizzata da un processo padre per attendere la terminazione di uno dei suoi processi figli. La funzione wait() blocca il processo padre finché non viene restituito un valore di stato da uno dei suoi processi figli terminati. In questo modo, il processo padre può ottenere il valore di stato del processo figlio terminato e liberare le risorse allocate al figlio.

Quando utilizzati insieme, il processo figlio utilizza exit() per terminare la sua esecuzione e restituire un valore di stato al processo padre, mentre il processo padre utilizza wait() per bloccare la sua esecuzione e attendere il valore di stato restituito dal figlio. In questo modo, il processo padre può ottenere il valore di stato del processo figlio e liberare le risorse allocate al figlio. Questo processo è noto come "terminazione controllata" del processo figlio da parte del processo padre.

59. Descrivere i passi principali per la creazione e terminazione di un processo, indicando le

principali chiamate di sistema.

La creazione e la terminazione di un processo in un sistema operativo moderno possono essere descritte attraverso i seguenti passaggi principali:

Creazione di un processo:

Fase di inizializzazione: il sistema operativo riserva la memoria per il processo e inizializza i registri necessari per l'esecuzione del codice del processo.

Fase di caricamento: il sistema operativo carica il codice del processo in memoria e inizia l'esecuzione del processo.

Fase di assegnazione delle risorse: il sistema operativo assegna le risorse necessarie per l'esecuzione del processo, ad esempio i file aperti, i semafori, le code di messaggi, etc.

Le principali chiamate di sistema coinvolte nella creazione di un processo includono:

fork(): crea un nuovo processo duplicando il processo chiamante.

exec(): sostituisce il codice del processo corrente con un nuovo programma.

exit(): termina l'esecuzione del processo corrente e restituisce un valore di stato al processo padre.

Terminazione di un processo:

Fase di liberazione delle risorse: il processo rilascia tutte le risorse che ha acquisito durante la sua esecuzione, come file aperti, memoria allocata, ecc.

Fase di notifica: il sistema operativo notifica il processo padre della terminazione del processo e restituisce un valore di stato del processo.

Fase di deallocazione: il sistema operativo dealloca la memoria occupata dal processo e libera tutte le risorse assegnate al processo.

Le principali chiamate di sistema coinvolte nella terminazione di un processo includono:

exit(): termina l'esecuzione del processo corrente e restituisce un valore di stato al processo padre.

wait(): blocca il processo padre finché uno dei suoi processi figli non termina.

kill(): invia un segnale a un processo per terminare la sua esecuzione in modo forzato.

exit_group(): termina tutti i processi del gruppo di processi corrente.

60. Descrivere il comportamento della famiglia di funzioni exec.

La famiglia di funzioni exec in Unix/Linux è utilizzata per sostituire il contenuto di un processo con un nuovo programma. Ci sono tre funzioni exec principali:

execl: questa funzione accetta una lista di argomenti di tipo stringa terminata da un puntatore nullo e sostituisce il programma corrente con un nuovo programma. Ad esempio, execl("/bin/ls", "ls", "-l", NULL) sostituirà il programma corrente con /bin/ls e gli passerà due argomenti: "ls" e "-l".

execv: questa funzione accetta un array di argomenti di tipo stringa terminato da un puntatore nullo e sostituisce il programma corrente con un nuovo programma. Ad esempio, char *args[] = {"ls", "-l", NULL}; execv("/bin/ls", args); sostituirà il programma corrente con /bin/ls e gli passerà due argomenti: "ls" e "-l".

execle e execve: queste funzioni sono simili a execl e execv, ma accettano anche un array di stringhe che rappresenta l'ambiente del nuovo programma.

In generale, quando una funzione exec viene chiamata, il processo corrente viene sostituito dal nuovo programma specificato. Il nuovo programma inizia l'esecuzione dal suo punto di ingresso, e riceve come

argomenti gli argomenti passati alla funzione exec. Il processo mantieni il suo PID e gli eventuali descrittori di file aperti, ma il codice, i dati e lo stack vengono sostituiti con quelli del nuovo programma.

Le funzioni exec sono comunemente utilizzate dai programmi che vogliono eseguire un altro programma come parte della loro esecuzione, ad esempio un interprete di comandi che esegue un comando specificato dall'utente.

61. Descrivere la sintassi di almeno una delle funzioni della famiglia exec.

La famiglia di funzioni exec comprende diverse funzioni, come execl, execle, execlp, execv, execve, execvp, ognuna delle quali ha una sintassi leggermente diversa.

Ad esempio, la sintassi della funzione execl è la seguente:

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ... /* (char *) NULL */ );
```

I parametri della funzione sono i seguenti:

path: una stringa che specifica il percorso del file eseguibile che si vuole eseguire. Il percorso può essere un percorso assoluto o relativo.

arg0: una stringa che rappresenta l'argomento zero del nuovo programma. In genere, questa stringa rappresenta il nome del programma stesso.

...: zero o più argomenti separati da virgole che vengono passati al nuovo programma. L'ultimo argomento deve essere un puntatore a NULL, che indica la fine degli argomenti.

La funzione execl sostituisce il processo corrente con il programma specificato da path. La funzione prende in ingresso il percorso del file eseguibile, l'argomento zero e una lista di argomenti. Il nuovo programma inizierà l'esecuzione dalla sua funzione main().

Ad esempio, il seguente codice utilizza la funzione execl per eseguire il programma ls:

```
#include <unistd.h>

int main() {
  execl("/bin/ls", "ls", "-l", NULL);
  return 0;
}
```

In questo esempio, la funzione execl sostituisce il processo corrente con il programma ls. Il parametro "/bin/ls" specifica il percorso del file eseguibile, "ls" rappresenta l'argomento zero e "-l" rappresenta il primo argomento passato al programma ls. Il parametro NULL indica la fine della lista di argomenti.

62. Descrivere come impostare la priorità di un processo, sia da terminale che tramite

chiamata di sistema.

La priorità di un processo è un valore che determina la quantità di tempo di CPU che il sistema operativo assegna a un processo. Maggiore è la priorità di un processo, maggiori sono le probabilità che il processo venga eseguito prima di altri processi in attesa. In questo modo, un processo con alta priorità può essere eseguito più rapidamente di un processo con bassa priorità.

Ecco come impostare la priorità di un processo:

Da terminale:

In Linux, è possibile impostare la priorità di un processo utilizzando il comando nice. La sintassi generale del comando nice è la seguente:

nice [opzioni] [comando [argomenti...]]

Il comando nice imposta la priorità del processo corrente o di un processo specificato nel comando comando. Il valore della priorità viene specificato da un numero intero tra -20 (priorità massima) e 19 (priorità minima).

Per impostare la priorità di un processo specifico, si utilizza il comando renice. La sintassi generale del comando renice è la seguente:

renice [opzioni] [priorità] -p [PID]

Il comando renice cambia la priorità del processo specificato dal PID. Il valore della priorità viene specificato da un numero intero tra -20 e 19.

Tramite chiamata di sistema:

In Linux, la priorità di un processo può essere impostata anche attraverso le chiamate di sistema setpriority() o sched_setscheduler().

La chiamata di sistema setpriority() è definita come segue:

#include <sys/resource.h>

int setpriority(int which, id_t who, int prio);

Il primo argomento which specifica il tipo di identificatore del processo, che può essere PRIO_PROCESS, PRIO_PGRP o PRIO_USER. Il secondo argomento who specifica l'identificatore del processo, che può essere il PID del processo o l'ID del gruppo di processi o l'ID dell'utente. Il terzo argomento prio specifica la nuova priorità del processo.

La chiamata di sistema sched_setscheduler() è definita come segue:

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);
```

Il primo argomento pid specifica il PID del processo da modificare. Il secondo argomento policy specifica la politica di scheduling, che può essere SCHED_FIFO, SCHED_RR o SCHED_OTHER. Il terzo argomento param è un puntatore a una struttura sched_param che contiene la priorità del processo. La priorità è specificata dal campo sched_priority della struttura sched_param.

Nota che la priorità del processo può essere modificata solo dal superutente o dal proprietario del processo stesso.

63. Descrivere l'attacco fork bomb, incluso il codice C per la sua esecuzione.

Un fork bomb è un tipo di attacco DoS (denial of service) in cui un programma crea ripetutamente nuovi processi fino a esaurire tutte le risorse di sistema disponibili, rendendo il sistema inutilizzabile.

Il codice C per l'esecuzione di un fork bomb è il seguente:

```
#include <unistd.h>

int main() {
  while(1) {
    fork();
  }
  return 0;
}
```

Questo programma utilizza la funzione fork() per creare un nuovo processo figlio che a sua volta creerà un altro processo figlio e così via. Ogni nuovo processo figlio crea una copia esatta del processo padre, inclusi

tutti i suoi descrittori di file e altre risorse di sistema. In questo modo, il numero di processi in esecuzione aumenta esponenzialmente, fino a esaurire tutte le risorse di sistema disponibili.

Per prevenire l'esecuzione di un fork bomb, è possibile limitare il numero massimo di processi che un utente può creare, impostando un limite di ulimit o impostando un valore massimo di max_user_processes nel file /etc/security/limits.conf. Inoltre, si possono utilizzare strumenti come cgroups per limitare il numero di processi per singolo utente o per singolo processo.

64. Cosa è un segnale, e cosa è un signal handler?

In un sistema operativo basato su Unix, un segnale (signal) è un meccanismo software utilizzato per comunicare eventi o interruzioni asincrone a un processo. I segnali sono usati per notificare un processo di eventi come la ricezione di dati in una socket, l'arrivo di un segnale da parte di un altro processo o la scadenza di un timer. In pratica, un segnale è un messaggio inviato a un processo che gli indica che qualcosa di importante è successo.

Un signal handler, o gestore di segnali, è una funzione che viene eseguita quando un processo riceve un segnale. La funzione viene chiamata in modo asincrono dal sistema operativo quando il segnale corrispondente viene ricevuto dal processo. Il signal handler può quindi eseguire azioni specifiche in risposta al segnale ricevuto, ad esempio terminare il processo o eseguire un'operazione di salvataggio dati.

Ogni segnale ha un numero univoco associato, definito da una costante in signal.h, come ad esempio SIGTERM per il segnale di terminazione. Ogni segnale può essere gestito da un signal handler specifico, che viene impostato attraverso la chiamata di sistema signal(). La sintassi di base della funzione signal() è la seguente:

```
#include <signal.h>

void (*signal(int signum, void (*handler)(int)))(int);
```

Dove signum è il numero del segnale da gestire e handler è il puntatore alla funzione da eseguire quando il segnale viene ricevuto. Il tipo di ritorno della funzione è un puntatore a una funzione che accetta un argomento intero (il numero del segnale).

65. Elencare almeno due segnali che è possibile inviare da tastiera verso un processo in foreground.

In un sistema Unix, quando un processo è in esecuzione in foreground, è possibile inviare segnali tramite la tastiera. Ecco due esempi di segnali che possono essere inviati da tastiera a un processo in foreground:

Ctrl-C (segnale SIGINT): Questo segnale viene inviato quando l'utente preme Ctrl-C sulla tastiera. Il segnale di interruzione (SIGINT) viene utilizzato per terminare un processo in modo pulito, permettendogli di liberare le risorse di sistema che sta utilizzando.

Ctrl-Z (segnale SIGTSTP): Questo segnale viene inviato quando l'utente preme Ctrl-Z sulla tastiera. Il segnale di stop (SIGTSTP) viene utilizzato per sospendere un processo in esecuzione. Quando il processo viene sospeso, viene inserito in background e il controllo viene restituito all'utente. Il processo può essere ripreso in un secondo momento utilizzando il comando fg o bg.

66. Elencare le macro corrispondenti ad almeno 5 segnali tra quelli elencati nella signal.h, e descrivere il comportamento di tali segnali.

Ecco le macro corrispondenti ad alcuni dei segnali elencati nella signal.h:

SIGINT - Macro: SIGINT. Comportamento: Questo segnale viene inviato al processo quando l'utente preme Ctrl-C sulla tastiera. Il segnale di interruzione (SIGINT) viene utilizzato per terminare un processo in modo pulito, permettendogli di liberare le risorse di sistema che sta utilizzando.

SIGTERM - Macro: SIGTERM. Comportamento: Questo segnale viene inviato al processo per richiedere la sua terminazione. Il segnale di terminazione (SIGTERM) viene utilizzato per terminare un processo in modo pulito, permettendogli di liberare le risorse di sistema che sta utilizzando.

SIGKILL - Macro: SIGKILL. Comportamento: Questo segnale viene inviato al processo per forzarne la terminazione immediata. Il segnale di kill (SIGKILL) viene utilizzato per terminare un processo in modo forzato, senza permettergli di liberare le risorse di sistema che sta utilizzando.

SIGUSR1 - Macro: SIGUSR1. Comportamento: Questo segnale è un segnale generico che può essere utilizzato per inviare una notifica personalizzata al processo. Il comportamento specifico del segnale dipende dall'applicazione.

SIGSEGV - Macro: SIGSEGV. Comportamento: Questo segnale viene inviato al processo quando viene tentato di accedere a una zona di memoria non autorizzata (ad esempio, leggere o scrivere in una zona di memoria non allocata). Il segnale di segmentation fault (SIGSEGV) viene utilizzato per indicare un errore di programmazione, come un puntatore non inizializzato o un accesso fuori dai limiti di un array.

67. Come è possibile visualizzare da shell l'elenco delle macro corrispondenti ai segnali, presenti nel nostro sistema?

Per visualizzare l'elenco delle macro corrispondenti ai segnali presenti nel sistema, è possibile utilizzare il comando kill -l dalla shell. Questo comando elenca tutti i segnali disponibili nel sistema, insieme alle relative macro definite nella libreria signal.h.

68. Descrivere il comportamento della chiamata signal, possibilmente aiutandosi con un esempio.

La chiamata di sistema signal() viene utilizzata per definire un gestore di segnale personalizzato (signal handler) per un particolare segnale. La sintassi di questa chiamata di sistema è la seguente:

void (*signal(int sig, void (*handler)(int)))(int);

dove sig è il segnale per il quale si vuole definire il gestore di segnale, e handler è il puntatore ad una funzione che rappresenta il gestore di segnale personalizzato.

La funzione handler prende in input un intero, che rappresenta il segnale ricevuto. Il comportamento della funzione handler dipende dal segnale in questione.

Ecco un esempio di come utilizzare la chiamata signal() per gestire il segnale SIGINT (generato dalla pressione dei tasti CTRL+C):

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sigint_handler(int sig) {
  printf("Ricevuto il segnale SIGINT (%d)\n", sig);
}

int main() {
  // Definizione del gestore di segnale personalizzato per SIGINT
  signal(SIGINT, sigint_handler);

  printf("Attendi il segnale SIGINT (CTRL+C)...\n");

  // Ciclo infinito
  while(1) {
    sleep(1);
  }

  return 0;
}
```

In questo esempio, il programma definisce una funzione sigint_handler() che viene chiamata ogni volta che viene ricevuto il segnale SIGINT. La funzione stampa un messaggio a schermo che indica il segnale ricevuto.

Nella funzione main(), la chiamata signal(SIGINT, sigint_handler) imposta il gestore di segnale personalizzato per il segnale SIGINT, in modo che la funzione sigint_handler() sia chiamata ogni volta che viene ricevuto il segnale.

Infine, il programma entra in un ciclo infinito che attende la ricezione del segnale SIGINT. Quando il segnale viene ricevuto, il gestore di segnale personalizzato viene chiamato, e la funzione sigint_handler() viene eseguita.

69. Descrivere le colonne della seguente tabella (ottenuta con ps -eo pid,pcpu,pmem,

start,comm|head -n 2)

PID %CPU %MEM STAT STARTED COMMAND

1 0.0 0.0 Ss Ss Feb 16 systemd

La tabella ottenuta con il comando "ps -eo pid,pcpu,pmem,start,comm|head -n 2" mostra le seguenti colonne:

PID: rappresenta l'identificativo numerico del processo (Process ID)

%CPU: indica la percentuale di CPU utilizzata dal processo, ovvero la sua attività computazionale

%MEM: indica la percentuale di memoria utilizzata dal processo

STAT: indica lo stato attuale del processo (ad esempio, S indica "sleeping", R indica "running", Z indica "zombie")

STARTED: indica l'ora e la data di avvio del processo

COMMAND: indica il nome del comando o del programma associato al processo (ad esempio, "systemd" in questo caso)

70. Elencare almeno due possibili approcci per limitare la generazione e persistenza di zombie

all'interno del sistema.

Ci sono almeno due approcci possibili per limitare la generazione e la persistenza dei processi zombie nel sistema:

Utilizzare la chiamata di sistema waitpid() per attendere la terminazione dei processi figli e ottenere il loro stato di uscita. In questo modo, il processo padre può liberare le risorse allocate per i processi figli e prevenire la creazione di processi zombie.

Utilizzare la chiamata di sistema signal() per registrare un gestore di segnali per il segnale SIGCHLD, che viene inviato quando un processo figlio termina. Il gestore di segnali può poi chiamare la chiamata di sistema wait() per attendere la terminazione dei processi figli e prevenire la creazione di processi zombie.

71. Descrivere come ignorare un segnale.

Per ignorare un segnale, è possibile utilizzare la funzione signal() passando come secondo argomento la costante SIG_IGN. Ad esempio, per ignorare il segnale SIGINT, si può utilizzare il seguente codice:

```
#include <signal.h>

int main() {
   // Ignora il segnale SIGINT
   signal(SIGINT, SIG_IGN);

   // Resto del codice
   ...
}
```

In questo modo, quando il processo riceve il segnale SIGINT, non verrà eseguito alcun signal handler, ma il segnale verrà ignorato.

72. Descrivere la sintassi e il comportamento della chiamata kill.

La chiamata di sistema "kill" viene utilizzata su sistemi operativi Unix e Unix-like per inviare un segnale a un processo specifico o a un gruppo di processi. La sintassi generale della chiamata "kill" è la seguente:

```
css

kill [options] pid [...]
```

dove "pid" è l'identificatore del processo o dei processi a cui inviare il segnale.

Le opzioni disponibili per la chiamata "kill" includono:

-s signal: specifica il segnale da inviare. Se non specificato, il segnale predefinito è SIGTERM (terminazione del processo).

-l: elenca tutti i segnali disponibili.

-n: mostra solo il numero di segnale senza inviare il segnale al processo.

-p: stampa il PID dei processi anziché inviare il segnale.

Il comportamento della chiamata "kill" dipende dal segnale inviato. Alcuni segnali comuni includono:

SIGTERM: termina il processo in modo pulito.

SIGKILL: termina immediatamente il processo.

SIGHUP: invia un segnale di hangup al processo.

SIGINT: invia un segnale di interruzione al processo.

In generale, la chiamata "kill" viene utilizzata per terminare un processo che non risponde o per inviare un segnale per richiedere una determinata azione da parte del processo. È importante notare che non tutti i segnali sono gestiti allo stesso modo da tutti i processi e alcuni processi possono ignorare o bloccare determinati segnali.

73. Descrivere la sintassi e il comportamento della chiamata alarm.

La chiamata di sistema "alarm" viene utilizzata su sistemi operativi Unix e Unix-like per impostare un timer per un determinato numero di secondi, dopo il quale viene inviato un segnale SIGALRM al processo corrente. La sintassi generale della chiamata "alarm" è la seguente:

arduino

unsigned int alarm(unsigned int seconds);

dove "seconds" è il numero di secondi dopo i quali inviare il segnale SIGALRM.

Il comportamento della chiamata "alarm" è il seguente:

Quando la chiamata "alarm" viene eseguita con un valore positivo di "seconds", viene impostato un timer interno con il valore specificato.

Se viene eseguita una seconda chiamata "alarm" prima che il timer sia scaduto, il valore del timer viene aggiornato con il nuovo valore specificato.

Quando il timer scade, il processo riceve il segnale SIGALRM, che può essere gestito o ignorato dal processo.

Se la chiamata "alarm" viene eseguita con un valore di "seconds" pari a 0, il timer corrente viene disabilitato e nessun segnale viene inviato.

La chiamata "alarm" viene spesso utilizzata per programmare una determinata azione dopo un determinato periodo di tempo. Ad esempio, potrebbe essere utilizzata per controllare il tempo di esecuzione di una determinata operazione o per eseguire un'attività a intervalli regolari. Tuttavia, poiché il segnale SIGALRM

viene inviato al processo corrente, è importante assicurarsi che il gestore del segnale sia correttamente configurato per evitare comportamenti indesiderati.

74. Cosa è il terminale di controllo, e come interagisce con i gruppi di processo?

Il terminale di controllo, noto anche come terminale del controllatore o terminale di controllo del gruppo, è un concetto utilizzato nei sistemi operativi Unix e Unix-like per indicare il terminale attraverso il quale un determinato gruppo di processi comunica con l'utente.

In un sistema Unix, un gruppo di processi è un insieme di processi correlati che condividono lo stesso identificatore di gruppo di processi (PGID). Il processo di login è il leader del gruppo di processi e tutti i processi figli creati dal processo di login diventano membri del gruppo di processi. Il processo di login è anche associato a un determinato terminale di controllo, che viene utilizzato per gestire l'input e l'output del gruppo di processi.

Quando un utente esegue un comando da terminale, il terminale invia l'input al processo di shell associato al terminale. La shell analizza l'input dell'utente e crea un nuovo processo figlio per eseguire il comando. Questo nuovo processo eredita l'identificatore del gruppo di processi e il terminale di controllo dal processo di shell.

Il terminale di controllo è importante perché influisce sul comportamento dei processi in esecuzione. Ad esempio, quando un utente preme CTRL-C o CTRL-Z da terminale, il terminale invia un segnale SIGINT o SIGTSTP al processo corrente. Se il processo corrente è il leader del gruppo di processi, il segnale viene inviato a tutti i processi del gruppo di processi.

Inoltre, quando un utente si disconnette da un terminale, il terminale invia un segnale SIGHUP al processo di login, che a sua volta invia un segnale SIGHUP a tutti i processi del gruppo di processi. Questo è utile per terminare tutti i processi associati al terminale di controllo quando un utente si disconnette.

In sintesi, il terminale di controllo è il meccanismo attraverso il quale un gruppo di processi comunica con l'utente. È importante comprendere come funziona il terminale di controllo e come interagisce con i gruppi di processi per gestire correttamente l'input e l'output dei processi in un sistema Unix.

75. Cosa succede se un processo tenta di leggere dal suo terminale di controllo senza essere

membro del gruppo del processo che controlla quel terminale?

Se un processo tenta di leggere dal suo terminale di controllo senza essere membro del gruppo del processo che controlla quel terminale, l'operazione di lettura fallirà e il processo riceverà un segnale di errore "EIO" (Input/Output error) o "ENOTTY" (Not a tty).

Questo perché il terminale è controllato da un processo che ha il diritto esclusivo di accedervi e di effettuare operazioni di input/output su di esso. Solo i processi che appartengono al gruppo di quel processo hanno il permesso di leggere o scrivere sul terminale. Se un processo che non appartiene al

gruppo del processo che controlla il terminale tenta di leggere dal terminale, il sistema operativo rileverà l'errore e impedirà l'accesso al terminale.

76. Come si può cambiare il gruppo di processi di appartenenza?

In un sistema operativo Unix-like come Linux, il gruppo di processi di appartenenza di un processo può essere cambiato utilizzando il comando setpgid o la funzione di libreria setpgid().

Il comando setpgid viene utilizzato per modificare il gruppo di processi di un processo esistente. La sintassi del comando è la seguente:

setpgid pid pgid

dove pid è il PID (Process ID) del processo e pgid è il PID del gruppo di processi di destinazione. Ad esempio, il seguente comando cambia il gruppo di processi del processo con PID 1234 al PID del gruppo di processi 5678:

setpgid 1234 5678

In alternativa, è possibile utilizzare la funzione di libreria setpgid() per modificare il gruppo di processi di appartenenza di un processo all'interno di un programma. La sintassi della funzione è la seguente:

int setpgid(pid_t pid, pid_t pgid);

dove pid è il PID del processo e pgid è il PID del gruppo di processi di destinazione. Ad esempio, il seguente codice C cambia il gruppo di processi del processo corrente al PID del gruppo di processi 5678:

setpgid(0, 5678);

È importante notare che solo i processi con lo stesso proprietario del processo possono cambiare il suo gruppo di processi. Inoltre, solo il leader di un gruppo di processi può cambiare il gruppo di processi del proprio gruppo.

77. Cosa si intende con IPC? Descrivere brevemente i principali meccanismi di IPC analizzati

nel corso.

IPC sta per "Inter-Process Communication" (Comunicazione Inter-Processo) ed è un insieme di tecniche che consentono ai processi di scambiare dati, informazioni e segnali tra loro su un sistema operativo.

I principali meccanismi di IPC che vengono utilizzati sono:

Pipe: Una Pipe è un canale unidirezionale che collega due processi in modo che il processo iniziale possa scrivere dati su di esso e il processo successivo possa leggere i dati. Le pipe sono utilizzate per trasferire grandi quantità di dati tra processi.

Message Queues: Una coda di messaggi è una struttura di dati che consente ai processi di scambiare messaggi attraverso il kernel del sistema operativo. I messaggi possono essere scambiati tra processi in modo asincrono e con una priorità specifica.

Shared Memory: La memoria condivisa consente ai processi di accedere alla stessa area di memoria nel kernel del sistema operativo, dove possono scrivere e leggere dati senza dover copiare i dati tra i processi.

Sockets: I socket sono una tecnologia di rete che consente ai processi di comunicare tra loro su una rete. Un processo può inviare dati a un socket su un altro computer o ricevere dati da un socket su un altro computer.

Signal: I segnali sono utilizzati per notificare a un processo di evento o di condizione particolare. Ad esempio, un segnale può essere inviato a un processo per indicare che è stato ricevuto un input da un utente o per avvisare un processo di terminare.

Questi meccanismi di IPC sono ampiamente utilizzati nella progettazione dei sistemi operativi e nelle applicazioni che richiedono la comunicazione tra processi.

78. Descrivere i passi per costruire e utilizzare un pipe anonimo, tramite codice C e tramite

comandi di shell.

Ecco i passaggi per costruire e utilizzare una pipe anonima, sia tramite codice C che tramite comandi di shell:

PIPE ANONIMO IN C

Creazione della pipe anonima utilizzando la funzione pipe().

c

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

Questa funzione restituisce 0 in caso di successo e -1 in caso di errore. La pipe anonima ha due estremità, una di scrittura e una di lettura, e pipefd[0] e pipefd[1] sono i file descriptor per le due estremità rispettivamente.

Creazione di un processo figlio utilizzando la funzione fork().

c

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Questa funzione crea un processo figlio che è una copia esatta del processo padre.

Nel processo figlio, chiudere l'estremità di lettura della pipe (poiché il processo figlio scriverà sulla pipe).

c

```
close(pipefd[0]);
```

Nel processo padre, chiudere l'estremità di scrittura della pipe (poiché il processo padre leggerà dalla pipe).

```
close(pipefd[1]);
```

Nel processo figlio, scrivere i dati sulla pipe utilizzando la funzione write().

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

Questa funzione scrive i dati nel file descriptor specificato (in questo caso, l'estremità di scrittura della pipe).

Nel processo padre, leggere i dati dalla pipe utilizzando la funzione read().

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Questa funzione legge i dati dal file descriptor specificato (in questo caso, l'estremità di lettura della pipe).

Nel processo figlio e nel processo padre, chiudere le estremità della pipe dopo aver finito di utilizzarle.

```
close(pipefd[0]);
close(pipefd[1]);
```

PIPE ANONIMO IN SHELL

Creazione di una pipe anonima utilizzando il comando pipe.

```
$ pipe
```

Questo comando crea una pipe anonima e restituisce i file descriptor per le due estremità della pipe.

Esecuzione del comando che scriverà sulla pipe. Ad esempio:

```
$ echo "Hello, world!" > /dev/fd/PIPE_WRITE_FD
```

dove PIPE_WRITE_FD è il file descriptor per l'estremità di scrittura della pipe.

Esecuzione del comando che leggerà dalla pipe. Ad esempio:

```
$ cat < /dev/fd/PIPE_READ_FD
```

dove PIPE_READ_FD è il file descriptor per l'estremità di lettura della pipe.

Chiudere le estremità della pipe dopo aver finito di utilizzarle.

$ exec 3>&- 4<&-

dove 3 e 4 sono i file descriptor per l'estremità di scrittura e lettura della pipe rispettivamente.

79. Descrivere le regole per pipe in lettura

Le pipe in lettura sono un meccanismo che permette di concatenare la lettura di più comandi in modo da elaborare dati in maniera sequenziale. Le regole per l'utilizzo delle pipe in lettura sono le seguenti:

Iniziare la pipe con il simbolo "|", che indica che l'output del comando precedente viene passato come input al comando successivo.

Posizionare la pipe tra i comandi che si desidera concatenare, come ad esempio: comando1 | comando2 | comando3.

Assicurarsi che i comandi utilizzati nella pipe abbiano un output che può essere utilizzato come input dal comando successivo.

Verificare che i comandi siano eseguiti in ordine corretto, in modo da ottenere il risultato desiderato.

Utilizzare i comandi di filtraggio come "grep", "sort" o "uniq" per elaborare i dati in maniera specifica.

Prestare attenzione alle espressioni regolari utilizzate nei comandi di filtraggio, in modo da ottenere il risultato desiderato.

Utilizzare il comando "head" o "tail" per limitare il numero di righe di output.

Seguendo queste regole, è possibile utilizzare le pipe in lettura per concatenare la lettura di più comandi in modo efficiente e sequenziale.

80. Descrivere le regole per pipe in scrittura

Le pipe in scrittura sono un meccanismo che permette di scrivere l'output di un comando direttamente in un file o in un altro comando. Le regole per l'utilizzo delle pipe in scrittura sono le seguenti:

Utilizzare il simbolo ">", seguito dal nome del file di destinazione, per scrivere l'output di un comando in un file. Ad esempio: comando1 > output.txt.

Utilizzare il simbolo ">>", seguito dal nome del file di destinazione, per aggiungere l'output di un comando a un file esistente senza sovrascrivere i dati già presenti. Ad esempio: comando1 >> output.txt.

Utilizzare il simbolo "|", seguito dal comando di destinazione, per passare l'output di un comando come input al comando successivo. Ad esempio: comando1 | comando2.

Assicurarsi che i comandi utilizzati nella pipe abbiano un input che può essere utilizzato come output dal comando precedente.

Verificare che i comandi siano eseguiti nell'ordine corretto, in modo da ottenere il risultato desiderato.

Utilizzare il comando "tee" per scrivere l'output di un comando sia su un file che su un altro comando. Ad esempio: comando1 | tee output.txt | comando2.

Seguendo queste regole, è possibile utilizzare le pipe in scrittura per scrivere l'output di un comando direttamente in un file o in un altro comando in modo efficiente e sequenziale.

81. Descrivere il comportamento della chiamata di sistema pipe.

La chiamata di sistema pipe() è utilizzata per creare una pipe, ovvero un canale di comunicazione tra due processi che consente di trasferire dati in maniera unidirezionale. La pipe viene creata utilizzando un array di interi con due elementi, dove il primo elemento rappresenta il file descriptor in lettura e il secondo elemento rappresenta il file descriptor in scrittura.

Il comportamento della chiamata di sistema pipe() è il seguente:

Viene creato un array di interi con due elementi.

Viene creato un nuovo canale di comunicazione tra processi e assegnato ai due elementi dell'array.

Il file descriptor del lato di scrittura della pipe viene restituito come valore di ritorno della chiamata di sistema pipe() al processo padre.

Il file descriptor del lato di lettura della pipe viene restituito come valore di ritorno della chiamata di sistema pipe() al processo figlio creato con una chiamata a fork().

I processi possono utilizzare i file descriptor per scrivere o leggere dati sulla pipe, rispettivamente utilizzando le chiamate di sistema write() e read().

In sintesi, la chiamata di sistema pipe() crea una pipe e restituisce i file descriptor per il lato di lettura e di scrittura, permettendo a due processi di comunicare tra loro attraverso il canale creato.

82. Descrivere i passi per costruire, utilizzare e distruggere un pipe con nome.

Una pipe con nome (o FIFO) è un file speciale che consente a processi diversi di comunicare tra loro attraverso un canale di comunicazione condiviso. La pipe con nome è caratterizzata da un nome che viene assegnato a livello di file system e viene creata utilizzando le seguenti operazioni:

Costruzione della pipe con nome:

Utilizzare la chiamata di sistema mkfifo() per creare un file speciale con il nome desiderato.

Questa operazione viene solitamente eseguita dal processo che intende utilizzare la pipe con nome per comunicare con altri processi.

Utilizzo della pipe con nome:

Utilizzare la chiamata di sistema open() per aprire la pipe con nome in modalità di lettura o scrittura.

Questa operazione viene eseguita dai processi che intendono utilizzare la pipe con nome per inviare o ricevere dati.

La pipe con nome può essere aperta simultaneamente da più processi, in modalità di lettura o scrittura.

Scambio di dati tramite la pipe con nome:

Utilizzare le chiamate di sistema write() e read() per inviare e ricevere dati attraverso la pipe con nome.

Le operazioni di scrittura e lettura possono essere eseguite da processi differenti, purché la pipe con nome sia aperta in modalità corretta.

Distruzione della pipe con nome:

Utilizzare la chiamata di sistema unlink() per rimuovere la pipe con nome dal file system.

Questa operazione viene solitamente eseguita dal processo che ha creato la pipe con nome, dopo aver verificato che nessun altro processo stia utilizzando ancora la pipe.

In sintesi, per costruire, utilizzare e distruggere una pipe con nome, occorre creare la pipe tramite la chiamata di sistema mkfifo(), aprire la pipe tramite la chiamata di sistema open(), scambiare dati utilizzando le chiamate di sistema write() e read(), e infine distruggere la pipe tramite la chiamata di sistema unlink().

83. Descrivere il comportamento della chiamata di sistema mknod, fornendo un esempio.

La chiamata di sistema mknod viene utilizzata per creare un file speciale nel file system, come ad esempio un dispositivo di carattere o blocco. Il comportamento della chiamata di sistema mknod varia a seconda del tipo di file speciale che si desidera creare.

La sintassi generale della chiamata di sistema mknod è la seguente:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

pathname è il percorso del file speciale da creare.

mode specifica i permessi del file speciale.

dev è il dispositivo associato al file speciale.

Ad esempio, se si desidera creare un dispositivo di carattere (ad esempio, una porta seriale), è possibile utilizzare la seguente chiamata di sistema:

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
   int fd;
   mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH;  // permessi del file speciale
   dev_t dev = makedev(4, 64);  // dispositivo associato al file speciale (esempio: /dev/ttyS0)

   // creazione del file speciale
   if (mknod("/dev/ttyS0", mode | S_IFCHR, dev) < 0) {
      perror("mknod");
      return 1;
   }

   // utilizzo del file speciale
   fd = open("/dev/ttyS0", O_RDWR);
   // ...
   close(fd);

   // rimozione del file speciale
   if (unlink("/dev/ttyS0") < 0) {
      perror("unlink");
      return 1;
   }

   return 0;
}
```

In questo esempio, la chiamata di sistema mknod viene utilizzata per creare un dispositivo di carattere denominato /dev/ttyS0 con permessi di lettura e scrittura per l'utente proprietario, il gruppo proprietario e

altri utenti. Il dispositivo associato è identificato da makedev(4, 64), che corrisponde alla porta seriale ttyS0 su un sistema Linux standard.

Dopo la creazione del file speciale, viene utilizzato nella chiamata di sistema open(), che consente di aprire il file speciale in modalità di lettura e scrittura. Alla fine del programma, il file speciale viene rimosso utilizzando la chiamata di sistema unlink().

84. Descrivere il comportamento della chiamata di sistema mkfifo, fornendo un esempio.

La chiamata di sistema mknod viene utilizzata per creare un file speciale nel file system, come ad esempio un dispositivo di carattere o blocco. Il comportamento della chiamata di sistema mknod varia a seconda del tipo di file speciale che si desidera creare.

La sintassi generale della chiamata di sistema mknod è la seguente:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

pathname è il percorso del file speciale da creare.

mode specifica i permessi del file speciale.

dev è il dispositivo associato al file speciale.

Ad esempio, se si desidera creare un dispositivo di carattere (ad esempio, una porta seriale), è possibile utilizzare la seguente chiamata di sistema:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd;
```

```
   mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH;  // permessi del file speciale
   dev_t dev = makedev(4, 64);  // dispositivo associato al file speciale (esempio: /dev/ttyS0)

   // creazione del file speciale
   if (mknod("/dev/ttyS0", mode | S_IFCHR, dev) < 0) {
      perror("mknod");
      return 1;
   }

   // utilizzo del file speciale
   fd = open("/dev/ttyS0", O_RDWR);
   // ...
   close(fd);

   // rimozione del file speciale
   if (unlink("/dev/ttyS0") < 0) {
      perror("unlink");
      return 1;
   }

   return 0;
}
```

In questo esempio, la chiamata di sistema mknod viene utilizzata per creare un dispositivo di carattere denominato /dev/ttyS0 con permessi di lettura e scrittura per l'utente proprietario, il gruppo proprietario e altri utenti. Il dispositivo associato è identificato da makedev(4, 64), che corrisponde alla porta seriale ttyS0 su un sistema Linux standard.

Dopo la creazione del file speciale, viene utilizzato nella chiamata di sistema open(), che consente di aprire il file speciale in modalità di lettura e scrittura. Alla fine del programma, il file speciale viene rimosso utilizzando la chiamata di sistema unlink()

85. Descrivere come si può creare un pipe anonimo ed un pipe con nome da shell, fornendo opportuni esempi di utilizzo.

In una shell Unix, è possibile creare un pipe anonimo e un pipe con nome utilizzando i comandi pipe e mkfifo, rispettivamente. Ecco come utilizzarli:

Pipe anonimo:

Il comando pipe crea un pipe anonimo, ovvero un canale di comunicazione tra due processi non correlati tramite un file descriptor. Il pipe anonimo può essere utilizzato per la comunicazione tra processi padre e figlio o tra due processi separati.

Esempio di utilizzo:

```
# Creazione di un pipe anonimo
pipe

# Esecuzione di un comando che scrive sul pipe
echo "Messaggio dal processo padre" > /dev/fd/3

# Esecuzione di un comando che legge dal pipe
cat <&3
```

In questo esempio, il comando pipe crea un pipe anonimo e restituisce due file descriptor: uno per la lettura e uno per la scrittura. Il file descriptor per la scrittura viene utilizzato dal comando echo per scrivere un messaggio sul pipe, mentre il file descriptor per la lettura viene utilizzato dal comando cat per leggere il messaggio dal pipe.

Pipe con nome:

Il comando mkfifo crea un pipe con nome, ovvero un file speciale nel file system che funge da canale di comunicazione tra due processi. Il pipe con nome può essere utilizzato per la comunicazione tra processi separati che condividono un file system comune.

Esempio di utilizzo:

```
# Creazione di un pipe con nome
mkfifo mypipe
```

```
# Esecuzione di un comando che scrive sul pipe
echo "Messaggio dal processo padre" > mypipe &

# Esecuzione di un comando che legge dal pipe
cat mypipe
```

In questo esempio, il comando mkfifo crea un file speciale chiamato mypipe. Il comando echo viene eseguito in background e scrive un messaggio sul pipe. Il comando cat legge il messaggio dal pipe e lo stampa a schermo.

È importante notare che il comando mkfifo blocca l'esecuzione del processo fino a quando non viene aperto il pipe in modalità di scrittura o di lettura. Inoltre, il pipe con nome persiste nel file system anche dopo la chiusura dei processi che lo utilizzano e deve essere esplicitamente rimosso tramite la chiamata di sistema unlink.

86. Descrivere gli attributi di dominio, tipo e protocollo di una socket

In C, gli attributi di una socket sono specificati attraverso la funzione socket() che accetta tre parametri:

Dominio (Domain): il primo parametro della funzione socket() specifica l'attributo di dominio, ovvero il tipo di ambiente di rete o di comunicazione locale in cui la socket opera. In C, gli attributi di dominio sono definiti come costanti simboliche definite in <sys/socket.h>. Alcuni esempi di attributi di dominio sono:

AF_UNIX (o AF_LOCAL): utilizzato per la comunicazione locale tra processi sullo stesso sistema operativo.

AF_INET: utilizzato per la comunicazione su Internet tramite il protocollo IP.

AF_INET6: utilizzato per la comunicazione su Internet tramite il protocollo IPv6.

AF_BLUETOOTH: utilizzato per la comunicazione tramite Bluetooth.

Tipo (Type): il secondo parametro della funzione socket() specifica il tipo di socket, ovvero il modello di comunicazione che viene utilizzato tra i processi. Anche in questo caso, i tipi di socket sono definiti come costanti simboliche in <sys/socket.h>. Alcuni esempi di tipi di socket sono:

SOCK_STREAM: utilizzato per la comunicazione affidabile e bidirezionale di flussi di byte (TCP).

SOCK_DGRAM: utilizzato per la comunicazione non affidabile di datagrammi di lunghezza fissa (UDP).

SOCK_RAW: utilizzato per la comunicazione a basso livello di pacchetti di rete.

SOCK_SEQPACKET: utilizzato per la comunicazione affidabile di datagrammi di lunghezza fissa.

Protocollo (Protocol): il terzo parametro della funzione socket() specifica il protocollo che viene utilizzato per la comunicazione sulla socket. Ci sono diversi protocolli disponibili, e la scelta dipende dal tipo di socket e dallo scopo della comunicazione. Alcuni esempi di protocolli sono:

IPPROTO_TCP: utilizzato con le socket di tipo SOCK_STREAM per la comunicazione affidabile di flussi di byte (TCP).

IPPROTO_UDP: utilizzato con le socket di tipo SOCK_DGRAM per la comunicazione non affidabile di datagrammi di lunghezza fissa (UDP).

IPPROTO_SCTP: utilizzato per la comunicazione affidabile di flussi di messaggi.

Per esempio, per creare una socket TCP/IP (AF_INET) di tipo SOCK_STREAM in C, il codice potrebbe essere:

```
#include <sys/socket.h>
#include <netinet/in.h>

int sockfd;
struct sockaddr_in serv_addr;

// creazione della socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);

// inizializzazione della struttura serv_addr
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(80);
inet_pton(AF_INET, "www.example.com", &serv_addr.sin_addr);

// connessione al server
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));

// utilizzo della socket per inviare e ricevere dati
send(sockfd, "GET / HTTP/1.1\r\nHost: www.example.com\r\n\r\n", strlen("GET / HTTP/1.1\r\nHost: www.example.com\r\n\r\n"), 0);
recv(sockfd, buffer, sizeof(buffer),
```

87. Descrivere i passi, lato server, per creare, utilizzare e distruggere una socket AF_UNIX.

88. Descrivere i passi, lato client, per creare, utilizzare e distruggere una socket AF_UNIX.

Ecco i passi per creare, utilizzare e distruggere una socket AF_UNIX lato client:

Creazione della socket: la creazione della socket lato client avviene attraverso la chiamata di sistema socket(), specificando come parametro il dominio AF_UNIX per indicare che si vuole utilizzare una socket di tipo Unix. Ad esempio:

```
int sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
if (sockfd == -1) {
   perror("socket creation failed");
   exit(EXIT_FAILURE);
}
```

Connessione al server: una volta creata la socket, il client deve connettersi al server. Ciò avviene attraverso la chiamata di sistema connect(), specificando come parametro l'indirizzo della socket del server. Ad esempio:

```
struct sockaddr_un server_address;
memset(&server_address, 0, sizeof(server_address));
server_address.sun_family = AF_UNIX;
strncpy(server_address.sun_path, "/tmp/server_socket", sizeof(server_address.sun_path) - 1);

if (connect(sockfd, (struct sockaddr *) &server_address, sizeof(server_address)) == -1) {
   perror("connect failed");
   exit(EXIT_FAILURE);
}
```

In questo esempio, si crea una struttura sockaddr_un per memorizzare l'indirizzo della socket del server. Viene inizializzata la famiglia di indirizzi AF_UNIX, e il percorso della socket del server viene copiato nella variabile sun_path della struttura. Successivamente, viene effettuata la connessione al server con la chiamata di sistema connect().

Comunicazione con il server: una volta stabilita la connessione, il client può inviare e ricevere dati dal server utilizzando le chiamate di sistema send() e recv(), rispettivamente. Ad esempio:

```
char buffer[1024];
```

```
int n = recv(sockfd, buffer, sizeof(buffer), 0);
if (n == -1) {
   perror("recv failed");
   exit(EXIT_FAILURE);
}

printf("Received message from server: %s\n", buffer);

const char *message = "Hello, server!";
if (send(sockfd, message, strlen(message), 0) == -1) {
   perror("send failed");
   exit(EXIT_FAILURE);
}
```

In questo esempio, il client riceve un messaggio dal server utilizzando la funzione recv(). Successivamente, invia un messaggio di risposta al server utilizzando la funzione send().

Chiusura della socket: una volta terminata la comunicazione con il server, il client deve chiudere la socket utilizzando la chiamata di sistema close(). Ad esempio:

```
close(sockfd);
```

Questo libero le risorse utilizzate dalla socket lato client.

89. Descrivere il comportamento di un server iterativo.

Un server iterativo è un tipo di server che gestisce le richieste dei client in modo sequenziale, ovvero gestisce una sola connessione alla volta. Il comportamento del server iterativo può essere descritto in diversi passi:

Creazione della socket: il server crea una socket attraverso la chiamata di sistema socket(), specificando il dominio, il tipo e il protocollo desiderati.

Assegnazione dell'indirizzo alla socket: il server assegna un indirizzo alla socket utilizzando la chiamata di sistema bind(), specificando l'indirizzo del server e la porta di ascolto.

Inizio dell'ascolto: il server inizia ad ascoltare le connessioni in ingresso attraverso la chiamata di sistema listen(), specificando la lunghezza massima della coda delle connessioni in attesa.

Accettazione delle connessioni: il server entra in un ciclo infinito in cui accetta le connessioni in ingresso attraverso la chiamata di sistema accept(). Questa chiamata blocca il server finché non viene ricevuta una nuova connessione.

Gestione della connessione: una volta accettata la connessione, il server esegue le operazioni richieste dal client attraverso la connessione stabilita. In questo tipo di server, le operazioni sono eseguite in modo sequenziale, ovvero una alla volta.

Chiusura della connessione: una volta completate le operazioni richieste dal client, il server chiude la connessione attraverso la chiamata di sistema close(). In questo modo la risorsa è liberata per ulteriori connessioni.

Ritorno al passo 4: una volta chiusa la connessione, il server torna al passo 4 per accettare una nuova connessione e ripetere il ciclo.

In pratica, il server iterativo gestisce le richieste dei client in modo sequenziale e non può gestire più di una connessione alla volta. Questo comportamento può essere adatto per piccole reti locali o per servizi non critici, ma non è ideale per reti di grandi dimensioni o per servizi ad alta disponibilità.

90. Descrivere il comportamento di un server concorrente.

Un server concorrente è un tipo di server che gestisce le richieste dei client in modo parallelo, ovvero può gestire più connessioni contemporaneamente. Il comportamento del server concorrente può essere descritto in diversi passi:

Creazione della socket: il server crea una socket attraverso la chiamata di sistema socket(), specificando il dominio, il tipo e il protocollo desiderati.

Assegnazione dell'indirizzo alla socket: il server assegna un indirizzo alla socket utilizzando la chiamata di sistema bind(), specificando l'indirizzo del server e la porta di ascolto.

Inizio dell'ascolto: il server inizia ad ascoltare le connessioni in ingresso attraverso la chiamata di sistema listen(), specificando la lunghezza massima della coda delle connessioni in attesa.

Accettazione delle connessioni: il server entra in un ciclo in cui accetta le connessioni in ingresso attraverso la chiamata di sistema accept(). Questa chiamata può essere bloccante o non bloccante, a seconda delle esigenze del server.

Creazione di un nuovo processo o thread: una volta accettata una nuova connessione, il server crea un nuovo processo o thread per gestire la connessione in modo concorrente. Il nuovo processo o thread esegue le operazioni richieste dal client attraverso la connessione stabilita.

Gestione della connessione: il processo o thread creato esegue le operazioni richieste dal client attraverso la connessione stabilita. In questo tipo di server, le operazioni sono eseguite in modo concorrente, ovvero contemporaneamente.

Chiusura della connessione: una volta completate le operazioni richieste dal client, il processo o thread creato dal server chiude la connessione attraverso la chiamata di sistema close(). In questo modo la risorsa è liberata per ulteriori connessioni.

Ritorno al passo 4: una volta chiusa la connessione, il processo o thread creato dal server torna al passo 4 per accettare una nuova connessione e ripetere il ciclo.

In pratica, il server concorrente può gestire più connessioni contemporaneamente attraverso la creazione di nuovi processi o thread. Questo comportamento è adatto per reti di grandi dimensioni o per servizi ad alta disponibilità, in cui la gestione delle richieste deve essere veloce ed efficiente. Tuttavia, la gestione di più connessioni contemporaneamente richiede una maggiore attenzione alla gestione delle risorse e alla sincronizzazione dei processi o thread creati dal server.

91. Descrivere il comportamento della funzione socketpair.

La funzione socketpair è una chiamata di sistema utilizzata per creare una coppia di socket connesse tra di loro. Questa chiamata di sistema è utilizzata principalmente per la comunicazione tra processi.

La funzione socketpair accetta tre parametri di input:

domain: il dominio delle socket. In genere, il dominio è AF_UNIX, che indica l'utilizzo di socket di tipo Unix domain.

type: il tipo di socket. In genere, il tipo è SOCK_STREAM, che indica l'utilizzo di socket di tipo stream.

protocol: il protocollo delle socket. In genere, il protocollo è 0, che indica l'utilizzo del protocollo di default.

La funzione socketpair crea una coppia di socket e le collega tra di loro. Le due socket sono identiche e possono essere utilizzate per la comunicazione bidirezionale tra i processi.

Una volta che la coppia di socket è stata creata, ogni processo deve utilizzare una delle due socket per la comunicazione. Ad esempio, il processo 1 potrebbe utilizzare la prima socket della coppia per inviare dati al

processo 2, mentre il processo 2 potrebbe utilizzare la seconda socket della coppia per inviare dati al processo 1.

In pratica, la funzione socketpair può essere utilizzata per creare una comunicazione bidirezionale tra due processi in modo semplice ed efficiente. Tuttavia, la gestione della comunicazione deve essere implementata dai programmi utilizzati nei processi coinvolti.

92. Descrivere la differenza tra Socket AF_UNIX ed AF_INET.

La differenza principale tra una socket AF_UNIX e una socket AF_INET sta nel tipo di indirizzo utilizzato per identificare il socket.

AF_UNIX (o AF_LOCAL) utilizza un indirizzo a livello di file system (pathname) per identificare il socket, mentre AF_INET utilizza un indirizzo IP e un numero di porta per identificare il socket.

In particolare, AF_UNIX viene utilizzato per la comunicazione locale all'interno di un singolo sistema operativo, mentre AF_INET viene utilizzato per la comunicazione su una rete TCP/IP.

Le socket AF_UNIX sono adatte per applicazioni che richiedono una comunicazione ad alta velocità e a bassa latenza, come ad esempio i server web. D'altra parte, le socket AF_INET sono adatte per applicazioni che richiedono una comunicazione su una rete, come ad esempio le applicazioni client-server.

Inoltre, le socket AF_UNIX consentono solo la comunicazione tra processi sullo stesso host, mentre le socket AF_INET consentono la comunicazione tra processi su host diversi. Infine, le socket AF_UNIX sono meno complesse da utilizzare rispetto alle socket AF_INET, che richiedono la specifica di un indirizzo IP e di una porta di rete.

93. Descrivere i passi, lato server e lato client, per creare, utilizzare e distruggere una socket

AF_INET.

Ecco i passi per creare, utilizzare e distruggere una socket AF_INET, sia dal lato server che dal lato client:

Lato Server:

Creazione della socket: utilizzare la funzione socket() per creare una nuova socket AF_INET. Questa funzione richiede come argomenti il dominio di comunicazione, il tipo di socket e il protocollo da utilizzare (in questo caso, AF_INET, SOCK_STREAM e IPPROTO_TCP).

Associazione della socket all'indirizzo IP e alla porta: utilizzare la funzione bind() per associare la socket all'indirizzo IP e alla porta del server. L'indirizzo IP è solitamente impostato su INADDR_ANY per indicare che il server accetterà le connessioni da qualsiasi indirizzo IP, mentre la porta può essere scelta arbitrariamente.

Mettere in ascolto la socket: utilizzare la funzione listen() per mettere in ascolto la socket e consentire al server di accettare le connessioni in arrivo dai client.

Accettare una connessione: utilizzare la funzione accept() per accettare una connessione in arrivo da un client. Questa funzione bloccherà il server finché non verrà ricevuta una connessione in arrivo.

Comunicazione con il client: utilizzare la socket accettata per comunicare con il client, utilizzando le funzioni di lettura e scrittura.

Chiusura della socket: una volta completata la comunicazione, utilizzare la funzione close() per chiudere la socket.

Lato Client:

Creazione della socket: utilizzare la funzione socket() per creare una nuova socket AF_INET.

Connessione al server: utilizzare la funzione connect() per connettersi al server, specificando l'indirizzo IP e la porta del server.

Comunicazione con il server: utilizzare la socket connessa per comunicare con il server, utilizzando le funzioni di lettura e scrittura.

Chiusura della socket: una volta completata la comunicazione, utilizzare la funzione close() per chiudere la socket.

In generale, l'ordine dei passi può variare a seconda dell'implementazione specifica. Inoltre, è importante gestire eventuali errori durante la creazione e l'utilizzo della socket.

94. Cosa rappresenta l'indirizzo di LOOPBACK?

L'indirizzo di loopback (127.0.0.1 per IPv4 e ::1 per IPv6) è un indirizzo speciale assegnato a una scheda di rete che consente a un dispositivo di comunicare con se stesso su una rete a livello di loopback o di loopback virtuale.

Quando un dispositivo invia dati a un indirizzo di loopback, i dati vengono inviati direttamente alla scheda di rete del dispositivo e non vengono trasmessi su una rete fisica. Ciò consente ai programmi di testare la loro capacità di comunicare con altri programmi sulla stessa macchina, senza dover effettivamente

trasmettere dati attraverso una rete. Inoltre, l'indirizzo di loopback viene spesso utilizzato per accedere ai servizi di rete locali, come un server web in esecuzione sulla stessa macchina.

95. Qual è il range possibile dei numeri di porta?

Il range possibile dei numeri di porta è da 0 a 65535.

Le porte dalla 0 alla 1023 sono comunemente note come "well-known ports" (porte ben conosciute) e sono riservate per l'uso da parte di servizi di rete standardizzati. Ad esempio, la porta 80 è associata al protocollo HTTP utilizzato dai server web, mentre la porta 25 è associata al protocollo SMTP utilizzato per inviare e-mail.

Le porte dalla 1024 alla 49151 sono comunemente note come "registered ports" (porte registrate) e possono essere utilizzate da applicazioni specifiche che richiedono l'uso di una porta.

Le porte dalla 49152 alla 65535 sono comunemente note come "dynamic ports" (porte dinamiche) e possono essere utilizzate da applicazioni che richiedono l'uso di una porta temporanea per la comunicazione di rete.

96. Come si può verificare la versione del kernel attualmente in uso?

Per verificare la versione del kernel attualmente in uso su un sistema Linux, è possibile utilizzare il comando "uname" con l'opzione "-r". Puoi fare così:

Apri un terminale o una console del tuo sistema e digita il seguente comando:

uname -r

Questo comando restituirà la versione del kernel attualmente in uso. Ad esempio, il risultato potrebbe essere "5.10.0-8-amd64" per un sistema Debian con kernel Linux versione 5.10.

Inoltre, se si desidera ottenere maggiori informazioni sul kernel, è possibile utilizzare il comando "uname" con altre opzioni. Ad esempio, "uname -a" restituirà informazioni dettagliate sul sistema, inclusa la versione del kernel, il nome dell'host, il tipo di macchina e altre informazioni pertinenti.

97. Scrivere un modulo kernel di hello world (funzioni di inizializzazione e cleanup).

Ecco un esempio di modulo kernel di "hello world" con funzioni di inizializzazione e cleanup:

```
#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>


static int __init hello_init(void)

{

   printk(KERN_INFO "Hello world!\n");

   return 0;

}
```

```
static void __exit hello_exit(void)
{
   printk(KERN_INFO "Goodbye world!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple hello world module");
```

Nel codice sopra, la funzione hello_init viene chiamata durante l'inizializzazione del modulo e la funzione hello_exit viene chiamata durante la pulizia del modulo. Entrambe le funzioni utilizzano la funzione printk per stampare messaggi sul registro del kernel.

Infine, le funzioni module_init e module_exit vengono utilizzate per registrare le funzioni di inizializzazione e pulizia del modulo. Le macro MODULE_LICENSE, MODULE_AUTHOR e MODULE_DESCRIPTION vengono utilizzate per fornire alcune informazioni sul modulo.

Per compilare e installare il modulo, salva il codice in un file chiamato hello.c, quindi esegui i seguenti comandi:

```
make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
sudo insmod hello.ko
```

Il primo comando compila il modulo, mentre il secondo carica il modulo nel kernel. Per rimuovere il modulo dal kernel, esegui il comando:

```
sudo rmmod hello
```

Nota: Questo è solo un esempio di modulo di "hello world" e non deve essere utilizzato in produzione senza essere adeguatamente testato e validato.

98. Come si carica un modulo nel kernel, come si può verificare la sua presenza, e come lo si

rimuove?

Per caricare un modulo nel kernel, puoi utilizzare il comando insmod seguito dal nome del file del modulo. Ad esempio, se il modulo si chiama "hello.ko", puoi eseguire il seguente comando:

sudo insmod hello.ko

Per verificare se un modulo è stato caricato correttamente nel kernel, puoi utilizzare il comando lsmod. Questo comando elenca tutti i moduli attualmente caricati nel kernel. Se il tuo modulo è presente nell'output, significa che è stato caricato correttamente.

Per rimuovere un modulo dal kernel, puoi utilizzare il comando rmmod seguito dal nome del modulo. Ad esempio, se il tuo modulo si chiama "hello", puoi eseguire il seguente comando:

sudo rmmod hello

È importante notare che un modulo può essere rimosso solo se non viene utilizzato da altri processi o moduli. Se un modulo dipende dal modulo che si sta cercando di rimuovere, sarà necessario rimuovere anche tutti i moduli dipendenti prima di poter rimuovere il modulo principale.

Inoltre, se il modulo è stato caricato durante l'avvio del sistema (ad esempio tramite /etc/modules), sarà necessario rimuoverlo da quel file per evitare che venga caricato nuovamente al prossimo avvio.

99. Cosa è una printk, e come può essere utilizzata?

In Linux, la funzione printk viene utilizzata per scrivere messaggi di log nel registro del kernel. Questi messaggi sono utili per il debug del kernel e degli eventuali driver o moduli del kernel che vengono eseguiti sul sistema.

La sintassi di base per printk è la seguente:

```
#include <linux/kernel.h>

void printk(const char *fmt, ...);
```

La funzione printk accetta un parametro di formato (simile a printf in C) e qualsiasi numero di argomenti successivi, a seconda del parametro di formato specificato. È possibile utilizzare i seguenti livelli di gravità per i messaggi di log:

KERN_EMERG: messaggi di emergenza

KERN_ALERT: messaggi di allerta

KERN_CRIT: messaggi critici

KERN_ERR: messaggi di errore

KERN_WARNING: messaggi di avviso

KERN_NOTICE: messaggi di notifica

KERN_INFO: messaggi informativi

KERN_DEBUG: messaggi di debug

Ad esempio, per scrivere un messaggio di log informativo utilizzando printk, si potrebbe utilizzare il seguente codice:

printk(KERN_INFO "Il mio messaggio di log informativo\n");

È possibile utilizzare la funzione printk in diversi contesti all'interno del kernel, ad esempio nelle funzioni di inizializzazione e pulizia dei moduli, nei driver del kernel, nelle funzioni del sistema di gestione delle interruzioni, ecc. È importante notare che printk ha un impatto sulla performance del sistema, quindi è necessario utilizzarlo con parsimonia e solo per lo scopo di debug.

I messaggi di log generati da printk possono essere visualizzati in diversi modi, ad esempio utilizzando il comando dmesg per visualizzare l'output del registro del kernel, utilizzando strumenti di monitoraggio come syslog-ng o rsyslog, o utilizzando strumenti di debugging come gdb o strace.

100. Cosa sono e come possono essere utilizzati il major number e il minor number di un

dispositivo? Come si possono assegnare ad un dispositivo, e come si possono visualizzare?

Il major number e il minor number di un dispositivo sono numeri interi utilizzati dal kernel di Linux per identificare un dispositivo specifico. Questi numeri sono assegnati dal kernel quando un dispositivo viene registrato e sono utilizzati dal kernel per associare il dispositivo con i driver corretti.

Il major number identifica il driver del dispositivo, mentre il minor number identifica una specifica istanza del dispositivo gestita dal driver. Ad esempio, se si ha una scheda audio con il driver snd, il major number sarà 116. Il minor number verrà assegnato dal driver per ogni istanza del dispositivo, ad esempio potrebbe essere 0 per la prima istanza, 1 per la seconda, e così via.

Per assegnare un major number e un minor number a un dispositivo, è necessario creare una voce nella directory /dev del sistema. Ad esempio, per creare un dispositivo chiamato /dev/mydevice, è possibile utilizzare i seguenti comandi:

sudo mknod /dev/mydevice c 250 0

sudo chmod 666 /dev/mydevice

In questo esempio, c indica che il dispositivo è di tipo carattere, mentre 250 è il major number e 0 è il minor number. La seconda riga imposta i permessi sul dispositivo per consentire a tutti gli utenti di leggere e scrivere.

Per visualizzare il major number e il minor number di un dispositivo, è possibile utilizzare il comando ls -l sulla voce del dispositivo in /dev. Ad esempio, per visualizzare il major number e il minor number del dispositivo /dev/sda, è possibile eseguire il seguente comando:

ls -l /dev/sda

L'output del comando mostrerà il major number e il minor number come parte dell'elenco di attributi del dispositivo.