UNIVERSITÀ DI PISA

Relazione sul progetto di Sistemi Operativi e Laboratorio

Samuele Iacoponi

A.A. 2022/2023

Sommario

Questo progetto consiste nell'implentare una **farm**, cioè uno schema di comunicazione tra 2 processi dove il processo ***Master***, composto da 1 o più *worker*, invia dei file *binari* al processo ***Collector*** dopo averci fatto delle operazioni.

1 Main

Per far comunicare i processi ho utilizzato 2 meccanismi:

1. **pipe:** è stata usata per notificare al processo *collector* di stampare la lista ordinata durante l'esecuzione;
2. **socket:** è stato usato per inviare i file con i risultati tra *master* e *collector.*

Dopo aver eseguito la *fork()* per creare i 2 processi, ho deciso di far fare da *padre* al processo ***Master***, e da *child* al ***Collector***.
Nel processo master è stato inserito il parsing dei comandi e una funzione ***masterfun*** che crea e avvia i thread worker. Una principale scelta implementativa è stata quello dell'uso di una **lista di supporto** che serve per salvare i file e gli argomenti passati al programma. Importante è la funzione **iscorrectfile** che controlla che i file passati come argomento siano validi, se lo sono possono essere inseriti nella lista concorrente che gli worker poi potranno usare per fare i calcoli, invece **recursiveSearch** visita ricorsivamente la cartella indicata come argomento

1.1 Segnali

In questo progetto era richiesto di gestire i segnali SIGHUP, SIGINT, SIGQUIT, SIGTERM, SIGUSR1. Per farlo nel processo *master* ho usato un thread che fa da signal handler, che viene terminato immediatamente nel caso viene inserito un argomento errato come opzione di un comando. Il segnale SIGPIPE è stato ignorato.
Per terminare il thread della gestione dei segnali viene inviato SIGUSR2

2 Master

Il *Master* è un processo multithreaded che è incaricato di creare n worker pari al numero passato come argomento del comando *-n*. Gli worker vengono creati con la funzione ***createworkers*** all'interno della funzione ***masterfun*** che viene invocata una volta fatto il parsing dei comandi, per farlo viene usato il comando *getopt* che permette di richiedere opzioni con argomenti obbligatori. Dopo aver creato i thread workers, la funzione *masterfun* comincia la copia dei file dalla lista di supporto alla lista condivisa **(sharedqueue)**.
La lista di supporto è stata utilizzata per gestire il comando *-q* che regola la dimensione della lista condivisa, quindi se questa lista è piena, la copia deve essere sospesa fino a quando un thread worker non ha consumato almeno un file. La struttura data usata per la lista di supporto e quella condivisa è la stessa:

1. **data**: il nome del file
2. **fd**: file descriptor del socket dove inviare il file

la differenza consiste nel fatto che, quando i nomi dei file vengono aggiunti nella lista di supporto il *file descriptor* è uguale a -1, mentre quando vengono aggiunti nella lista condivisa dai thread worker viene indicato anche il file descriptor corretto, in modo che i thread workers sappiano dove inviare il file

2.1 Gestione degli errori

Per gestire gli errori delle *System calls* utilizzo la macro vista a lezione

```
#define SYSCALL(r,c,e) if((r=c)==-1) { \
perror(e); \
exit(errno); \
}
```

In caso di argomenti errati come ad esempio, un numero come numero di workers, ho utilizzato 3 variabili:

- wrongNargument: l'argomento di *-n* non è un numero
- wrongQargument: l'argomento di *-q* non è un numero
- wrongTargument: l'argomento di *-t* non è un numero

Se si verifica uno di questi errori non vengono caricati i file nella lista di supporto e quindi il master si interrompe subito.

2.2 Attesa tra le richieste

Per gestire l'attesa ho usato le **variabili di condizione**. Attraverso il comando *-t ⟨delay⟩* le richieste vengono mandate con un ritardo agli worker, per l'attesa è stato usato *pthread_cond_timedwait*, usata per risvegliare dall'attesa nel caso dell'arrivo di un segnale come SIGINT per la richiesta di terminazione, l'attesa viene interrotta dal thread che fa da signal handler con una *signal()*.
Se invece l'attesa non viene interrotta e il timer è scaduto allora la funzione restituisce **ETIMEDOUT** che viene gestita.
Dato che la funzione *pthread_cond_timedwait* deve essere chiamata con la lock acquisita, quest'ultima viene fatta solo se l'argomento del comando *-t* è != 0

2.3 Workers

I thread vengono creati con la funzione *create_workers*, poi cominciano a estrarre in testa dalla lista i file aprendoli con *fopen* in modalità *rb*.
Sempre usando *variabili di condizione* se la lista è vuota gli workers si mettono in attesa fino a

quando non vengono risvegliati dall'inserimento di un file nella coda condivisa dal thread master. L'operazione che viene fatta sul file è la seguente:

$$\sum_{i=0}^{N-1} i * file[i]$$

3 Collector

Il collector è il processo figlio che ha il compito di ricevere il nome dei file e i risultati della somma dal master.
Per inviare i file ho deciso di creare una struttura dati contente appunto

1. nome del file
2. risultato della somma

chiamata **collectedfiles**.

Come da specifica il collector deve stampare, non solo la lista dei file in ordine crescente di somma alla fine, ma anche quando il processo master riceve SIGUSR1.

3.1 Ordinamento

Per l'ordinamento dei file e dei loro risultati è stato utilizzato il ***mergesort***, un algoritmo di ordinamento che applica la tecnica del *dividi e conquista*.
Consiste nel dividere la lista in 2 ricorsivamente poi le metà vengono ordinate e poi fuse tra loro in modo da ottenere la lista ordinata, questo algoritmo porta vantaggi e svantaggi:
Vantaggi:

- Efficiente: ha una complessita Ø(n*log(n)) sia in caso di lista ordinata che nel caso di liste non ordinate
- Stabile: perchè mantiene uguale l'ordine degli elementi uguali.

Svantaggi:

- Richiede più spazio per la fusione delle liste, quindi può risultare meno efficiente con liste molto grandi

3.2 Segnali

La richiesta era quella di mascherare tutti i segnali gestiti dal master, per farlo ho usato ***sigprocmask*** con opzione *SIG_BLOCK*, il quale è stato utilizzabile in quando il processo è single-thread.

3.3 Select

Durante l'esecuzione del collector ho utilizzato una select in quanto il master comunicava sia con la *socket* che con la *pipe*, così da poter capire quale dei 2 era pronto per inviare dati.

- **Se i dati provengono dalla socket:** il collector riceve la lunghezza del file, il nome del file, e la somma relativa a quel file e poi salva tutto in una lista della struttura dati *collectedFiles*
- **Se i dati provengono dalla pipe:** significa che il master ha ricevuto il segnale SIGUSR1

3.4 Stampa risultati

La stampa dei risultati viene fatta sia alla fine del programma, oppure anche durante l'esecuzione quando il processo master ha ricevuto SIGUSR1. Quando questo avviene il master attraverso il thread signal handler invoca la funzione ***requestprint()*** che consiste nel cambio del valore della variabile *print* da 0 ad 1, la scrittura al collector attraverso la pipe e di conseguenza la stampa

3.5 Terminazione

Il collector viene terminato quando non sono più presenti elementi nella coda condivisa nel processo master, in quanto la socket viene chiusa e il collector viene risvegliato sulla read e aggiorna il valore della variabile **endconnection** uscendo sia dal ciclo for che dal ciclo while e poi eseguendo le operazioni di pulizia

4 Note e funzioni di utilità

La *comunicazione* è stata implementata con **un'unica socket** condivisa tra tutti gli workers, questo vuol dire che è stato necessario aggiungere una sincronizzazione per far in modo che ognuno aspetti di scrivere se la socket è occupata. Per realizzare questa sincronizzazione è stata utilizzata una variabile mutex *mtx_socket*.

4.1 Funzioni di utilità

Sono state usate delle funzioni utilità:

- **Pthread_mutex_...**: le funzioni come la lock, wait, signal sono state riscritte per aggiungere un controllo sugli errori, come visto a lezione è stata messa la 'P' maiuscola rispetto alla funzione standard
- **isNumber:** una funzione vista a lezione che permette di capire se l'argomento è un numero oppure no
- **NULL_CHECK:** una macro per controllare se un puntatore è null, usata principalmente per controllare il valore di ritorno di una malloc

5 Esecuzione del programma

La cartella viene fornita con i codici .c, cartella con gli headers, il file per eseguire i test e il makefile.

5.1 Makefile

Per eseguire il programma è necessario eseguire il comando ***make*** questo creerà le cartelle:

- **src:** dove verranno spostati i file .c
- **obj:** dove verranno creati i file oggetto
- **bin:** dove verranno creati i file eseguibili della farm e generafile per creare i file su cui verranno eseguiti i test e spostato lo script bash per i test

Una volta creati i file eseguibili, come da richiesta, è stato creato il target ***test***, quindi lanciando il comando ***make test*** verrà eseguito lo script bash contenente i test. Per l'esecuzione ho aggiunto il comando *chmod +x* per rendere eseguibile lo script.
Un altro target è stato implementato, ***clean*** per rimuovere i file oggetto e tutti i file creati durante i test. Viene eseguito lanciando il comando ***make clean.***