UNIVERSITÀ DI PISA

Corso di Sistemi operativi e laboratorio a.a. 20/21
Docente Massimo Torquati
Progetto SOL – Corso A e B – a.a. 22/23

Fagioli Elia
Matricola: 580115

Sommario:
Il progetto *farm* è volto a rappresentare un sunto del programma del corso di Sistemi Operativi, in particolare coinvolge gli aspetti affrontati durante gli Assegnamenti svolti attraverso le lezioni di laboratorio.
Nello specifico la mia implementazione del progetto affronta vari aspetti quali: fondamenti di programmazione in C, gestione di strumenti di debug quali GDB e valgrind, realizzazione di un Makefile, corretta gestione di path e directory, chiamate di sistema, gestione di thread POSIX e processi, comunicazione socket (in locale), gestione dei segnali con conseguente realizzazione di maschere e signal handler ed infine utilizzo di script bash per il testing e in generale della bash Unix nelle varie operazioni.

Scopo del progetto:
Lo scopo del progetto è di realizzare uno schema di comunicazione tra processi e thread implementando i corretti meccanismi di mutua esclusione sulle risorse e di comunicazione tra entità di sistema e allo stesso tempo gestire i segnali.

Makefile – comandi:
- **make all**: compilazione e collegamento per i target farm, generafile e collector
- le regole per i target farm, generafile e collector specificano dipendenze e azioni per compilare e collegare i rispettivi target con i file oggetto appropriati
- regola per compilare i file oggetto
- **make test**: il comando testa lancia il comando “bash test.sh”
- **make clean** pulisce il sistema dai file generati dalla compilazione e dal test.

Descrizione dell'architettura:
L’architettura è composta da:
- file header “.h”
- file “.c”
- file oggetto “.o”
- e da uno script Bash test.sh.

La directory **includes** contiene gli **Header**, i file di intestazione:
- *strutture.h*: L’header strutture.h raccoglie al suo interno:
    - l’insieme degli include utilizzati all’interno del progetto
        - include delle varie librerie standard
        - #include<sys/socket.h> e #include<sys/un.h> per lavorare con il socket in locale
        - #include<pthread.h> per i thread worker
        - #include<sys/stat.h> #include<dirent.h>per i file e le directory
        - #include<signal.h> per gestire i segnali
        - #include<sys/wait.h> per attendere terminazione dei processi figli

    - la definizione (extern) degli elementi per la comunicazione su socket che sono necessari al master e ai thread worker
    - flag (extern) che serve a comunicare ai thread worker quando il master ha terminato l'inserimento di elementi nella coda.
    - Le struct per la coda e per gli argomenti della comunicazione socket.
- *worker_thread.h*: fornisce una dichiarazione della funzione worker_thread per poter essere utilizzata in masterWorker.c visto che l'implementazione di worker_thread è su un altro file.

La directory **src** contiene l'insieme dei file sorgente del progetto:

- *generafile.c*: codice per la generazione di file binari contenenti valori long.
  (fa parte del materiale per il testing del progetto)
- *masterWorker.c*: rappresenta l'implementazione del processo principale di questo progetto.
    - Gestisce i vari segnali tramite signal_handler (sigaction).
    - Genera il processo *collector* tramite una *fork* e l'utilizzo di *execl* sul figlio.
    - Il processo masterWorker (padre) stabilisce una connessione socket locale con il processo *collector*: la sincronizzazione per lo stabilire questa connessione è svolta tramite un meccanismo temporizzato di chiamate *connect* nella funzione set_connection() del master per un massimo di MAX_RETRIES attendendo la chiamata *listen* del processo Collector sulla socket.
    - Esegue il casting degli argomenti di argv tramite *parse_options* ottenendo: il numero di thread worker da generare, la dimensione massima della coda concorrente, il tempo di attesa tra l'invio di due richieste successive e la lista di filename da inserire nella coda concorrente.
    - Inizializza la coda, crea un pool di thread worker e attende la loro terminazione (questa indica l'avvenuta comunicazione di tutti gli elementi al processo collector.
    - Infine il processo chiude la connessione socket (condivisa con i thread worker) e aspetta la terminazione del figlio (collector) tramite la wait(NULL).
- *worker_thread.c*: contiene il comportamento dei thread worker generati dal processo masterWorker
    - maschera i segnali che vengono gestiti dal masterWorker
    - il singolo worker entra in un ciclo fino a quando non terminano le elaborazioni delle richieste inserite nella coda da parte del master.
    - Opera in mutua esclusione:
        - Per fare la pop dalla coda concorrente
        - Per comunicare sulla singola connessione col collector
    - Non compie in mutua esclusione l'operazione di lettura del file binario dove calcola il result da passare al collector perché le richieste (i file name) sono separate.
- *collector.c*: rappresenta l'implementazione del master della comunicazione socket.
    - Il processo collector maschera i segnali che vengono gestiti dal processo masterWorker.
    - Il processo collector si mette in ascolto sulla connessione stabilita con il masterWorker e riceve dei socket_args inviati singolarmente dai thread worker che elaborano il contenuto dei file name passati loro come argomenti, la *write* in risposta ai thread indica l'avvenuta ricezione dei socket_args.
    - Il processo collector esce da un ciclo while(true) quando riceve la chiusura della connessione socket da parte del master, successivamente stampa (ordinando in maniera crescente), libera la memoria allocata dinamicamente e termina.

Descrizione delle scelte di progettazione:

Le principali scelte di progettazione richieste dalla consegna del progetto erano:

- Implementare il codice del programma in file diversi
- La scelta di quale dei processi svolge il ruolo di master nella comunicazione socket
- La scelta tra singola connessione socket o una connessione per ogni thread.

Motivazione delle scelte progettuali:

La separazione in file diversi: masterWorker, collector e worker_thread è stata eseguita in base alla specifica della consegna del progetto, i codici sorgente in questione sono stati inseriti nella cartella src.

È compito del Makefile gestire la compilazione dei file con il riferimento ai file di intestazione (nella cartella includes) nella cartella obj che conterrà quindi i file oggetto .o.
Gli eseguibili si trovano all'esterno di queste sottodirectory.

La scelta del processo collector come master della comunicazione socket è stata fatta perché è suo il compito di tener traccia dei risultati ricevuti dai vari client (thread worker) per poi restituirli in maniera ordinata al termine della complessiva richiesta del processo masterWorker.
In sostanza il collector è un server che elabora la richiesta di masterWorker che vuole ordinare per valore crescente i result di una lista di file name.

La scelta tra singola connessione e una connessione per ogni worker è stata fatta sulla base di questo ragionamento: considerando che il numero di thread worker è variabile e la struttura dati passata al collector dai worker è relativamente piccola (long + una stringa di massimo 255caratteri) è più conveniente utilizzare la soluzione con una singola connessione socket.
Ciò perché l'overhead di creare e gestire più connessioni potrebbe superare il beneficio di avere più connessioni per processare i dati in parallelo, soprattutto se il numero di worker è relativamente basso.
Questo non significa che la scelta di una connessione per thread sia sbagliata ma nella mia implementazione ho dei risparmi nell'utilizzare una singola connessione, aprendola e chiudendola una sola volta.
In caso di utilizzo di una connessione per ogni thread avrei dovuto gestire il multiplexing sul server tramite funzione select di <select.h>
La mia valutazione si basa sulla considerazione che:
Se il numero di worker è relativamente basso e la dimensione dei dati da scambiare è contenuta (come in questo caso), utilizzare una singola connessione potrebbe essere la soluzione migliore.
Al contrario, se il numero di worker è elevato e i dati da scambiare sono molto grandi, potrebbe essere conveniente utilizzare più connessioni per sfruttare il parallelismo e ottimizzare le prestazioni complessive del sistema.

Implementazione:
L'implementazione dei file del progetto segue lo schema architetturale spiegato precedentemente.
Andando più nel concreto, analisi dell'implementazione dei meccanismi principali:

- masterWorker.c

In quello che rappresenta il processo padre di tutta l'architettura avviene un'interpretazione degli argomenti passati da linea di comando all'eseguibile farm (così verrà chiamato l'eseguibile generato da masterWorker.c).
Il parsing di argv viene svolto tramite la funzione *parse_options* dopo aver realizzato Options e passandolo come argomento alla funzione stessa.
L'implementazione di questo meccanismo di lettura viene effettuata tramite getopt.
Questa funzione della libreria standard di C interpreta gli argomenti da linea di comando e funge in aggiunta da filtro quando al nostro programma vogliamo passare delle opzioni specifiche che possono richiedere argomenti o meno (":" nella definizione delle opzioni).
La funzione scorre gli ulteriori elementi di argv (oltre alle opzioni ci sono i file name) riconoscendoli come nomi dei file passati al programma e inserisce il tutto nella lista file di Options e verrà processata dalla funzione *path_check_to_queue* per inserire i nomi dei file nella coda in maniera concorrente.
Il codice del masterWorker assegna la funzione signal_handler ai segnali richiesti dalla consegna tramite sigaction (della libreria sigset) che è thread-safe (a differenza di signal).
Gestione dei segnali:

- nel primo caso viene impostata a true una variabile globale *sig_received* che termina l'inserimento delle richieste nella coda concorrente se quest'ultimo non era già terminato e attende l'elaborazione dei worker della coda e la terminazione di collector con la sua conseguente stampa dei valori.
- Anche nel caso di SIGUSR1 viene impostata a true la variabile *sigusr1_received* che rimanda a una funzione di invio di richiesta di stampa in cui viene acquisita la mutex sulla connessione del processo (e non sulla coda concorrente) e viene inviato un socket_args al collector contenente la

stringa “STAMPA RESULTS”, quando il collector riceve questo pacchetto provvede a ordinare e stampare gli elementi fino a quel momento ricevuti e a continuare la normale esecuzione del programma, infatti dopo il lancio della funzione il masterWorker ricomincia l’iterazione corrente di inserimento nella coda.

Il masterWorker fa una fork() lanciando tramite execl il processo collector e successivamente si occupa di inizializzare la connessione ad esso tramite la funzione set_connection, questa itera per un MAX_RETRIES numero di volte eseguendo una connect a distanza di 1 secondo da ogni chiamata fino a quando il processo figlio effettua la listen.
Questo meccanismo di sincronizzazione permette al master di attendere che ci sia qualcuno in ascolto delle sue richieste.
Il masterWorker inizializza una coda concorrente e passerà come argomento dei thread il puntatore a questa coda, successivamente genera quindi il pool di n thread worker.
La funzione ricorsiva *path_check_to_queue* è svolta dal processo masterWorker per effettuare l’inserimento dei file name nella coda concorrente delle richieste, la funzione naviga la file_list di *options* verificando se il file name passato sia un file vero e proprio, inoltre la gestione del passaggio come argomento (opzione -d) di una directory è gestita aggiungendo i file alla lista tramite una funzione ricorsiva.
Ogni file riconosciuto come tale dalla funzione viene inserito tramite mutua esclusione con i thread worker nella coda concorrente, questo è reso possibile tramite due condition variable nella struct della coda.
A questo punto ciò che resta da fare al processo masterWorker è impostare a true un flag che segnala ai worker che l’inserimento è terminato e attendere la loro terminazione e conseguentemente chiudere la connessione col collector.
Chiusura di connessione che una volta ricevuta dal collector lo farà terminare e la wait(NULL) nel masterWorker serve proprio ad attendere tale terminazione.

- worker_thread.c

Come prima cosa vengono mascherati i segnali (che devono essere gestiti dal processo masterWorker).
Dopo il casting del puntatore alla coda passatogli come argomento il singolo thread entra in un while(true) e compie le operazioni di pop in mutua esclusione col masterWorker e con gli altri worker e successivamente tramite un’altra mutex sulla connessione socket invia il file path insieme al result calcolato dalla lettura del file di quest’ultimo sulla socket verso il collector.
Il calcolo del risultato del file non è effettuato all’interno delle lock perché la gestione della sincronizzazione permette che venga assegnato un file per volta.
Il worker termina quando la coda è vuota e il master ha terminato la funzione di inserimento comunicandolo ai thread tramite il flag globale di fine put. Oppure quando alla ricezione di un segnale di interruzione il master termina anticipatamente la funzione di inserimento e i thread svuotano la coda prima di terminare.

- collector.c

Rappresenta l’implementazione del processo collector, viene lanciato dal figlio di masterWorker tramite execl, passando oltre al path e al comando la lista degli argomenti che è vuota; infatti, il main del collector ha void come argomento.
Come prima cosa il processo maschera i segnali, che devono essere gestiti solamente da masterWorker.
Effettua la funzione cleanup() che fa unlink del socket.sck ovvero lo rimuove dal sistema in caso di mancata cancellazione in un’esecuzione errata precedente.
Con la funzione atexit(cleanup) si specifica che la funzione va chiamata anche al termine del processo.
Viene creata una variabile server_addr di tipo struct sockaddr_un, che rappresenta l'indirizzo del server e viene inizializzata a tutti 0 da memset.
Tramite il campo sun_family viene messo AF_LOCAL che sta a rappresentare l’utilizzo in locale della socket Unix, il path socket.sck finisce invece nel campo sun_family.
Le successive operazioni per impostare il server in ascolto:

- bind associa server_fd all’indirizzo specificato in server_addr.
- Listen mette il socket in modalità di ascolto con il parametro 1 che indica il numero massimo di connessioni.
- Accept serve ad accettare la connessione in entrata che arriverà dal masterWorker e il descrittore del masterWorker verrà messo in client_fd.

Successivamente itera in un while(true) andando ad effettuare una *read* ad ogni iterazione e salvando il numero di bytes letti in una variabile intera n.
In base ai bytes letti distingue tra una ricezione di un socket_args, un errore o la chiusura della connessione, in quest'ultimo caso esce dal ciclo, stampa l'array di results, lo dealloca e termina.
L'array di socket_args chiamato results è incrementato dinamicamente tramite una realloc ad ogni ricezione.
È presente anche un controllo per la ricezione del segnale di stampa quando il master riceve SIGUSR1.

Test e valutazione:
Descrizione dei test effettuati:
Per testare il programma viene utilizzato lo script fornito come materiale *test.sh*.
Questo script bash verifica se è stato generato l'eseguibile *generafile* del file omonimo .c sempre facente parte del materiale fornito.
Successivamente crea un file *expected.txt* con una serie di risultati attesi al suo interno.
Utilizza l'eseguibile generafile per creare dei file binari con dei long al proprio interno e alcuni finiscono in una directory testdir che può a sua volta contenere sotto-directories.
Successivamente alla generazione dei file per il testing, lo script test.sh si occupa di verificare 5 testcase:

1) Chiama farm con 2 thread worker e lunghezza di coda 1.
   Successivamente confronta il contenuto di expected.txt con l'output del programma tramite il comando "diff"
2) Riesegue il test precedente stavolta con 8 worker e lunghezza di coda 16
3) Con 1 worker e dimensione di coda 1, questa volta passa l'opzione delay 1000, ovvero richiede 1000 secondi tra l'inserimento di una richiesta e la successiva da parte del master.
   Invia il segnale SIGTERM dopo circa 5 secondi e verifica il codice di uscita del processo sia corretto.
4) Esegue il programma farm e controlla eventuali problemi utilizzando valgrind
5) Sempre tramite valgrind riesegue farm questa volta controllando se ci sono perdite di memoria

In aggiunta a questi test ne sono stati fatti di ulteriori come:

- la generazione di directory di cui alcune anche vuote
- la realizzazione di file con nome superiore ai 255 (cosa non possibile in sistemi Unix)
- la gestione del segnale SIGUSR1 (non previsto in test.sh)
- uno "stress test" con numero elevato di thread generati
- un test con dimensione della coda superiore al numero dei file
- …

Valutazione delle prestazioni
Le prestazioni del progetto rispecchiano a pieno l'analisi fatta all'inizio, in particolare l'utilizzo della singola connessione non provoca alcun tipo di ritardo sull'architettura, inoltre c'è un corretto utilizzo dell'allocazione e della conseguente deallocazione di memoria verificabile tramite valgrind.
La mutua esclusione, dove è utilizzata, viene implementata nella maniera corretta evitando deadlock e race condition, in particolare nella possibilità di comunicare al collector un thread per volta e per gestire i casi in cui la coda concorrente si riempie o si svuota durante l'inserimento e le pop dei worker.
La gestione dei segnali viene effettuata correttamente da parte del processo master anche durante il test 3 e anche simulando un'esecuzione per testare la ricezione di SIGUSR1.
Lo scopo del progetto è stato realizzato e il programma sviluppato risulta idoneo ai requisiti.