RELAZIONE PROGETTO

Gabriele Cristofaro
Mat. 567757 (CORSO A)
Github: https://github.com/gabriele0011/OS_project_farm

Descrizione generale

Il progetto implementa due processi che comunicano tramite una sola connessione socket. Il processo MasterWorker genera il processo Collector mediante una chiamata di sistema *fork()*, di cui si esamina il valore di ritorno per distinguere i due processi. Il processo MasterWorker, inizialmente, nel *main()*, si occupa di effettuare il parsing della riga di comando, acquisendo le informazioni di setting, se presenti, e i file in input, che vengono controllati e poi accettati (se risultano regolari). I file vengono memorizzati in una lista, utilizzata poi per riempire la coda concorrente dei task da elaborare tenendo conto della capienza massima supportata.

Il processo MasterWorker genera il processo Collector dopo aver momentaneamente gestito i segnali, che verranno successivamente gestiti in modo permanente. Il processo Collector gestisce i segnali in maniera definitiva appena viene generato. I due processi si connettono mediante la socket ed iniziano a comunicare. Il processo Collector ha la possibilità di ricevere "messaggi" dai thread, che inviano i risultati ottenuti elaborando i file, la richiesta di chiusura del sistema causata dalla ricezione di specifici segnali o la stampa dei risultati parziali. I risultati vengono recepiti uno alla volta, poiché il processo Collector è single-threaded. A tal proposito, ho ritenuto non particolarmente vantaggioso impostare il progetto facendo in modo che il Collector fungesse da server e i thread da client. In sostanza, sarebbe cambiata la gestione del sistema di comunicazione socket, che avrebbe previsto l'installazione di una socket per ogni thread. Nel Collector, sarebbe stato necessario implementare un server sequenziale con l'utilizzo della *select()*. Questo meccanismo sarebbe particolarmente vantaggioso nel caso in cui anche il processo Collector fosse multi-threaded,

permettendo una ricezione multipla dei risultati da client multipli. Nel mio caso, ho gestito l'invio dei messaggi da parte dei thread e la ricezione di questi ultimi (tramite socket) garantendo, attraverso delle variabili mutex, mutua esclusione nell'invio e nella ricezione. Dunque, in ogni instante è possibile che al massimo un thread comunichi con il processo Collector, che si tratti di ricezione di risultati o di altri eventi.

Il codice è stato interamente scritto da me, senza l'utilizzo di librerie o codice prodotto da terzi. È stato testato su macchine virtuali Linux Ubuntu 20.04.4 LTS e Xubuntu (fornita nel corso).

Suddivisione dei file che compongono il progetto

I file con estensione .c sono associati ai rispettivi file header che hanno diversa estensione ma stesso nome, a differenza di generafile.c che non ne possiede uno.

main.c

Nel file main.c è definita la funzione *main()* che effettua una chiamata alla funzione *parsing(),* che appunto esegue il parsing della riga di comando. Inoltre, sono definite altre due funzioni ovvero *file_check()* e *take_from_dir()*. La prima verifica che un file passato come argomento sia regolare, mentre l'altra esplora una directory passata come argomento (e le rispettive sotto directory ricorsivamente) prelevando i file contenuti all'interno.

master_thread.c

Implementazione del processo MasterWorker. L'omonima funzione contiene il codice che implementa il processo, all'interno del quale viene generato anche il processo Collector.

collector.c

Implementazione del processo Collector, mediante l'omonima funzione, e della routine qs_*cmp()* prevista dall'algoritmo di ordinamento QuickSort usato al suo interno per ordinare i risultati.

pool_worker.c

Implementazione della pool di thread e di ulteriori funzioni utilizzate dai thread. La procedura di creazione, *create_pool_worker()*, viene invocata dal processo MasterWorker per generare i thread. Inoltre, sono presenti le funzioni *thread_start()* e *thread_proc()* che implementano rispettivamente il lato consumatore e l'elaborazione successiva una volta ottenuto un task. Infine, la funzione *send_res()* invia i risultati calcolati in precedenza al processo Collector.

aux_function.c

Contiene la definizione di funzioni ausiliare utilizzare nel progetto, tra cui *sub_timespec()* per calcolare il tempo trascorso tra due istanti di tempo, *msleep()* simile ad una sleep a cui viene passato come argomento il tempo espresso in millisecondi. Infine, funzioni di controllo per il parsing e funzioni *write_n()* e *read_n()* che includono read e write con controllo sul numero di byte attesi.

conc_queue.c

Implementazione di una coda concorrente di tipo FIFO con funzioni di inserimento, estrazione, deallocazione, e stampa.

generafile.c

Funzione che genera dei file binari, fornita nella consegna.

list.c

Implementazione di una lista con rispettive funzioni di inserimento, estrazione, deallocazione, stampa, rimozione di un nodo, ricerca.

Scelte implementative

Gestione dei segnali

I segnali vengono gestiti tramite signal handlers e signal mask. Il

processo MasterWorker ignora inizialmente i segnali da gestire mascherandoli mediante una signal mask finché i gestori permanenti non saranno installati e installando loro un handler *SIG_IGN*. Il processo Collector installa i propri handler definitivi e resetta la signal mask ereditata. Infine, il processo MasterWorker installa i propri handler definitivi e resetta la maschera precedentemente settata. Il segnale SIGPIPE viene gestito da entrambi i processi mediante un handler, assegnando SIG_IGN e di conseguenza ignorandolo. Inoltre, è stato gestito il segnale SIGCHLD con un apposito handler nel processo MasterWorker. Il processo Collector invia un segnale di questo tipo al processo padre (in questo caso il processo MasterWorker) una volta terminato. Di default il segnale verrebbe ignorato, nel nostro caso è utilizzato per acquisire l'informazione di terminazione del Collector nel processo MasterWorker.

Comunicazione socket

La comunicazione socket è unica. Il processo MasterWorker si mette in ascolto, mentre il Collector cerca di connettersi. È previsto un timer di riconnessione da parte del processo che fa la *connect()*, ovvero il Collector, dopo il quale non sarà possibile stabilire la connessione. Questo meccanismo è utile nel momento in cui un processo non risponda e/o per sincronizzare la connessione.

Gestione dei task e thread

È stato implementato un meccanismo produttore-consumatore per la gestione dei task verso i thread con una coda concorrente, variabili mutex e condition variables.

Output del Collector

L'output del processo Collector viene ordinato tramite l'algoritmo *QuickSort*, efficiente su un array di n elementi, all'interno del quale vengono memorizzati in precedenza i risultati ricevuti da ordinare. Oltre al caso di ricezione del segnale SIGUSR1, i risultati vengono stampati ordinati anche alla chiusura del sistema.

Test aggiuntivi

È stato inserito un *test 6* aggiuntivo che invia un segnale di tipo SIGURS1 per verificare che il processo Collector stampi i risultati parziali ottenuti fino a quell'istante.