Petrocchi Gianmarco 599116

Progetto Farm

Progetto corso Sistemi Operativi e Laboratorio; 09/01/2023

Premessa

Il progetto ***farm*** consiste nella realizzazione di un programma composto da due processi, denominati rispettivamente *MasterWorker* e *Collector. MasterWorker* è un processo multi-thread composto da un thread Master e da 'n' thread Worker.
Il programma prende come argomenti una lista di nomi di file binari contenenti numeri lunghi ed un certo numero di argomenti, che serviranno per poter configurare l'esecuzione del *MasterWorker* e l'ambiente su cui andranno a lavorare i thread worker. Il processo *Collector* viene generato dal processo *MasterWorker,* i due processi comunicano attraverso una connessione socket AF_UNIX.
Il nome del generico file, che viene letto dal *MasterWorker,* viene inviato ad uno dei thread worker tramite una coda sincronizzata condivisa. Il thread worker si occupa di leggere dal disco il contenuto dell'intero file il cui nome ha ricevuto in input, e di effettuare un calcolo sugli elementi letti e quindi inviare il risultato ottenuto, unitamente al nome del file, al processo *Collector* tramite la connessione socket precedentemente stabilita. Il processo *Collector* attende di ricevere tutti i risultati dai worker ed al termine stampa i valori ottenuti sullo standard output, ordinando la stampa in ordine crescente di risultato.
Gli argomenti che opzionalmente possono essere passati al processo *MasterWorker* sono i seguenti:

- -n <nthread> specifica il numero di thread worker del processo *MasterWorker* (valore di default 4)
- -q <qlen> specifica la lunghezza della coda concorrente (valore di default 8)
- -d <directory-name> specifica una directory in cui sono contenuti file binari ed eventualmente altre directory contenenti file binari; i file binari dovranno essere utilizzati come file di input per il calcolo
- -t <delay> specifica un tempo in millisecondi che intercorre tra l'invio di due richieste successive al thread worker da parte del thread Master (valore di default 0)

Il processo *MasterWorker* deve gestire i segnali SIGHUP, SIGINT, SIGQUIT, SIGTERM, SIGUSR1. Alla ricezione del segnale SIGUSR1 il processo *MasterWorker* notifica il processo *Collector* di stampare i risultati ricevuti sino a quel momento (sempre in modo ordinato). Alla ricezione dei segnali SIGINT, SIGQUIT, SIGTERM e SIGHUP, il processo deve completare i task eventualmente presenti nella coda dei task da elaborare, non leggendo più eventuali altri file in input, e quindi terminare dopo aver atteso la terminazione del processo *Collector* ed effettuato la cancellazione del socket file.
Il processo *Collector* maschera tutti segnali gestiti dal processo *MasterWorker.* Il segnale SIGPIPE deve essere gestito opportunamente dai due processi.

Struttura del progetto

Il progetto è suddiviso in vari file, ognuno dei quali ha un proprio file header (.h) con i metodi e variabili che necessita per il funzionamento:

- *Main.c:* file la cui unica funzionalità è quella di generare due processi tramite il metodo fork(). Il processo padre è il *MasterWorker*, il processo figlio è il *Collector.* Riceve in input i vari parametri da riga di comando, che verranno passati al processo padre per poter procedere con la configurazione.
- *Master_Thread.*c: processo padre che riceve in input i parametri passati da riga di comando. Nel caso in cui l'utente non scrive nessun argomento su CLI, verranno utilizzati dei valori di default che sono dichiarati all'interno del file *parameters.h.* Il *Master_Thread* si occupa di: assegnare ai segnali che devono essere gestiti un handler, questo viene fatto subito in maniera tale da non incorrere in system call interrotte; creare il socket per la comunicazione con il processo figlio (*Collector*); creare il pool di thread worker e la coda sincronizzata attraverso i parametri ricevuti; mandare a intervalli regolari di tempo, specificato dal parametro opzionale -t, i nomi dei file binari ai thread worker usando la coda creata; terminare l'esecuzione liberando la memoria allocata dinamicamente.
- *Collector.c:* processo figlio che comunica con il *MasterWorker* attraverso il socket AF_UNIX. Riceve i nomi dei file binari e i risultati calcolati dai thread worker per ogni file. Questi vengono stampati in ordine crescente di

risultato. Per poter garantire questo tipo di stampa, il *Collector* utilizza una linked list che inserisce gli elementi in ordine crescente di risultato. L'implementazione di questo tipo di struttura dati è presente nel file *ascending_queue.c.*

- *Thread_Worker.c:* thread che vengono creati dal *MasterWorker,* il quale passa a loro i parametri necessari: socket per la comunicazione con il *Collector;* puntatore alla coda concorrente per poter estrarre i nomi dei file; due puntatori a flag per la gestione dei segnali che vengono catturati dal *MasterWorker,* e a seconda del flag settato il thread worker comunicherà al *Collector* quello che dovrà fare; mutex che viene utilizzato per gestire l'accesso concorrente al socket. I thread worker si occupano di accedere in modo concorrente alla coda per prelevare i nomi dei file e calcolare il risultato sulla base dei numeri lunghi letti. Il risultato e il nome del file vengono comunicati al *Collector* attraverso il socket, il quale è comune a tutti i thread worker, quindi non si ha un socket diverso per ogni thread worker. Proprio per questo motivo viene usato un mutex, in maniera tale che due scritture di due thread worker diversi non si sovrappongano, mandando quindi un dato errato al processo *Collector.*
- *queue.*c: file contenente l'implementazione della coda sincronizzata che conterrà i nomi dei file. Per realizzare la coda è stata implementata una linked list con accesso sincronizzato di tipo FIFO (First In First Out). Per rendere le operazioni di inserimento ed estrazioni costanti (O(1)), la coda è stata realizzata attraverso due puntatori: uno per la testa, che viene manipolato solo se la lista è vuota, in caso di operazione di inserimento, e per l'estrazione; uno per la coda che viene manipolato in caso di operazioni di inserimento. Gli elementi di questa coda sono la coppia <filename, lunghezza del filename>, tengo traccia della lunghezza del filename perché serve per poter scrivere il numero esatto di byte nella socket. A differenza delle classiche linked list questa ha una sua dimensione, la quale viene settata attraverso il parametro opzionale -q <qlen> passato attraverso CLI.
- *ascending_queue.*c: file contenente l'implementazione della coda che viene utilizzata dal processo *Collector* per stampare i risultati in ordine crescente. Gli elementi di questa coda sono la coppia <filename, result>. Questa lista non è sincronizzata, in quanto viene usata solo dal processo *Collector.*
- *mystring.*c: file contenente alcuni metodi per la creazione e concatenazione di stringhe, utili per la creazione dei path assoluti dei file presenti nella directory specificata con il flag -d.
- *utilities.*c: file contenente metodi per la lettura e scrittura su socket, questi due metodi sono denominati rispettivamente "writen" e "readn"; metodi per il controllo dei tipi di parametri passati da CLI; metodi per il controllo di file e directory regolari.

Scelte implementative

- La scrittura delle varie stringhe che avvengono sul socket viene fatta tramite il metodo ***writen(),*** il quale prende come parametri di ingresso: la socket su cui andare a scrivere, il contenuto da andare a scrivere che è una stringa o un valore numerico, il numero di bytes da scrivere. Il metodo utilizza la system call ***write()*** all'interno di un ciclo while fino a quando non sono stati scritti il numero di byte passati come argomento. Durante la scrittura possono avvenire degli errori, quindi il metodo ***writen()*** restituirà determinati valori: se il metodo ***write()*** imposta il valore di errno allora viene restituito -1; se viene chiuso il socket dal processo lettore, allora la scrittura non avviene e viene restituito 0; se non accade nessuno di questi errori viene restituito il numero di bytes scritti.
  Per ogni scrittura viene controllato il valore restituito dal metodo ***writen()***. Se restituisce -1 oppure 0 viene terminato il processo scrittore, che sia il MasterWorker o il singolo thread worker.
  Il protocollo di comunicazione, per spedire le stringhe sul socket, è il seguente: prima scrittura che contiene la lunghezza della successiva stringa da spedire; seconda scrittura che contiene la stringa effettiva.
  Viene fatto ciò perché così il processo lettore crea un buffer, per poter contenere la stringa effettiva che viene acquisita con la seconda lettura, che sia della lunghezza giusta così da non sprecare ulteriore memoria.

La lettura delle varie stringhe dal socket viene fatta tramite il metodo ***readn(),*** il quale come parametri ha gli stessi della ***writen()*** specificati sopra. Anche la logica di questo metodo è simile al metodo ***writen()*** con la sola differenza che nel ciclo while usa la system call ***read()***.
Per ogni scrittura viene controllato il valore restituito dal metodo ***readn().*** Se restituisce -1 oppure 0 viene terminato il processo lettore, ovvero il Collector.

- La configurazione del *MasterWorker* e dell'ambiente di esecuzione per i thread worker viene fatto attraverso un ciclo while che usa la funzione ***getopt()***, la quale permette di identificare i parametri specificati da CLI e di prendere i valori con la variabile globale ***optarg*** del metodo. Tutti i valori sono delle stringhe e a seconda del parametro incontrato viene processato in maniera diversa:
    - nel caso di -n, il valore di optarg viene prima convertito in un numero con il metodo ***isNumber*** definito nel file *utilities.h* e poi castato ad intero. Il metodo ***isNumber*** restituisce -1 se il parametro passato non è un numero, altrimenti fa la conversione e restituisce il risultato ottenuto. Possono verificarsi degli errori nella specifica del valore per il parametro -n: valore non numerico (quindi ***isNumber*** restituisce -1); valore numerico ma minore o uguale a 0. Se è andato tutto bene si usa il valore specificato da CLI, altrimenti si usa il valore di default (4).
    - nel caso di -q, il valore di optarg viene prima convertito in un numero con il metodo ***isNumber*** e poi castato ad intero. Possono verificarsi degli errori nella specifica del valore per il parametro -q: valore non numerico; valore numerico ma minore o uguale a 0. Se è andato tutto bene si usa il valore specificato da CLI, altrimenti si usa il valore di default (8).
    - nel caso di -d, il valore di optarg viene controllato se è un path di una directory esistente con il metodo ***isDir*** definito nel file *utilities.h*, il quale restituisce 0 in caso affermativo, -1 in caso contrario. Quindi se il valore restituito dal metodo è 0 viene usato il valore di optarg come eventuale directory da scansionare/visitare, altrimenti viene usato il valore di default (NULL).
    - nel caso di -t, il valore di optarg viene prima convertito in un numero con il metodo ***isNumber*** e poi castato ad intero. Possono verificarsi degli errori nella specifica del valore per il parametro -t: valore non numerico; valore numerico ma minore di 0. Se è andato tutto bene si usa il valore specificato da CLI, altrimenti si usa il valore di default (0).

- La creazione del pool di thread worker viene fatto con il metodo ***spawn_threads()*** che restituisce un array di tipo ***pthread_t***. Tutti i thread worker hanno i seguenti argomenti: socket per la comunicazione con il processo *Collector*, coda sincronizzata per l'estrazione dei nomi dei file, puntatori ai flag "flag_print" e "flag_others_signals" che monitorano la ricezione dei segnali SIGUSR1, SIGINT, SIGQUIT, SIGTERM e SIGHUP da parte del processo *MasterWorker*, mutex per regolare l'accesso al socket per la scrittura. Tutti questi parametri vengono passati con il metodo ***pthread_create()*** attraverso una struct chiamata ***thread_parameter***, definita nel file *Thread_Worker.h*. Il processo *MasterWorker* attende la terminazione dei thread worker con una ***pthread_join().***

- La lettura e l'inserimento in coda dei nomi dei file specificati su CLI, viene fatto con un for che cicla sull'array degli argomenti passati, ovvero ***char* argv[]***.
Questo array ha la seguente forma (con tutti i parametri specificati):
<nome_eseguibile> -n <nthread> -q <qlen> <lista di nomi dei file> -d <directory-name> -t <delay>
I parametri possono essere intercambiati a piacere, tranne il <nome_eseguibile> che non deve essere specificato da CLI ma è il primo argomento presente all'interno dell'array ***char* argv[].***
Quando l'indice del ciclo for è uguale a 0 non viene fatto niente perché ***argv[0]*** contiene <nome_eseguibile>; quando l'indice identifica un elemento ***argv[i]*** del tipo "-n", "-q", "-d" o "-t" viene incrementato di uno il valore dell'indice, in questo modo quando il for incrementerà a sua volta l'indice ci troveremo due posizioni

più avanti e salteremo quindi il valore del parametro specificato, ad esempio: considerando il contenuto dell'array specificato sopra, quando l'indice è uguale a 1 avremo ***argv[1] = -n***, l'indice assumerà valore 2 (quindi ***argv[2] = <nthread>***) e poi il for incrementerà nuovamente l'indice a 3 e avremo ***argv[3] = -q.***
Quando l'indice del for identifica un nome di un file binario: questo viene controllato se è un nome di un file esistente, se sì viene inserito all'interno della coda sincronizzata, il processo *MasterWorker* viene interrotto per un totale di millisecondi pari a quelli specificati con il parametro -t (0 di default). In questo caso l'indice del for viene incrementato una sola volta (direttamente dal ciclo for), perché quando si trova il primo nome del file significa che gli altri si troveranno tutti di seguito.
Non viene usata quindi una struttura dati ausiliaria dove vengono inseriti i nomi dei file per poi in seguito inserirli nella coda sincronizzata, perché se ne avessimo specificati tantissimi sprecheremo due volte O(n) tempo, con n pari al numero dei file. La prima volta con il ciclo for per inserirli nella ipotetica struttura dati, e una seconda volta visitando l'intera struttura dati per metterli nella coda sincronizzata condivisa.

- La scansione/visita dell'eventuale directory specificata da CLI, viene fatta attraverso il metodo ricorsivo ***scan_dir()*** definito nel file *Master_Thread.h*. Il metodo prende il path della directory, la apre e visita il contenuto di essa attraverso il metodo ***readdir()*** usato in un ciclo while fino a quando non restituisce NULL. Il metodo ***readdir()*** restituisce un puntatore ad una struttura di tipo ***dirent*** (definita nel file *dirent.h*), la quale contiene diverse informazioni sul file presente nella directory. Il ciclo while attua queste operazioni: controlla se il nome della struttura dirent è diverso da "." e da "..". Se sì viene usato il metodo ***str_concatn()***, definito nel file *mystring.h*, per concatenare il path della directory passato al metodo ***scan_dir()***, il carattere '/' e il nome della struttura dirent, così si costruisce una stringa che rappresenta il path assoluto del file. Si controlla se quest'ultimo identifica un file regolare oppure una directory, se ci troviamo nel primo caso inseriamo il nome del file nella coda sincronizzata, altrimenti richiamiamo ricorsivamente il metodo ***scan_dir()*** passando come parametro il path appena creato, in modo tale da scansionare/visitare il possibile contenuto della sottodirectory.

- La terminazione del *MasterWorker* può avvenire per diverse cause:
    1. Il processo MasterWorker ha terminato di inserire tutti i nomi dei file nella coda sincronizzata.
    2. Durante l'esecuzione del processo Collector è avvenuto un errore prima di instaurare la connessione con il processo MasterWorker.
    3. Durante l'esecuzione del processo MasterWorker è avvenuto un errore prima di instaurare la connessione con il processo Collector.
    4. È avvenuto un errore durante la lettura sulla socket da parte del processo Collector.
    5. È avvenuto un errore durante la scrittura sulla socket da parte del processo MasterWorker.

  Nel primo caso di terminazione viene usato il metodo ***shutdown_master_thread().*** Quest'ultimo opera nella seguente maniera: per ogni thread presente all'interno del pool di thread inserisce nella coda la stringa "finish", in maniera tale che ognuno dei thread ne estrae una copia e termina la propria esecuzione, la quale viene attesa dal processo *MasterWorker* con il metodo ***pthread_join()***. Successivamente viene usato il socket per spedire al processo Collector la stringa "finish", in maniera tale che questo processo possa terminare stampando la lista dei risultati ordinati in ordine crescente e liberando successivamente tutta la memoria allocata. Attende la terminazione del processo Collector con il metodo ***waitpid(),*** e succesivamente viene chiuso il socket di comunicazione e viene liberata tutta la memoria allocata dal MasterWorker.
  Per il secondo e il terzo caso vengono eseguiti rispettivamente i metodi ***shutdown_master_thread_prematurely()*** e ***shutdown_collector(),*** i quali si limitano a liberare quella poca memoria allocata, in quanto non è stata effettivamente instaurata la connessione tra i due processi.

Per il quarto e il quinto caso si verificano i valori restituiti dai metodi ***readn()*** e ***writen()***, ovvero se è stato restituito -1 o 0. Quindi si controlla se è avvenuto un errore nell'esecuzione dei due metodi, oppure se uno dei due estremi della comunicazione ha chiuso il socket.

- La coda sincronizzata, usata per l'inserimento e l'estrazione dei nomi dei file, è resa tale grazie all'uso di un mutex (denominato *mutex_queue*) e di due condition variables per mettere in attesa i thread del verificarsi di determinati eventi. Queste due condition variables sono associate rispettivamente: all'evento che la coda è vuota; all'evento che la coda è piena, in quanto quest'ultima ha una sua dimensione prefissata. Queste due condition variables sono fondamentali, in quanto l'utilizzo della coda segue il paradigma producer-consumer: il producer è il processo *MasterWorker* che inserisce i nomi dei file nella coda, non può inserirli se la coda è piena; il consumer sono i thread worker che estraggono i nomi dei file dalla coda, non possono estrarli se la coda è vuota.

Gestione segnali

Il processo *MasterWorker* e *Collector* devono gestire i segnali SIGINT, SIGHUP, SIGTERM, SIGQUIT, SIGPIPE, SIGUSR1. Questi segnali vengono mascherati dai due processi in maniere differenti: il processo *Collector* ignora tutti i segnali assegnandogli SIG_IGN. Il processo *MasterWorker* ignora solamente il segnale SIGPIPE, per tutti gli altri è associato un signal handler che modifica il valore di due flag sig_atomic_t denominati rispettivamente "flag_print", "flag_others_signals".
Il primo flag viene modificato quando viene catturato il segnale SIGUSR1, il secondo viene modificato quando vengono catturati uno tra i segnali SIGINT, SIGQUIT, SIGHUP e SIGTERM.
L'aggiornamento dei due flag viene visto dai thread worker, i quali comunicheranno al processo *Collector* ciò che dovrà fare attraverso la socket. Se viene intercettato il segnale SIGUSR1 il flag "flag_print" viene incrementato di valore, il thread worker scrive sulla socket la stringa "print" che indica al *Collector* di stampare la lista ordinata dei risultati, e il valore di "flag_print" viene decrementato dal thread worker stesso. Questo lavoro viene fatto finché il valore del flag non è uguale a 0, quando raggiunge questo valore significa che tutti i segnali SIGUSR1 sono stati gestiti dai thread worker. Quindi se viene catturato 10 volte il suddetto segnale, anche in un tempo molto breve, avremo 10 stampe da parte del processo *Collector* della lista ordinata. La modifica del valore del flag è resa atomica grazie alla definizione della variabile stessa, ovvero come variabile sig_atomic_t, quindi gli aggiornamenti di essa non vengono persi e vengono fatti in un singolo ciclo di clock.
La cattura del segnale SIGUSR1 non implica la terminazione del processo *MasterWorker* né tanto meno del processo *Collector*.
Se viene intercettato uno tra i segnali SIGINT, SIGQUIT, SIGTERM e SIGHUP viene modificato il valore del flag "flag_others_signals". Il processo MasterWorker eseguirà il metodo "shutdown_master_thread", il funzionamento di esso è citato nel paragrafo *"Scelte implementative"*.
La ricezione di questi quattro segnali da parte del processo *MasterWorker* implica la terminazione di quest'ultimo e del processo *Collector,* il quale stamperà la lista ordinata dei risultati ricevuti fino a quel momento.

Test di esecuzione

I test sono presenti nel Makefile, al suo interno ci sono 7 target: "compilation" che compila sia generafile.c che Main.c i quali hanno rispettivamente come target "compilation_generafile" e "compilation_farm"; "test" che esegue lo script test.sh; "test1" e "test2" che eseguono rispettivamente lo script test1.sh e test2.sh presenti nella cartella script; "compilation_farm_debug_signals" che compila il file Main.c aggiungendo il flag -DDEBUG, in questo modo è possibile fare il testing dei segnali perché verrà stampato sullo *stderr* il pid del processo padre a cui mandare i segnali. Per eseguirli basta posizionarsi all'interno della cartella dove c'è Makefile e digitare da CLI "make <nome target>" per eseguire un target specifico, altrimenti digitare "make" per eseguire il primo tag del Makefile ovvero "compilation".
Di seguito saranno mostrati altri test di esecuzione, riguardo a casistiche non comprese negli script bash.

```
osboxes@osboxes:~/Desktop/ProgettoSOL$ ./farm file1.dat file2.dat file3.dat file4.dat file5.dat file6.dat file7.dat
Parent process 12610

Will use default value for option -q
Will use default value for option -n
Will use default value for option -t
No directory name set from command line
Child process 12611

file6.dat isn't a regular file
file7.dat isn't a regular file
Final result
103453975 file2.dat
153259244 file1.dat
293718900 file3.dat
584164283 file4.dat

Child process 12611 has terminated successfully
Parent process 12610 has terminated successfully
```

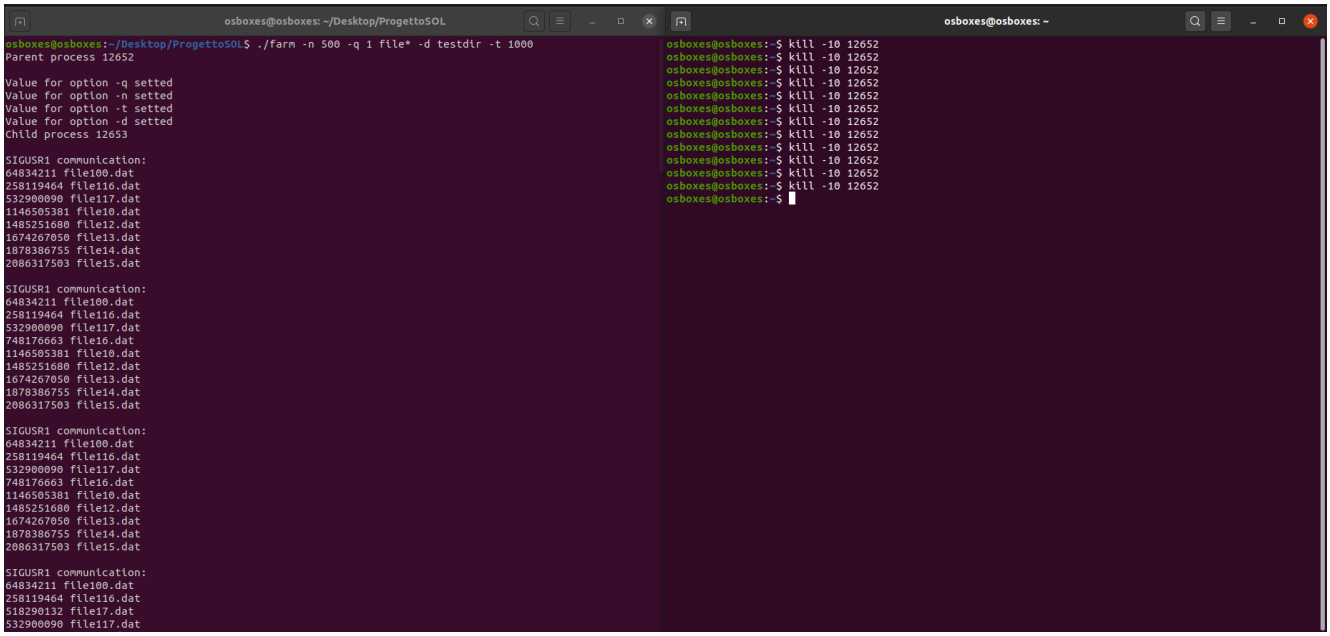
1. Nessun parametro viene passato da riga di comando, vengono usati i valori di default.

```
osboxes@osboxes:~/Desktop/ProgettoSOL$ ./farm -n pippo -q 2 file1.dat file2.dat file3.dat file4.dat file5.dat file6.dat file7.dat
Parent process 12624

Option -n require an integer parameter
Processo figlio 12625 terminato
Parent process 12624 has terminated unsuccessfully
```

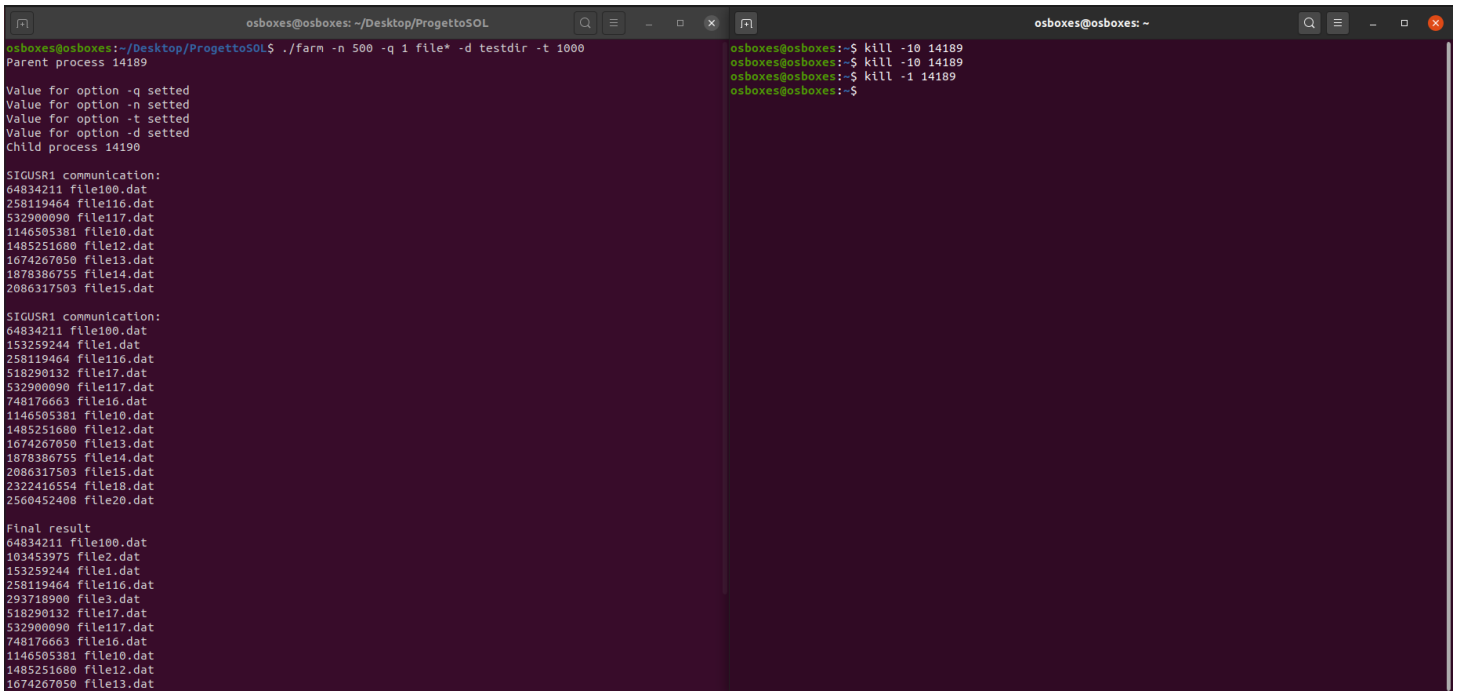
2. Passaggio di valore errato per parametro -n. In questo caso viene passata una stringa.

```
osboxes@osboxes:~/Desktop/ProgettoSOL$ ./farm -n 2 -q 3 file* -d Esempi
Parent process 12627

Option -d require a legit directory name
Processo figlio 12628 terminato
Parent process 12627 has terminated unsuccessfully
```

3. Passaggio di un nome di directory non esistente/valido



4. Viene mandato ripetutamente il segnale SIGUSR1 (kill -10 <ppid>). Si avranno un numero di stampe pari al numero di segnali mandati



5. Viene mandato due volte il segnale SIGUSR1 e successivamente il segnale SIGINT (kill -1 <ppid>)