Report di Sistemi Operativi
DisastrOS IPC Message Queue

Giorgia D'Armiento
1711864
darmiento.1711864@studenti.uniroma1.it

INTRO

Linux supporta tre tipi di meccanismi di comunicazione tra processi apparsi per la prima volta in Unix System V; questi tre meccanismi sono: code di messaggi, semafori e memoria condivisa.
In questo progetto andiamo ad implementare il meccanismo IPC basato sulle code di messaggi a partire dal sistema operativo disastrOS visto durante le lezioni del corso di sistemi operativi.

Una coda di messaggi è un elenco collegato di messaggi archiviati all'interno del kernel e identificati da un identificatore di coda di messaggi. Le code di messaggi consentono ai processi di comunicare in modo asincrono inviando messaggi l'uno all'altro, questi messaggi vengono archiviati in una coda, in attesa di essere elaborati e vengono eliminati dopo essere stati elaborati.

Aggiungiamo le seguenti syscall per la gestione del IPC con le code di messaggio:

1) msgQueueCreate()
2) msgQueueOpen()
3) msgQueueClose()
4) msgQueueUnlink()
5) msgQueueWrite()
6) msgQueueRead()

Viene creata una nuova coda da **msgQueueCreate()** o viene aperta una coda esistente da **msgQueueOpen()**.
Viene chiuso il descrittore associato alla coda di messaggi con **msgQueueClose()**, viene rimossa la coda di messaggi una volta che tutti i descrittori associati ai processi che l'hanno aperta sono stati chiusi con **msgQueueUnlink()**.
I nuovi messaggi vengono aggiunti alla fine di una coda da **msgQueueWrite()**, che aggiunge il messaggio puntato da msg_ptr alla coda a cui si riferisce mqdes, in questa syscall si specificano anche la dimensione del messassgio msg_len e la priorità, priority, da dare al messaggio. Nella syscall è presente la printf relativa alla msg_queue_list, che stampa la coda di messaggi.
I messaggi vengono prelevati da una coda da **msgQueueRead()**. Non dobbiamo recuperare i messaggi in un ordine first-in, first-out, ma in base alla loro priorità, rimuove quello con priorità più alta (più vecchio) dalla coda a cui si riferisce mqdes e lo inserisce nel buffer puntato da msg_ptr, specifichiamo anche la dimensione msg_len del messaggio.

Tutti i processi possono scambiare informazioni attraverso l'accesso a una coda di messaggi di sistema comune. Il processo di invio (producer) inserisce un messaggio su una coda che può essere letto da un altro processo (consumer) .

Ad ogni coda di messaggi viene assegnata un'identificazione in modo che i processi possano selezionare la coda di messaggi appropriata. Nel nostro caso chiamiamo tale identifricatore name, ovvero il nome assegnato alla coda di messaggio, per identificarla tramite un puntatore. Di conseguenza aggiungiamo anche la funzione Findbyname per la ricerca di coda tramite il nome.

1) **disastrOS_msgQueueCreate**

La **int disastrOS_msgQueueCreate(const char *name)** crea la coda di messaggi con il nome preso in input, la funzione prende in input il nome della coda di messaggi da creare.

Allochiamo la subqueue e la coda tramite pull allocator e inseriamo le strutture allocate nella resources_list e msg_queue_list.

La funzione ritorna 0 in caso di successo e il relativo errore DSOS_EMQ_CREATE altrimenti.

2) **disastrOS_msgQueueOpen**

La **int disastrOS_msgQueueOpen(const char *name)** apre la coda di messaggi con il nome preso in input, prende in input il nome della coda da aprire.

Troviamo il puntatore della coda di messaggi con il nome richiesto, tramite la funzione findbyname e lo assegna ad un puntatore, se non esiste ritorna l'errore DSOS_EMQ_NOEXIST.

Creiamo il descrittore per la risorsa in questo processo, in caso di errore ovvero l'allocazione non è avvenuta correttamente restituisce DSOS_EMQ_NOFD.
Allochiamo sia il descrittore che il suo puntatore correttamente e incrementiamo il contatore dell'ultimo file descriptor, aggiungiamo entrambe le risorse appena create alla lista dei descrittori aperti associati alla coda e dei puntatori ai descrittori.

La funzione ritorna il file descriptor associato al descrittore del processo in caso di successo, altrimenti un codice di errore (DSOS_EMQ_NOFD e DSOS_EMQ_NOEXIST).

3) **disastrOS_msgQueueClose**

La **int disastrOS_msgQueueClose(int mqdes)**: prende in input il descrittore (file descriptor) mqdes associato alla coda di messaggi da chiudere.

Dichiariamo il file descriptor della coda di messaggi che si vuole chiudere, troviamo il file descriptor corrispondente alla coda di messaggi e lo assegniamo al puntatore desc, se la coda di messaggi è già chiusa, lo comunichiamo con l'errore.
Stacchiamo il descrittore (deallochiamo) e il suo puntatore dalla lista dei descrittori e lista dei puntatori e ci assicuriamo che questo avvenga con le free.
Se tutti i descrittori dei processi che hanno aperto la coda vengono deallocati dalla lista dei descrittori, quindi tutti i processi che hanno aperto la coda chiudono i descrittori associati ad essa, la coda verrà distrutta, quindi eseguiamo la funzione msgQueueUnlink.

Ritorna 0 in caso di successo, DSOS_EMQ_CLOSE altrimenti.

4) disastrOS_msgQueueUnlink

La **int disastrOS_msgQueueUnlink(const char *name)**: prende in input il nome della coda di messaggi da rimuovere.

Specifichiamo il nome della coda di messaggi che si vuole chiudere, trova il puntatore della coda di messaggi con il nome richiesto, tramite la funzione findbyname e assegna NULL al descrittore, se il puntatore è diverso da NULL, il descrittore della coda di messaggi mette tutti i puntatori dei suoi descrittori a NULL.
Se il descrittore della coda di messaggio è diverso da zero, non esiste la coda di messaggio con quel nome e ritorna l'errore, se il descrittore della coda di messaggi ha dimensione maggiore di 0, ci sono descrittori aperti nella coda di messaggi, stacchiamo e deallochiamo il puntatore alla coda di messaggi e il suo descrittore e ci assicuriamo che questo avvenga correttamente con le free di verifica.

La coda verrà distrutta una volta che tutti i processi che l'hanno aperta chiudono i propri descrittori associati alla coda.

Ritorna 0 in caso di successo, DSOS_EMQ_UNLINK altrimenti.

5) disastrOS_msgQueueWrite

La **int disastrOS_msgQueueWrite(int mqdes, const char *msg_ptr, unsigned msg_len, unsigned int priority)**: prende in input il messaggio, puntato da msg_ptr, da inserire nella coda di messaggi identificata dal descrittore mqdes; viene specificata la lunghezza del messaggio puntato da msg_ptr e la priorità assegnata al messaggio.

Dichiariamo il descrittore della coda di messaggi desc tramite la ricerca del file descriptor mqdes, se desc è diverso da zero e quindi la ricerca non è stata completata, la coda di messaggi con il file descriptor non è stata aperta per il processo in running.

Dichiariamo la coda di messaggi mq e se la dimensione della coda di messaggi è quanto il massimo numero di messaggi per msg queue, la coda di messaggi è piena e si ritorna zero, se la lunghezza del messaggio è maggiore della massima lunghezza del messaggio, il messaggio è troppo grande e si ritorna l'errore.

Definiamo la sottocoda di priorità e allochiamo un nuovo messaggio nella subqueue con priorità, in particolare, viene allocato il messaggio tramite Message_alloc, se msg è diverso da zero allora l'allocazione fallisce.
Il messaggio viene inserito nella lista di messaggi associata alla sottocoda con quella priorità.

Viene ritornata la lunghezza del messaggio in caso di successo, DSOS_EMQ_WRITE altrimenti.

6) disastrOS_msgQueueRead

La **int disastrOS_msgQueueRead(int mqdes, char *msg_ptr, int unsigned msg_len)**: prende in input il messaggio, puntato da msg_ptr, da inserire nella coda di messaggi identificata dal descrittore mqdes; viene specificata la lunghezza del messaggio puntato da msg_ptr (dimensione del buffer puntato da msg_ptr =256), per poter verificare che la dimensione del messaggio da scrivere sia entro questo valore e la priorità assegnata al messaggio.

Dichiariamo il descrittore della coda di messaggi desc tramite la ricerca del file descriptor mqdes, se desc è diverso da zero e quindi la ricerca non è stata completata, la coda di messaggi con il file descriptor non è stata aperta per il processo in running.

Dichiariamo la coda di messaggi mq e inizializziamo a NULL la subqueue, eseguiamo un ciclo for per la subqueue che legge i messaggi in ordine di priorità nel caso in cui ci siano, se non ci sono messaggi da leggere nella subqueue viene restituito l'errore.
Prendiamo il primo messaggio della lista di messaggi (quello con priorità più alta) e verifichiamo la lunghezza del messaggio, e lo inserisce nel buffer a cui punta msg_ptr tramite la funzione strcpy. Rimuovo messaggio da lista di messaggi della sottocoda e decremento la size.

Questa funzione ritorna la lunghezza del messaggio letto in caso di successo, altrimenti DSOS_EMQ_READ.

GESTIONE DEGLI ERRORI

DSOS_EMQ_CREATE →errore in caso il name inserito è pari a NULL oppure non è possibile allocare una nuova coda di messaggio o un puntatore alla coda di messaggio con name.

DSOS_EMQ_NOFD →errore in caso di file descriptor non valido.

DSOS_EMQ_NOEXIST →errore in caso di coda di messaggi non trovata.

DSOS_EMQ_CLOSE →errore in caso di fallimento nella deallocazione del puntatore al descrittore o del descrittore con fd.

DSOS_EMQ_UNLINK →errore se non esiste la coda di messaggio con quel nome oppure errore in caso di fallimento della deallocazione del message queue o del message queue ptr per il message queue.

DSOS_EMQ_WRITE → errore nel caso in cui la coda di messaggi con fd=mqdes non è stata aperta per il processo in running, oppure viene superata la massima lunghezza dei messaggi MAX_TEXT_LEN, oppure in caso di fallimento nell'allocazione di un nuovo messaggio.

DSOS_EMQ_READ →errore nel caso in cui la coda di messaggi con fd=mqdes non è stata aperta per il processo in running, errore restituito nel caso in cui non si è in grado di leggere i messaggi perché la coda di messaggi è piena, oppure il messaggio non raggiunge la lunghezza minima.

GESTIONE COSTANTI

MAX_NUM_MESSAGES_PER_MSG_QUEUE →che rappresenta il numero massimo di messaggi contenuti nella message queue, se il valore è < 3 è possibile testare l'errore causato dalla coda piena.

MAX_NUM_PRIORITIES → indica il numero di sottocode che verranno create nella coda di messaggi.

MAX_TEXT_LEN → rappresenta la dimensione del messaggio scritto, se il valore è < 6 testiamo errori di scrittura.

MAX_NUM_MESSAGES → indica numero massimo di messaggi che si possono scrivere se < 9 causerà segmentation fault nel momento in cui cerca di allocare l'ultimo messaggio.

ESECUZIONE

Per testare il funzionamento del progetto, eseguire:
- git clone https://github.com/giorgiadarmi/Progetto-disastrOS.git

- cd ProgettoDisastrOS

- make

- ./disastrOS_test per l'esecuzione