Progetto Sistemi Operativi 2024-2025

Autori

Nome/Cognome	Matricola	Email
Davide gioetto	1129099	davide.gioetto@edu.unito.it
Massimo Doni	1118325	massimo.doni@edu.unito.it

1. Compilazione

Per compilare, eseguire i comandi nella root del progetto:

```
mkdir bin build
make
```

2. Esecuzione

Per eseguire il progetto, utilizzare il seguente comando nella root:

```
./bin/manager_main
```

3. Configurazione

Le configurazione sono modificabili dal file `config.conf`. Il file di configurazione presenta un parametro aggiuntivo `log_level` che permette di impostare la verbosità dell'output, lasciato a 2 l'output soltanto le informazioni richieste dall'implementazione del progetto, impostandolo a 0 mostrerà anche informazioni utili al debugging.
Per provare le configurazione `config_explode.conf` o `config_timeout.conf` è possibile copiare il contenuto di una delle due dentro `config.conf` oppure rinominarne una in `config.conf`.

4. Struttura del Progetto

Il progetto segue una struttura classica. Il codice è suddiviso in due cartelle: `src/` per contenere i sorgenti e `include/` per contenere le intestazioni. `build/` contiene i file compilati e `bin/` i file eseguibili che vengono lanciati durante la simulazione.

5. Scelte Implementative

Tendenzialmente i processi seguono questo flusso di esecuzione:

```c
int main() {
    setup();
    while (1) {
        start();
        core();
    }
}
```

- `setup()`: funzione che viene chiamata all'inizio dell'esecuzione di un processo, si occupa di inizializzare delle IPC Facilities, dati di configurazione, variabili globali...
- `start()`: funzione invocata prima dell'inizio di ogni giornata, ogni processo in questa fase si organizza per iniziare la nuova giornata e quando è pronto lo comunica al manager tramite un semaforo, quanto tutti i processo sono pronti il manager dal il via alla simulazione.
- `core()`: dove avviene effettivamente l'esecuzione della giornata, ad esempio un worker dentro il core attende l'arrivo di nuove richieste di servizio da parte degli utenti.
  Alla fine della giornata i processi escono dal core e ritornano nello start a meno che la simulazione non sia terminata.

5.1 Manager

> **Info**
>
> file sorgente: `src/manager_main.c`

I compiti che esegue il manager in una giornata della simulazione:

1. Aggiunge utenti se è stato richiesto.
2. Avvia la simulazione.
3. Raccoglie tutti i dati.

5.2 User

> **Info**
>
> file sorgente: `src/user_main.c`

1. Nella fase di start richiede ad un processo clock di essere segnalato tramite una notifica ad una certa ora della giornata, ovvero l'ora in cui lo user intende andare in posta. La scelta dell'orario viene fatta casualmente dallo user, questa richieste viene collocata nel momento in cui c'è meno coda nel range di due ore dalla funzione `calendar.c/find_best_time()`.
2. Comunica che ha finito la fase di preparazione pre-giornata e attende il segnale del manager per iniziare.
3. Nel core lo user rimane in attesa di uno di questi segnali (5.7):
   - `DAY_ENDED`: segnale che la giornata è giunta al termine, esce dal core.
   - `CLOCK_NOTIFC`: segnale mandato dal clock che indica che è ora di andare in posta.
     - In seguito effettua un richiesta di biglietto e attende un segnale `TICKET_RESP`, in base all'esito del biglietto decine se mandare una richiesta di servizio al worker indicato dalla biglietteria. In caso affermativo lo user si mette in attesa della risposta del servizio da parte del lavoratore.
4. Manda le statistiche al manager.

5.3 Worker

> **Info**
>
> file sorgente: `src/worker_main.c`

1. Nella fase di pre-giornata, come per lo user, si organizza con il clock per essere avvisato quando dovrà andare in pausa (pausa che viene fatta secondo una probabilità).
2. Comunica che è pronto all'inizio della giornata e attende il segnale del manager.
3. Inizia la simulazione e tramite dei semafori tenta di trovare un posto, se lo trova attende le richieste di servizio (`SERVICE_REQ`) da parte degli user, altrimenti rimane in attesa che un operatore vada in pause (`SEAT_FREE`) (5.7).
4. Al termine della giornata manda le statistiche al manager.

5.4 Ticket Dispenser

> **Info**
>
> file sorgente: `src/ticket_dispenser_main.c`

1. Non ha nessuna attività particolare nella fase di pre-giornata.
2. Attende il segnale (5.7) di `TICKET_REQ`, controlla la disponibilità del servizio ed eroga il biglietto. Per controllare la disponibilità accede alla shared memory degli sportelli.

3. Non manda nessuna statistica al termine della giornata.

5.5 Clock

> **Info**
>
> file sorgente: `src/clock_main.c`

Processo clock che è stato creato per sincronizzare i vari orari dei processi, ovvero si occupa di comunicare quando gli utenti devono andare in posta, quando i lavoratori devono andare in pause e quando la giornata è finita.

1. Nella fase pre-giornata si salva le richieste di notifica degli user e degli worker.
2. Procede nella giornata, il passare del tempo è codificato come ciclo while con un attesa all'interno

```c
// pseudo codice
int min_count = 0;
int min_in_giorno = 8 * 60;
while (min_count < min_in_giorno) {
    sleep(N_NANO_SECS);
}
```

3. Manda le notifica all'orario richiesto.
4. Comunica che la giornata è finita.

5.6 Sportello (Seat)

Risorsa rappresentata da un shared memory di struct cosi definite:

```c
typedef struct
{
  int seats_taken; // 1 se lo sportello è occupato da un operatore
  int msg_notify_worker_id;
  int sem_notify_worker_count;
} SeatInfo;
```

Ogni sportello è rappresentato da una di queste struct. La funzione `seats.c/seats_try_take_seat()` è usata dai worker per provare ad ottenere uno sportello.
L'accesso alla shared memory è sincronizzato tramite un algoritmo lettori scrittori con priorità agli scrittori (algoritmo)

5.7 Comunicazione tra Processi

Tendenzialmente i processi nel core ricevono i messaggi (chiamati nella documentazione anche segnali) tramite la funzione `notifications.c/get_notifications()`. Questa funzione rimane in attesa che il semaforo `SEM_NOTIFY_<NOME_PROCESSO>_ID` venga incrementato (`val > 0`), una volta incrementato legge un messaggio dalla `MSG_NOTIFY_<NOME_PROCESSO>_ID`. La lettura dalla coda di messaggi avviene con priorità, ad esempio il segnale che indica al processo che la giornata (`DAY_ENDED`) è finita ha la priorità più alta.

6. Librerie esterne

È stata adottata una libreria log.c per migliorare il logging del progetto soprattutto in fase di debug.