Relazione del Progetto di Sistemi Operativi

Niccolò Alleyson 833974

Manuele Castiglioni 849966

Giovanni Chiappino 846676

Università degli Studi di Torino - Dipartimento di Informatica Piero della Francesca

Sommario

Struttura del progetto

- Makefile: automatizzazione della compilazione e dell'esecuzione per agevolare il programmatore
- Script Bash: dinamicità della struttura del progetto in relazione all'automatizzazione, separazione di codice sorgente ed eseguibili
- File sorgente: codice C, librerie, file oggetto

Struttura del progetto

Makefile

Utility del linguaggio di programmazione C per l'automatizzazione della compilazione e dell'esecuzione tramite utilizzo della system call ***make***.

Il comando cerca dentro la directory un file denominato Makefile dentro al quale vengono date delle direttive su come compilare i file, come salvarli e dove, quali file vanno rimossi a fine della compilazione.

Permette di risparmiare molto lavoro di scrittura su terminale ed automatizzare il controllo degli errori e dei warning.

```
CC=gcc
CFLAGS=-Wall
DEPS=shared.h
ODIR =../build
DIR = build

.PHONY: all

all: build student gestore clean

build:
   cd ..; ./mkbuild

$(ODIR)/%.o: %.c $(DEPS)
   $(CC) -std=c99 -c -o $@ $< $(CFLAGS)

student: $(ODIR)/student.o $(ODIR)/shared.o
   $(CC) -std=c99 -o $(ODIR)/$@ $^ $(CFLAGS)

gestore: $(ODIR)/gestore.o $(ODIR)/shared.o
   $(CC) -std=c99 -o $(ODIR)/$@ $^ $(CFLAGS)
```

```
execute:
	cd $(ODIR); ./gestore

clean:
	rm -f $(ODIR)/*.o
```

Il segmento di codice qui riportato istruisce la system call sul tipo di compilatore che deve usare (**gcc**), come deve essere compilato il file (**$(ODIR)** rappresenta la cartella destinazione dei file oggetto) ed eventuali flag (**$(CFLAGS)** dice al compilatore di utilizzare *Wall* per segnalare meglio warning ed errori).

Script Bash

È stato utilizzato uno script Unix per dividere i file oggetto dai file sorgente durante la compilazione.

```
#!/bin/bash

DIR="build"
if [ ! -d "$DIR" ]
	then mkdir -p "$DIR"
fi
```

Il codice sopra riportato controlla l'eventuale presenza di una cartella **build**, e se non presente la crea. Altrimenti, i file oggetto vengono semplicemente salvati al suo interno.

Struttura del progetto

- ***shared.h***

  Il file header contiene le librerie, i prototipi delle funzioni che vengono richiamate dai file sorgenti e le strutture utilizzate da questi ultimi. Contiene anche delle macro che sono risultate utili in fase di debugging e direttive del preprocessore per poter lavorare con valori predefiniti.

- ***shared.c***

  Contiene l'implementazione delle funzioni del file header, richiamate poi dai file principali del progetto.

- ***gestore.c***

  Processo principale del progetto, si occupa della creazione dei processi figli e della creazione della memoria condivisa, dei semafori e delle code di messaggi.

  La fase di spawning viene effettuata da una funzione specifica, dopodiché mediante una chiamata excve i processi figli si specializzano eseguendo il codice contenuto nel file student.c.

  Il gestore dopo aver creato i processi figli e le varie IPC, imposta un handler per gestire i segnali utilizzati durante la simulazione e, dopo aver atteso che tutti i processi abbiano scritto le loro informazioni nella memoria condivisa, setta un allarme e va in pausa.

  Quando scatta l'allarme manda un segnale ai processi figli, che aspettano di ricevere un messaggio dal gestore contenente il voto finale, stabilito in base all'analisi delle loro informazioni contenute nella memoria condivisa.

Una volta terminato, il gestore stampa delle statistiche sulla simulazione e termina.

```
switch(signum){
        case SIGALRM:
            for (int i = 0; i < POP_SIZE; i++) {
                kill(pst->stdata[i].student_pid, SIGUSR1);
            }
            break;

        case SIGINT:
            for(int i = 0; i < POP_SIZE; i++){
                kill(pst->stdata[i].student_pid, SIGTERM);
            }
            semctl(semid, 2, IPC_RMID);
            shmdt(pst);
            shmctl(memid, IPC_RMID, NULL);
            remove_queue(msgid);
            remove_queue(lastid);
            exit(EXIT_FAILURE);
    }
```

Il codice sopra riportato (contenuto nel file *shared.c*) gestisce i segnali che vengono inviati al gestore. In particolare il segnale **SIGINT** (Ctrl + C) fa sì che il gestore chiuda tutte le IPC e uccida mediante un segnale **SIGTERM** tutti i suoi figli. È stato utilizzato per gestire eventuali errori imprevisti durante lo sviluppo ed evitare che rimanessero strutture dati attive.

- ***student.c***

Codice relativo ai processi di tipo student. Lo studenti inizializza le proprie variabili e le scrive nella memoria condivisa dopo essersi connesso alle varie IPC create dal gestore, dopodiché imposta il proprio handler per il segnale SIGUSR1.

In seguito si mette in cerca di altri studenti per formare un gruppo secondo le specifiche del progetto.

Se il processo studente non trova nessuno da invitare o tutti rifiutano i suoi inviti chiude il gruppo da solo.

Alla fine (se è riuscito a chiudere il gruppo dopo aver trovato gli altri compagni) imposta una maschera per il segnale **SIGUSR1** e si mette in attesa di ricevere il voto dal gestore, altrimenti non imposta la maschera e si mette in attesa quando riceve il segnale dal gestore.

Dopo aver ricevuto il voto finale tramite messaggio, il processo stampa i suoi dati e termina.

```
     if (((pst->stdata[st_ind].leader && st_nof_el != nelem_team) ||
                pst->stdata[st_ind].team == 0) &&
pst->stdata[st_ind].nof_invites > 0)
            {
                if (pst->stdata[st_ind].leader) {
                    if ((pod = find_team_mate(st_ind)) != -1) {
                        wait_answer = invite(st_ind, pod, st_mark_ca);
                    }
                } else {
                    if ((pod = find_team_mate(st_ind)) != -1) {
```

```
                        wait_answer = invite(st_ind, pod, st_mark_ca);
                    } else {
                        if ((pod = find_random_mate(st_ind)) != -1) {
                            wait_answer = invite(st_ind, pod,
st_mark_ca);
                        }
                    }
                }
            } /* search mates */
```

Nel codice sopra riportato il processo corrente cerca un processo da invitare nel proprio gruppo. Se lo trova, gli invia un messaggio tramite la coda di messaggi per invitarlo.

Il codice varia a seconda che il processo corrente sia già leader di un gruppo o meno: il leader cerca solo altri elementi vantaggiosi per il proprio gruppo, mentre un processo non leader se non trova un elemento idoneo da invitare ne invita uno a caso.

```
    /* Find the perfect mate to build the team */
    pid_t find_team_mate(int ind)
    {
       pid_t pid = -1;
       for(int i = 0; (i < POP_SIZE && pid == -1); i++) {
           if(i != ind) {
```

```
                if(pst->stdata[i].team == 0
                    && pst->stdata[ind].fclass == pst->stdata[i].fclass
                    && pst->stdata[ind].nof_elems ==
pst->stdata[i].nof_elems) {

                    // Team mate with better or equal mark than mine
                    if (pst->stdata[i].mark_ca >=
pst->stdata[ind].mark_ca) {
                        return pst->stdata[i].student_pid;
                    }

                    // Team mate with better mark than mine diminished by
3
                    if (pst->stdata[i].mark_ca >=
(pst->stdata[ind].mark_ca - 3)) {
                        return pst->stdata[i].student_pid;
                    }
                }
            }
        }
        return pid;
    }
```

Nel codice sopra riportato il processo chiamante legge le informazioni scritte nella memoria condivisa alla ricerca di un processo che abbia voto migliore o uguale al proprio per invitarlo a formare il gruppo. In alternativa, controlla se il voto dell'altro processo rientra nel range di tolleranza (voto del processo chiamante meno 3) e lo invita.

Risultati

Quasi tutti i processi riescono a formare un gruppo e ricevono il proprio voto dal gestore.

Alcuni processi rimangono soli a causa del loro voto troppo basso e quindi chiudono il gruppo da soli. Una piccola percentuale di processi non riescono a formare un gruppo con il numero di elementi richiesti e quindi formano il gruppo casualmente, ricevendo un voto più basso ed insufficiente.

La simulazione termina nel tempo previsto (1 secondo), non rimangono IPC in memoria e tutti i processi terminano correttamente.