UNIVERSITÀ DEGLI STUDI DI GENOVA

Università degli Studi di Genova

Programmazione Concorrente Algoritmi Distribuiti

Lorenzo Vaccarecci

Anno Accademico 2024/2025

Indice

Capitolo 1

Introduzione

1.1 Perchè usare/sviluppare sistemi concorrenti/distribuiti?

Perchè in questo modo siamo più efficienti, distribuiamo dei dati, condividiamo delle risorse, usiamo reti di computer e per comunicare.

1.2 Perchè è difficile concettualmente?

Perchè se ho n processi e m istruzioni per processo, ho

$$\frac{(n \cdot m)!}{(m!)^n}$$

possibili "percorsi" che un'esecuzione concorrente/distribuita può avere.

1.3 Programma concorrente

E' un insieme di processi che si eseguono in parallelo (programmi che si eseguono su processori diversi ma anche programmi multi-task che si eseguono sullo stesso processore dando l'illusione del parallelismo) e che comunicano (scambio di dati tramite messaggi o memoria condivisa che permette la sincronizzazione).

1.4 Esecuzione concorrente

L'esecuzione di un programma concorrente è composto da un insieme di processi ciascuno dei quali dispone di un insieme di istruzioni atomiche. Ogni processo ha un puntatore verso la prossima istruzione da eseguire e l'esecuzione del programma concorrente è un intreccio arbitrario delle istruzioni dei vari processi (interleaving), la prossima istruzione è dunque scelta fra quelle puntate dai vari processori: tanti possibili scenari (1.2).

1.5 Comunicazione via memoria condivisa

I processi hanno un accesso comune alla memoria che può essere tramite variabile condivisa o con strutture più complesse. Se si usa una variabile condivisa deve essere possibile scrivere e leggere e bisogna fare attenzione alla gestione dell'accesso (se transazione atomica o non atomica). Ci sono vari casi possibili:

- **Multiple Reader/Multiple Writer**
- **Multiple Reader/Single Writer**

Capitolo 2

Processi e Thread

2.1 Processi

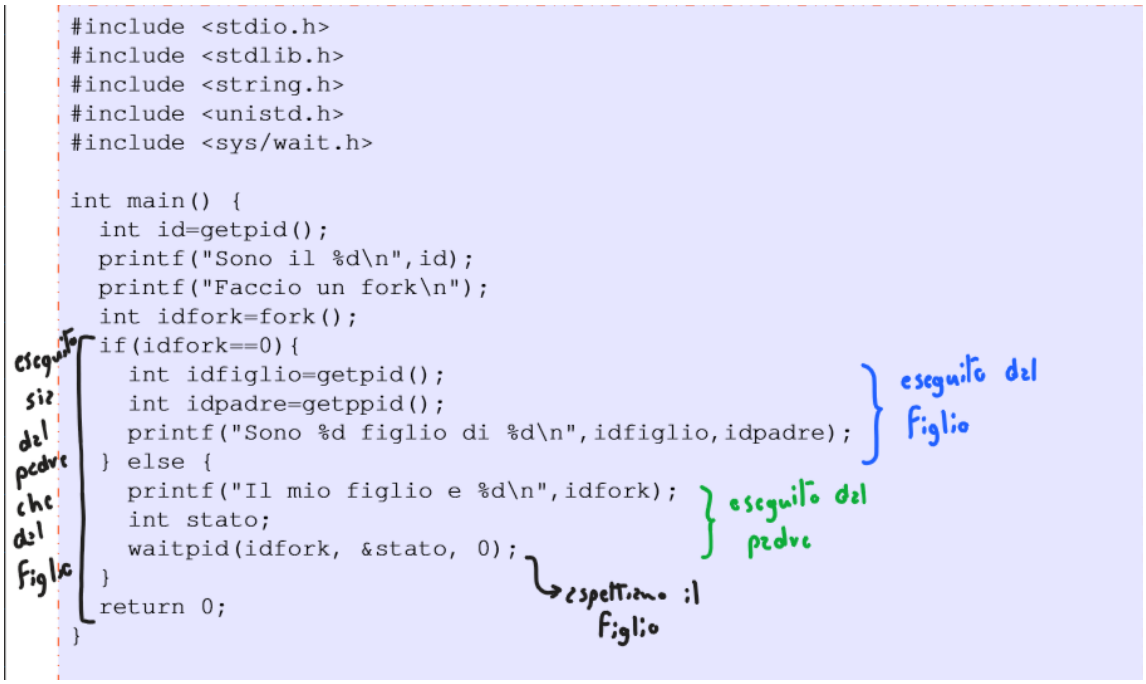
Un processo è un programma in corso di esecuzione. Ogni processo necessita di uno spazio di indirizzamento dedicato (stack) e di input e output.
Più processi possono essere eseguiti sulla stessa macchina in un modo quasi simultaneo, un processo non ha accesso allo spazio di indirizzamento degli altri e permette di avere un'illusione di parallelismo.

2.1.1 Unix

Creazione di un processo

```
pid_t fork(void);
```

Ritorna 0 al figlio e il PID del figlio al padre.
`fork()` crea un processo figlio che è una copia esatta del padre, con lo stesso codice, dati e spazio di indirizzamento. Le variabili non sono condivise.



Esecuzione di un comando

```
int execvp(const char *file, char *const argv[]);
```

Dove `file` è il nomde del comando che vogliamo eseguire e l'array `argv` contiene:

- Il nome del comando nel primo campo
- Gli argomenti del comando
- `NULL` per indicare la fine dell'array in posizione finale obbligatoria

2.1.2 Pipe

Servono per la comunicazione fra processi. Le pipes sono dei canali unidirezionali considerati come dei descrittori di file.


```
int pipe(int fd[2]);
```

Dove `fd[0]` è il descrittore di lettura e `fd[1]` è il descrittore di scrittura. Dopo il forkk, entrambi i processi possono scrivere e leggere ma perchè funzioni bisogna chiudere il descrittore che non si usa.

2.2 Thread

Un thread è un filo d'esecuzione dentro un programma, è eseguito da un processo e possono esserci più thread in un processo (processo multi-thread). Ogni thread è diverso e ha come attributi:

- Un puntatore di esecuzione o `PC` (Program Counter)
- Uno stack

2.2.1 POSIX

```
#include <pthread.h>
```

Bisogna anche dire al momento della compilazione che si desidera usare questa libreria con il flag `-pthread` o `-lpthread`.

Creazione di un thread

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)
                                    (void *), void *arg);
```

Per condividere le variabili si usa la keyword `volatile`.

Locks

L'accesso ai dati condivisi deve essere protetto:

- Un thread che desidera accedere a un dato condiviso richiede il lock
- Se il lock è libero, prosegue altrimento rimaane bloccato finchè il lock non è libero
- Quando ha finito, rilascia il lock

Questi lock sono condivisi tra i threads.
E' buona pratica far liberare il lock dal thread che lo ha preso.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
  int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Queste due funzioni ritornano 0 se tutto è andato a buon fine.

Capitolo 3

Interleaving e Mutua esclusione

3.1 Istruzioni atomiche

Un'istruzione atomica è eseguita completamente senza essere interrotta, in questo corso supponiamo che ogni riga nei programmi sarà atomica.

```
1            x = x + 1;
2            if(x == 2) { // Questo if potrebbe essere interrotto e x potrebbe
     cambiare
3                x = 3;
4            }
5
```

3.2 Ipotesi di atomicità

Le ipotesi possono cambiare a seconda del compilatore, il processore, ecc.

Evitare di scrivere istruzioni che manipolano più di una volta la stessa variabile condivisa tra più processi o thread senza un'adeguata sincronizzazione.

- **Registri atomici**: operazione di lettura e di scrittura atomiche
- **Test-and-set**: si può testaare e modificare una variabile senza essere interrotto
- **Swap**: si può cambiare il valore di un registro locale e di uno condiviso in modo atomico.
- ...

3.3 Interleaving

Def 3.3.1. Uno stato di un programma concorrente P è una tupla con:

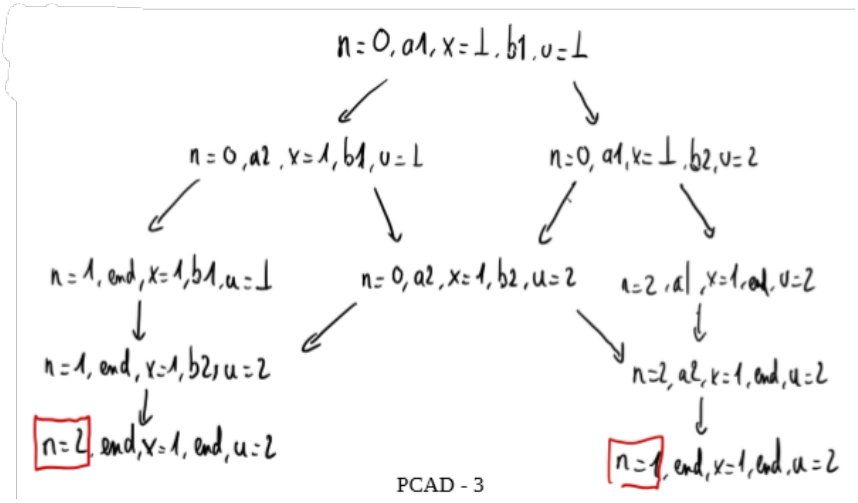
- un puntatore d'istruzioni per ciascun processo
- un valore per ciascuna variabile (locale o condivisa)

Def 3.3.2. Per due stati s_1 e s_2 si scrive $s_1 \to s_2$ quando si può passare da s_1 a s_2 usando una delle istruzioni puntata in s_1.

Def 3.3.3. Il diagramma degli stati di P è il sistema di transizione $TS_P = (S, \to, s_0)$ dove:

- S è l'insieme degli stati di P
- s_0 è lo stato iniziale di P
- $\to$ è la relazione di transizione inclusa in $S \times S$

Un cammmino in TS_P è uno scenario possibile per P.



I cammini in TS_P corrispondono a tutti gli intrecci possibili per P. Facciamo quindi l'ipotesi che tutti questi intrecci siano possibili.

3.3.1 Semantica ragionevole

- **Sistemi con un processore**: c'è una successione d'istruzioni
- **Sistemi con più processori**: ogni processo è legato a un processore e l'interleaving non corrisponde esattamente alla realtà (azioni parallele), ma è corretta se non ci sono conflitti sulle risorse.
- **Sistemi distribuiti**: diversi computer, nessuna variabile condivisa, comunicazione tramite messaggi. Molto diversa rispetto ai primi due casi, ma la semantica rimane interessante se uno ci aggiunge l'invio e la ricezione di messaggi.

3.4 Mutua esclusione