



UNIVERSITÀ DEGLI STUDI
DI GENOVA

UNIVERSITÀ DEGLI STUDI DI GENOVA

Programmazione Concorrente Algoritmi Distribuiti

Lorenzo Vaccarecci

Indice

1	Processi e Thread	2
1.1	Processi	2
1.1.1	Unix	2
1.1.2	Pipe	3
1.2	Thread	3
1.2.1	POSIX	3
2	Interleaving e Mutua esclusione	4
2.1	Istruzioni atomiche	4
2.2	Ipotesi di atomicità	4
2.3	Interleaving	4
2.3.1	Semantica ragionevole	5
2.4	Mutua esclusione	5

Capitolo 1

Processi e Thread

1.1 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.

1.1.1 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.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int id=getpid();
    printf("Sono il %d\n",id);
    printf("Faccio un fork\n");
    int idfork=fork();
    if(idfork==0){
        int idfiglio=getpid();
        int idpadre=getppid();
        printf("Sono %d figlio di %d\n",idfiglio,idpadre);
    } else {
        printf("Il mio figlio e %d\n",idfork);
        int stato;
        waitpid(idfork, &stato, 0);
    }
    return 0;
}
```

eseguito dal padre che crea il figlio

eseguito dal figlio

eseguito dal padre

aspettiamo il figlio

Esecuzione di un comando

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

Dove `file` è il nome 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

1.1.2 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 `fork`, entrambi i processi possono scrivere e leggere ma perchè funzioni bisogna chiudere il descrittore che non si usa.

1.2 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

1.2.1 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 altrimenti rimane 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 2

Interleaving e Mutua esclusione

2.1 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
```

2.2 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 testare e modificare una variabile senza essere interrotto
- **Swap:** si può cambiare il valore di un registro locale e di uno condiviso in modo atomico.
- ...

2.3 Interleaving

Def 2.3.1. Uno stato di un programma concorrente P è una tupla con:

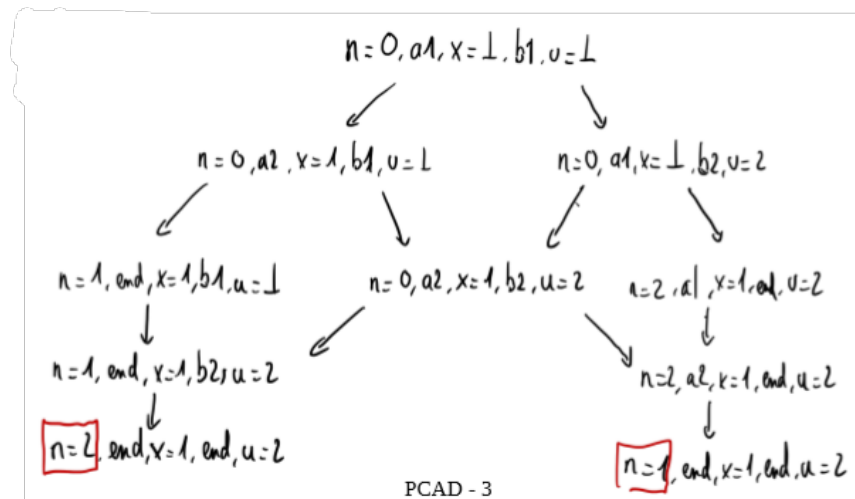
- un puntatore d'istruzioni per ciascun processo
- un valore per ciascuna variabile (locale o condivisa)

Def 2.3.2. Per due stati s_1 e s_2 si scrive $s_1 \rightarrow s_2$ quando si può passare da s_1 a s_2 usando una delle istruzioni puntata in s_1 .

Def 2.3.3. Il diagramma degli stati di P è il sistema di transizione $TS_P = (S, \rightarrow, s_0)$ dove:

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

Un cammino 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.

2.3.1 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.

2.4 Mutua esclusione