Processi e Thread

Traduzione e riorganizzazione delle slide originali

Corso: Sistemi di Calcolo 2
Docente: Riccardo Lazzeretti
Crediti speciali: Daniele Cono D'Elia, Leonardo Aniello, Roberto Baldoni

Fonti

Le presenti note sono una traduzione e riorganizzazione delle slide *"Processes and Threads"*, in larga parte adattate da *Operating Systems: Internals and Design Principles* (6a ed.) di William Stallings (cap. 4) e dal tutorial "POSIX threads Programming" di Blaise Barney. Le citazioni e gli esempi di codice seguono il materiale originale.

Nota: le figure presenti nelle slide originali sono qui descritte testualmente quando necessario.

Indice

1 Roadmap

- **Processi**: `fork()`, `wait()`.
- **Thread**: possesso delle risorse ed esecuzione.
- **Caso di studio**: *Pthreads*.
- **Elaborazione simmetrica multiprocessore** (SMP).

2 Che cos'è un processo?

Un **processo** è un'entità dinamica caricata in memoria principale (RAM) e generata da un programma. Più precisamente, è una sequenza di attività (task) controllate da un programma (*scheduler*) che ha luogo su un processore sotto la gestione/supervisione del sistema operativo.

2.1 Ruolo dei processi

Molti requisiti che un sistema operativo deve soddisfare possono essere espressi in termini di processi:

- Esecuzione interlacciata.
- Allocazione delle risorse e politiche.
- Creazione di processi da parte dell'utente e comunicazione inter-processo.

2.2 Elementi del processo

Un processo comprende:

- Codice del programma (eventualmente condiviso).
- Un insieme di dati.
- Attributi che descrivono lo stato del processo durante l'esecuzione.

Durante l'esecuzione, un processo ha tipicamente i seguenti elementi:

- Identificatore.
- Stato.
- Priorità.
- Program counter.
- Puntatori di memoria.
- Dati di contesto (registri).
- Informazioni di stato I/O.
- Informazioni di contabilità.

2.3 Process Control Block (PCB)

Il **PCB** contiene gli elementi del processo, è creato e gestito dal sistema operativo e consente di supportare la multiprogrammazione.

3 Chiamate di sistema Unix: creare e gestire processi

3.1 Come creare nuovi processi

La chiamata `fork()` crea un **processo figlio** come duplicato del **processo padre**. Padre e figlio eseguono *concorrentemente*; entrambi possono a loro volta invocare `fork()`, generando un *albero di processi* la cui radice è un processo speciale creato dal sistema all'avvio.

3.2 Bootstrapping (avvio del sistema)

All'accensione o al reset della macchina, un *bootstrap program*:

- Inizializza registri CPU, controller di dispositivo, memoria.
- Carica il sistema operativo in memoria.
- Avvia l'esecuzione dell'OS.

L'OS avvia il primo processo (es. `init`) e quindi attende eventi (interruzioni hardware o *trap* software).

3.3 `fork()`: comportamento e valori di ritorno

- Restituisce `-1` in caso di errore.
- Restituisce `0` nel processo figlio.
- Restituisce il *pid* del figlio nel processo padre.

Il figlio eredita una copia identica della memoria del padre, i registri della CPU e tutti i file aperti. L'esecuzione riprende dalla prima istruzione successiva alla `fork()` in *entrambi* i processi.

3.4 Esempio di flusso con `fork()`

Listing 1: Schema tipico d'uso di `fork()` con `wait()`

```
pid_t ret = fork();
switch (ret) {
  case -1:
    perror("fork");
    exit(1);
  case 0:  // Codice del figlio
    /* ... */
    exit(0);
  default: // Codice del padre
    /* ... */
    wait(0);
}
```

3.5 Esecuzione di un programma: famiglia `exec*()`

Se il figlio deve eseguire un *nuovo* programma, si usa `exec*()`: l'OS rimpiazza l'immagine del processo corrente (testo, dati, stack) con quella del nuovo programma. Il nuovo programma deve avere una `main()`. La `exec*()` **non ritorna**. In questo corso si assume spesso che il figlio esegua codice definito nel programma del padre.

3.6 Terminazione ordinata: `exit()` e `_exit()`

`exit(status)` Salva il valore *status*, esegue le funzioni registrate con `atexit()` / `on_exit()`, svuota i buffer (`fflush()`), chiude i file aperti (non quelli condivisi con altri processi) e alla fine invoca `_exit(status)`.

`_exit(status)` Dealloca la memoria; se il processo ha figli attivi, questi vengono adottati da `init`. Se il padre è vivo, il valore di uscita viene mantenuto fino alla chiamata `wait()`, lasciando il processo in stato *zombie*; se il padre non è vivo, il figlio termina completamente.

3.7 Attendere il figlio: `wait()` e `waitpid()`

- `wait(int *status)` blocca finché *un* figlio termina; ritorna il pid del figlio terminato o `-1` se non esistono figli.
- `waitpid(pid_t pid, int *status, int options)` consente di attendere un figlio specifico.

3.8 Processi con Python (OS API)

In Python, la creazione dei processi è delegata all'OS tramite il modulo `os`:

Listing 2: Esempio basilare con fork/wait in Python

```
import os

ret = os.fork()
if ret == 0:
    # Codice del figlio
    # ...
    os._exit(0)
else:
    # Codice del padre
```

```
    # ...
    os.wait()
```

4 Processi e Thread

Un processo ha due caratteristiche fondamentali:

Esecuzione/scheduling segue un percorso di esecuzione interlacciabile con altri processi.

Possesso delle risorse include uno spazio di indirizzamento virtuale che contiene l'immagine del processo.

Il sistema operativo tratta queste due caratteristiche in modo in larga parte indipendente: l'*unità di dispatch* è il **thread** (o *lightweight process*), mentre l'*unità di possesso delle risorse* è il **processo** (o *task*).

4.1 Multithreading

La capacità del sistema operativo di supportare percorsi multipli e concorrenti di esecuzione all'interno dello stesso processo. Esempi:

- Ambienti Java come singolo processo con più thread.
- Sistemi moderni (Windows, Solaris, UNIX) con più processi e più thread.

4.2 Struttura di un processo in un OS multithread

- Spazio di indirizzi virtuale con l'immagine del processo.
- Accessi protetti a CPU, altri processi, file, risorse I/O.

4.3 Caratteristiche di un thread

Ogni thread dispone di:

- Stato di esecuzione (running, ready, blocked).
- Contesto salvato quando non è in esecuzione.
- Stack di esecuzione.
- Dati statici per variabili locali per-thread.
- Accesso alla memoria e alle risorse del processo (condivise tra i thread del processo).

Un thread può essere visto come un *program counter indipendente* all'interno di un processo.

4.4 Benefici dei thread

- Creazione/terminazione più rapida rispetto ai processi.
- Context switch tra thread più veloce di quello tra processi.
- Comunicazione tra thread senza invocare il kernel (condivisione di memoria).

4.5 Esempi d'uso in sistemi *single-user*

- Lavoro in foreground e background.
- Elaborazione asincrona.
- Maggiore velocità (sovrapposizione con attese I/O).
- Struttura modulare del programma.

4.6 Azioni a livello di processo che impattano i thread

- Sospendere un processo sospende tutti i suoi thread (stesso address space).
- Terminare un processo termina tutti i thread.

4.7 Stati di esecuzione e transizioni

Stati: *running*, *ready*, *blocked*. Transizioni tipiche: *spawn* (creazione di un nuovo thread), *block* (p.es. su I/O), *unblock*, *finish* (rilascio di contesto e stack). Un tema chiave: il blocco di un thread può comportare il blocco dell'intero processo? (dipende dall'implementazione dei thread).

4.8 Esempio: Remote Procedure Call (RPC)

- Con un singolo thread: le due RPC verso server diversi sono serializzate.
- Con un thread per server: le RPC possono procedere in parallelo, aggregando i risultati più rapidamente.

4.9 Multithreading su un uniprocessore

Anche su una sola CPU, il multithreading può migliorare la *responsiveness* sovrapponendo attese I/O e computazione.

4.10 Categorie di implementazione dei thread

ULT (User-Level Threads) Gestione dei thread a livello utente; il kernel non è a conoscenza dei thread.

KLT (Kernel-Level Threads) Thread supportati dal kernel (leggermente chiamati *light-weight processes*); scheduling a livello di thread. Esempio: Windows.

Approcci combinati Creazione e molte primitive in user-space; più ULT mappati su KLT (es. Solaris, con $k = u$).

4.11 Pro e contro

Vantaggi ULT

- Scheduling specifico dell'applicazione.
- Switch tra thread senza passaggio in *kernel mode*.
- Portabili su qualunque OS via librerie user-level.

Svantaggi ULT

- Una system call bloccante in un thread blocca tutti i thread del processo.
- Non sfruttano appieno architetture multiprocessore/multicore.

Vantaggi KLT

- Il kernel può schedulare thread dello stesso processo su CPU diverse.
- Se un thread è bloccato, il kernel può eseguirne un altro dello stesso processo.
- Le routine del kernel possono essere multithread.

Svantaggi KLT

- Lo switch tra thread richiede un passaggio in kernel mode.

5 PThreads (POSIX Threads)

5.1 Definizione

Su sistemi UNIX, implementazioni di thread che aderiscono allo standard IEEE POSIX 1003.1c sono dette **Pthreads**. Le primitive sono definite nel file header `pthread.h`.

5.2 Perché usare Pthreads

- Migliorare le prestazioni del programma.
- Minor overhead rispetto alla creazione di processi.
- Minor consumo di risorse per l'esecuzione.

5.3 Confronto empirico: `fork()` vs `pthread_create()`

Tempi per eseguire 50 000 creazioni (valori indicativi, "real/user/sys"):

Piattaforma	`fork()`			`pthread_create()`		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 core/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 core/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 core/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 core/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpu/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpu/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpu/node)	104.5	48.6	47.2	2.1	1.0	1.5
Intel 2.4 GHz Xeon (2 cpu/node)	54.9	1.5	20.8	1.6	0.7	0.9
Intel 1.4 GHz Itanium2 (4 cpu/node)	54.5	1.1	22.2	2.0	1.2	0.6

5.4 Progettare programmi a thread

Per sfruttare Pthreads, organizzare il programma in compiti indipendenti e concorrenti (ad es. `routine1` e `routine2`) eseguibili in parallelo.

5.5 Modelli di programmazione a thread

Manager/Worker Un thread manager assegna lavoro ai thread *worker*, gestendo input e suddivisione dei compiti.

Pipeline Il compito è suddiviso in sotto-operazioni in serie ma eseguite concorrentemente da thread diversi (una catena di montaggio).

5.6 Memoria condivisa e *thread safety*

Tutti i thread condividono la stessa memoria globale e possiedono dati privati. Il programmatore deve sincronizzare l'accesso ai dati condivisi per evitare corruzione e *race conditions*. Un codice è *thread-safe* se può essere eseguito da più thread senza interazioni indesiderate (p.es. librerie che modificano strutture globali devono adottare meccanismi di sincronizzazione).

5.7 Creazione e terminazione di thread (API)

Creare `pthread_create()` crea un nuovo thread ed avvia la funzione specificata. Il numero massimo di thread è dipendente dall'implementazione. I thread sono *peer* e possono creare altri thread.

Terminare Modi comuni:

- Ritorno dalla funzione di start.
- `pthread_exit()` a lavoro concluso.
- `pthread_cancel()` invocato da un altro thread.
- `exit()` (termina l'intero processo!).
- Se il `main` termina senza `pthread_exit()`, i thread possono continuare solo se `pthread_detach()` o altri accorgimenti sono usati.

Nota: `pthread_exit()` non libera automaticamente tutte le risorse (p.es. file aperti): occorre *cleanup*.

5.8 Esempio Pthreads in C

Listing 3: Pthreads: creare 5 thread e salutare

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define NUM_THREADS 5
6
7  void* printHello(void *arg) {
8    int threadID = *(int*)arg;
9    printf("Ciao! Sono il thread #%d!\n", threadID);
10   pthread_exit(NULL);
11 }
12
13 int main(void) {
14   pthread_t threads[NUM_THREADS];
15   int ret, t;
16   for (t = 0; t < NUM_THREADS; t++) {
17     printf("Nel main: creo il thread %d\n", t);
```

```
    int *arg = malloc(sizeof(int));
    *arg = t;
    ret = pthread_create(&threads[t], NULL, printHello, (void*)arg);
    if (ret != 0) { printf("ERRORE: codice %d\n", ret); exit(-1); }
  }
  pthread_exit(NULL);
}
```

5.9 Possibile output (non deterministico)

```
Nel main: creo il thread 0
Nel main: creo il thread 1
Ciao! Sono il thread #0!
Nel main: creo il thread 2
Ciao! Sono il thread #2!
Ciao! Sono il thread #1!
Nel main: creo il thread 3
Nel main: creo il thread 4
Ciao! Sono il thread #3!
Ciao! Sono il thread #4!
```

6 Thread in Java

6.1 Modi d'uso

- Estendere `Thread` e ridefinire `run()`.
- Implementare `Runnable` e passare l'istanza a `new Thread(...)`.

Il metodo `start()` avvia il thread (che invoca `run()`). `join()` consente di attendere il termine di un altro thread.

6.2 Esempio 1: sottoclasse di Thread

```
public class SimpleThread extends Thread {
  public SimpleThread(String str) { super(str); }
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println(i + " " + getName());
    }
  }
}

public class TwoThreadsTest {
  public static void main (String[] args) throws InterruptedException {
    SimpleThread t1 = new SimpleThread("Jamaica");
    SimpleThread t2 = new SimpleThread("Fiji");
    t1.start();
    t2.start();
    t1.join();
    System.out.println("Thread Jamaica terminato. Non mi interessa il thread Fiji.");
  }
}
```

6.3 Esempio 2: implementare Runnable

```
public class SimpleThread implements Runnable {
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println(i + " Ciao");
    }
  }
}

public class TwoThreadsTest {
  public static void main (String[] args) {
    SimpleThread s1 = new SimpleThread();
    SimpleThread s2 = new SimpleThread();
    Thread t1 = new Thread(s1);
    Thread t2 = new Thread(s2);
    t1.start();
    t2.start();
  }
}
```

6.4 Esempio 3: condividere lo stesso Runnable

```
public class SimpleThread implements Runnable {
  int i = 0;
  public void run() {
    while (i < 10) {
      System.out.println(i + " Ciao");
      i++;
    }
  }
}

public class TwoThreadsTest {
  public static void main (String[] args) {
    SimpleThread s = new SimpleThread();
    Thread t1 = new Thread(s);
    Thread t2 = new Thread(s);
    t1.start();
    t2.start();
  }
}
```

In questo caso i due thread condividono la stessa memoria (e variabili).

7 Thread in Python

La libreria standard offre il modulo `threading` con le primitive necessarie.

```
import logging
import threading
import time

def thread_function(name):
    logging.info("Thread %s: start", name)
    time.sleep(2)
    logging.info("Thread %s: finish", name)
```

```
9
10 if __name__ == "__main__":
11     format = "%(asctime)s: %(message)s"
12     logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")
13
14     logging.info("Main: prima di creare il thread")
15     x = threading.Thread(target=thread_function, args=(1,))
16     logging.info("Main: prima di avviare il thread")
17     x.start()
18     logging.info("Main: attendo che il thread finisca")
19     x.join()
20     logging.info("Main: fatto")
```

8 Elaborazione parallela e SMP

8.1 Visione tradizionale

Tradizionalmente il calcolatore è visto come una macchina sequenziale: un processore esegue istruzioni una alla volta; ogni istruzione è a sua volta una sequenza di operazioni. Approcci alla parallelizzazione includono **SMP** e **cluster**.

8.2 Tassonomia di Flynn

SISD Una sola unità di controllo (istruzione) e un solo flusso di dati.

SIMD La stessa istruzione eseguita su insiemi di dati diversi da processori diversi.

MISD Un flusso di dati trasmesso a più processori, ciascuno con *diversa* sequenza di istruzioni.

MIMD Più processori eseguono simultaneamente diverse sequenze di istruzioni su dati diversi.

8.3 Architetture e organizzazione SMP

Una tipica organizzazione **SMP** prevede più CPU simmetriche che condividono memoria principale e I/O, con un unico OS che le gestisce.

8.4 Considerazioni progettuali per OS multiprocessore

- Concorrenza di processi e thread.
- Scheduling.
- Sincronizzazione.
- Gestione della memoria.
- Affidabilità e tolleranza ai guasti.