Laboratorio di Reti e Sistemi Distribuiti

11: Introduzione ai Thread Posix

Roberto Marino, PhD[1]
roberto.marino@unime.it

[1]Dipartimento di Matematica, Informatica, Fisica e Scienze della Terra
Future Computing Research Laboratory
Università di Messina

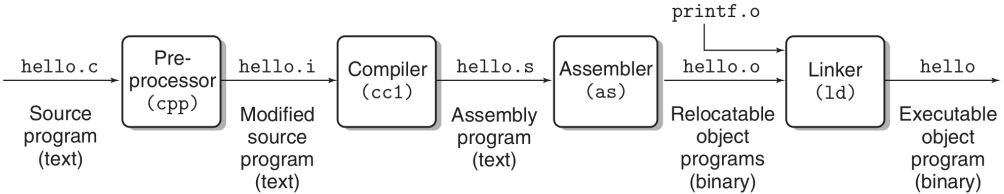
Last Update: 26th March 2025

Dal codice sorgente ai thread/processi

Perchè scendere nel dettaglio?

1. **Ottimizzare le performance dei programmi**: ad esempio, uno switch è sempre più efficiente di una serie di if/then? Che differenza c'è tra for e while?
2. **Comprendere gli errori ed i warning del linker**: qual'è la differenza tra variabili statiche e globali? che differenza c'è tra librerie statiche e dinamiche?
3. **Evitare problemi di sicurezza**: risolvere i buffer overflow, che comportano seri problemi di sicurezza su internet

Dal sorgente al codice compilato



Dal sorgente al file oggetto

1. **Preprocessing (cpp)**: Traduce le direttive al preprocessore
2. **Compilazione (cc)**: Produce un file di testo che contiene codice assembly
3. **Assemblamento (as)**: Produce codice macchina rilocabile (binario non eseguibile)
4. **Linker (ld)**: Link di eventuali librerie o altri file oggetto e produzione di un binario eseguibile

hello.c

code/intro/hello.c

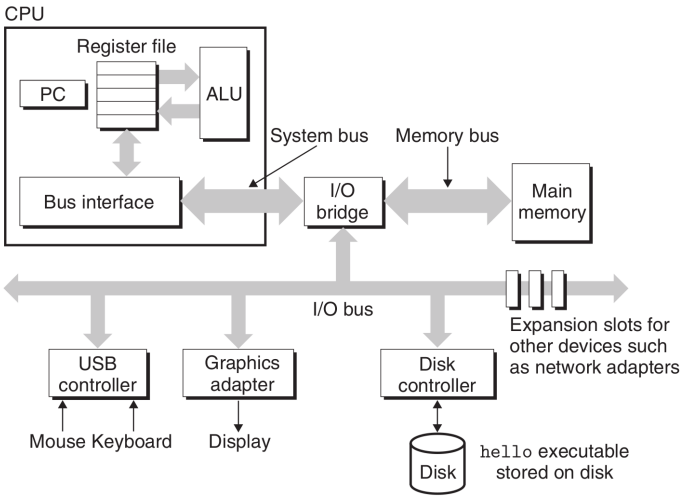
```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("hello, world\n");
6      return 0;
7  }
```

code/intro/hello.c

hello.s

```
1   main:
2       subq    $8, %rsp
3       movl    $.LC0, %edi
4       call    puts
5       movl    $0, %eax
6       addq    $8, %rsp
7       ret
```

Architettura di un elaboratore (binario)

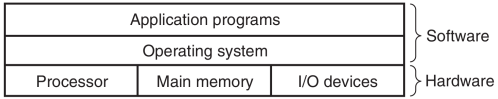


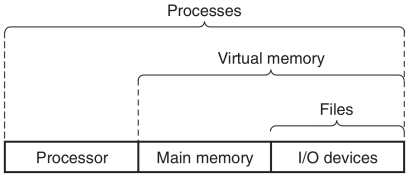
Elementi dell'elaboratore: modello semplificato

1. **Memoria principale**: mantiene momentaneamente in memoria il codice ed i dati. Array lineare indirizzabile.
2. **Registri di uso generale e program-counter (PC)**: ad ogni istante punta ad una istruzione in memoria centrale
3. **ISA (Instruction Set Architecture)**: Load, Store, Operate, Jump
4. **Microarchitettura**: descrive come l'ISA è implementata a livello di circuiti combinatori

Operating system: Abstraction is all you need!

1. **Processo**: astrazione principale, programma in esecuzione
2. **Memoria Virtuale**: ogni processo ha l'illusione di avere l'uso esclusivo della memoria
3. **File**: dispositivi di I/O (dischi, rete, video). Everything is a file!





Processi: definizioni

Processo

Un **processo** è l'astrazione del sistema operativo per un programma in esecuzione. Più processi possono essere eseguiti contemporaneamente sullo stesso sistema e ogni processo sembra avere l'uso esclusivo dell'hardware.

Concorrenza

Per **concorrenza** si intende che le istruzioni di un processo sono interlacciate con quelle di un altro processo. Nella maggior parte dei sistemi ci sono più processi da eseguire che CPU.

Processi: definizioni

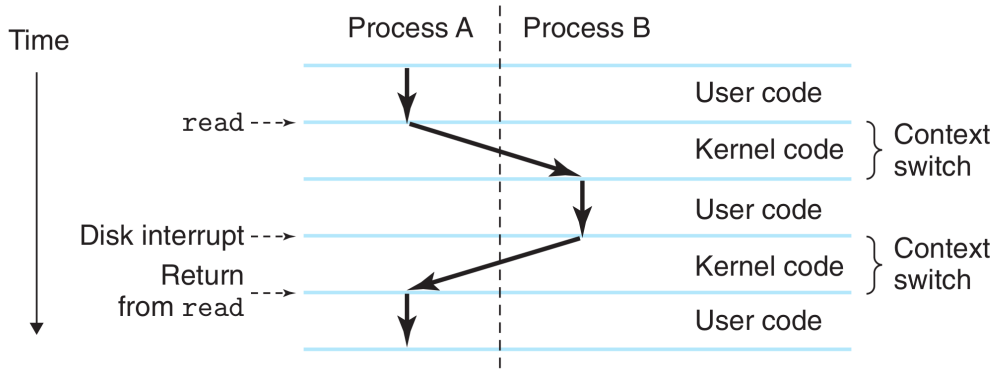
Multiprogrammazione

I sistemi tradizionali potevano eseguire solo un programma alla volta, mentre i più recenti processori multi-core possono eseguire più programmi contemporaneamente. In entrambi i casi, singola CPU può apparire come se stesse eseguendo più processi simultaneamente, facendo passare il processore da uno all'altro. Il sistema operativo esegue questo interlacciamento (interleaving) con un meccanismo noto come commutazione di contesto (context switching). Per semplificare il resto della trattazione consideriamo solo un sistema uniprocessore contenente una singola CPU.

Context Switch per terminazione

Esempio classico: ci sono due processi contemporanei, **il processo shell e il processo hello**. Inizialmente, il processo shell è in esecuzione da solo, in attesa di input sulla riga di comando. Quando gli chiediamo di eseguire il programma hello, la shell esegue la nostra richiesta invocando una funzione speciale nota come **chiamata di sistema**, che passa il controllo al sistema operativo. Il sistema operativo salva il contesto della shell, crea un nuovo processo hello e il suo nuovo contesto, quindi passa il controllo al nuovo processo. Dopo la terminazione di hello, il sistema operativo **ripristina il contesto del processo di shell** e gli passa di nuovo il controllo, dove attende il successivo comando.

Context Switch su interrupt

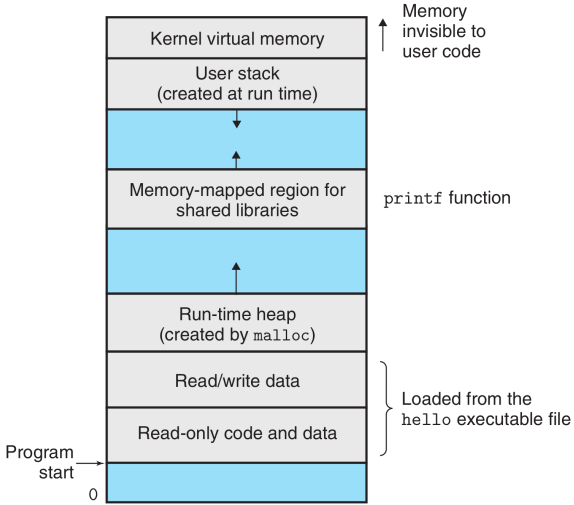


Threads

Sebbene normalmente si pensi a un processo come a un unico flusso di controllo, nei sistemi moderni un processo può in realtà essere costituito da più unità di esecuzione, chiamate **thread**, ciascuna delle quali viene eseguita nel contesto del processo e condivide lo stesso codice e gli stessi dati globali.

1. E' più facile condividere informazioni tra thread che tra processi, la loro creazione è più efficienti
2. Il Multi-threading è un modo per rendere i programmi più veloci quando sono presenti più processori/core.

Memoria virtuale di processo



Memoria virtuale di processo

1. Attenzione! Ogni processo suppone di avere il pieno controllo della memoria!
2. **Program code and data:** Il codice comincia allo stesso indirizzo per tutti i processi ed è seguito dalle variabili globali del programma. La dimensione è fissata al momento della creazione del processo.
3. **Heap:** Si espande/contrae a runtime a seguito di chiamate `malloc()` e `free()`
4. **Shared libraries:** (es. printf() function)
5. **Stack:** usato per implementare le chiamate di funzione. Cresce dinamicamente durante l'esecuzione ogni volta che viene chiamata una funzione e si contrae ad ogni return.
6. **Kernel virtual memory:** riservato per il kernel. Il processo non può accedervi.

La memoria virtuale funziona grazie ai meccanismi della rilocazione e della paginazione e sfrutta un componente della CPU chiamato **MMU** (Memory Management Unit).

Concorrenza

Definizione generale di concorrenza

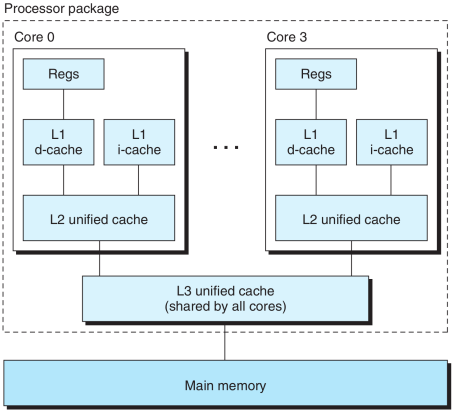
Il termine **concorrenza** (piuttosto ambiguo!) si riferisce al concetto generale di sistema con attività multiple e simultanee che concorrono a sfruttare risorse condivise, mentre il termine parallelismo si riferisce all'uso della concorrenza per rendere un sistema più veloce su più processori. Il parallelismo può essere sfruttato a più livelli di astrazione in un sistema informatico. In questa sede ne evidenziamo tre, partendo dal livello più alto a quello più basso della gerarchia del sistema.

Per capire meglio:

- Una esecuzione concorrente può non girare parallelo (caso single-core)
- Una esecuzione parallela è necessariamente concorrente.

Tipi di concorrenza

- **Concorrenza simulata**: sistemi monoprocessore → thread switching
- **Multiprocessore** (multicore, no pipelining)
- **Multiprocessore** (hyperthreading, instruction pipelining)



Creare un processo: fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();  // Crea un nuovo processo
    if (pid < 0) {
        perror("Errore nella fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Questo blocco viene eseguito dal processo figlio
        printf("Sono il processo figlio. Il mio PID e %d\n", getpid())
    ;
        printf("Il PID del mio processo padre e %d\n", getppid());
    } else {
        // Questo blocco viene eseguito dal processo padre
        printf("Sono il processo padre. Il PID del mio figlio e %d\n",
     pid);
    }
```

Creare un processo: fork()

1. fork() crea un nuovo processo. Se ha successo, il processo figlio riceve 0 come valore di ritorno, mentre il padre riceve il PID del figlio.
2. Nel processo figlio (pid $==$ 0), viene eseguito il codice relativo al figlio.
3. Nel processo padre (pid $>$ 0), viene eseguito il codice relativo al padre.

```
1 #include <unistd.h>
2 pid_t fork(void);
```

Fork-exec

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();
    if (pid < 0) {
        perror("Errore nella fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Il processo figlio esegue "ls -l"
        execl("/bin/ls", "ls", "-l", (char *)NULL);
        perror("Errore nell'exec");
        exit(EXIT_FAILURE);
    } else {
        wait(NULL);
        printf("Il processo figlio e terminato.\n");
    }
    return 0;
```

Fork-exec

1. Il processo figlio chiama execl() per eseguire ls -l.
2. Il padre attende la terminazione del figlio con wait(NULL).
3. Se execl() ha successo, il codice dopo di esso nel figlio non viene eseguito.

```
1 #include <unistd.h>
2 int execl(const char *path, const char *arg, ..., (char *) NULL);
```

Pthread Create

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

- thread: Puntatore a pthread_t che conterrà l'ID del thread.
- attr: Attributi del thread (NULL per default).
- start_routine: Funzione che il thread eseguirà.
- arg: Argomento passato alla funzione del thread.
- Ritorno: 0 se successo, codice di errore in caso contrario.

Pthread Join, Exit

```c
int pthread_join(pthread_t thread, void **retval);
```

- thread: ID del thread da attendere.
- retval: Puntatore al valore di ritorno del thread (NULL se non interessa).
- Ritorno: 0 se successo, codice di errore altrimenti.

```c
void pthread_exit(void *retval);
```

Descrizione: Termina il thread corrente e restituisce retval.