DISPENSA DI
SISTEMI OPERATIVI

A.A. 2019-2020

Studenti dell'Alma Mater
Università di Bologna

Contents

1 SISTEMI OPERATIVI A.A. 2019-2020

1.1 Regole di contribuzione (versione 1.3 beta)

1. Contribuire con appunti chiari e concisi
2. Annotare i contenuti più importanti, tralasciando i dettagli specifici
3. Formattare le note in sintassi Markdown: https://daringfireball.net/projects/markdown/syntax, https://www.markdownguide.org/basic-syntax/
4. In caso di contribuzioni frequenti, iscriversi a https://t.me/joinchat/A6FVFQ7snFXU_gEzkl_bwA
5. Utilizzare la lingua italiana

1.2 Strumenti del corso

- Wiki del corso: http://so.v2.cs.unibo.it/
- Gruppo telegram: https://t.me/joinchat/Ck1TqRJ-sLIpQUjFQjc3xg
- Bot del corso: https://t.me/so_cs_unibot, assieme al relativo sorgente
- Mailing list del corso: https://lists.cs.unibo.it/cgi-bin/mailman/listinfo/so
- Lezioni live: http://www.cs.unibo.it/~renzo/live/
- Sito web: http://www.cs.unibo.it/~renzo/so/
- Vecchi lucidi: https://www.cs.unibo.it/~renzo/so/old/lucidi
- Mail: renzo@cs.unibo.it
- Ricevimento studenti: mercoledì ore 13:00-15:00

2 PRESENTAZIONE DEL CORSO (23/09/2019)

2.1 Compiti a casa

- Avere GNU-Linux sui propri sistemi
- Sperimentare Esercizi di "lettura" programmi in C 2019/20
- Creare gruppi (max. 4 persone)
- Iscriversi alla mailing list
- Attivare l'account di laboratorio

La registrazione di un account nella Wiki del corso richiederà l'inserimento di un captcha. Visitare questa pagina per sapere più in dettaglio come fare.

3 INTRODUZIONE (25/09/2019)

3.1 Cos'è un sistema operativo?

Non esiste consenso generale su cosa costituisca un *sistema operativo*, ma una definizione spesso adottata è quella di ritenere tale il *kernel*, un componente in esecuzione dalla fase di boot fino a quella di spegnimento.

Un sistema operativo, in generale, mira ad astrarre e semplificare l'utilizzo dell'hardware su cui opera. Le astrazioni da esso offerte possono essere quelle della memoria, della gestione dei processi e dell'IPC (*Inter Process Communication*), del file system, dei dispositivi di I/O ecc, e sono attuate dalle *chiamate di sistema* (*syscall*), nella fattispecie un insieme di funzioni. Internamente, le chiamate di sistema sono realizzate mediante trap.

Diamo qualche esempio dei compiti dei sistemi operativi:

- **Gestione dei processi**: in un tipico sistema operativo sono solitamente presenti più processi. Sarà allora importante scegliere una politica che consenta una condivisione del tempo di calcolo adeguata. Questo tipo di regola viene detta *scheduling*
- **Condivisione delle risorse**: poiché più processi possono accedere simultaneamente allo stesso hardware (tastiere, monitor, stampanti, ...), è bene regolare il suo utilizzo in modo tale da evitare inconsistenze nella lettura e scrittura dei dati

In un sistema operativo sono di solito presenti dei cosiddetti *programmi di sistema*, utili al suo funzionamento (es. i comandi di utility del sistema GNU). Vi sono inoltre *programmi applicativi*, che fanno uso dei servizi del sistema operativo ma non sono sua parte integrante. Nel corso del tempo la complessità dei sistemi operativi si è evoluta così tanto da ispirare una linea di progettazione minimale: quella dei *microkernel*. In un microkernel sono presenti solo le funzionalità essenziali del sistema, mentre le altre sono offerte come programmi di sistema.

Una classe molto diffusa di sistemi operativi è quella *UNIX-like*, dei sistemi cioè ispirati al sistema operativo UNIX. Lo standard che definisce i criteri di conformità a sistemi UNIX-like è lo IEEE 1003, anche detto POSIX.

4 COMPILATORI (30/09/2019)

4.1 Cross-compiling

Abbiamo visto a lezione una spiegazione semplificata dei cross-compilatori per mezzo di analogie linguistiche. Vediamo di essere più precisi con la trattazione seguente.

Supponiamo di avere un compilatore scritto in codice macchina, e voler ideare un sistema attraverso il quale cambiare macchina sia semplice.

Situazione iniziale: ho un compilatore scritto in linguaggio macchina M_1 che traduce da un linguaggio di alto livello (che chiameremo L) in M_1.
Obiettivo: avere un compilatore scritto in linguaggio macchina M_2 che traduca da L a M_2.

Passi logici:

- Scrivo un compilatore da L a M_1 in L. Lo sforzo è ridotto in quanto uso un linguaggio di alto livello.
- Modifico il compilatore in modo che gli opcode siano quelli di M_2 e traduca quindi in M_2. Ho quindi un compilatore scritto in L che traduce L in M_2. Lo sforzo è "minimo", in quanto non ho dovuto programmare in linguaggio macchina, ma solo scrivere un compilatore in un linguaggio di alto livello.
- Compilo il compilatore (scritto in L, da L a M_2) con il compilatore della situazione iniziale (scritto in M_1, da L a M_1).
- Eseguo il compilatore appena compilato (programma eseguibile M_1, da L a M_2), dandogli in pasto il compilatore scritto in L da L a M_2.
- Ho ora un compilatore scritto in M_2, che traduce da L a M_2

4.2 Approfondimento linguaggio C

Viene affrontata la lettura dei primi programmi C alla pagina Prin C ples. Si compiono anche esperimenti per

- Scrivere programmi C privi della libreria standard: `nolibc`
- Scrivere programmi C privi del sistema operativo: applicazione delle nozioni teoriche di cross-compiling su una macchina concreta *[quale macchina? aggiungere riferimento]*

Il codice sorgente e lo storico dei comandi svolti a lezione è consultabile alla pagina History commentata del 2019 settembre 30.

4.2.1 Istruzioni C utili

- `asm(const char*)` genera codice assembly
- `register`: parola chiave che mette un valore in un registro specificato con asm()

- `volatile`: keyword del linguaggio C che, applicata alla dichiarazione di una variabile, informa l'ottimizzatore che la memoria a cui fa riferimento può essere modificata da fonti esterne al programma, e così facendo previene l'intervento di ottimizzazioni su di essa

4.2.2 Note sull'esperimento di scrittura di un programma senza SO

Le note sono relative al file "noSO.c", vedere la history linkata sopra

- `DDRC`: Data Direction Register C (dove "C" si riferisce al "banco C")
- `PORTC`: valore dei bit del banco C
- L'errore `cannot find entry symbol _start` ci fa pensare che il main non sia il programma principale (link precedente, righe 89-92)
- La system call che fa partire il processo lascia `argc` e `argv` sullo stack, proprio con lo `_start`
- Il programma `start` mette sullo stack `argc` e `argv`, chiama `main` e quando termina main lascia il return value sullo stack e fa la syscall ad exit

4.3 Comandi utili

- `gcc -E`: preprocessore
- `gcc -S`: assembler
- `gcc -c`: codice oggetto
- `$$`: pid della shell
- `$?`: ultimo valore ritornato alla shell
- `gcc -O2`: ottimizzatore
- `gcc -static`: linkaggio statico
- `nm`: elenco dei simboli che stanno nell'eseguibile
- `ld`: linkaggio
- `as`: compila l'assembly in un .o

5 APPROFONDIMENTO C, INSTALLAZIONE DISTRO DEBIAN (02/10/2019)

5.1 Approfondimento linguaggio C

Si continua la lettura dei sorgenti C alla pagina Esercizi di "lettura" programmi in C 2019/20. Si è visto che

- `char *spoint = "hello"` è un puntatore variabile ad un valore costante (possiamo riassegnare il puntatore `spoint`, ma non modificare la stringa alla quale *inizialmente* punta)
- `char sarr[] = "hello"` è una "variabile array", ovvero un puntatore costante. È cioè possibile modificare i valori individuali con la notazione `sarr[x]` ma non riassegnare `sarr` ad un altro indirizzo
- Nell'esercizio *alignment*, i `long int` possono essere messi solo in locazioni di memoria tali che `posizione % 4 == 0`. Le strutture, invece, solo in posizioni tali che `posizione % 8 == 0`. Quindi, la prima struttura salta `7` byte prima di inserire il long int, dopo l'assegnamento del primo byte al char
- Lo switch è un costrutto che internamente fa una `goto` alla label corrispondente. La keyword `break` è stata introdotta solo in seguito

5.1.1 Operatore virgola

L'operatore virgola del linguaggio C svolge diversi ruoli, tra cui

- Separatore di dichiarazioni di variabile: ad esempio `int a = 1, b = 2;`
- Operatore che valuta il primo operando e scarta il risultato, poi valuta il secondo operando e lo ritorna. Se ad esempio `b = (3, 5)` allora `b` vale `5`
- Terminatore di istruzioni (al posto di `;`): esempio: `i++, j--` (nota: alcune istruzioni devono terminare con `;`, come l'ultima istruzione del programma)

5.1.2 Preprocessore C

Le istruzioni del linguaggio C precedute dal carattere `#` sono quelle gestite dal *preprocessore* e vengono dette **direttive**. Esempi di questi direttive possono essere

- `#include <header.h>`. Inserisce tutto il contenuto testuale di `header.h` nel file corrente
- `#define LABEL VALUE`. Sostituisce sintatticamente tutte le occorrenze di `LABEL` nel codice con `VALUE`
- `#define MACRO(X, ...) ()`. Definisce una *macro*
  - Effettua sempre una sostituzione sintattica
  - Tra il nome e la tonda aperta non ci va lo spazio
  - La macro vuole un'espressione, quindi per fare un blocco bisogna usare un `do{ ... } while(0);`

– Se all'interno della macro compare `#X`, viene sostituito il parametro attuale, con 'text ## X' concateno *[migliorare]*

5.2 Installazione sistema Debian

La seconda parte della lezione è dedicata all'installazione passo-passo di una distribuzione Debian su macchina virtuale.

6 STRUMENTI DI SVILUPPO SOFTWARE (07/10/2019)

6.1 Approfondimento linguaggio C

Si continua la lettura dei sorgenti C alla pagina Esercizi di "lettura" programmi in C 2019/20. Abbiamo visto

- Diversi modi per bufferizzare file in memoria: `getchar()`, `putchar()`; funzioni `fopen()`, `fwrite()`, `fread()`, `fclose()`, ...
- Che l'accesso ai file in UNIX è mediato da due insiemi di primitive diverse: quello del sistema operativo (syscall `read()`, `write()`, ...) e quello delle primitive della libreria C (`fread()`, `fwrite()`, `fgets()`, ...)
- L'unificazione delle interfacce dalla linea di comando tramite librerie apposite: `getopt`, `argp` (non accennato a lezione)

6.2 Strumenti software

Abbiamo accennato a quattro classi di strumenti fondamentali per l'informatico nello sviluppo software: sistemi di building, sistemi di versionamento, sistemi di generazione di file di build, sistemi di pacchettizzazione.

6.2.1 Sistemi di building

Automatizzano il processo di compilazione, richiedendo all'utente non più che qualche istruzione da riga di comando per generare un eseguibile/libreria. Esempi concreti di tali strumenti sono

- Make (linguaggio C e affini)
- Maven (linguaggio Java)
- Gradle (linguaggio Java)

6.2.2 Sistemi di versionamento

Sono utili al versionamento di un progetto software, consentendo operazioni quali il ritorno a una versione precedente, il branching (la ramificazione del lavoro), lo sviluppo di gruppo parallelo ecc. Alcuni sistemi di questo tipo sono

- Git
- Subversion
- Mercurial
- CVS

6.2.3 Sistemi di generazione di file build

Sviluppare manualmente file di build per un particolare sistema è un'operazione laboriosa e poco flessibile. I sistemi di generazione di file di build astraggono le particolarità di un dato SO/architettura e generano file di costruzione software,

i quali a loro volta verranno processati da una fase di compilazione automatica (vedere punto 1). `CMake` è uno strumento di questo genere.

6.2.4 Sistemi di pacchettizzazione

Utilizzati nella distribuzione di software *[verificare voce]*.

7 EMULAZIONE CON QEMU, APPROFONDIMENTO CMAKE (09/10/2019)

7.1 Approfondimento linguaggio C

Si continua la lettura dei sorgenti C alla pagina Esercizi di "lettura" programmi in C 2019/20. Si è visto che

- Nel linguaggio C il valore numerico `0` assume il valore booleano `false`, mentre tutti gli altri `true`
- `!!x`, dove `x` può essere una variabile intera, converte tale variabile all'insieme `{0, 1}`
- `NULL` è definito come `0`, quindi `if (puntatore)` è equivalente a `if (puntatore != NULL)`

7.1.1 Operatori bit-a-bit

Gli operatori logici appartengono a una classe differente di quella degli *operatori bit-a-bit* (*bitwise operators*)

- Gli operatori logici sono: `&&`, `||`, `!`
- Quelli bit-a-bit: `&`, `|`, `~` (not), `^` (xor), `<<` (shift sinistro), `>>` (shift destro)

Possiamo vedere un esempio di utilizzo degli operatori bit-a-bit nel sorgente seguente

7.1.1.1 Esempio di programma C

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

// && || ! sono operatori logici
// & | ~ ^ << >> sono operatori bit-a-bit

int main (int argc, char *argv[]) {
        int i = atoi(argv[1]);

        printf("%x\n", i);

        i |= 0x6;
        printf("%x\n", i);

        i &= ~0x9;
        printf("%x\n", i);

        return EXIT_SUCCESS;
}
```

In C siamo anche in grado di esprimere interi in basi diverse da quella decimale

- `0b1100 = 12`, `0b1110 = 14` (da binario a decimale)
- `012 = 10`, `024 = 20` (da ottale a decimale)
- `0x12 = 18` (da esadecimale in decimale)

7.1.2 Aritmetica dei puntatori.

Siano `p, q` due puntatori e `x` una variabile di tipo `int`, allora:

- `p + x` = puntatore spostato di `x` celle (il valore effettivamente incrementato è pari a `x * sizeof(*p)`)
- `p - x` = come sopra, in direzione opposta
- `p - q = intero` (numero di celle comprese tra i due puntatori)
- `printf("%p", &var)` stampa l'indirizzo della variabile var

7.1.3 Tipo di dato `struct`

`size_t offsetof(struct_name, data_field)` ritorna la distanza in byte del campo dall'inizio della struttura.

7.1.4 Tipo di dato `union`

- Nelle union più valori di tipi diversi condividono la stessa zona di memoria
- Tipicamente utilizzate per ottimizzare l'uso della memoria
- L'occupazione in memoria della union è pari a `max {sizeof(data_fields)}`
- Può essere dichiarato con molti campi differenti, ma solo uno di questi può contenere un valore in un dato momento

7.2 Esperimento di emulazione con QEMU

QEMU è una macchina virtuale capace di emulare il linguaggio hardware di un'altra macchina. A differenza di KVM, QEMU fa traduzione dinamica di binari: prende un eseguibile che è per un processore diverso, e usa il codice macchina del powerpc e lo traduce in codice macchina Intel per farlo eseguire.

7.3 Approfondimento CMake

[Aggiungere contenuti.]

7.4 Compiti per casa

- Esplorare l'elenco delle system-call.

8 INTRODUZIONE ALLE SYSTEM CALL (16/10/2019)

8.1 Approfondimento linguaggio C

Si continua la lettura dei sorgenti C alla pagina Esercizi di "lettura" programmi in C 2019/20. Abbiamo visto che

- `unsigned int` è un tipo di dato diverso da `int`: il primo è l'insieme degli interi positivi rappresentabili nella macchina ospite, il secondo quello degli interi con segno

8.1.1 Funzioni variadiche

Le funzioni variadiche (in inglese *variadic functions*) sono funzioni definite dal programmatore in grado di ricevere un numero variabile, e potenzialmente illimitato, di parametri attuali. Consultare la man page `man 3 stdarg` per maggiori informazioni. La f. `printf()`, della libreria standard è un classico esempio di funzione variadica (magg. informazioni, per es. sulle stringhe di formato, possono essere trovate in `man 3 printf`).

Un esempio di realizzazione di `printf()` svolto a lezione è il seguente

8.1.1.1 Esempio di programma C

```c
#include <stdio.h>
#include <stdarg.h>

int vrprintf(const char *format, va_list ap);

static int putcx(int c) {
    if (c != 0) {
        putchar(c);
        return 1;
    } else
        return 0;
}

static char backchar[128] = {
        [0 ... 127] = 0,
        ['a'] = '\a',
        ['b'] = '\b',
        ['f'] = '\f',
        ['n'] = '\n',
        ['r'] = '\r',
        ['t'] = '\t',
        ['v'] = '\v',
```

```
        ['\\'] = '\\',
        ['\''] = '\'',
        ['"'] = '"',
        ['?'] = '?',
};

static int put_backslash(char escapeChar) {
    return putcx(backchar[escapeChar & 0x7f]);
}

static int rvrp_int(int val) {
    if (val == 0)
        return 0;
    else
        return rvrp_int(val / 10) + putcx('0' + val % 10);
}

static int vrp_int(int val) {
    if (val != 0) {
        if (val < 0)
            return putcx('-') + rvrp_int(-val);
        else
            return rvrp_int(val);
    } else
        return putcx('0');
}

static int vrp_string(char *s) {
    switch (*s) {
        case 0:
            return 0;
        case '\\':

            return put_backslash(*(s+1)) ? vrp_string(s + 2) + 1 : 0;
        default:
            return putcx(*s) + vrp_string(s + 1);
    }
}

static int vrp_percent(const char *format, va_list ap) {
    switch (*format) {
        case 0:
            return 0;
        case '%':
            return putcx(*format) + vrprintf(format + 1, ap);
        case 'd':
```

```
            return vrp_int(va_arg(ap, int)) + vrprintf(format + 1, ap);
        case 's':
            return vrp_string(va_arg(ap, char *)) + vrprintf(format + 1, ap);;
        default:
            printf("ERROR\n");
            return 0;
    }
}

int vrprintf(const char *format, va_list ap) {
    switch (*format) {
        case 0:
            return 0;
        case '%':
            return vrp_percent(format + 1, ap);
        case '\\':
            return put_backslash(*(format+1)) ? vrprintf(format + 2, ap) + 1 : 0;
        default:
            return putcx(*format) + vrprintf(format + 1, ap);
    }
}

int rprintf(const char *format, ...) {
    int rval;
    va_list ap;
    va_start (ap, format);
    rval = vrprintf(format, ap);
    va_end(ap);
    return rval;
}


int main() {
    int v;

    v = rprintf("hello world\n");
    printf("%d\n", v);
    v = printf("hello world\n");
    printf("%d\n", v);
    v = rprintf("hello world %d\n", 10);
    printf("%d\n", v);
    v = printf("hello world %d\n", 10);
    printf("%d\n", v);
    v = rprintf("hello world %s %d\n", "piripicchio", 42);
    printf("%d\n", v);
    v = printf("hello world %s %d\n", "piripicchio", 42);
```

```
    printf("%d\n", v);
    v = rprintf("hello world %% \"%s\" %d\n", "piripicchio\tbackslash", 42);
    printf("%d\n", v);
    v = printf("hello world %% \"%s\" %d\n", "piripicchio\tbackslash", 42);
    printf("%d\n", v);
    v = rprintf("%%\n");
    printf("%d\n", v);
    v = printf("%%\n");
    printf("%d\n", v);
}
```

Osserviamo che

- `va_list` è il tipo di dato usato per recuperare i parametri addizionali di una funzione
- `void va_start(va_list ap, paramN)` inizializza `ap` per recuperare parametri addizionali dopo `paramN`
- `void va_end(va_list ap)` va chiamata prima che la funzione da cui va_start() era stata invocata ritorni

8.2 Bitfields per tipi di dato `struct`

I bitfields sono una sintassi per tipi di dato `struct` che consente assegnare bit individuali a determinati campi. Vediamo subito un

8.2.1 Esempio di programma C

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

struct stest {
        // Il valore che segue i due punti indica il numero di bit assegnato al
        // campo corrispondente
        unsigned int a: 2;
        unsigned int b: 4;
        unsigned int c: 2;
};

union utest {
        struct stest s;
        unsigned char c;
};


int main (int argc, char *argv[]) {
        union utest u;
```

```
        if (argc < 2) {
                fprintf(stderr, "%s: <integer>\n", argv[0]);
                exit(EXIT_FAILURE);
        }

        u.c = atoi(argv[1]);
        printf("%x %x %x\n", u.s.a, u.s.b, u.s.c);

        return EXIT_SUCCESS;
}
```

8.3 Direttive al compilatore

Sono comandi (funzioni, macro) comprese da una particolare implementazione di compilatore C, utilizzate ad es. per ottimizzare il codice generato.

Un esempio è fornito da `__builtin_expect()`, che dato un costrutto `if-else` suggerisce al compilatore quale ramo del costrutto ha maggiori probabilità di essere eseguito. L'utilità di tale direttiva è dovuta alla tecnica di pipelining adottata in molte architetture di elaboratori moderne, che consiste in un precaricamento delle istruzioni da eseguire in un file eseguibile.

Esempio d'uso: `if(__builtin_expect(condizione, default))` *[aggiungere sorgente gist]*.

8.4 Introduzione alle system call

Abbiamo iniziato ad affrontare il concetto di *syscall*, contestualmente a quelle dei sistemi operativi UNIX. Sulla wiki del prof. Davoli è presente un ''Catalogo" delle System Call.

Una **System call** (trad. chiamata di sistema) è una primitiva del sistema operativo che consente accesso alle funzionalità da esso offerte. Questi servizi possono essere la gestione dell'input/output, della memoria, del filesystem, dei processi e dell'IPC (Inter Process Communication), della rete ed altro ancora.

L'invocazione di una syscall è **realizzata mediante trap**, che consente l'esecuzione di codice kernel e il passaggio dallo user-mode al kernel-mode non appena ci si imbatte su di essa. Le *trap* sono particolari tipi di interrupt software, ovvero funzioni in grado di fermare il processo in esecuzione. La differenza sostanziale tra interrupt hardware e software è che i primi (causati da dispositivi hw esterni al processore) possono arrivare in maniera asincrona rispetto al clock della CPU.

Tutte le syscall restituiscono un valore intero. Questo può anche essere rappresentato come `size_t` o altri tipi particolari.

8.4.1 Syscall `fork()`

Come detto in precedenza, il PID (process ID) è il numero identificativo di un processo.

La syscall `fork()`, chiamata nel processo A, crea un processo figlio B e restituisce al processo genitore il PID di B, mentre a B restituisce 0.

Un esempio di utilizzo della syscall `fork()` è dato dal seguente:

8.4.1.1 Esempio di programma C

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


int main (int argc, char *argv[]) {
        if (fork())
                puts("AAA");
        else
                puts("BBB");

        return EXIT_SUCCESS;
}
```

Un errore potenzialmente dannoso, chiamato *Fork bomb*, è quello di mal posizionare una `fork()`. Senza una qualche condizione di uscita, la procedura fa partire due processi figli, che a loro volta faranno la stessa cosa. Diventa quindi impossibile fermare completamente questa sequenza di sottoprocessi.

Due syscall affini a quella appena esaminata sono: * `execve()`, che sostituisce il codice del processo chiamante con quello di un programma (file) specificato come argomento. * `getpid()`, che ritorna il PID del processo chiamante.

Nota: per quanto riguarda il tipo di PID, è preferibile utilizzare `pid_t` piuttosto che `int`, perchè il primo è portabile.

8.5 Approfondimento shell

La shell è un interprete dei comandi che permette un dialogo diretto fra utente e sistema operativo. Ne esistono svariate realizzazioni, il cui nome termina solitamente in `sh` (bash, ksh, ash, zsh...). La sintassi tipica di un comando di shell è data dalla forma `<verbo> [complementi] [complemento oggetto]`. Il linguaggio che una shell interpreta viene detto linguaggio di scripting. I comandi di un linguaggio di scripting posso essere classificati in comandi built-in, alias, programmi esterni ecc. Un programma scritto in un linguaggio di scripting si dice *script*. La **differenza tra un linguaggio di programmazione e linguaggio di scripting** risiede nel loro livello di astrazione. Nei primi l' astrazione è

piuttosto bassa, e il livello di responsabilità affidato al programmatore è molto alto. Nei secondi, invece, l'astrazione è più elevata, al punto che i comandi stessi sono solitamente programmi esterni. Le singole istruzioni hanno una *granularità* maggiore nei linguaggi di scripting.

Per poter eseguire uno script da linea di comando bisogna assegnare i permessi di esecuzione, tramite ad es. il comando `chmod u+x <percorso file>`.

9 APPROFONDIMENTO SYSTEM CALL (21/10/2019)

9.1 Approfondimento system call

Viene esaminato il comportamento delle seguenti system call: `fork()`, `execve()`, `wait()`, `waitpid()`, `open()`, `close()`, `read()`, `write()`, `dup()`. Per maggiori informazioni: `man 2 <nome syscall>`.

9.2 Processi zombie e processi demoni

Un processo che ha duplicato se stesso può attendere il termine di un processo figlio tramite la chiamata `wait()` (o simili). Quando un processo termina, il sistema operativo è autorizzato a liberare le risorse che questi aveva allocato, ma non a rimuovere la sua voce dalla tabella dei processi. Qui è infatti contenuto lo stato di ritorno del processo, che deve poter esser letto dal processo genitore tramite `wait()`. Un processo che ha terminato, ma il cui genitore non ha ancora letto lo stato di ritorno, è detto **zombie**. Tale modalità non ha nulla di anomalo ed è assunta brevemente da ogni processo in uno stato intermedio tra quello di terminazione e di `wait()`.

Un **processo demone** è caratterizzato dal fatto di eseguire in background per l'intero periodo di accensione. In sistemi UNIX-like, processi demoni sono avviati da `init`, o invocati da processi regolari che invocano una `fork()`, terminano se stessi immediatamente, e passano (implicitamente) il controllo del processo figlio ad `init`. In tali sistemi il nome di processi demone termina per distinzione in `d`.

9.3 Approfondimento shell

Secondo la filosofia UNIX, un comando deve: fare una cosa, farla bene, e consentire operazioni più complesse tramite la concatenazione (mediante pipe) con altri programmi. Se ad esempio voglio visualizzare un file, filtrare le parti di mio interesse e riordinarle, posso eseguire: `cat <percorso_file> | grep <regex> | sort`, dove `<percorso_file>` è il percorso del file da visualizzare e `<regex>` un'espressione regolare da usare come filtro.

9.3.1 Comandi bash

- `$?` è la variabile che contiene il valore di ritorno del comando precedente
- `cat`: concatena più contenuti di file
- `dd`: disk duplicate, serve per copiare velocemente dei file, anche su dischi, con opzioni aggiuntive
- `strace`: mostra quali syscall sono invocate dall'eseguibile passato come argomento

9.4 Annotazioni varie [da sistemare]

- Una trap è un interrupt software. Possono esservi due tipi di interrupt (chiamate in grado di interrompere il processo corrente): software e hardware. Alla fine di ogni ciclo il processore controlla se nel vettore di interruzioni c'è stata caricata un'interruzione. Se trova qualcosa sospende le sue attività correnti, salva il proprio stato, ed esegue una funzione chiamata interrupt handler (oppure una routine di servizio, ISR) per gestire l'evento
- Gli interrupt hardware (causati da dispositivi hw ESTERNI al processore) possono arrivare in maniera asincrona rispetto al clock del processore. Gli interrupt software sono dovuti al programma in esecuzione in quel momento.
- Il Kernel è caricato sempre alle stesse posizioni nella RAM
- exit(return_value) conclude un processo, fornendo il valore di ritorno all'ambiente ospite
- environment: variabili che i processi possono vedere
- se il processo diventa orfano il valore di ritorno viene passato alla gerarchia superiore, se non ritorna niente è un processo zombie
- L'elenco degli errori è in /usr/include/asm-generic/errno-base.h
- Stream: sequenza di byte, c'è una successione di byte (non si sa quanto sarà lunga)
- Buffer: contenitore, zona di memoria in cui si tengono temporaneamente dei byte

System call importanti

- waitpid() aspetta che il processo figlio con un determinato PID termini
- man wait contiene le macro per dedurre le informazioni dallo status settato opportunamente da waitpid()
- per fare i processi demoni il padre lancia il processo e si suicida senza fare wait
- open: apre un file
- read: legge una serie di bit da un file
- `*buf`: buffer in cui scrive ciò che ha letto
- count: dimensione del buffer che vuole leggere
- write: scrive su un buffer

9.4.1 Funzioni della `libc`

- strerror() converte un intero nella stringa rappresentante l'errore
- perror() stampa l'errore in maniera standard

9.5 Streams [da sistemare]

- Gli standard stream (stream = sequenza di byte, non si sa quanto sarà lunga!) sono canali pre-connessi di comunicazione in input e output tra un programma ed il suo ambiente quando il programma viene eseguito.

- I tre canali sono chiamati standard input (`stdin`), standard output (`stdout`) and standard error (`stderr`).
- standard input: canale da cui il processo legge per acquisire informazioni dall'utente
- standard output: canale su cui il processo scrive informazioni normalmente
- standard error: canale su cui il processo scrive informazioni di errore
- Attaccate al file descriptor ci sono le strutture dati associate, che vengono rimosse con la close()
- I file descriptor sono degli indici di un vettore che puntano ai rispettivi files, con la dup abbiamo poi due indici che puntano allo stesso file

10 FILE SYSTEM (23/10/2019)

10.1 Gestione del file system

10.1.1 File descriptor

Un *file descriptor* (fd) è un numero intero utilizzato per identificare un file aperto da un qualsiasi processo.

10.1.2 Tabelle del file system

Il kernel utilizza un sistema di tabelle per la gestione dei file:

- La tabella dei *file descriptor*, più propriamente un vettore, è associata ad **ogni processo attivo**. Tiene traccia di quali file descriptor sono correntemente in uso. Ad ognuno di questi può essere associata una corrispondente voce nella tabella dei file aperti
- La tabella dei file aperti (*open file table*) gestisce informazioni relative ai file aperti, quali le modalità di accesso (lettura, scrittura, ...), la posizione attuale del cursore ecc. Per garantire un accesso rapido essa è mantenuta in memoria centrale. E' unica e condivisa da tutti i processi che ne fanno richiesta. Questo costrutto, a sua volta, gestisce riferimenti alla tabella vnode
- La tabella vnode contiene informazioni sul file quali l'ampiezza in byte, info. di proprietà, permessi ecc. Anche questa tabella è globale. Per ulteriori info vedere:
  - https://stackoverflow.com/questions/27345342/inode-vs-vnode-difference
  - https://www.usna.edu/Users/cs/wcbrown/courses/IC221/classes/L09/Class.html

Gli esempi pratici di programmi in C seguenti sono intesi ad illustrare molti dei dettagli e dei comportamenti di chiamate di sistema per la gestione dei file. Iniziamo da

10.1.3 Esempio di programma C

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


int main (int argc, char *argv[]) {
        int fd1 = open("/tmp/myfile", O_WRONLY | O_TRUNC | O_CREAT, 0666);

        switch (fork()) {
```

```
                default:
                        write(fd1, "Montagna", 8);
                        break;
                case 0:
                        write(fd1, "Ciao", 4);
                        break;
                case -1:
                        exit(1);
        }

        close(fd1);
        return EXIT_SUCCESS;
}
```

Il programma in questione tenta di aprire il file `/tmp/myfile` in sola scrittura, troncando nel caso sia già presente o creandone uno nuovo qualora sia inesistente. In seguito alla chiamata a `fork()` il processo figlio erediterà una copia esatta della tabella dei file descriptor, comprendente il valore di `fd1` che farà riferimento a un'opportuna voce nella tabella dei file aperti. Sapendo che la syscall `write()` è **atomica**, possiamo pertanto dire che il contenuto scritto in `/tmp/myfile` sarà

`MontagnaCiao`

o

`CiaoMontagna`

in base all'ordine in cui i due processi vengono schedulati.

Proseguiamo con la seguente variazione su tema

10.1.4 Esempio di programma C

```c
int main (int argc, char* argv[]) {
    int fd1 = open("/tmp/myfile", O_WRONLY | O_TRUNC | O_CREAT, 0666);
    // Crea un'altra voce nella open files atable
    int fd2 = open("/tmp/myfile", O_WRONLY | O_TRUNC | O_CREAT, 0666);

    // fd2 punta alla stessa voce di fd1 nella open files table
    fd2 = dup(fd1);
    write(fd1, "Montagna", 8);
    write(fd2, "Ciao", 4);

    close(fd1);
    close(fd2);
}
```

In questo esempio è bene ricordare che una chiamata a `fd2 = dup(fd1)` fa in modo da rendere `fd1` ed `fd2` sinonimi, in maniera tale da consentire intercambiabile il loro uso. Ora, sebbene l'invocazione di due `open()` comporti l'apertura di due voci distinte nella tabella dei file aperti *[verificare voce]*, in seguito alla chiamata a `dup()` si avranno due fd con lo stesso riferimento alla TFA, che condividono anche il cursore di scrittura. Così non fosse stato *[verificare voce]*, le due scritture con `fd1` ed `fd2` avrebbero generato sovrapposizioni.

10.2 System call `pipe()`

La chiamata di sistema `int pipe(int pipefd[2])` crea un canale di dati unidirezionale tra il file descriptor `pipefd[0]` e `pipefd[1]`: ogni byte scritto in `pipefd[1]` verrà memorizzato in un buffer collegato a `pipefd[0]`. Consideriamo il seguente

10.2.1 Esempio di programma C

```c
int main(int argc, char* argv[]) {
    int pfd[2];
    size_t nread;
    char buffer[20];

    // 'pfd' non è inizializzato e conterrà valori casuali, ma
    // ciò è irrilevante ai fini del nostro esempio
    printf("(pfd[0], pfd[1]) = (%d, %d)\n", pfd[0], pfd[1]);

    pipe(pfd);
    write(pfd[1], "prova", 5);
    nread = read(pfd[0], buffer, 19);
    // La terminazione manuale è necessaria, in quanto non è
    // effettuata dalle chiamate di sistema
    buffer[nread] = '\0';
    printf("%s\n", buffer);

    close(pfd[0]);
    close(pfd[1]);
}
```

Un utilizzo di `pipe()` lievemente più sofisticato è il seguente, in cui si fa uso della syscall `fork()`

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

- `exec()` è una famiglia di syscall con molte varianti, in base ai suffissi aggiunti: p.e. 'execlp()' lancia un comando al posto di eseguire un file
- La libreria C fa buffering, ad esempio questo programma stampa "Montagna\nMontagna\n" https://paste.pics/b4cf358737e7d5cc892738af0616657a questo perché la printf non chiama write() ma mette la stringa in un

buffer in memoria. Al momento della fork, "montagna" è sia nel buffer del padre che del figlio. Quando si stampa '\n' il buffer viene svuotato il "problema" sta nel fatto di aver usato insieme system call e funzioni di C. Usando write() al posto di printf() l'output del programma sarebbe stato solo "Montagna"

- L'utilità della condivisione, da parte di padre e figlio, del sistema di file aperti è quella di evitare particolari conflitti indesiderati; un esempio può verificarsi nel caso entrambi stiano modificando lo stesso file nello stesso momento.

10.3 System call `lseek()`

Dato un file descriptor `fd`, la syscall `lseek()` permette di spostare il cursore di lettura/scrittura (posizione attuale nel file, "file offset" in inglese) di tale file descriptor ad una posizione arbitraria.

È possibile porsi la seguente domanda: aprendo un nuovo file in scrittura, e spostandosi ad una posizione arbitraria `n` e ivi scrivendovi, cosa accadrà a tutti i byte precedenti? La risposta è che il file system sarà in grado di considerare i buchi così creati evitando di allocare memoria eccessiva in loro conto. La situazione appena descritta è riproducibile dal programma seguente

```
#include <...>

int main (int argc, char *argv[]) {
    int fd1 = open("/tmp/myfile", O_WRONLY | O_TRUNC | O_CREAT, 0666);
    lseek(fd1, 1000000000, SEEK_SET);
    write(fd1, "A", 1);
    close(fd1);

    return 0;
}
```

Esempio https://snipboard.io/DctCal.jpg

10.4 Struttura tipica del file system

La struttura di un file system UNIX assume la forma di DAG (Grafo Diretto Aciclico). È molto frequente ritrovare le seguenti directory al livello radice di un file-system UNIX:

- `/etc`: file di configurazione del sistema
- `/var`: tutti gli spool, i log, files temporanei
- `/bin`: i binari (eseguibili) dei comandi principali
- `/lib`: librerie
- `/usr`: directory utenti (può essere condivisa)
- `/usr/bin` e `/usr/lib`: comandi e librerie condivise
- `/home`: home directory utenti

- `/dev`: dispositivi, visti come file secondo la filosofia everything is a file, otherwise, it is a process
- `/mnt`: cartella adibita al montaggio temporaneo di dispositivi (es. periferiche)

Il sistema operativo fornisce due chiamate di sistema fondamentali per la gestione delle directory: `mount` e `umount`; la loro invocazione richiede permessi di root. Esiste un analogo comando da terminale `mount <sorgente> <destinazione>`.

10.5 inode e link

Un *inode* è la struttura dati che contiene informazioni relative ad un oggetto del file system (es. file o directory) quali la locazione dei suoi dati, informazioni sui permessi, su date di accesso, di modifica ecc. Ogni inode è identificato univocamente da un intero, che indicizza nella tabella degli inode: questa è unica, e gestita dal sistema operativo. È bene non confondere la nozione di percorso con quella di inode: mentre un percorso è una delle tante possibili associazioni per un inode, e fa parte di una molteplicità, un inode è univoco.

10.5.1 Link fisici e link simbolici

Dato un file F, è possibile creare una o più associazioni ad esso dette *collegamenti* o *link*. Vi sono due tipi di link:

- Link fisico (detto anche *hard link*): copia esatta di F e di relative stats e opzioni. Si riferisce al medesimo inode del file originale. Una modifica di F si trasmette anche ai rispettivi hard links e viceversa. L'effettiva eliminazione dell'inode di F avviene al momento della cancellazione di F e di tutti i suoi hard links.
- Link simbolico (detto anche *soft link* o *symlink*): file che memorizza il percorso di F. La lettura del percorso avviene a runtime, e se non esiste viene sollevato un errore. Ha ovvimanete un inode differente da quello del file originario.

Per la manipolazione dei link fisici, è possibile affidarsi alle syscall `link()` e `unlink()`; per quelli simbolici `symlink()`. L'utilizzo dei link simbolici è in parte legittimato dall'impossibilità di creare link fisici tra dispositivi diversi (es. tra un disco fisso e una periferica).

10.5.1.1 File temporanei

È possibile gestire file temporanei, la cui durata cioè è equivalente a quella del processo che li mantiene, invocando `open()` seguita da `unlink()`. Il file sarà cancellato al termine del processo.

10.5.2 `stat()`

La syscall `stat()` (e il corrispettivo comando da shell) visualizza informazioni dettagliate riguardato un file. Tale funzione, e le sue varianti, restituiscono una struttura `stat` del linguaggio C con i seguenti campi

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */
    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */
#define st_atime st_atim.tv_sec      /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

10.5.3 Permessi

È possibile manipolare i permessi di un dato file o directory con la syscall `chown` (***ch****ange* ***own****er*).

10.5.4 Tipi di files

I sistemi UNIX-like seguono la filosofia everything is a file: nel tentativo di riutilizzare lo stesso codice, dispositivi esterni, terminali, socket e altre entità vengono gestite con le stesse chiamate per la gestione dei file. Vengono così introdotti più "tipi" di file, tra cui:

- File normali
- Directory
- Dispositivi a blocchi (*block devices*)
- Dispositivi a caratteri (*character devices*)

11 FILE E DIRECTORY (28/10/2019)

11.1 `fcntl()`

La syscall `fcntl(int fd, int cmd, ...)` offre operazioni di controllo sui file descriptor, tra cui:

- Duplicazione di file descriptor, ma con una modalità diversa da `dup()`
- Settaggio dei flag del file descriptor. In realtà solamente un flag, `FD_CLOEXEC`, è attualmente supportato
- Settaggio dei flag di stato degli open files, inizializzati da `open()`. Sono ignorati i flag di modalità di accesso (lettura/scrittura) e di creazione
- Lock di files, anche su porzioni diverse dello stesso, in maniera tale che più processi vi possano scrivere senza sovrapposizioni Le operazioni di `fcntl()` sono funzionalità ben determinate e globali; si applicano ai file in quanto stringhe di caratteri.

11.2 `ioctl()`

`ioctl(int fd, unsigned long request)` offre operazioni di controllo su file speciali (che rappresentano quindi dei device). Supponiamo di avere un device (ad esempio una scheda audio). Se sappiamo in che formato deve essere l'audio basta fare scrittura sulla scheda (o meglio sul file descriptor riferito al device) e avremo il suono. Tuttavia potrebbero esserci dei parametri da modificare che non hanno il corrispettivo tra `read()` e `write()` (ad esempio "avvolgi il nastro"). Per tutte le operazioni che non coinvolgono read e write si usa `ioctl()`.

11.2.1 Parentesi: implementazione syscall

L'aspetto *everything is a file* dei sistemi UNIX consente di gestire ogni dispositivo con le stesse syscall dei file. Una domanda che può naturalmente sorgere allora è: come realizzare, a livello implementativo, un meccanismo tale per cui una *stessa* syscall `f()` si comporti in maniera diversa sulla base del dispositivo?
La risposta è che i driver dei dispositivi possiedono puntatori a funzioni che specificano il comportamento da assumere quando viene invocata una chiamata di sistema su di essi. L'effetto così realizzato è molto simile al polimorfismo.

11.2.2 `mknod()`

`mknod()` serve a creare file speciali. Un esempio di file speciale può essere dato da file `pts` associati a canali di input/output di determinati terminali. Proviamo a creare un file di questo genere nel seguente programma (sono necessari permessi di root per l'esecuzione)

11.2.3 Esempio di programma C

```
#include <stdio.h>
#include <sys/types.h>
```

```c
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    // Crea un device associando i due ID passati come parametro
    // il primo denota la classe a cui appartiene il dispositivo,
    // il secondo invece indica un possibile dispositivo di quella classe
    dev_t mydev = makedev(136, 5);

    // Assegna il device ad un file, in questo caso "/tmp/newterm"
    // S_IFCHR è un flag che segnala che il file è a caratteri
    if (mknod("/tmp/newterm", S_IFCHR | 0666, mydev) < 0)
        perror("mknod");

}
```

L'esecuzione di questo programma comporterà la creazione di un file a caratteri (flag `S_IFCHR`) sul quale sarà possibile scrivere e leggere (sulla base dei permessi ad esso assegnati). File di questo genere si trovano solitamente sotto la locazione `/dev/pts`, il loro nome è un indice numero da `0` in poi, e ognuno di essi è associato a un terminale aperto nella macchina corrente.

11.3 Syscall per la gestione delle directory

Esistono chiamate di sistema dedicate alla gestione delle directory, tra le quali

- `mkdir()` crea una nuova directory
- `rmdir()` rimuove una directory, solo se vuota
- `chdir()` sostituisce la directory corrente del processo con quella indicata come argomento
- `getcwd()` restituisce il percorso della directory corrente. *NOTA: esiste una funzione della libreria C omonima che fa qualcosa di più rispetto alla systemcall. La systemcall piazza il path in un buffer passato* dal chiamante.
- `openat()` specifica (indirettamente, tramite il primo parametro) il percorso da utilizzare come nuova directory corrente
- `getdents()`: vedi paragrafi successivi
- `opendir()`, `openfdir()`, `readdir()`, `rewinddir()`: vedi paragrafi successivi

È possibile effettuare operazioni sulle directory tramite file descriptor come su file. Per riuscirci è sufficiente invocare `open()` con il suo percorso come argomento. Il sorgente seguente ne dà un esempio concreto

11.3.1 Esempio di programma C

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

void printcwd();

int main(int argc, char *argv[]) {
    printcwd();

    // Salva il file descriptor della cwd
    int pwdfd = open(".", O_RDONLY | O_DIRECTORY);

    // Cambia la cwd in quella in argv[1]
    chdir(argv[1]);
    printcwd();

    // Cambia la cwd in quella salvata in precedenza
    fchdir(pwdfd);
    printcwd();

    close(pwdfd);
}

void printcwd(void) {
    char *cwd = getcwd(NULL, 0);

    printf("PWD= %s\n", cwd);

    free(cwd);
}
```

Problematiche di programmazione concorrente
La gestione del file system precedentemente illustrata è pensata per programmi single-thread. È possibile infatti che, mentre un thread tenti di aprire un file in un determinata directory, un altro stia per cambiarla, alterando il risultato della `open()` successiva. Per ovviare a questo problema esistono syscall terminanti in `at`, quali `openat()`, `fchownat()`, `fstatat()`, `linkat()`, ecc.

11.3.2 `getdents()`

La syscall `getdents()` ricava informazioni legate a directory. Ognuna delle strutture gestite ha il seguente formato

```c
struct linux_dirent {
    unsigned long  d_ino;     /* Inode number */
```

```
    unsigned long  d_off;     /* Offset to next linux_dirent */
    unsigned short d_reclen;  /* Length of this linux_dirent */
    char           d_name[];  /* Filename (null-terminated) */

    /*
    // Visto che "d_name" è di lunghezza indefinita ci sarà un padding e l'ultimo byte rappr

    char           pad;     // Zero padding byte
    char           d_type;  // File type (only since Linux
                            // 2.6.4); offset is (d_reclen - 1)
    */
}
```

Con questa chiamata è possibile implementare un comando con le stesse finalità di `ls`, come nell'esempio seguente (`myls.c`)

11.3.3 Esempio di programma C

```
#define GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/syscall.h>   /* For SYS_xxx definitions */

#define BUFSIZE 8192
char buf[BUFSIZE];

int main(int argc, char* argv[]) {
    int dirfd;
    char *path = argc > 1 ? argv[1] : ".";
    ssize_t rl;
    ssize_t offset;

    dirfd = open(path, O_RDONLY | O_DIRECTORY);

    while ((rl = syscall(__NR_getdents64, dirfd, buf, BUFSIZE)) > 0) {
        printf("%d\n", rl);

        // Incomincio da 0, e ciclo fino a quando l'offset non supera la
        // lunghezza del buffer che ho letto

        offset = 0;

        while (offset < rl) {
```

```
            // Prende la entry successiva a quelle che ho già letto
            struct dirent64 *de = (void*)buf + offset;

            printf("%s\n", de->d_name);

            offset += de->d_reclen;
        }
    }

    close(dirfd);
}
```

Benché efficace, questa chiamata risulta di difficile utilizzo, al punto che solitamente si preferisce affidarsi a chiamate della libreria C quali `opendir()` e `fdopendir()`. Un esempio di utilizzo di tali chiamate è dato nel seguente programma C che svolge lo stesso compito del precedente

11.3.4 Esempio di programma C

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>

#define BUFSIZE 8192
char buf[BUFSIZE];

int main(int argc, char* argv[]) {
    char *path = argc > 1 ? argv[1] : ".";
    DIR *d = opendir(path);

    if (d) {
        struct dirent *de;

        while (de = readdir(d) != NULL)
            printf("%s\n", de->d_name);
    }

    closedir(d);
}
```

11.4 Pipe tra processi

Nelle lezioni precedenti abbiamo visto la creazione di pipe (un canale di comunicazione FIFO) tra processi "parenti". Può sempre sorgere la necessità di far comunicare tra loro processi estranei, dove le tecniche delle volte precedenti risultano insufficienti. Per risolvere questo problema i sistemi UNIX forniscono una utility di sistema, `mkfifo`, che consente la creazione di pipe e l'associazione di nomi sotto forma di percorso.

In quest'ultimo programma vediamo un esempio di comunicazione mediante pipe tra due processi. Prima della sua esecuzione sarà necessario creare i due file pipe mediante il comando `mkfifo`, e passare il loro percorso come argomento

11.4.1 Esempio di programma C

```c
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
    int fdin, fdout;
    char *buf = NULL;
    char *ibuf[1024];
    size_t buflen = 0;
    ssize_t len;

    fdin = open(argv[1], O_RDONLY | O_NONBLOCK);
    fdout = open(argv[2], O_WRONLY);

    while (1) {
        if (getline(&buf, &buflen, stdin) <= 0)
            break;

        write(fdout, buf, len);

        if ((len = read(fdin, ibuf, 1024)) < 0)
            break;

        write(1, ibuf, len);
    }
}
```

12 INTRODUZIONE ALLA CONCORRENZA (30/10/2019)

12.1 File speciali a blocchi

Torniamo ad uno dei programmi della volta precedente, dove abbiamo utilizzato la syscall `mknod()` per la creazione di un dispositivo terminale. Modifichiamo tale sorgente per la creazione di un altro tipo di file, un dispositivo a blocchi con id `(254, 0)`

12.1.1 Esempio di programma C

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    dev_t mydev = makedev(254, 0);

    // S_IFBLK perché il file è a blocchi
    if (mknod("/tmp/newterm", S_IFBLK | 0666, mydev) < 0)
        perror("mknod");
}
```

Con questo programma siamo stati in grado di creare un file speciale, cioè un device che corrisponde ad un alias al device `(254, 0)` di sistema, ovvero la radice del file system *[verificare voce]*.

12.2 Chat tra processi

Un altro programma lasciato in sospeso durante l'ultima lezione è la chat tra processi, dove ci si era avvalsi di due pipe nominate come canali di ingresso e di uscita. Riprendiamolo

12.2.1 Esempio di programma C

```c
#include <stdio.h>
#include <fcntl.h>

// Versione vecchia: getline attende input da rigirare all'altro programma
// La prima pipe viene chiamata con /tmp/myf1, la seconda /tmp/myf2
int main(int argc, char *argv[]) {
    int fdin, fdout;
    char *buf = NULL;
```

```
    char *ibuf[1024];
    size_t buflen = 0;
    ssize_t len;

    fdin = open(argv[1], O_RDONLY | O_NONBLOCK);
    fdout = open(argv[2], O_WRONLY);

    while (1) {
        if (getline(&buf, &buflen, stdin) <= 0)
            break;

        write(fdout, buf, len);

        if ((len = read(fdin, ibuf, 1024)) < 0)
            break;

        write(1, ibuf, len);
    }
}
```

Possiamo realizzare un'implementazione diversa dello stesso programma adottando un sistema ad eventi, mediato dalla syscall `poll()`. Per evento, in questo caso, intendiamo una sequenza di bit che si possono accendere *[verificare]*.

12.2.2 Esempio di programma C

```
#include <...>
#include <poll.h>

#define BUFSIZE 1024

int main (int argc, char** argv) {
    int fdin = open(argv[1], O_RDWR);
    int fdout = open(argv[2], O_RDWR);
    struct pollfd pfd[] = {{STDIN_FILENO, POLLIN}, {fdin, POLLIN}, 0};
    char *buf[BUFSIZE];
    ssize_t len;

    while (1) {
        poll(pfd, 2, -1);

        if (pfd[0].revents & POLLIN) {
            if ((len = read(STDIN_FILENO, buf, BUFSIZE)) < 0)
                break;
```

```
            write(fdout, buf, len);
        }

        if (pfd[1].revents & POLLIN) {
            if ((len = read(fdin, buf, BUFSIZE)) < 0)
                break;

            write(STDOUT_FILENO, buf, len);
        }
    }
}
```

Anche questa volta abbiamo utilizzato una coppia di pipe `/tmp/myf1`, `/tmp/myf2`. In questa implementazione tuttavia cambia il modo con cui ricaviamo dati in lettura, che avviene leggendo gli eventi nel costrutto `while` principale. Rimane possibile leggere informazioni dall'utente, per poter essere inviate all'altro processo, mediante console.

Quest'ulteriore versione del programma fa invece utilizzo della syscall `select()` al posto di `poll()`:

12.2.3 Esempio di programma C

```
#include <...>
#include <sys/select.h>

#define BUFSIZE 1024

int main (int argc, char** argv) {
    int fdin = open(argv[1], O_RDWR);
    int fdout = open(argv[2], O_RDWR);
    fd_set inset;
    char *buf[BUFSIZE];
    ssize_t len;

    while (1) {
        int max;
        FD_ZERO(&inset);
        FD_SET(STDIN_FILENO, &insert);
        FD_SET(fdin, &insert);

        max = ((STDIN_FILENO < fdin) ? fdin : STDIN_FILENO) + 1;
        select(max, &inset, NULL, NULL, NULL);

        if (FD_ISSET(STDIN_FILENO, &inset)) {
```

```
            if ((len = read(STDIN_FILENO, buf, BUFSIZE)) < 0)
                break;

            write(fdout, buf, len);
        }

        if (FD_ISSET(fdin, &inset)) {
            if ((len = read(fdin, buf, BUFSIZE)) < 0)
                break;

            write(STDOUT_FILENO, buf, len);
        }
    }
```

12.3 Concorrenza

A partire da questa lezione, iniziamo ad occuparci di temi legati alla concorrenza. La **concorrenza** di un sistema è la capacità di far avanzare più compiti in uno stesso intervallo di tempo, e va distinta dalla nozione di **parallelismo**. Nel contesto dei calcolatori moderni, un sistema si dice parallelo quando possiede più unità di processamento (CPU o *cores*) capaci di eseguire computazioni in maniera perfettamente parallela. Un calcolatore può effettuare calcoli in maniera concorrente senza con ciò essere necessariamente parallelo, in quanto la concorrenza si può ottenere con tecniche di *time-slicing*, dove più compiti sono eseguiti in alternanza.

Un modo con cui ottenere sistemi concorrenti è la duplicazione dei processi, come abbiamo già visto. Nel tempo, però, questa tecnica è stata perfezionata ed è nata l'idea di *thread*, detto anche processo leggero. Un thread è un'esecuzione di codice parallela a quella di altri thread o processi, e si differenzia da un processo per

- Minor uso di risorse a tempo di creazione e cancellazione
- Condivisione di certe aree di memoria con altri thread dello stesso processo
- Appartenenza ad un processo: ad ogni processo possono essere associati uno o più thread.

La syscall per creare un nuovo thread nel processo chiamante si chiama `pthread_create()` ed è ottenibile tramite la libreria `pthread`. Un'altra chiamata analoga è la syscall `clone()` la quale, invece di essere usata direttamente, è spesso adoperata internamente per la creazione di thread.

Nota: la libreria `pthread` non fa parte della libreria standard del C, quindi va collegata tramite comando da console al momento del linking con `gcc -o <nome eseguibile> <nome file> -lpthread`.

Possiamo far uso delle tecniche di concorrenza per realizzare una versione ancora diversa del programma chat precedente. L'approccio di questa versione sarà

completamente diverso, in quanto non abbiamo a che fare con programmazione ad eventi, ma con due cicli che eseguono simultaneamente.

12.3.1 Esempio di programma C

```
#include <...> // vedere dichiarazioni nei programmi precedenti
#include <pthread.h>

#define BUFSIZE 1024

void* filecat (void *arg);

int main(int argc, char * argv[]) {
    int fdin = open(argv[1], O_RDWR);
    int fdout = open(argv[2], O_RDWR);
    int fdp0[2] = {STDIN_FILENO, fdout};
    int fdp1[2] = {fdin, STDOUT_FILENO};
    pthread_t t0, t1;

#if 1
    // In questo caso ci sono due thread, t0 e t1
    pthread_create(&t0, NULL, filecat, fdp0);
    pthread_create(&t1, NULL, filecat, fdp1);
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);
#else
    // In questo caso ci sono due thread, quello del main e quello creato
    pthread_create(&t0, NULL, filecat, fdp0);
    filecat(fdp1);
#endif
}

void *filecat (void *arg) {
    int *fd = arg;
    char *buf[BUFSIZE];
    ssize_t len;

    while(1) {
        if ((len = read(fd[0], buf, BUFSIZE)) < 0)
            break;

       write(fd[1], buf, len);
    }
}
```

12.4 Problematiche di concorrenza

Questo primo approccio ci consente di dare un'occhiata (sebbene ancora da lontano) alle principali problematiche che si incontrano quando si ha a che fare con la programmazione concorrente.

12.4.1 Esempio di programma C

```
int saldo;

int op_bancaria(int x) {
    saldo = saldo + x;
}
```

Poniamo che tanti processi eseguano queste operazioni simultaneamente, e una persona vada contemporaneamente a depositare 100 euro ed a versare 100 euro. Siamo sicuri che l'operazione funzionerebbe? Risposta: no. In uno pseudocodice macchina, l'operazione `saldo = saldo + x` non è atomica; viene infatti spezzata nelle tre seguenti operazioni, per semplicità mostrate in pseudocodice macchina

```
LOAD Rx, saldo
ADD Rx, x
STORE Rx, saldo
```

12.5 Segnali

Un segnale è una notifica inviata ad un particolare processo/thread nel momento in cui si verifica un determinato evento. I segnali sono numerati, l'utente può quindi decidere cosa fare in al momento della ricezione di determinati segnali.

12.5.1 Esempi di segnali

- Segnale 9 (`SIGKILL`): termina un programma, qualunque sia la condizione in cui esso si trovi. E' l'unico segnale che non può essere gestito in alcun caso
- Segnale 11 (`SIGSEGV`): segnalato dal kernel nel momento in cui si accede ad una parte di memoria per la quale non si ha autorizzazione
- Segnale 14 (`SIGALARM`): segnala al kernel quando deve essere inviato un segnale. (uso: in 3sec send sigalarm)

Nota importante: fare `kill -9 pid` è diverso da fare `kill pid` senza segnale. L'esempio seguente cerca di rendere conto di tale fatto

12.5.2 Esempio di programma C

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

```c
void handler(int sig);

int main(int argc, char *argv[]) {
   signal(SIGTERM, handler);

   while (1) {
       printf(".");

       fflush(stdout);
       sleep(1);
   }
}

void handler(int sig) {
   printf("No, non voglio morire\n");
}
```

Questo programma stampa un punto al secondo. Con il comando `kill <pid esempio>` viene inviato al processo il segnale di terminazione, `SIGTERM`. Alla ricezione del segnale il processo esegue il codice contenuto nell'handler e quindi stampa "No, non voglio morire", dopodiché riprende la normale esecuzione del programma. Con `kill -9 <pid_esempio>` il processo viene terminato istantaneamente.

La syscall `signal()` stabilisce che, al ricevimento di un particolare segnale (dato in input da console oppure da un'altra procedura del programma stesso), venga attivata una procedura o handler. Vediamo nel seguente sorgente C un esempio di *[spiegare intenzione del programma]*

12.5.3 Esempio di programma C

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

void handler(int sig);

//  (2)

int main(int argc, char *argv[]) {
    pid_t pid;
    signal(SIGTERM, handler); //  (1)

    switch (pid = fork()) {
        case 0:
```

```
            while (1) {
                printf(".");
                // La procedura fflush(<stream>) serve a forzare lo svuotamento
                // del buffer e mostrare il tutto a video
                fflush(stdout);

                sleep(1);
            }
        default:
            sleep(10);
            kill(pid, SIGTERM);
            sleep(5);
            kill(pid, SIGTERM);
            sleep(5);
            kill(pid, SIGKILL);
    }
}

void handler(int sig) {
    printf("No, non voglio morire\n");
}
```

Dopo i `10` e `15` secondi dall'avvio di questo programma viene stampata la scritta `No, non voglio morire`; al secondo `20` il processo viene ucciso. La scarsa espressività di questo metodo (per esempio, quale dovrebbe essere il comportamento se durante l'handling di un SIGTERM ne viene ricevuto un secondo?) ha motivato l'introduzione di `sigaction()`. Quest'ultima syscall serve a ***[specificare funzionamento della procedura]***

Sostituendo a (1)

```
struct sigaction new = {
    .sa_handler = handler,
    .sa_flags = SA_SIGINFO
    // SA_SIGINFO, serve per richiedere al gestore di avere la struttura
    // completa che altrimenti sarebbe pesante se non necessaria
    // *[da scrivere meglio in italiano]*
};
sigaction(SIGTERM, &new, NULL);
```

E sostituendo a (2)

```
void action(int sig, siginfo_t *info, void *arg) {
    printf("No, non voglio morire %d\n", info->si_pid);
}
```

Si ottiene, con l'utilizzo di `sigaction()`, una realizzazione più moderna.

13 CONCORRENZA (04/11/2019)

13.1 Programmazione concorrente

Riprendiamo i contenuti di programmazione concorrente della volta scorsa, dove abbiamo incontrato vari "paradigmi" di implementazione di una chat tra processi

1. Ad eventi, basandosi su syscall come `poll()` e `select()`
2. Concorrente, con l'uso di due semplici `write()` su thread diversi

Come abbiamo già avuto modo di accennare, il paradigma di programmazione concorrente non è privo di svantaggi. Anche operazioni semplici come

```
int saldo;

int op_bancaria(int x) {
    saldo = saldo + x;
}
```

possono rilevarsi problematiche (in questo caso vediamo che `saldo = saldo + x` non è atomica).

13.1.1 Processo

Vogliamo ora introdurre alcune nozioni basiche e teoriche che ci saranno utili nel seguito quando discuteremo di sistemi operativi in maniera più approfondita. Partiamo con la definizione di processo.

Un **processo** è l'atto di esecuzione di un programma. È un'entità dinamica (che può mutare stato nel tempo) che si contrappone a quella di programma, entità statica che specifica solo una sequenza di istruzioni, senza determinare la loro distribuzione temporale.

Assioma di *finite progress* Ogni processo viene eseguito ad una velocità *finita*, ma sconosciuta.

13.1.2 Stato di un processo

Lo stato di un processo dipende da quello del program counter, da quello del processore e da quello della memoria. Esso può assumere i seguenti stati

- Ready: pronto per iniziare l'esecuzione, o riprenderla se l'aveva precedentemente interrotta (ciò non vuol necessariamente dire che al momento sia in esecuzione)
- Running: in esecuzione (nota: può accadere che un processo running ritorni ready)
- Blocked: in attesa di un eventuale riavvio (o terminazione)
- Ended: terminato definitivamente

13.1.3 Parallelismo

Si danno due principali tipologie di parallelismo:

- Parallelismo "reale", dovuto a più unità di processamento parallele che eseguono effettivamente codice in maniera simultanea
- Parallelismo "apparente", dovuto a una singola unità di processamento (core, ...) che opera un'alternanza (**interleaving**) di diversi compiti

È possibile domandarsi se lo studio di una di queste tecniche possa essere ricondotto a quello dell'altra, cioè se esse siano equivalenti sotto certi aspetti. Cerchiamo di darne una spiegazione con il seguente esempio:

```
void aggiorna(int val) {
    LD R0, saldo
    // (1)
    ADD R0, val
    // (2)
    STO R0, saldo
}
```

[Il paragrafo sottostante presenta serie inaccuratezze concettuali - ovvero?] In un contesto di parallelismo reale (es. tra più core di una stessa CPU) un altro processo può accedere contemporaneamente alla stessa locazione di memoria (`saldo`) per poi effettuare operazioni diverse e salvare il risultato. In questo modo si sovrascrive il risultato dell'operazione che ha salvato per prima il risultato.

Viceversa, in un contesto di parallelismo apparente (es. un solo core), se l'operazione non è atomica, si rischia di mettere in pausa il processo prima che venga eseguita la `ADD`. Se ciò accade, e un altro processo accede alla stessa locazione del saldo, leggerà il risultato non aggiornato.

Diremo che un programma (o una sequenza di processi), è soggetto a ***race condition*** quando il risultato della sua esecuzione dipende dalla distribuzione temporale delle sue istruzioni, (o delle sue esecuzioni).

Sezione critica = porzione di programma che accede a una o più risorse condivise tra più processi.

13.1.4 Notazioni

Iniziamo ad introdurre un paio di notazioni utili ad esprimere l'esecuzione concorrente di più processi

```
P : process {

}

Q : process {
```

```
}
```

Con questa prima scrittura, indichiamo che i processi `P` e `Q` sono in esecuzione concorrente. All'interno di `{ ... }` sarà possibile specificare il codice da loro eseguito.

Un altro tipo di notazione per esprimere lo stesso concetto è

```
inst1

inst2a // inst2b

inst3
```

Qui intendiamo l'esecuzione di `inst1`, seguita dall'esecuzione *concorrente* di `inst2a` e `inst2b`, seguita (solo quando sia `inst2a` che `inst2b` avranno terminato) da quella di `inst3`. All'interno di questo corso questa notazione sarà meno utilizzata.

Osservazione: non esiste alcun modo deterministico per risolvere problemi di programmazione concorrente.

13.1.5 Operazioni atomiche

Un'*operazione atomica* è un'istruzione che può essere eseguita completamente, o niente affatto. Ad es. l'assegnazione di una costante può non essere atomica su architettetture RISC che eseguono linguaggi d'alto livello compilati; questo perché potrebbe non esistere nessuna operazione assembly che in un unico ciclo di clock copi un valore in una locazione di memoria.

Nel corso dei nostri studi considereremo atomiche le seguenti operazioni:

- Assegnamento di una costante
- Lettura di una variabile
- Eventuali operazioni comode per specifiche architetture

Data una generica istruzione `i`, la notazione `< i >` esprime che `i` è considerata atomica

13.1.6 Casi di concorrenza

La concorrenza di un sistema può manifestarsi nei seguenti casi:

- Sistemi multiprocesso: sebbene un processo singolo pensi di operare individualmente, esso sta condividendo risorse con tutti gli altri processi presenti nel sistema
- IPC (Inter Process Communication): più processi che si scambiano vicendevolmente messaggi
- Sistema multiprogramma: all'interno del processo stesso compare la concorrenza *[specificare il significato]*

13.1.7 Proprietà dei programmi

Definiamo *proprietà* di un programma un attributo che persiste a prescindere da un suo possibile storico di esecuzione. Possiamo classificare tali proprietà in

- *Safety*: reso informalmente come *nothing bad happens*, esprime il fatto che il programma non commette azioni indesiderate
- *Liveness*: informalmente *something eventually happens*, esprime la mancanza di attese infinite. Garantisce la terminazione di un programma

13.1.7.1 Sistemi distribuiti e problema del consenso

[questo paragrafo necessita di alcune sistemazioni per migliorarne la comprensione]

Possiamo illustrare un esempio di proprietà dalla teoria dei sistemi distribuiti.

Nei sistemi distribuiti può capitare che un processo termini in maniera anomala, tuttavia il risultato finale deve essere comunque raggiunto. Gli altri processi devono proporre un risultato e alla fine tutti devono convergere allo stesso valore: tale condizione viene detta *consenso*. Questo tipo di problema non si pone qualora si parli di un sistema non distribuito

Assunto del *silent failure* Se il processo sbaglia termina senza restituire nulla.

Queste sono le proprietà del problema del consenso:

1. safety1: tutti i processi corretti ritornano lo stesso valore
2. safety2: i processi ritornano un valore proposto
3. liveness: tutti i processi corretti *eventually* decidono

senza le quali si ottengono i seguenti effetti

1. Senza la prima: `consensus(x) = x`
2. Senza la seconda: `consensus(x) = 42`
3. Senza la terza: `consensus(x) = while true`

13.1.8 Deadlock e starvation

Due termini molto importanti nel contesto della concorrenza sono *deadlock* e *starvation*. Definiamoli

Deadlock È una condizione indesiderata che vede due o più processi in un'attesa circolare. Più in particolare, se consideriamo due processi `P1` e `P2`, e supponiamo che `P1` detenga una risorsa `R1` e `P2` una risorsa `R2`, la condizione di deadlock si verifica se

- `P1` è in attesa di `R2` e `P2` in attesa di `ucidi-so.shtmlR1`
- `P1` non può rilasciare `R1` prima di aver acquisito `R2`
- `P2` non può rilasciare `R2` prima di aver acquisito `R1`

Starvation: Sia `A` un processo e siano `R1`, ..., `RN` altri processi di priorità più alta di `A`. Se lo scheduler dei processi alloca la CPU prima ai processi con priorità più elevata e alla terminazione di un processo Ri ne viene creato un altro di priorità più elevata di A, allora A non verrà mai eseguito. Si dice che A è in *starvation*.

13.2 Linguaggi di shell scripting

Nella seconda parte di questa lezione affrontiamo nozioni base di linguaggi di shell scripting.

Come abbiamo già accennato in lezioni precedenti, una *shell* è un programma che offre accesso ai servizi del sistema operativo; quelle di cui ci interesseremo noi sono shell testuali (in contrapposizione a quelle grafiche) che si interfacciano all'utente tramite una CLI (*command line interface*). Tale programma viene detto shell (*guscio* in inglese) perché è il livello software più esterno di un sistema operativo. Come vedremo, una shell può agire sia come interpete di comandi che di script scritti in linguaggi appositi.

Le prime shell ad apparire in ordine storico furono `sh` (Bourne shell) e `csh`, che hanno influenzato molte delle più popolari di oggi, tra cui `bash`, `ksh`, `zsh` ecc.

I primi caratteri di uno script shell (detti shebang) sono solitamente formati da

```
#!<eseguibile>
```

dove `<eseguibile>` sarà un comando un grado di interpretare il sorgente corrente. Osserviamo che `<eseguibile>` può essere il nome di un programma qualunque, e molto spesso è rimpiazzato da quello di linguaggi di programmazione (es. Python).

Tanto per mostrare la versatilità di questo metodo, un esempio un tantino patologico è dato da

13.2.0.1 Esempio di script shell

```
#!/bin/cat

ls -l /proc/$$/exe
# Qui sotto sarà possibile scrivere qualunque cosa
```

Un altro dallo stile più tradizionale è invece

13.2.0.2 Esempio di script shell

```
#!/bin/bash

ls -l /proc/$$/exe
```

13.2.1 Variabili

Si danno due tipi principali di variabili nei ling. di shell

- Locali, es. `numero = 42; echo $numero`
- D'ambiente (*environment variables*), visibili da più processi e rese tali dalla keyword `export`

Un comando utile per mostrare tutte le variabili d'ambiente è `printenv`.

Date delle variabili, possiamo sostituirle automaticamente all'interno di doppi apici, come in

```
echo "$a, $b, ..., $z"
```

Gli apici singoli hanno un effetto diverso

```
echo '$a, $b, ..., $z'
```

in questo caso verrà stampato letteralmente `$a, $b, ..., $z`.

Per ottenere i valori ricevuti dalla linea di comando è possibile leggere le variabili `$1`, `$2`, ..., `$N`, dove `N` è il numero massimo di argomenti passati. La variabile `$*` contiene un'espansione di `$1`, ..., `$N` separate da spazi.

13.2.2 Cattura dell'output

Possiamo eseguire un programma e ricevere il suo output, da memorizzare o da passare come argomento di un altro programma, con le seguenti scritture

- `sh cmd1 $(cmd2)`
- "`sh cmd1 \`cmd2`"

Ad esempio

```
cat $(ls *.sh)
cat `ls *.sh`
```

13.2.3 Intermezzo: glob patterns

L'esempio appena considerato ci permette di dare un accenno ai cosiddetti glob patterns, una particolare sintassi dei ling. di shell per specificare succintamente un insieme di file. Vediamo ad es. che

- `*.sh` è sostituito da tutti i file terminanti in `.sh`
- `ciao?.txt` è sostituito da tutti i file `ciao?.txt`, dove `?` è *un solo* carattere qualsiasi
- `ciao[0-9].txt` è sostituito da tutti i file `ciaoX.txt`, dove `X` è un valore compreso tra `0` e `9`

13.2.4 Costrutti condizionali

Sono disponibili anche i classici costrutti condizionali, quali `if` e `case`, e quelli iterativi come `for` e `while`. Diamone un esempio concreto

13.2.4.1 Esempio di script shell

```
#!/bin/bash

echo uno

if gcc prog.c then
    echo compilazione okay
else
    echo compilazione ko
    exit 2
fi

case $1 in
    y*) echo yes
        ;;
    no) echo nono
        ;;
    *) echo non capisco
        ;;
esac

./a.out
```

13.2.4.2 Cortocircuitazione degli operatori logici

Anche nei linguaggi di scripting shell gli operatori logici seguono la regola della cortocircuitazione. Ad esempio

```
gcc programma.c || echo "programma errato"
```

13.2.5 Utilità di sistema

Vediamo ora un insieme ristretto di programmi, dette utilità di sistema, molto utili nella programmazione di script shell.

13.2.5.1 `grep`

Il comando `grep` si usa per cercare occorrenze di stringhe all'interno di file di testo (o canale di lettura) con il formalismo delle espressioni regolari.

L'invocazione più basilare del comando è data da `grep <testo> <percorso_file>`, ad esempio

```
grep ciao /tmp/myfile
```

È possibile utilizzare il flag `-q` per invocare `grep` in modalità silenziosa. Verrà ritornato il valore `0` in caso di corrispondenze.

13.2.5.2 `find`

Il comando `find` è uno strumento utile a cercare file, all'interno di un percorso specificato, mediante opportuni filtri.

13.2.5.3 `sort`

Il comando `sort` è utilizzato per ordinare il contenuto di file di testo, linea per linea.

13.2.5.4 `expr`

Il comando `expr` valuta un'espressione e scrive il suo risultato sullo standard output.

13.2.5.5 `sed`

`sed` è un editor di testo guidato da script, da utilizzare per modificare file come, ad esempio, la sostituzione di un termine a tutte le sue occorrenze.

14 CONTROLLO DELLA CONCORRENZA (06/11/2019)

La volta precedente abbiamo accennato a problemi inerenti la programmazione concorrente quali race condition, deadlock e starvation. In questa lezione forniremo strumenti utili a risolverli.

14.1 Tassonomia dei computer paralleli

Prima di addentrarci nell'argomento soffermiamoci su due punti in particolare.

Per prima cosa, ricordiamo che nel contesto della programmazione concorrente possiamo considerare due modelli di memoria

- Condivisa: adottata nella comunicazione tra più thread di uno stesso processo
- Separata: adottata nella comunicazione tra più processi

In secondo luogo, ricordiamo che viene definita una tassonomia dei computer paralleli (dovuta a Flynn), che classifica i calcolatori sulla base di due *flussi*: quello sulle istruzioni e quello sui dati

- **SISD** (Single Instruction, Single Data): macchina di Von Neumann classicamente intesa
- **SIMD** (Single Instruction, Multiple Data): macchine dotate di una sola unità di controllo (es. un solo program counter) capaci di elaborare più flussi di dati simultaneamente (es. molteplici data path in una ALU, processamento di materiale audio-visivo, ...)
- **MISD** (Multiple Instruction, Single Data): macchine dove più flussi di istruzioni operano sullo stesso dato. Gli esempi di tale topologia sono rari, benché alcuni autori considerano sistemi con pipeline di tipo MISD
- **MIMD** (Multiple Instruction, Multiple Data): più unità di processamento che operano individualmente su più flussi di dati (es. CPU multicore a memoria condivisa, o più macchine che comunicano tramite IPC)

14.1.1 Sezioni critiche

Perché si abbia un problema di *sezione critica* devono valere le seguenti proprietà:

1. **Mutua esclusione**: solo un processo alla volta può eseguire codice della sezione critica
2. **No deadlock**: i processi non si devono bloccare a vicenda tentando di entrare nella sezione critica
3. **No delay non necessari**: un processo fuori dalla sezione critica non deve ritardare l'accesso di altri processi
4. **No starvation**: un processo che richiede di entrare nella sezione critica prima o poi deve riuscirci

Una prima idea di risoluzione del problema di race condition sarebbe individuare una sezione critica e renderla atomica. Fare cioè qualcosa come, nell'esempio dell'operazione bancaria della volta scorsa

```
[enter cs]
    load r0 saldo
    add r0 importo
    store r0 saldo
[exit cs]
```

I progettisti degli anni '60, che si posero per primi la sfida di risolvere tali problemi, si domandarono se non fosse possibile introdurre un modello generale dei processi utile ad affrontare la problematica.

Una possibile proposta di modello può essere data da

```
process P:
    while true:
        [cs enter]
            /* codice critico */
        [cs exit]

        /* codice non critico */

process Q:
    while true:
        [cs enter]
            /* codice critico */
        [cs exit]
        /* codice non critico */
```

sufficientemente espressivo da risolvere la problematica di deadlock.

14.1.2 Algoritmo di Dekker

L'algoritmo di Dekker propone una soluzione al problema della mutua esclusione nella programmazione concorrente. Cercheremo di giungere alla versione definitiva passando prima per quattro realizzazioni inesatte, utili ad illustrare le problematiche incontrate.

14.1.2.1 Prima realizzazione

Consideriamo questa prima versione

```
turn = P

process P:
    while true:
        while (turn != P)
```

```
            ;

        /* codice critico */
        turn = Q
        /* codice non critico */

process Q:
    while true:
        while (turn != Q)
            ;

        /* codice critico */
        turn = P
        /* codice non critico */
```

Osserviamo che tale realizzazione

- Risolve il problema della mutua esclusione
- Risolve il problema del deadlock
- Risolve il problema della starvation

ma non risolve il problema dei ritardi non necessari. Supponiamo infatti che `P` sia un processo veloce, e `Q` un processo molto lento. Il "ritmo" di esecuzione di `P` sarà allora rallentato. Infatti, quando `P` esce dalla sezione critica e poi tenta di rientrarci, se `Q` è molto lento `P` è forzato ad aspettare che `Q` entra ed esca dalla sezione critica.

14.1.2.2 Seconda realizzazione

Per rimediare alle imperfezioni del metodo precedente, possiamo introdurre due parametri (flag) booleani che indicano quale processo si trova correntemente nella sezione critica.

```
inP = false
inQ = false

process P:
    while true:
        while (inQ)
            ;

        inP = true /* (1) */

        /* codice critico */
        inP = false
        /* codice non critico */

process Q:
```

```
    while true:
        while (inP)
            ;

        inQ = true /* (2) */

        /* codice critico */
        inQ = false
        /* codice non critico */
```

Notiamo subito che anche questo secondo tentativo non è privo di difetti. Considerato infatti che `inP = false` e `inQ = false` inizialmente, è molto probabile che già dalla prima esecuzione entrambi i processi entrino nella sezione critica, violando il primo requisito.

14.1.2.3 Terza realizzazione

Tentiamo una variazione della versione precedente spostando il settaggio dei flag a `true` prima dell'entrata nei cicli. *[verificare]*

```
inP = false
inQ = false

process P:
    while true:
        inP = true

        while (inQ) /* (1) */
            ;

        /* codice critico */
        inP = false
        /* codice non critico */

process Q:
    while true:
        inQ = true

        while (inP) /* (2) */
            ;

        /* codice critico */
        inQ = false
        /* codice non critico */
```

Questa versione, anch'essa inesatta, non previene invece il sorgere di deadlock. Se infatti i due processi si trovano rispettivamente ai punti `(1)` e `(2)` del sorgente

entreranno entrambi in un ciclo infinito, poiché nessuno dei due setterà la propria flag a `false` se non avrà prima eseguito la sezione critica.

14.1.2.4 Quarta realizzazione

In questa quarta implementazione alteriamo la semantica dei parametri `inP` e `inQ`, rinominandoli opportunamente in `needP` e `needQ`, esprimendo l'intenzione di entrare nella sezione critica.

```
turn = P

process P:
    while true:
        needP = true

        while (needQ)
            needP = false;
            /* Aspetta un po' */
            needP = true;

        /* Codice critico */
        needP = false
        /* Codice non critico */

process Q:
    while true:
        needQ = true

        while (needP)
            (1)
            needQ = false;
            /* aspetta un po' */
            needQ = true;

        /* codice critico */
        needQ = false
        /* codice non critico */
```

L'errore commesso in questa implementazione, che conclude gli esempi fallaci mostrando una starvation, è facilmente individuabile immaginando i processi come perfettamente sincronizzati (o in fase) in esecuzione su due processori concorrenti: all'arrivo di entrambi a `(1)`, la riga immediatamente successiva sarà eseguita sia da `P` sia da Q, e dopo un'uguale attesa contemporanea continueranno a fermarsi a vicenda. Entrambi, quindi saranno in una situazione di starvation.

14.1.2.5 Implementazione definitiva

```
bool needP, needQ;
turn = P;

process P:
    while true:
        needP = true

        while (needQ)
            if (turn != P)
                needP = false
                while (turn != P)
                    ;
                needP = true

        /* codice critico */

        needP = false
        turn = Q
        /* codice non critico */

process Q:
    while true:
        needQ = true

        (2)
        while (needP)
            if (turn != Q)
                needQ = false

                while (turn != Q)
                    ;
                needQ = true

        /* codice critico */
        needQ = false
        turn = P
        /* codice non critico */
```

Nella prossima lezione affronteremo una dimostrazione formale della correttezza dell'algoritmo di Dekker.

14.1.3 Esempio di programma C

Alla pagina del professor Davoli è possibile ottenere un'implementazione C di programmi che illustrano le tecniche di gestione della concorrenza appena illustrate.

14.2 Linguaggi di shell scripting

Come di consueto, cambiamo argomento nella seconda parte di lezione tornando ad occuparci di shell scripting, lasciato in sospeso la volta precedente.

14.2.1 Costrutto `for`

Un costrutto shell non esplorato la scorsa volta è il `for`, che possiamo illustrare con un semplice esempio

14.2.1.1 Esempio di script shell

```
for i in $ls$ do
    echo file: $i
done
```

14.2.2 Directory utente

Notiamo che ogni directory utente possiede un numero di script avviati in diversi momenti d'esecuzione della shell

- `.bashrc`, l'insieme di istruzioni avviate all'avvio di una shell interattiva
- `.bash_logout`, l'insieme di istruzioni eseguite all'uscita di una shell *di login*
- `.inputrc`, istruzioni destinate all'utility `readline`

Eseguire tali script da una shell in riga di comando (es. `./nomescript`) non produce alcun effetto, in quanto lo script è letto ed eseguito in una *sottoshell*, distrutta al termine dell'esecuzione. Per ovviare al problema è possibile utilizzare il comando `source`, o il suo alias `.`, come `source <nomescript>` o `. <nomescript>`.

14.2.3 Comandi di utilità

In questa sezione estendiamo il glossario di comandi utilità iniziato la volta scorsa.

14.2.3.1 `set`

Il comando builtin `set` permette di abilitare (`-`) e disabilitare (`+`) determinati flag per le opzioni della shell, quali ad esempio

- `set -f` disabilita il globbing
- `set -x` attiva la stampa dei comandi, e relativi parametri, nel momento in cui vengono eseguiti
- `set -e` abilita l'uscita immediata se un'istruzione ritorna un valore diverso da `0`

Invocato senza parametri, `set` mostra tutti i valori correntemente in uso. Osserviamo che esso è disponibile in Bash, ma non in tutte le shell.

14.2.3.2 `test`

`test` fornisce le classiche operazioni di confronto relazionali (es. `==`, `<`, `>`, ...), logico (es. `&&`, `||`, ...), di controllo su file ecc.. Osserviamo che può essere utilizzato in forma più concisa con `[`.

Vediamone un semplice esempio in

14.2.3.3 Esempio di script shell

```
#!/bin/bash

i=0

while [ $i -ne 10 ] do
    echo $i
    sleep 1

    i=‘expr $1 + 1‘
    # In alternativa: ((i = i + 1))
done
```

Osserviamo che `[` altro non è che il nome di un programma (solitamente situato in `/usr/bin/[`) che svolge la stessa funzione di `test`, ma che richiede `]` come ultimo argomento (il quale è giusto un carattere con nulla di speciale). Ciò rende molto chiara la necessità degli spazi tra `[` e i suoi argomenti.

14.2.3.4 `which`

`which` restituisce il percorso completo del nome di comando ricevuto come argomento, ad es.

```
$ which cat
> /usr/bin/cat
```

14.2.3.5 `cat`

[spostare in sezione più adatta, non relativa a `cat`] Illustriamo un utilizzo peculiare del comando `cat`

```
#!/bin/bash

cat > lettera << STOP_HERE

ciao $1, ti trovo in forma oggi!
hai vinto 1000 euro
STOP_HERE
```

Dentro al file `lettera` viene scritto ciò che compare dopo il comando `cat`fino al tag `STOP_HERE`. Questo è un esempio di *here is document* : si può ***[c'è su wikipedia, completare]***

15 CONTROLLO DELLA CONCORRENZA - SEMAFORI (11/11/2019)

15.1 Algoritmo di Dekker

Ci siamo lasciati l'ultima volta con la versione finale dell'algoritmo di Dekker. In questa lezione affrontiamo la dimostrazione della sua correttezza.

```
bool needP, needQ;
turn = P;

process P:
    while true:
        needP = true

        while (needQ)
            if (turn != P)
                needP = false
                while (turn != P)
                    ;
                needP = true

        /* codice critico */
        needP = false
        turn = Q
        /* codice non critico */

process Q:
    while true:
        needQ = true
(2)
        while (needP)
            if (turn != Q)
                needQ = false

                while (turn != Q)
                    ;
                needQ = true

        /* codice critico */
        needQ = false
        turn = P
        /* codice non critico */
```

Ricordiamo che per dimostrare la correttezza di questo algoritmo bisogna poter soddisfare i quattro vincoli dovuti alle sezioni critiche.

15.1.1 Dimostrazione

15.1.1.1 Mutua esclusione

Procediamo per assurdo, supponendo che entrambi i processi stiano lavorando nella sezione critica. Usciti dalla sezione critica abbiamo che sia `needP` che `needQ` sono `true`. Supponiamo che sia entrato prima `P` nella sezione critica, senza perdita di generalità. A questo punto, visto che abbiamo supposto che `Q` sia nella regione critica deve aver necessariamente passato la guardia del while, e quindi needP deve essere falso, il che è assurdo poichè abbiamo visto che `needP` è vero.

15.1.1.2 Assenza di deadlock

Procediamo per assurdo, supponendo che né `P` né `Q` possano entrare nella sezione critica.

1. `P` e `Q` devono essere bloccati nel primo `while`
2. esiste un istante t dopo il quale `needp` e `needq` sono sempre `true`
3. supponiamo (senza perdita di generalità) che all'istante t, `turn = Q`
4. l'unica modifica a `turn` può avvenire solo quando `Q` entra nella sezione critica
5. dopo t, `turn` resterà sempre uguale a `Q`
6. `P` entra nel primo ciclo, e mette `needp = false`
7. ASSURDO!

15.1.1.3 Assenza di delay non necessari

Se `Q` non sta eseguendo codice critico si ha `needQ = false`. Allora `P` può entrare nella sezione critica.

15.1.1.4 No starvation

Supponiamo che `P` richieda di accedere alla sezione critica. Si hanno due casi: se `Q` non sta eseguendo codice critico allora `P` entra subito. Altrimenti, non appena `Q` uscirà dalla sezione critica si avrà che è il turno di `P`.

15.2 Algoritmo di Peterson

In questo algoritmo, appena inizia l'esecuzione di `P`, viene dichiarato il turno di `Q`, e viceversa.

```
bool needP, needQ;
turn = P;

process P:
    while true:
        needP = true
        turn = Q
```

```
        while (needQ && turn != P)
            ;

        /* codice critico */
        needQ = false
        /* codice non critico */
```

`processQ: <duale>`

L'algoritmo quindi stabilisce che `P` entri nella sezione critica solo se `Q` non ha bisogno di entrarci o ha dato la precedenza a `P`. Se ci sono `N` processi, Peterson fa una sorta di "torneo" con tutti gli altri processi. Per passare allo stadio successivo bisogna aver "vinto" contro un altro processo. Se un processo ha passato `N` stadi accede alla regione critica.

Ecco il codice di Peterson multiprocesso

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>

#define N 5

volatile int stage[N];
volatile int last[N];

void *proc(void *arg) {
    uintptr_t i = (uintptr_t) arg;
    int j, k;

    for(;;) {
            for (int j=0; j < N; j++) {
                    stage[i] = j;
                    last[j] = i;

                    for (k=0; k < N; k++) {
                            if (i != k)
                                    while (stage[k] >= stage[i] && last[j] == i);
                    }
            }
    }

    printf("%din ", i); fflush(stdout);
```

```
    usleep((rand() % 100000) * (rand() % 10));

    printf("%dout\n", i);
    stage[i] = 0;
    usleep((rand() % 100000) * (rand() % 10));
}


int main(int argc, char * argv[]) {
        uintptr_t i;
        srandom(time(NULL));
        pthread_t t[N];

        for (i = 0;  i < N; i++)
                pthread_create(t + i, NULL, proc, (void *) i);

        pause();
        return 0;
}
```

Questi algoritmi ci insegnano che possiamo gestire una sezione critica usando solo l'atomicità dell'assegnamento. È da osservare che questo è il problema alla base della programmazione concorrente, ed è abbastanza complesso. Inoltre, bisogna osservare che l'uso continuo di procedure di attesa, detto ***busy waiting***, è un grosso spreco di risorse. Occorre, dunque, implementare un modo di gestire la concorrenza più efficiente, ovviamente basandosi sui risultati appena ottenuti.

Si può creare un'istruzione macchina capace di gestire la mutua esclusione, che tuttavia non è completamente corretta dal punto di vista della risoluzione del problema della sezione critica.

15.2.1 Istruzione Test&Set

Ci si può chiedere se non si possa affidare la risoluzione dei problemi di concorrenza con un'unica istruzione. Il meccanismo dell'istruzione Test&Set è proprio questo: visto ad alto livello, il primo processo che raggiunge la sezione critica esprime, con una "risorsa condivisa", il suo stato. La risorsa condivisa non è rilasciata finché il primo processo non ha terminato la sezione critica. Una possibile definizione dell'istruzione può essere questa:

Test&Set: `TS(x, y) <y = x, x = 1>` Le due operazioni vengono eseguite sequenzialmente, ma siccome deve essere un'istruzione atomica o vengono eseguite entrambe o nessuna delle due.

```
g = 0;

process P:
    while true:
```

```
        do
            TS(g, l) // l è passata per riferimento
        while (l == 1);

        /* codice critico */
        g = 0
        /* codice non critico */

process Q:
    /* Uguale */
```

Questa istruzione copia il valore della variabile globale nella cella di una variabile locale per gestire lo stato della sezione critica. In questo modo, ogni processo può utilizzare la propria variabile locale per controllare il permesso di esecuzione. Inizialmente si ha `g = 0`, ed indica cioè che la CS è libera al momento. `P` decide di entrare nella CS; cambia quindi il valore di `g`. A questo punto esegue il codice della CS. Al termine di ciò, il valore di `g` viene resettato a zero. In sostanza, il processo che riesce per primo a settare la variabile globale ad 1 ottiene il permesso di entrare nella CS. Questa variabile globale è quindi solitamente chiamata *lock*.

Questo procedimento evita mutua esclusione, deadlock e delay non necessari, ma non riesce a risolvere la problematica di starvation. Infatti, con una combinazione particolarmente sfortunata di tempi di esecuzione di `P` e `Q`, il primo processo potrebbe completare la sezione critica e richiedere nuovamente la "precedenza" prima che l'altro processo abbia la possibilità di fare qualcosa. Senza un sistema di turni, infatti, questo risultato è inevitabile *[nota dello scrittore]*. Esistono molteplici implementazioni di Test&Set. Infatti, le necessità che questa funzione deve sostenere sono perlomeno due: la dipendenza dalla variabile globale e l'espressività *[appunto dello scrittore, NON esplicitamente detto dal prof]*.

15.3 Semafori

Un semaforo (*semaphore* in inglese) è un tipo di dato astratto usato nella programmazione concorrente munito delle seguenti operazioni

- Inizializzazione, es. `semaphore(int init)`
- `P()` (dall'olandese *proberen*, provare), detta anche `wait()`. Viene invocata per *attendere* un segnale, quali il verificarsi di un evento o il rilascio di una risorsa
- `V()` (dall'olandese *verhogen*, incrementare), detta anche `signal()`. Viene invocata per *emettere* un segnale, quali il verificarsi di un evento o il rilascio di una risorsa

Possiamo abbozzarne una realizzazione nel paradigma OOP

```
class semaphore {
    private int val;
```

```
    semaphore (int init): val = init;
    void P (void);
    void V (void);
}
```

Descrizione informale dei due metodi:

- `P()` permette al processo chiamante di dichiarare l'entrata all'interno di una risorsa condivisa o, eventualmente, di mettersi in coda per farlo.
  - se `val` è maggiore di zero, il processo inizia ad eseguire il codice della CS
  - se `val` è minore o uguale a zero, il processo entra nella coda di attesa
- `V()` incrementa `val` di 1. Permette quindi l'operazione inversa, ovvero segnala il rilascio della risorsa da parte del chiamante, ed eventualmente estrae un processo dalla coda di attesa

Prima di illustrare un esempio di utilizzo, osserviamo che se n_P è il numero di operazioni di tipo P completate ed n_V quelle del tipo V completate, vale la seguente invariante:

$$n_P \leq n_V + init \tag{1}$$

o, equivalentemente:

$$init + (n_V - n_P) \geq 0 \tag{2}$$

Si osservi che l'invariante è una proprietà di *safety* dato che non riguarda in nessun modo la temporizzazione delle esecuzioni dei processi.

Un semaforo può essere utilizzato nel modo seguente

```
s = new Semaphore(1);

process p1, p2, p3, ...:

while true {
    s.P();

    [cs enter]
    foo();
    [cs exit]

    s.V();

    bar();
}
```

Supponiamo che il semaforo abbia valore iniziale `1` e che il ciclo venga eseguito indefinitamente. Il primo processo può dunque effettuare `s.P()` e il conseguente `foo()`. Se, ora, un altro processo tenta di effettuare la `s.P()` non verrebbe rispettata l'invariante, quindi dovrà attendere che il processo precedente effettui `s.V()`

```
semaphore s0(0)
semaphore s1(0)

process Q:
    <print a>
    s0.V()
    s1.P()
    <print b>

process T:
    <print 1>
    s1.V()
    s0.P()
    <print 2>
```

Supponiamo che non conti l'ordine con cui vengono stampati `a` ed `1`, ma che volessimo stampare `b` e `2` (anche qui l'ordine non conta) solo dopo che sia `a` che `1` siano già stati stampati. Allora basta introdurre i semafori come descritto nel codice per sincronizzare i due processi.

15.3.1 Implementazione dei semafori

Ci poniamo ora il problema di fornire una realizzazione dei semafori, che tenga conto dei dettagli implementativi su sistemi monoprocessore e multiprocessore.

[il paragrafo necessita di miglioramenti - integrare con la parte sulle slide]

15.3.1.1 Sistema monoprocessore

Se abbiamo una macchina monoprocessore e disattiviamo gli interrupt, ciò che accade è che la sequenza di programmi in esecuzione procederà nell'ordine seguito fino a quel momento. Si potrebbero implementare i semafori a livello kernel, in modo che i processi utente possano chiedere al kernel un accesso alla sezione critica. Per bloccare un processo, basta toglierlo dalla coda dei processi ready, in modo che lo scheduler non lo scelga.

Si può usare la mutua esclusione degli interrupt per implementare i semafori *[verificare]*. Per le problematiche più di alto livello si userà l'implementazione dei semafori del kernel.

```
class semaphore {
    int val;
```

```
    queue q; /* coda dei processi bloccati su questo semaforo */
    semaphore(x) : val = x;

    P() {
        cs_enter() /* disabilita gli interrupt */
        if val == 0:
            q.enqueue(getpid())
            suspend(getpid()) /* toglie dalla coda dei processi ready */
        else:
            val--
        cs_exit() /* riabilita gli interrupt */
    }

    V() {
        cs_enter() /* disabilita gli interrupt */
        if q.empty(): /* non c'è nessun processo bloccato */
            val++
        else:
            resume(q.dequeue()) /* reinserisce nella coda dei processi ready */
        cs_exit() /* riabilita gli interrupt */
    }
}
```

Fairness: il processo che viene fatto ripartire è quello che stava attendendo da più tempo (proprio perché `q` è una coda, utilizza cioè la modalità FIFO).

Tutti i semafori mostrati d'ora in poi all'interno del corso saranno *fair*, a meno di non dichiarare espressamente il contrario.

15.3.1.2 Sistema multiprocessore

In questi casi `cs_enter()` e `cs_exit()` vengono implementate attraverso la Test&Set poiché siccome ciascun core ha il suo gestore degli interrupt disabilitarli significherebbe farlo solo per il core sul quale il processo corrente è attualmente in esecuzione mentre per gli altri core non cambierebbe nulla. Vengono consumati dei cicli macchina per impostare la mutua esclusione ma poi libera la CPU.

15.4 Problemi classici

Vogliamo ora affrontare un numero di problemi di concorrenza classici di una certa rilevanza storica

- Problema del produttore-consumatore
- Problema del buffer limitato
- Problema dei lettori-scrittori
- Problema dei filosofi a cena (detto anche "dei cinque filosofi")

15.4.1 Problema del produttore-consumatore

Si danno un processo produttore `producer` e un processo consumatore `consumer`, tale per cui `producer` vuole trasferire valori a `consumer`. La comunicazione avviene attraverso una *singola* variabile condivisa. Il problema principale risiede nel fatto che `producer` deve aspettare che `consumer` abbia effettivamente utilizzato un risultato prima di crearne uno nuovo, altrimenti il vecchio valore verrebbe sovrascritto. Pertanto i due processi devono essere sincronizzati.

15.4.1.1 Soluzione mediante semafori

```
semaphore canWrite(1)    /* all'inizio è possibile scrivere sul buffer */
semaphore canRead(0)    /* all'inizio non si può leggere dal buffer */
int buf;

process producer:
    while true:
        int val = generate()
        canWrite.P()
        buf = val
        canRead.V()

process consumer:
    while true:
        int val
        /* Se arrivasse prima il consumatore si fermerebbe qui, perchè  */
        /* il valore del semaforo sarebbe 0 e non potrebbe decrementarlo */
        canRead.P()
        val = buf
        canWrite.V()
        process(val)
```

15.4.2 Problema del buffer limitato

Questo problema è simile a quello del produttore-consumatore, ma invece di avere un singolo elemento di scambio ce ne sono una quantità limitata `N`. Proviamo a scriverne un'implementazione modificando il codice del produttore-consumatore in maniera opportuna

```
semaphore canWrite(N)    /* all'inizio è possibile scrivere sul buffer */
semaphore canRead(0)    /* all'inizio non si può leggere dal buffer */
queue buf;

process producer:
    while true:
        int val = generate()
        canWrite.P()
```

```
        buf.enqueue(val)
        canRead.V()

process consumer:
    while true:
        canRead.P()
        int val = buf.dequeue()
        canWrite.V()
        process(val)
```

L'utilizzo delle primitive `enqueue()` e `dequeue()` dell'esempio appena riportato presenta un difetto, in quanto le due chiamate sono prive di atomicità: questo rende il programma facilmente soggetto a race condition. L'esempio seguente mira a correggere questa imperfezione con l'utilizzo del meccanismo di blocco `mutex()`, implementato qui come un semaforo binario.

[eventualmente integrare con le slide sulla concorrenza da 119 in poi: https://www.cs.unibo.it/~renzo/so/old/lucidi/so-02-concorrenza-4p.pdf]

```
semaphore canWrite(N)    /* all'inizio è possibile scrivere sul buffer */
semaphore canRead(0)    /* all'inizio non si può leggere dal buffer */
semaphore mutex(1)       /* protegge le operazioni sulla coda */
queue buf;

process producer:
    while true:
        int val = generate()
        canWrite.P()
        mutex.P()
        buf.enqueue(val)
        mutex.V()
        canRead.V()

process consumer:
    while true:
        canRead.P()
        mutex.P()
        int val = buf.dequeue()
        mutex.V()
        canWrite.V()
        process(val)
```

16 SEMAFORI BINARI (13/11/2019)

16.1 Problema del buffer limitato

L'ultima volta ci siamo lasciati con la risoluzione del problema del buffer limitato, proponendo una realizzazione d'alto livello. È però opportuno soffermarsi anche su una realizzazione di livello più basso, che faccia uso di una coda realizzata tramite array circolare

```
int buf[N]
semaphore ok2Read(0), ok2Write(N)
int front = 0, back = 0

process producer:
    while true:
        int val = generate()
        ok2Write.P()
        buf[front] = val
        front = (front + 1) % N
        ok2Read.V();

process consumer:
    while true:
        int val
        ok2Read.P();
        val = buf[back]
        back = (back + 1) % N
        ok2Write.V()
        process(val);
```

Si noti che, nel caso di un solo produttore ed un solo consumatore, le variabili `front` e `back` non sono condivise: non è quindi necessario un semaforo `mutex` per proteggerne l'accesso. Nella versione con più produttori o più consumatori, il semaforo ausiliario è necessario, il che trasforma il codice precedente in:

```
int buf[N]
semaphore ok2Read(0), ok2Write(N)
semaphore mutexr(1), mutexw(1)
int front = 0, back = 0

process producer:
    while true:
        int val = generate()
        ok2Write.P() /* (1) */
        mutexw.P() /* (2) */
        buf[front] = val
        front = (front + 1) % N
```

```
        mutexw.V()
        ok2Read.V();

process consumer:
    while true:
    int val
        ok2Read.P();
        mutexr.P()
        val = buf[back]
        back = (back + 1) % N
        mutexr.V()
        ok2Write.V()
        process(val);
```

È opportuno che il metodo `P()` di `mutexw` e `mutexr` venga invocato solo dopo la verifica del permesso di esecuzione della CS. Invertendo le righe `(1)` e `(2)`, il `mutexw` potrebbe rimanere inaccessibile sia per il consumer sia per il producer. *[verificare]*

16.2 Semafori binari

Una variante dei classici semafori è quella dei **semafori binari**, in cui nulla cambia se non che il proprio valore può assumere solo valori `0` e `1`. L'invariante per questo particolare caso assumerà la forma

$$0 \leq init + n_V - n_P \leq 1 \tag{3}$$

Da ciò si deduce che il blocco del semaforo avviene quando $init + n_V - n_P = 1$ e viene invocato `V()` oppure quando vale `0` e viene invocato `P()`.

Si noti che un semaforo binario non è un semaforo: infatti un semaforo può assumere valori diversi da `0` e `1`, cosa che un semaforo binario non può fare.

16.2.1 Equivalenza espressiva

Possiamo porci la seguente domanda: semafori binari e semafori "normali" hanno lo stesso potere espressivo? Per rispondere a tale domanda tentiamo di scrivere degli algoritmi per implementare una delle versioni attraverso l'altra, e viceversa.

Partiamo dalla realizzazione di semafori binari da quelli generali, supponendo di disporre di primitive per semafori generali

```
class semaphore:
    init(int v)
    P()
    V()
```

Vorremo allora ottenere

```
class binary_semaphore:
    semaphore S0, S1

    Binit(s):
        S0.init(s)
        S1.init(1 - s)

    BP():
        S0.P()
        S1.V()

    BV():
        S1.P()
        S0.V()
```

Ad ogni semaforo binario facciamo quindi corrispondere due semafori generali, che regolano l'attivazione delle operazioni `BP()` e `BV()`.

Adesso ci cimentiamo nella realizzazione di semafori generali a partire da quelli binari. Partendo dunque da primitive di semafori binari quali

```
class binary_semaphore
    Binit(int v)
    BP()
    BV()
```

puntiamo a realizzare primitive di semafori generali come

```
class semaphore:
    int val
    pid_t_queue q
    binary_semaphore mutex(1) # protegge le operazioni
    binary_semaphore procsem[N] = {0, ..., 0} # vedere nota (1)

    init(int v):
        mutex.BP()
        val = v
        mutex.BV()

    P():
        mutex.BP()

        if val == 0:
            q.enqueue(getpid()) # mette il processo chiamante in coda
            # sblocca il mutex, altrimenti al blocco del chiamante si ha deadlock
            mutex.BV()
            procsem[getpid()].BP() # blocca chiamante
```

```
        else:
            val--
            mutex.BV()

    V():
        mutex.BP()

        if !q.empty:
            # sblocca q.dequeue(), il primo della coda
            procsem[q.dequeue()].BV()
        else:
            val++

        mutex.BV()
```

(1) In questa riga viene dichiarato un vettore di dimensione predefinita `N`, contenente semafori binari inizializzati a `0`. Ad ognuno dei binari sarà associato un processo in attesa. Al momento della call di `P()` su un semaforo binario, il processo chiamante si bloccherà.

Sarebbe poi errato fornire una versione alternativa quale

```
class semaphore:
    int val
    int nblocked
    binary_semaphore mutex(1)
    binary_semaphore blocked(0)

    init(int v):
        mutex.BP()
        val = v
        mutex.BV()

    P():
        mutex.BP()

        if val == 0:
            nblocked++
            mutex.BV()
            blocked.BP()
        else:
            val--
            mutex.BV()

    V():
        mutex.BP()
```

```
        if nblocked > 0:
            nblocked--
            blocked.BV()
        else:
            val++

        mutex.BV()
```

perché tale realizzazione non sarebbe *fair*.

Questa realizzazione finale generalizza l'implementazione dei semafori per un numero indefinito di processi. Invece di memorizzare i semafori binari in un vettore di dimensione `N`, si utilizza una coda.

```
class semaphore:
    int val
    binary_semaphore_queue q
    binary_semaphore mutex(1)

    init(int v):
        mutex.BP()
        val = v
        mutex.BV()

    P():
        mutex.BP()

        if val == 0:
            s = new binary_semaphore(0)
            q.enqueue(s)
            mutex.BV()
            s.BP()
            free(s)
        else:
            val--
            mutex.BV()

    V():
        mutex.BP()

        if !q.empty():
            q.dequeue().BV()
        else:
            val++

        mutex.BV()
```

Il successo di quest'ultima implementazione ci mostra che semafori generali sono equivalenti a semafori binari.

Nota: nelle prossime realizzazioni verrà utilizzata, solitamente, la notazione `P()` e `V()` sia per i semafori generali che per quelli binari.

16.3 Problemi classici

Torniamo alla discussione dei classici problemi di concorrenza della volta scorsa, riprendendo per primo il problema dei filosofi a cena.

16.3.1 Problema dei filosofi a cena

Questo problema prevede la presenza di cinque filosofi che siedono ad una tavola rotonda con un piatto di spaghetti davanti: ognuno di loro, vista la scarsa cultura culinaria dei creatori del problema, ha bisogno di due forchette per mangiare gli spaghetti. Ogni filosofo ha una forchetta alla propria sinistra e il tavolo è circolare, quindi ognuno dei cinque avrà una forchetta a destra e una a sinistra, per un totale di cinque posate. Ogni filosofo nella sua vita o pensa o mangia, seguendo il seguente algoritmo:

```
process philosopher:
    while !dead:
        think()
        eat()
```

Perciò, uno degli obiettivi principali della realizzazione sarà quello di non far morire di fame nessun filosofo (starvation). La scelta del numero di filosofi non è casuale: cinque è infatti il più basso numero che non ammette soluzione "ovvia", cioè simmetrica o per mutua esclusione. Un filosofo con una forchetta morirebbe di fame; il "ritmo di mangiata" di due filosofi sarebbe mutualmente esclusivo; con tre filosofi ognuno sarebbe in competizione con tutti gli altri; quattro filosofi si alternerebbero tra numeri pari e numeri dispari.

Abbozziamo una prima soluzione:

```
semaphore stick[5] = {1, 1, 1, 1, 1}

process philo[i], i = 0, ..., 4
    while true:
        think()
        stick[i].P() # (1)
        stick[(i + 1) % 5].P()
        eat()
        stick[i].V()
        stick[(i + 1) % 5].V()
```

questa soluzione non è ammissibile in quanto se tutti i filosofi partono ad uguale velocità e vengono fatalmente interrotti in `(1)`, il riprendere dell'esecuzione di uno qualunque di essi comporterà una condizione di deadlock.

Proviamo a sistemare i difetti della versione precedente introducendo un vettore di semafori che indica se un filosofo sta mangiando o meno

```
semaphore wait2eat[5] = {1, 1, 1, 1, 1}
bool eating[5] = {false, false, false, false, false}

process philo[i], i = 0, ..., 4
    while true:
        think()

        # Se uno dei due vicini sta mangiando
        if eating[(i + 1) % 5] || eating[(i + 4) % 5])
            wait2eat[i] = true

        eating[i] = true
        eat()
        eating[i] = false

        # se il vicino a sx di 2 non sta mangiando
        if (!eating[(i + 3) % 5]
            # risveglia se in attesa quello a sinistra
        # se il vicino a dx di 2 non sta mangiando
            # risveglia se in attesa quello a destra
```

Questa soluzione porta a starvation. Supponiamo infatti che ci sia una congiura: i filosofi da `1` a `4` si mettono d'accordo per far morire il filosofo `0`.

- Il filosofo 1 inizia a mangiare
- I filosofi 0 e 2 non possono mangiare
- Il filosofo 3 inizia a mangiare
- Il filosofo 3 smette e dice al 4 di mangiare
- Il filosofo 4 inizia a mangiare
- Il filosofo 1 smette e dice al 2 di mangiare
- Il filosofo 2 inizia a mangiare In tal modo il filosofo 0 non avrà mai modo di mangiare.

Diagramma di Gantt:

```
|____1____|___2___|___1___|___2___|___1___|___2___| ...
|__3__|___4___|___3___|___4___|___3___|___4___| ...
```

17 SEMAFORI - PASSAGGIO DEL TESTIMONE (18/11/2019)

17.1 Passaggio del testimone

Nelle implementazioni di semafori viste finora abbiamo incontrato schemi in cui un dato processo `P1` entrava ed usciva da una sezione critica, consentendovi l'accesso ad altri processi. In certe occasioni potremmo voler ideare uno schema in cui, non appena un ipotetico `P1` ha finito di lavorare nella sezione critica, voglia permetterne l'accesso ad un *arbitrario* processo `P2` che soddisfi certe condizioni. Vorremmo fare in modo che tra la sez. critica di `P1` e quella di `P2` non vi siano processi che entrino nella sezione critica in tempi intermedi. Realizzeremo questa gestione tramite la tecnica del **passaggio del testimone** (*passing the baton* in inglese), cimentandoci inizialmente in una modifica dell'implementazione vista nella lezione precedente.

17.1.1 Esempio di pseudocodice

```
class semaphore:
    int val
    binary_semaphore_queue q
    binary_semaphore mutex(1)

    init (int v):
        mutex.P()
        val = v
        mutex.V()

    P():
        mutex.P()

        if val == 0:
            s = new binary_semaphore(0)
            q.enqueue(s)
            mutex.V()
            s.P()
            free(s)

        val--
        mutex.V()

    V():
        mutex.P()
        val++

        if !q.empty()
```

```
        # Riattiva un processo bloccato, senza rilasciare la sezione critica
        q.dequeue().V()
    else
        mutex.V()
```

In questa nuova implementazione, la `V()` non rilascia necessariamente `mutex`. Di conseguenza, in caso di una chiamata di `dequeue()` (da parte di `P1`), nessun processo potrà eseguire il codice di `P()` finché `P2` (appena risvegliato) non avrà rilasciato la mutua esclusione.

17.2 Problemi classici

17.2.1 Problema dei filosofi a cena.

Durante la lezione precedente non è stata trovata una soluzione a questo problema. Le due realizzazioni provate presentavano infatti problemi di deadlock e starvation. Com'è stato anticipato, l'unica soluzione ammissibile è rompere la simmetria. Possiamo quindi stabilire che uno dei filosofi sia mancino, ovvero che necessita di prendere per prima la posata opposta rispetto a quella dei colleghi.

[completare il paragrafo]

```
semaphore stick[5] = {1, 1, 1, 1, 1};

process philo[1..4];
    think()

    eat()
    stick[MAX(i, (i+1) % 5)].V()
```

17.2.2 Problema dei lettori-scrittori

Un ulteriore problema classico di concorrenza è il seguente: una risorsa è condivisa tra un certo numero di processi di due tipi. I lettori accedono alla risorsa per leggerne il contenuto, mentre gli scrittori accedono alla risorsa per aggiornarne il contenuto.

Posto r il numero dei lettori e w quello degli scrittori, vorremo mantenere le seguenti invarianti

1. $r > 0 \wedge w = 0$, *oppure*
2. $r = 0 \wedge w = 1$

che corrispondono alle seguenti condizioni

1. Più processi possono leggere mentre nessuno di essi scrive
2. Un processo ha il permesso di scrivere interrompendo la lettura di tutti gli altri

Vediamo di darne un esempio di implementazione nel seguente

17.2.2.1 Primo tentativo di implementazione

```
reader: process (1..n)
    while true:
        start_read()
        read()
        end_read()

writer: process (1..m)
    while true:
        start_write()
        write()
        end_write()

"""
Le seguenti procedure fanno uso, implicitamente,
di un semaforo per gli scrittori ed uno per i lettori.
Entrambi i semafori sono muniti delle solite operazioni
e di una rispettiva coda per i processi.
"""

# Effettua <await(nw == 0) -> nr++>
start_read():
    mutex.P()

    if !(nw == 0):
        s = new sem(0)
        qr.enqueue(s)
        mutex.V()
        s.P()
        free(s)

    nr++
    f()

# Effettua <nr-->
end_read():
    mutex.P()
    nr--
    f()

# Effettua <await(nw == 0 && nr == 0) -> nw++>
start_write():
    mutex.P()
    if !(nw == 0 && nr == 0):
        s = new sem(0)
```

```
        qw.enqueue(s)
        mutex.V()
        s.P()
        free(s)
    nw++
    f()

# Effettua <nw-->
end_write():
    mutex.P()
    nw--
    f()
```

dove con `f()` nello pseudocodice qui sopra intendiamo la sostituzione del frammento seguente *[sostituire con funzione incapsulata]*

```
if nw == 0 && !qr.empty():
    qr.dequeue().V()
else if nr == 0 && nw == 0 && !qw.empty():
    qw.dequeue().V()
else:
    mutex.V()
```

Osserviamo che con la notazione `<A -> B>` indichiamo l'esecuzione di `B` solo in seguito al soddisfacimento della condizione di `B`, in gergo senza soluzione di continuità *[verificare]*.

Per esprimere in modo più chiaro il codice scritto sopra, possiamo sostituire le occorrenze di `f()`, eliminando opportunamente le condizioni del costrutto `if-else` che non possono verificarsi

```
start_read():
    mutex.P()

    if !(nw == 0):
        s = new sem(0)
        qr.enqueue(s)
        mutex.V()
        s.P()
        free(s)

    nr++

    if !qr.empty:
        qr.dequeue().V()
    else
        mutex.V()
```

```
end_read():
    mutex.P()
    nr--

    if !qr.empty():
        qr.dequeue().V()
    else if nr == 0 && !qw.empty():
        wq.dequeue().V()
    else
        mutex.V()

start_write():
    mutex.P()

    if !(nw == 0 && nr == 0):
        s = new sem(0)
        qw.enqueue(s)
        mutex.V()
        s.P()
        free(s)

    nw++
    mutex.V()

end_write():
    mutex.P()
    nw--

    if !qr.empty():
        qr.dequeue().V()
    else if !qw.empty():
        qw.dequeue().V()
    else:
        mutex.V()
```

Purtroppo, visto il favorimento dei processi lettori, la soluzione appena introdotta induce a starvation. Possiamo rimediare a questo difetto con la strategia seguente

- L'arrivo di uno scrittore durante l'esecuzione di diversi processi lettore comporta il rifiuto di tutti i processi lettore successivi. Questi saranno inseriti in una coda ed eseguiti quando il processo scrittore avrà terminato
- I processi scrittori favoriscono la precedenza ai processi lettori, ed in secondo luogo quella dei processi scrittori

Per concretizzare queste idee è necessario ridefinire le procedure `start_read()`, `end_read()`, `start_write()` ed `end_write()`.

17.2.2.2 Secondo tentativo di implementazione

```
/* ... */

start_read():
    mutex.P()

    if !(nw == 0 && qw.empty()):
        s = new sem(0)
        qr.enqueue(s)
        mutex.V()
        s.P()
        free(s)

    nr++

    if !qr.empty:
        qr.dequeue().V()
    else
        mutex.V()

end_read():
    mutex.P()
    nr--

    if !qr.empty():
        qr.dequeue().V()
    else if nr == 0 && !qw.empty():
        wq.dequeue().V()
    else
        mutex.V()

start_write():
    mutex.P()

    if !(nw == 0 && nr == 0):
        s = new sem(0)
        qw.enqueue(s)
        mutex.V()
        s.P()
        free(s)

    nw++
    mutex.V()

end_write():
```

```
mutex.P()
nw--

if !qr.empty():
    qr.dequeue().V()
else if !qw.empty():
    qw.dequeue().V()
else:
    mutex.V()
```

17.3 Considerazioni

Per quanto utili, i semafori presentano i seguenti difetti:

- Sono strutture dati di basso livello
- Gestione manuale della mutua esclusione
- Gestione manuale delle code per evitare la starvation

Si può astrarre il problema come la gestione delle sezione critiche, quindi tutte le implementazioni hanno l'idea di mutua esclusione, e inoltre vanno gestite delle condizioni da verificarsi per l'attivazione dei semafori. Vedremo come risolvere queste problematiche in una delle lezioni successive, quando avremo parlato di monitor.

17.4 Linguaggi di shell scripting

Sfruttiamo la seconda metà della lezione ritornando ai concetti di linguaggi di shell.

`/dev/null` è un device che scarta tutto ciò che viene scritto su di esso, e provando a leggere da esso viene restituito `EOF`. Questo device può risultare utile nella valutazione delle condizioni booleane in cui sono presenti comandi di cui non interessa l'output: infatti l'output si può mandare su questo device, e quindi scartare.

Ad esempio, l'output dei comandi `cmp` e `diff` sotto descritti può non essere rilevante:

17.4.1 Comandi di utilità

In questa sezione continuiamo il glossario di comandi di utilità lasciato in sospeso le volte scorse.

17.4.1.1 `cmp`

Il comando `cmp` confronta due file byte per byte.

17.4.1.2 `diff`

Il comando `diff` confronta due file linea per linea.

17.4.1.3 `patch`

Il comando `patch` effettua la patch di un file utilizzando un dump effettuato da `diff`.

17.4.1.4 `find`

Il comando `find` ricerca risorse all'interno di un file system mediante un sistema di filtri. Esempi

```
# Cerca tutti i file terminanti in '.sh' sotto il percorso '/etc'
find /etc -name "*.sh"

# Cerca tutti i file chiamati 'hw.c' nel percorso './' ed esegui 'ls -l' su di essi
find . -name hw.c -exec ls -l {} \;
```

I parametri di `find` sono raggruppati in due categorie

- Di ricerca : `-name`, `-user`, ...
- Di azione: `-exec`, `-execdir`, `-show`, `-print`, ...

17.4.1.5 `xargs`

Il comando `xargs` riceve in standard input valori da passare come parametro di un secondo comando.

Ad esempio, eseguendo `xargs ls` e digitando `dir` vengono elencati i file nella directory `./dir`.

L'esempio seguente, invece, scrive nel file `all_C` tutti i testi dei file `.c` presenti nella directory corrente, uno di seguito all'altro.

```
find . -name "*.c" | xargs cat > all_C
```

18 MONITOR (20/11/19)

18.1 Materiale on-line

Sono a disposizione dello studente strumenti per la sperimentazione di semafori e monitor nei linguaggi C e Python alla pagina del professor Davoli. La compilazione può essere effettuata mediante l'apposito `Makefile`.

18.2 Monitor

Abbiamo già accennato, la volta scorsa, ad alcuni difetti principali dei semafori. Vogliamo ora introdurre degli strumenti di programmazione concorrente, detti **monitor**, che automatizzino molte delle gestioni di concorrenza che erano prima affidate al programmatore.

Quando scriviamo il codice con i monitor non dobbiamo preoccuparci delle interazioni non volute perché siamo sicuri che solo un processo alla volta sta eseguendo il codice del monitor.

Nota: d'ora in avanti utilizzeremo la keyword `entry` con un significato simile a quello del `public` di C. La scriveremo quindi per indicare procedure del monitor visibili anche all'esterno.

18.2.1 Esempi in pseudocodice

```
monitor conto_corrente:
    /* variabile privata */
    int saldo

    /* Aggiorna il saldo del conto corrente */
    entry operazione(int valore):
        saldo += valore

monitor producer_consumer:
    int buffer
    bool isfull = false

    condition full    /* isfull == true */
    condition empty    /* isfull == false */

    entry void put(int value):
        if (isfull)
            empty.wait()

        buffer = value
        isfull = true
        full.signal()
```

```
entry int get():
    if (!isfull)
        full.wait()

    int tmp = buffer
    isfull = false
    empty.signal()
    return tmp
```

Nello pseudocodice appena introdotto abbiamo fatto uso delle variabili `full` ed `empty`, dette **variabili di condizione**. Una variabile di condizione è una variabile munita di due operazioni:

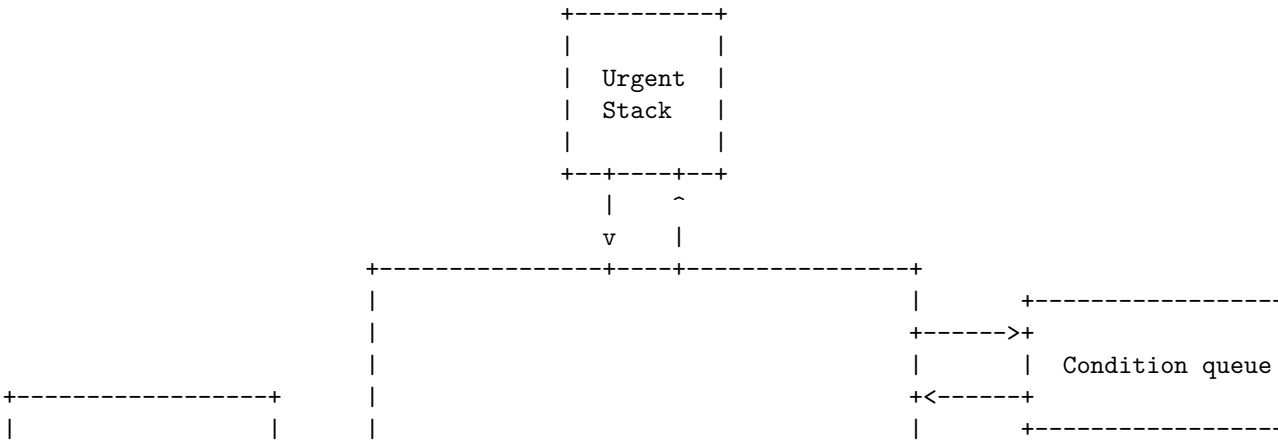
- `wait()`: attende il verificarsi di una certa condizione
- `signal()`: segnala ad un dato processo il verificarsi di una certa condizione

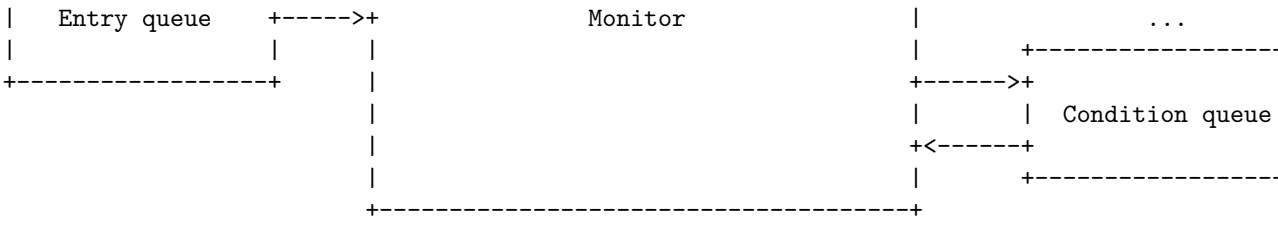
18.2.2 Implementazione dei monitor

Per gestire l'entrata nel monitor viene usata una coda detta *entry queue*.

Supponiamo che un processo `P` entri nel monitor e si fermi su una particolare variabile di condizione `CV` poiché il valore di essa non rispetta le sue esigenze. A questo punto, `P` viene posto fuori dal monitor ed inserito nella *condition queue* relativa alla variabile di condizione su cui esso ha invocato `wait()`. Quando poi un processo `Q` lancerà una `signal()` su `CV`, se esiste un processo nella coda relativa a questa variabile di condizione (nel nostro caso `P`), esso viene risvegliato e rientra nel monitor. Il processo che ha chiamato la `signal()`, invece, viene posto in uno *urgent stack* (altrimenti sarebbero presenti due processi nel monitor contemporaneamente).

La *urgent stack* viene completamente svuotata prima che venga preso in ingresso un nuovo processo dalla *entry queue*.





L'implementazione del monitor di cui sopra potrebbe essere utilizzata nel modo seguente da una coppia di processi produttore-consumatore

```
process producer:
    while true:
        x = produce()
        pc.put(x)

process consumer:
    while true:
        x = pc.get()
        consume(x)
```

18.2.3 Produttore-consumatore multiplo

Possiamo realizzare un monitor per il problema produttore-consumatore con buffer multiplo come segue

```
monitor producer_consumer:
    queue q
    condition ok2put    /* q.length() < N */
    condition ok2get    /* q.length() > 0 */

    entry void put(int value):
        if (q.length() >= N)
            ok2put.wait()

        q.enqueue(value)
        ok2get.signal()

    entry int get()
        if (q.length() == 0)
            ok2get.wait()

        tmp = q.dequeue()
        ok2put.signal()

        return tmp
```

Si noti che, nei monitor, la mutua esclusione è già garantita. Di conseguenza

l'implementazione rimane la stessa sia per due processi che per un numero arbitrario `N`.

18.2.4 Potere espressivo

Vogliamo ora dimostrare che monitor e semafori possiedono lo stesso potere espressivo. Affronteremo tale prova come nella volta dell'equivalenza tra semafori e semafori binari, implementando un costrutto mediante l'altro, e viceversa.

18.2.4.1 Semafori implementati con monitor

Partiamo con la realizzazione di un semaforo avvalendoci di un monitor come primitiva a disposizione

```
monitor semaphore:
    int value
    condition ok2p    /* value > 0 */

    entry init(unsigned v):
        value = v

    entry P():
        if value == 0:
            ok2p.wait()

        value--

    entry V():
        value++
        ok2p.signal()
```

Da notare il ruolo dello *urgent stack*: se viene mandata una `signal()` il processo che prima era in coda di attesa ritorna nel monitor, finendo dunque l'invocazione di `P()` che aveva iniziato e lasciando il valore del semaforo a `0`. Avviene dunque una sorta di "passaggio del testimone".

Differenze tra monitor e semafori Benché similari, semafori e monitor divergono sotto i seguenti aspetti

- `wait()` è sempre bloccante, mentre `P()` si blocca solo in base al valore del semaforo
- `V()` è una chiamata bloccante, quindi se in seguito arriverà una `P()` non si bloccherà perchè era stata fatta una `V()` in precedenza, mentre la `signal()` viene persa se non ci sono processi in coda
- `signal()` si comporta come se passasse il testimone, ovvero trasmette immediatamente la segnalazione

18.2.4.2 Monitor implementati con semafori

Date le primitive dei semafori, ci poniamo ora lo scopo di realizzare quelle dei monitor.

```
semaphore mutex(1)
stack urgent

/* Definiamo per motivi di chiarezza una struttura 'condition' con un solo campo */
struct condition:
    queue q

monitor_enter():
    mutex.P()

wait(condition cond):
    wait_semaphore = new semaphore(0)
    cond.q.enqueue(wait_semaphore)
    __exit()
    wait_semaphore.P()
    free(wait_semaphore)

signal(condition cond):
    if (!cond.q.empty()):
        urgent_semaphore = new semaphore(0)
        urgent.push(urgent_semaphore)
        cond.q.dequeue().V()
        urgent_semaphore.P()
        free(urgent_semaphore)

monitor_exit():
    __exit()
    mutex.V()
```

dove `__exit()` è implementata come segue:

```
void __exit():
    if !urgent.empty():
        urgent.pop().V()
    else
        mutex.V()
```

[aggiungere conclusioni sull'equivalenza espressiva]

19 PRESENTAZIONE PROGETTO (25/11/19)

Il corso di sistemi operativi comprende una parte di progetto, riguardante lo sviluppo di un piccolo sistema operativo, *BiKaya*. È possibile ricavare i requisiti di progetto alla pagina del corso di Sistemi Operativi.

Osserviamo che:

- I registri delle due architetture, $\mu MPS2$ e μARM, non sono propriamente tali, ma fanno riferimento ad indirizzi di bus sui quali è possibile scrivere o leggere informazioni in maniera asincrona.

20 MESSAGE PASSING (27/11/19)

Nell'ultima lezione di teoria abbiamo introdotto i monitor, utilizzandoli nella risoluzione dei problemi del produttore-consumatore e del buffer limitato. Adesso non ci rimane che impiegarli per la risoluzione dei problemi dei cinque filosofi e dei lettori-scrittori.

20.1 Problemi classici

20.1.1 Problema dei lettori-scrittori

```
/* i = 1..NREADERS */
reader_i: process
    while true:
        rw.beginread()
        /* read */
        value = rw.read()
        rw.endread()


/* i = 1..NWRITERS */
writer_i: process
    while true:
        rw.beginwrite()
        /* write */
        rw.endwrite()


rw monitor:
    /* ww = number of waiting writers */
    int nr = 0, nw = 0, ww = 0
    condition ok2read    /* nw == 0 */
    condition ok2write   /* nr == 0 && nw == 0*/

    entry beginread():
        if nw > 0 || ww != 0:
            ok2read.wait()

        nr++
        ok2read.signal()

    entry endread():
        nr--

        if nr == 0:
            ok2write.signal()
```

```
    entry beginwrite():
        if nr > 0 || nw > 0:
            ww++
            ok2write.wait()
            ww--

        nw++

    entry endwrite():
        nw--
        ok2read.signal()

        if nr == 0:
            ok2write.signal()
```

L'invocazione di `ok2read.signal()` in `beginread()` è necessaria, in quanto omettendola solo il primo lettore in coda inizierebbe a leggere una volta giunto il suo turno; così, invece, ogni lettore chiama il successivo a cascata.

20.1.2 Problema dei cinque filosofi

Tentiamo una risoluzione del problema dei cinque filosofi con l'ausilio dei monitor.

```
/*  i = 0..4 */
process philo_i:
    while true:
        think()
        stick[i].get()    /* (1) */
        stick[(i+1) % 5].get()

        eat()

        stick[i].put()
        stick[(i+1) % 5].put()


monitor generic_stick:
    condition ok2get
    bool available = true

    entry get():
        if !available:
            ok2get.wait()

        available = false
```

```
    entry put():
        available = true
        ok2get.signal()
```

Ricordiamo che la soluzione appena abbozzata porterebbe a deadlock, in quanto un sospendersi dei cinque processi all'istruzione `(1)` porterebbe l'acquisizione della bacchetta sinistra di ogni filosofo, impedendo l'ottenimento di quella destra al ritorno di uno qualunque tra essi. Sistemiamo tali difetti nella versione seguente

```
process philo_i: i = 0..4
    while true:
        think()
        philo.eat_begin(i)
        eat()
        philo.eat_end(i)


monitor philo:
    bool iseating[5]
    condition ok2read[5]

    entry eat_begin(int i)
        if (iseating[(i+1) % 5] || iseating[(i+4) % 5]):
            ok2eat[i].wait()
        iseating[i] = true

    entry eat_end(int i):‘
        iseating[i] = false

        if (!iseating[(i + 2) % 5])
            ok2eat[(i+1) % 5].signal()
        if (!iseating[(i+3) % 5])
            ok2eat[(i+4) % 5].signal()
```

Quest'ulteriore soluzione porta a starvation (congiura dei filosofi). Cerchiamo allora di iterare nuovamente il processo di miglioramento, nella rielaborazione seguente.

```
monitor philo:
    bool avstick[5]    /* available sticks */
    condition ok2get[5]

    entry eat_begin(int i):
        if !avstick[i]:
            ok2get[i].wait()
        avstick[i] = false
```

```
        if !avstick[(i+1) % 5]:
            ok2get[(i+1) % 5].wait()
        avstick[(i+1) % 5] = false

    entry eat_end(int i):
        avstick[i] = true
        ok2get[i].signal()

        avstick[(i+1) % 5] = true
        ok2get[(i+1) % 5].signal()
```

Questa soluzione non porta a deadlock, in quanto bisognerebbe che tutti i filosofi siano bloccati al `wait()` del secondo `if` perchè ciò si verifichi. Detto informalmente, significa che il vicino avrebbe dovuto trovare una bacchetta libera e l'altra occupata, così come il vicino del vicino - prima del vicino - e così via.

20.1.3 Dimostrazione dell'assenza di deadlock

Procediamo per assurdo. Chiamiamo i filosofi `A`, `B`, `C`, `D`, `E`. Il filosofo `A` trova la bacchetta alla sua sinistra libera e quella alla sua destra occupata: quindi la bacchetta alla sua destra era stata presa da `B`, che a sua volta aveva trovato la bacchetta alla sua destra occupata perché era stata precedentemente presa da `C`, e così via. Si ha che il filosofo `E` avrebbe trovato occupata la bacchetta alla propria destra perché presa da `A`. Questo è assurdo: `E` era entrato nel monitor prima di `A`, quindi visto che per ipotesi `A` trova la bacchetta alla propria sinistra (e quindi alla destra di `E`) libera, questa era libera anche all'ingresso di `E` nel monitor.

20.2 Message passing

Finora abbiamo sempre considerato modelli con processi comunicanti in una zona di memoria **condivisa**. Affronteremo adesso lo studio di programmi dove i processi in esecuzione lavorano su spazi di memoria **privati e inaccessibili da fuori**.

La comunicazione tra processi di questo tipo avviene mediante lo scambio di messaggi. Un messaggio è un insieme di informazioni formattate da un processo *mittente* e interpretate da un processo *destinatario*. L'invio di un messaggio `m` dal processo `P` al processo `Q` corrisponde alla copia delle informazioni di m dallo spazio di memoria privato di `P` a quello relativo a `Q`. Questo procedimento è mediato dal sistema operativo.

Detto ciò, è bene osservare che il message passing non è un meccanismo di sola sincronizzazione tra processi, come lo erano semafori e monitor. Esso rappresenta infatti una vera e propria comunicazione di informazioni, piuttosto che di semplici segnali.

Le primitive a disposizione, `send()` e `receive()`, utilizzano messaggi in formato

arbitrario: per esprimere una strutturazione all'interno di questi messaggi sarà necessario accordarsi su un protocollo mittente-destinatario. Alcuni esempi di uso di queste primitive in pseudocodice sono:

- `send(dest, msg)` nella quale il processo destinatario deve essere specificato
- `msg = recv(sender)` nella quale può essere indicato un particolare mittente, oppure non specificarne nessuno, utilizzando quindi la keyword `ANY`

Studieremo tre modelli basati su comportamenti diversi di invio e ricezione di messaggi:

1. **Sincrono**: sia `send()` che `receive()` sono bloccanti
2. **Asincrono**: solo `receive()` è bloccante
3. **Completamente asincrono**: nessuna delle due chiamate è bloccante. Nota: non essendo implementabile, l'interesse di questo metodo è riservato ad ambienti puramente accademici.

Nel seguito adotteremo la seguente nomenclatura:

1. Nel modello sincrono, le primitive disponibili saranno `ssend()` e `srecv()`
2. In quello asincrono, `asend()` e `arecv()`
3. In quello puramente asincrono, `asend()` e `nb-receive()`

20.2.1 Problemi classici

Possiamo affrontare lo studio dei classici problemi di concorrenza anche nel modello di comunicazione tra processi per scambio di messaggi a memoria privata.

20.2.1.1 Produttori-consumatori

20.2.1.1.1 Message-passing sincrono

```
process producer:
    while true:
        x = produce()
        ssend(server, x)

process consumer:
    while true:
        x = srecv(server)
        consume(x)

process server:
    while true:
        x = srecv(producer)
        ssend(consumer, x)
```

L'utilizzo di un processo `server`, nella realizzazione appena mostrata, serve come intermediario della comunicazione. Questo processo funziona quindi come uno *switch*: fa in modo che il messaggio in quesione venga trasmesso solo ed esclusivamente al processo destinatario *[verificare]*. Nell'esempio seguente rivediamo lo stesso problema mediante un passaggio di messaggi asincrono.

20.2.1.1.2 Message-passing asincrono

```
process producer:
    while true:
        x = produce()
        asend(consumer, x)
        arecv(consumer)

process consumer:
    while true:
        x = arecv(producer)
        asend(sender, ACK)
        consume(x)
```

Questa realizzazione prevede l'attesa di una conferma (*acknowledgement*) nel processo `producer` da parte di `consumer`, ma è immediato notare come ci si riduca al caso della comunicazione sincrona dell'esempio precedente. Per ricavare i benefici della comunicazione asincrona sarà necessario creare un offset tra i messaggi inviati dal produttore e le conferme, in modo da inviare il messaggio $i + 1$ solo dopo aver ricevuto la conferma del messaggio i.

```
process producer:
    bool first = true

    while true:
        x = produce()

        if (first)
            first = false
        else
            arecv(consumer)

        asend(consumer, x)

process consumer:
    while true:
        x = arecv(producer)
        asend(sender, ack)
        consume(x)
```

20.2.2 Implementazione del message passing sincrono con quello asincrono

Come nel modello a memoria condivisa, possiamo domandarci se la comunicazione sincrona sia riconducibile a quella asincrona, e viceversa. Nell'esempio seguente mostriamo innanzitutto che vale la riconducibilità della comunicazione sincrona in quella asincrona

```
asend(dst, <getpid(), msg>)
msg = arecv(sender)

ssend(dst, msg):
    asend(dst, msg)
    arecv(dst)

srecv(sender):
    <realsender, msg> = arecv(sender)
    asend(realsender, ACK)
    return msg
```

20.2.3 Implementazione del message passing asincrono con quello sincrono

```
asend(dst, msg):
    ssend(server, <getpid(), dst, msg>)

arecv(sender):
    ssend(server, <getpid(), NULL, sender>)
    return srecv(server)

q.enqueue(sender, dest, msg)
q.dequeue(dest, sender)    /* NULL se non c'è match */
waitfrom[processes]    /* NULL se non aspetta */

server:
    while true:
        (sender, dest, msg) = srecv(ANY)

        if dest == NULL:
            msgdest = sender
            msgsender = msg
            retmsg = q.dequeue(sender, msg)

            if retmsg == NULL:
                waitfrom[msgdest] = msgsender
            else:
                ssend(msgdest, retmsg)
```

```
else:
    if (waitfrom[dest] == sender || waitfrom[dest] == ANY):
        ssend(dest, msg)
        waitfrom[dest] = NULL
    else:
        q.enqueue(sender, dest, msg)
```

Per passare da sincrono ad asincrono si è dovuto aggiungere un processo, a differenza del passaggio inverso in cui è stato sufficiente aggiungere una libreria. Per questo il potere espressivo del sincrono è minore.

21 MESSAGE PASSING (CONT.) (02/12/2019)

21.1 Problemi classici

Continuiamo a soffermarci sul tema della comunicazione tra più processi con modello di memoria privata.

21.1.1 Problema dei filosofi a cena

Vediamo di impostare e dare una soluzione al problema dei cinque filosofi adottando le primitive di comunicazione tra più processi a memoria separata.

```
/* i = 0..4 */
process philo[i]:
    while true:
        asend(stick[i], getpid())
        arecv(stick[i])
        asend(stick[(i + 1) % 5], (getpid(), 'REQ'))
        arecv(stick[(i + 1) % 5])
        eat()
        asend(stick[i], (getpid(), 'RELEASE'))
        arecv(stick[(i + 1) % 5], (getpid(), 'RELEASE'))

/* i = 0..4 */
process stick[i]:
    while true:
        sender = arecv(ANY)
        asend(sender, ACK)
        arecv(sender)
```

Quando un filosofo vuole la bacchetta, spedisce un messaggio al server del tipo `asend(stick[i], getpid())`. Questo sarà anche il primo messaggio che ciascun filosofo manda.

21.2 Message passing asincrono senza server

A partire dal message passing asincrono è possibile creare quello "ANY asincrono", implementando quindi queste nuove primitive:

- `aasend(dest, msg)    /* ANY asend() */`
- `msg = aarecv()    /* ANY arecv() */`

Nel prossimo esempio proviamo ad implementare un meccanismo di message passing senza l'utilizzo di un server. Ricordiamo che, in questo caso, la ricezione può avvenire solo da "ANY".

```
asend(dest, msg):
    aasend(dest, (getpid(), msg))
```

```
arecv(sender):
    aasend(getpid(), (getpid(), TAG))

    while true:
        <sx, msg> = aarecv()

        if (sx == getpid() && msg == TAG):
            break

        mailbox.add(sx, msg)

    while (msg = mailbox.get(sender)) == NULL:
        <sx, msg> = aarecv()
        mailbox.add(sx, msg)

    return msg
```

21.3 Decalogo della programmazione concorrente

Alla relativa pagina della wiki del professor Davoli è presente una lista di criteri che un'implementazione di algoritmo concorrente deve poter rispettare. Nella pagina vengono mostrati errori di implementazione ricorrenti.

21.4 Introduzione a Python

Iniziamo a parlare di Python, un linguaggio imperativo orientato agli oggetti general-purpose, con diversi aspetti di programmazione funzionale. È prevalentemente interpretato, ma è anche presente una fase di compilazione che genera un codice intermedio più rapido da eseguire.

Tra le caratteristiche importanti è possibile evidenziare:

- Mancanza di un sistema di tipi. Il tipo di una particolare variabile viene assunto in base al valore assegnatole
- Utilizzo dell'indentazione e del carattere `:` per la definizione di blocchi
- Precisione variabile per gli interi. L'interprete (o compilatore) è in grado di aumentare dinamicamente lo spazio di memoria dedicato ad una variabile per evitare overflow
- Vi sono due tipi principali di strutture dati: mutabili ed immutabili. Un oggetto immutabile (es. stringa, tupla) non può essere modificato dopo la sua creazione
- Tutte le funzioni ritornano un valore, o al più `None`
- Per gestire l'iterazione di un ciclo `for` è anche possibile utilizzare una qualsiasi sequenza di elementi come, ad esempio, una lista

Fare riferimento alle pagine Esercizi di "lettura" programmi in Python 2019/20 e Esempi didattici in Python per approfondire le diverse caratteristiche del

linguaggio.

22 APPROFONDIMENTO PYTHON (04/12/2019)

22.1 Interpolazione di stringhe

Python possiede diversi meccanismi di interpolazione di stringhe, ovvero costrutti sintattici e chiamate di libreria capaci di sostituire stringhe segnaposto con valori concreti all'interno di stringhe di partenza.

22.1.1 Stringhe di formato

Il primo di questi metodi consiste nell'utilizzo dell'operatore `%` abbinato a stringhe di formato (es. `%s`, `%d`, ...) molto simili a quelle del linguaggio C. Abbiamo ad esempio

```
>>> name, age = "Renzo", 42
>>> print("Hello, %s! You are %d" % (name, age))
Hello, Renzo! You are 42
```

L'operatore `%` è anche in grado di ricevere un dizionario come secondo argomento, dal quale ricavare valori da sostituire nella stringa formattata, a patto di assegnare nomi individuali ai campi da sostituire

```
>>> person = {'name': 'Renzo', 'age': 42}
>>> print("Hello, %(name)s! You are %(age)d" % person)
Hello, Renzo! You are 42
```

22.1.2 Metodo `str.format()`

Il metodo `format` del tipo `str` permette la realizzazione di un meccanismo di interpolazione di stringhe alternativo a quello dell'operatore `%`. I campi rimpiazzati sono indicati da parentesi graffe contenenti l'indice del parametro.

Considerando il codice in esempio:

```
name, age = "Renzo", 42
"Hello, {}. You are {}".format(name, age)
```

si otterrà l'output `'Hello, Renzo. You are 42'`.

22.1.3 f-strings

Le f-string sono stringhe con una `f` al loro inizio (può essere anche maiuscola). Nelle f-string possono comparire parentesi graffe contenenti espressioni che saranno rimpiazzate con i propri valori. Le espressioni sono valutate a runtime. La sintassi è simile a quella di `str.format()` ma meno verbosa, di conseguenza più leggibile e comoda da utilizzare.

Supponendo che `name, age` siano le variabili dell'esempio precedente:

```
>>> f"Hello, {name}. You are {age}"
'Hello, Renzo. You are 42'
```

Siccome le f-string sono valutate a runtime, è possibile inserirvi qualsiasi espressione valida. Ad esempio:

```
>>> f"{2 * 37}"
'74'
```

All'interno delle f-string si possono anche effettuare chiamate di funzioni:

```
>>> def to_lowercase(input):
            return input.lower()

>>> name = "Renzo"

>>> f"{to_lowercase(name)}"
'renzo'

>>> f"{name.upper()}"
'RENZO'
```

22.2 Decoratori di funzione

Nell'introdurre i decoratori di funzione è necessario passare per il concetto di **funtore**, nel nostro contesto una funzione che ritorna un'altra funzione. È funtore ad esempio la funzione `addx()` del codice seguente

```
#!/usr/bin/env python3
import sys

def addx(x):
    def __add(y):
        return x + y

    return __add

add2 = addx(2)
add42 = addx(42)

n = int(sys.argv[1])
print(add2(n), add42(n))
```

Questo codice definisce una sottofunzione che somma al proprio argomento il valore passato alla funzione esterna. La sottigliezza sta nel fatto che la funzione esterna ritorna proprio la sottofunzione. In sostanza, la procedura esterna crea un'altra procedura. Nell'esempio appena riportato `add2` e `add42` sono funzioni che sommano al valore del proprio parametro `2` e `42` rispettivamente.

I funtori permettono di modificare facilmente la definizione di una procedura e, grazie a ciò, si possono evitare lunghi e dispendiosi procedimenti di iterazione o ricorsione. Ad esempio, grazie ai funtori si può calcolare un particolare valore

della successione di Fibonacci molto più velocemente che con una modalità ricorsiva (con la tecnica della *memoization*).

Terminologia: * **Funtore**: funzione che restituisce un'altra funzione * **Decoratore**: funzione che prende in input una funzione ed esegue delle istruzioni su di essa. Viene invocato anteponendo `@<decorator>` alla definizione della funzione da "decorare"

22.3 Funzioni variadiche

Il linguaggio Python possiede un supporto sintattico built-in per la definizione di funzioni variadiche (v. lezione del 16/10/19). Una funzione

```
def f(*x):
    return sum(x)
```

può essere invocata con un numero di parametri arbitrario:

```
f(10)              # 10
f(3, 4)            # 3 + 4 = 7
f(1, 2, ..., n)   # 1 + 2 + ... + n
```

Così facendo il tipo del parametro formale `x` sarà `tuple`.

Una modalità di passaggio alternativa di parametri attuali, contrapposta a quella posizionale consueta, consiste nella specifica del nome dei parametri formali. Ad es.

```
def f(a, b)
    return a + b

f(b=10, a=20)
```

produrrà lo stesso effetto dell'invocazione

```
f(20, 10)
```

Forti di ciò, è possibile estendere la definizione di funzioni variadiche posizionali con quella di funzioni variadiche nominative. In altre parole, data

```
def f(**x):
    for key, value in x.items():
        print(key, value)
```

essa potrà essere invocata come

```
f(a=10, b=20, ..., z=260)
```

dove i nomi di variabili `a`, `b`, ..., `z` sono arbitrari e affidati al codice chiamante, mentre `x` è nota alla funzione `f` come variabile di tipo `dict`. L'output prodotto sarà perciò

```
a 10
b 20
...
z 260
```

23 APPROFONDIMENTO PYTHON E BASH (09/12/2019)

23.1 Approfondimento linguaggio Python

Proseguiamo il nostro approfondimento sul linguaggio Python.

Una delle caratteristiche che non abbiamo ancora esplorato è l'utilizzo del comando da console `python`. Osserviamo innanzitutto che `python` è, in molti sistemi UNIX-like, un alias alla versione più recente dell'interprete; qualora si voglia invocare una versione specifica, è possibile inserirla come suffisso nel nome del comando, es. `python3`, `python2.7` ecc.

L'invocazione di `python` da riga di comando avvia l'interprete del linguaggio, in grado di leggere e valutare istruzioni riga per riga. Quando queste vengono inserite, ad esempio in

```
>>> 2 * 4 - 3
```

i risultati dell'utlima valutazione vengono memorizzati nella variabile `_`. È altresi possibile utilizzare il comando `python` in modi differenti; ad esempio, conosciuto il nome `x` di un modulo, è possibile eseguirlo con

```
$ python -m x
```

23.1.1 Costanti stringa multilinea

Il linguaggio possiede capacità built-in per definire stringhe costanti multilinea, con l'ausilio dei tre apici `'` o `"` concatenati. Esempio

```
print('''Usage:
                -s: do domething
                -o: other
        ''')
```

L'applicazione di tali stringhe è più vasta di quanto si potrebbe inizialmente immaginare. Con esse, è infatti possibile

- Introdurre commenti multilinea, in sostituzione di quanto fatto con il carattere `#`
- Servirsi delle cosiddette docstring, una convezione di documentazione per funzioni, metodi, classi e moduli

23.1.2 `range()`

La funzione `range()`, di vasta utilità, invocata come `range(a, b)` restituisce un oggetto iterabile contenente la sequenza dei numeri `a, a + 1, ..., b - 1`, in cui cioè l'estremo destro è escluso.

```
    # Stampa i numeri 0, 1, ..., 9
    for i in range(10):
        print(i)
```

`range()` si avvale anche di un terzo parametro, opzionale, che controlla la distanza tra un elemento e quello successivo. È così possibile

```
# Definire una sequenza intervallata da costanti maggiori di 1
range(1, 100, 5)
# Costruire sequenze decrescenti, es. 9, 8, ..., 0
range(9, -1, -1)
```

La funzione `range()` è un buon punto di partenza per comprendere anche altre caratteristiche del linguaggio. Ad esempio, dato un qualsiasi oggetto iterabile, come una stringa `s`, è possibile selezionare una qualsivoglia sottosequenza mediante operazioni di *splicing*

```
>>> s = 'velocissimo'
>>> s[-1]
'o'
>>> s[::-1]
'omissicolev'
>>> s[::2]
'vlcsio'
```

Per oggetti sequenziali modificabili, come le liste, è possibile sfruttare la sintassi appena vista per realizzare operazioni di modifica, come

```
>>> l = [3, 4, 5]
>>> l[0:0] = [1, 2]
>>> l
[1, 2, 3, 4, 5]
```

23.1.3 Separazione dell'output in `print()`

Alcune primitive del linguaggio prendono parametri nominativi. Una su tutte, `print()` può accettare un parametro nominativo `sep` per separare i parametri nella stampa.

```
>>> print(1, 'two', 3, sep = ' | ')
1 | two | 3
```

23.1.4 Lettura da file

Per leggere un file `f` possiamo semplicemente usare un `for` che abbia come range il file stesso.

```
import sys

with open(sys.argv[1], 'r') as f:
```

```
    for line in f:
        print('--- ', line)
```

In questo modo vengono lette solo metà delle righe (???): questo perché sia il file che `print()` vanno a capo. Abbiamo due modi per risolvere il problema:

```
1) print('--- ', line.strip()) che toglie i blank in fondo
2) line.split()
```

usando `for line in enumerate(f):` vengono inseriti gli indici di riga.

L'esempio sottostante fa un macello(sistemare)

```
for n, line in enumerate(f):
    print('---', n, line.split())
```

Ad esempio, una funzione estremamente utile di ciò potrebbe essere contare la frequenza delle parole di un testo, oppure la lista delle righe in cui compare una data parola. Implementiamo la seconda versione:

```
import sys

words = {}

with open(sys.argv[1], 'r') as f:
    for n, line in enumerate(f):
        for w in line.split():
            words[w] = words.get(w, []) + [n+1]

print(words)
```

Versione alternativa con le tuple:

```
words[w] = words.get(w, ()) + (n+1, )
```

altra roba

```
words.setdefault(w, []).append(str(n+1))
```

Se non esiste, la crea vuota e fornisce il puntatore all'ingresso della lista. Questa versione è la più efficiente tra tutte. A questo punto ci manca solo di stampare in ordine alfabetico:

```
for w in sorted(words)
    print(w, ', '.join(words[w])))
```

La versione finale creata è questa:

```
#!/usr/bin/python3
import sys

words = {}
```

```
with open(sys.argv[1], 'r') as f:
    for n, line in enumerate(f):
        for w in line.split():
            words.setdefault(w.strip(',.;:', []).append(n+1))

for w in sorted(words):
    print(w, ', '.join(map(str, words[w])))
```

Nota: non bisogna chiudere esplicitamente il file. Non appena il `for` in cui il file viene aperto si termina (tramite indentazione), il file viene automaticamente chiuso.

23.1.5 `is`

`is` ritorna `True` se e solo se i due l-valori operandi possiedono lo stesso identificativo, cioè mappano alla stessa zona di memoria. Se le variabili corrispondo semplicemente a valori uguali, senza coincidere in memoria allo stesso oggetto, `is` ritorna `False`. Ad esempio

```
a = [1, 2, 3]
b = [1, 2, 3]
>>> a == b
true
>>> a is b
false
>>> b = a    # assegna il riferimento, rendendoli di fatto la stessa cosa
>>> a is b
true
```

23.1.6 Metodo `setdefault` dei dizionari (boh non so quanto serva)

```
>>> d = {}
>>> d.setdefault("sempre", []).append(1)
>>> d
{'sempre': [1]}
>>> d.get("caro", []).append(1)
>>> d
{'sempre': [1]}
```

23.1.7 `if-then-else` in python

Gode di una sintassi molto descrittiva, al contrario dell'operatore ternario `x ? y : z` di `C`

```
def f(x):
    return "vero" if x == 42 else "falso"
```

23.1.8 Generatori

Vediamo come creare vari generatori: nel nostro caso, genereremo quadrati perfetti.

Generando con la list comprehension

```
gen0 = (x**2 for x in range(10))
```

Funzioni per creare generatori:

```
def gen1(limit):
    for x in range(limit):
        yield x*x

for i in gen1(10):
    print(x)
```

Facendo `yield` non si esce dalla funzione: si restituisce il valore di ritorno, poi l'esecuzione continua. La si può vedere come un processo in un sistema operativo: si ferma, per poi ripartire.

Nell'esempio sottostante, la stampa avviene fino a quando il quadrato stesso non sia maggiore di 10.

```
def gen2():
    x = 0

    while True:
        yield x*x
        x += 1

for i in gen2():
    print(i)

    if i > 10:
        break
```

23.2 Approfondimento linguaggio Bash

23.2.1 `sed`

Il programma `sed` (`s`tream `ed`itor) è un editor speciale che permette di modificare file in modo automatico, ad esempio tramite script.

23.2.2 `awk`

Il linguaggio `awk` è un linguaggio di scripting, composto da un insieme di azioni che si possono eseguire su flussi di dati. Principalmente viene usato per processare testo e come strumento di acquisizione di dati da file.

24 CLASSIFICAZIONE DEI SISTEMI OPERATIVI (03/03/2020)

24.1 Architettura dei Sistemi Operativi

Possiamo suddividere i sistemi operativi in due macrocategorie, a seconda della loro struttura:

- Sistemi con struttura semplice
- Sistemi con struttura a livelli

24.1.1 Sistemi con struttura semplice

I sistemi a struttura semplice sono caratterizzati dall'assenza di particolari sofisticazioni, e costituiti perlopiù da un insieme di procedure cooperanti. Esempi di sistemi a struttura semplice sono MS-DOS e FreeDOS, dalle funzionalità rudimentali e che non prevedono ad esempio multitasking o protezione da codice maligno. Un altro caso di sistema operativo semplice è UNIX stesso, monolitico e suddiviso in due parti, kernel e programmi di sistema.

24.1.2 Sistemi con struttura a strati

La tipologia di sistemi a strati (o a livelli) è caratterizzata da una suddivisione in livelli delle funzionalità. Il suo vantaggio principale è la modularità, che semplifica la gestione della complessità realizzativa e del debugging, ma presenta anche svantaggi quali la difficoltà nel definire i vari livelli software e la presenza di overhead. Per ovviare all'inefficienza, è talvolta concesso a livelli superiori di accedere a funzionalità di livelli più inferiori di quello immediatamente accanto.

24.1.3 Modelli storici

Alcuni modelli astratti di sistema operativo proposti in passato sono:

- The OS (ideato da Dijkstra)
- Venus OS

24.1.3.1 Note aggiuntive

Per concludere questa sezione osserviamo infine che, come visto a lezione

- Le shell sono di rado parte integrante di un kernel, il quale dispone al più di una console di emergenza
- Alcuni sviluppatori di sistemi operativi sono soliti includere nel kernel *sottosistemi d'ambiente*, utili nella compatibilità con software di terze parti
- Con il termine *controller* si indica una componente hardware, mentre con *driver* una parte software.

24.2 Tassonomia dei kernel

Possiamo suddividere i kernel in quattro categorie:

- Kernel monolitici
- Microkernel
- Kernel ibridi
- ExoKernel

24.2.1 Kernel monolitici

I kernel monolitici sono costituiti da un'unica, folta collezione di procedure tra loro correlate, che espongono servizi all'esterno mediante chiamate di sistema. Benché siano offerte capacità di modularizzazione (es driver di dispositivi), l'immagine del kernel a runtime è memorizzata e gestita in un unico ambiente. Pertanto, sebbene le implementazioni di questi sistemi risultino solitamente molto efficienti, la presenza di un solo bug in un modulo può comportare il fallimento dell'intero sistema, rendendo i kernel monolitici una scelta meno adatta in contesti che richiedono alte misure di sicurezza.

Esempi moderni di kernel monolitici sono dati da OpenVMS, Linux, BSD, SunOS, AIX, MULTICS ecc.

24.2.2 Microkernel

I microkernel nascono come risposta alla constatazione della crescente complessità dei sistemi operativi col procedere del loro sviluppo. Per affrontare questa difficoltà, un microkernel prevede la rimozione dal suo interno di tutte le parti inessenziali al funzionamento, fornendo una gestione minimale dei processi e della memoria. Ogni servizio addizionale sarà realizzato da entità dette *server* ed erogato ad altre entità dette *client*: la loro comunicazione avviene mediante meccanismi di **message passing**. Per mantenere fede alla loro minimalità, i microkernel prevedono tipicamente solo due system call, `send()` e `receive()` per inviare e ricevere, rispettivamente, messaggi a e da un server.

L'adozione di microkernel ha una serie di conseguenze, tra cui

- Semplicità di realizzazione
- Flessibilità: l'aggiunta di un servizio comporta solo il lancio di un nuovo processo server, non la ricompilazione del kernel
- Portabilità: il codice del kernel può essere facilmente trasportato su altre architetture
- Sicurezza: il crollo di un servizio non ha ripercussioni sull'intero sistema
- Inefficienza: lo scambio di messaggi server-client comporta un certo overhead

Un esempio di microkernel è Minix: il kernel è composto da gestore di processi e task (i thread del kernel). Questi ultimi funzionano come processi demoni e fanno quindi da gestori per i messaggi in arrivo dal livello superiore.

24.2.2.1 Message passing

Una delle funzionalità di base del microkernel è sostanzialmente quella di intermediare la comunicazione tra i vari processi clienti e serventi. Essa è chiaramente implementata tramite message passing e, partendo da ciò, siamo in grado di scrivere l'API standard di gran parte dei sistemi operativi:

```
int open(char* file, ...) {
    msg = < OPEN, file, ... >
    send(msg, file-server)
    fd = receive(file-server)

    return fd
}
```

24.2.3 Kernel ibridi (o micro kernel modificati)

I kernel ibridi sono semplicemente micro kernel che mantengono parte del codice in kernel space. Mantenendoli in kernel space, non ci sono i vantaggi di affidabilità di cui godono i microkernel, ma, così come per i kernel monolitici, non c'è overhead per il message passing o il cambio di contesto (user-kernel mode). Un esempio è Windows NT.

24.2.4 ExoKernel

Sono di interesse puramente sperimentale. Ri-espongono l'interfaccia hardware ai programmi utente e si limitano ad implementare semplici controlli di sicurezza.

24.3 Macchine Virtuali

Le virtual machines sono l'implementazione via software di macchine hardware. In sostanza, emulando una particolare macchina virtuale M_V su una qualsiasi macchina reale M_0, si possono eseguire programmi scritti in L_V. Con questo meccanismo è quindi mascherata l'implementazione effettiva della macchina che l'utente sta utilizzando. Le implementazioni di macchine virtuali portano diversi vantaggi:

- Consentono di far coesistere SO differenti
- Possono far funzionare SO monotask in un sistema multitask e "sicuro"
- Possono essere emulate architetture hardware differenti

Vi sono però anche alcuni svantaggi:

- La soluzione non è efficiente
- Si presentano difficoltà nel condividere risorse

25 SCHEDULING (04/03/2020)

25.1 Macchine Virtuali (cont.)

Il linguaggio ISA (Instruction Set Architecture) interfaccia la parte hardware con quella software. Il sistema operativo solitamente può accedere, tramite questo insieme di istruzioni, direttamente alle parti della macchina (bus, controller, ecc.). Alle altre applicazioni software ciò non è permesso.

25.1.1 Modalità di virtualizzazione

Si distinguono più modalità di virtualizzazione:

- Macchina virtuale a livello di processo *(process VM)*: permette ad un programma di essere eseguito allo stesso modo su qualsiasi piattaforma. Viene eseguita come una normale applicazione all'interno di un SO ospite e supporta un singolo processo. Il suo scopo è fornire un ambiente indipendente dalla piattaforma hardware e dal SO ospite. Vengono virtualizzati sia l'hardware che il sistema operativo.
- Macchina virtuale a livello di sistema *(system VM)*: permette l'esecuzione di un completo SO, anche con un ISA diverso da quello della macchina reale. Viene virtualizzato esclusivamente e completamente l'hardware.
- Macchine virtuale a livello di system call: ***[aggiungere spiegazione]***

[Integrare le voci e sistemarle] Due esempi importanti di software di hardware virtualization sono QEMU e SVM. Il primo both PVM and SVM, cross-compiles different archs or for same (KQEMU) parsing code and removing privileged, so dynamic translation. XEN only SVM, uses para-virtualization:

25.1.2 Paravirtualizzazione

La paravirtualizzazione è una tecnica di virtualizzazione che presenta alle VM un'interfaccia simile, ma non identica, all'interfaccia hardware-software della macchina ospite. Lo scopo di quest'interfaccia modificata è ridurre il tempo trascorso ad eseguire operazioni sostanzialmente più difficili da compiere in un ambiente virtuale rispetto ad un ambiente non virtualizzato.

25.2 Progettazione dei sistemi operativi

Non esistono sistemi operativi universalmente ottimi, ma sistemi più adatti a seconda delle particolari esigenze. Nella costruzione di un sistema operativo sarà pertanto il caso di considerare

- Obiettivi e vincoli entro i quali il sistema dovrà funzionare
- Le limitazioni inferiori imposte dall'hardware
- Le limitazioni superiori imposte dai tipi di programmi che si suppone debbano essere eseguiti
- Portabilità e sua estensione

Nota: per quanto riguarda l'ultimo caso, stabilire correttamente la lista dei **parametri** utilizzati permette di ricompilare facilmente il kernel per architetture diverse.

25.3 Process Control Block

Un processo è l'insieme delle azioni rivolte al completamento di una attività. Dal punto di vista del calcolatore, per lo svolgimento di un processo è necessario un descrittore di processo (process control block, *PCB*), costituito da:

- Codice da eseguire
- Dati oggetto dell'esecuzione
- Stack per la gestione dell'esecuzione (chiamate di funzione, passaggio parametri, ecc.)
- Insieme di attributi contenente tutte le informazioni utili per l'esecuzione del codice

Le informazioni contenute nel descrittore appartengono quindi a tre macroaree, che esamineremo una per una:

- Identificazione del processo
- Stato del processo
- Controllo del processo

25.3.1 Informazioni di identificazione del processo

Come già visto nelle lezioni precedenti viene utilizzato un numero intero per identificare univocamente un processo (**process ID**). Ad esso si possono poi aggiungere gli ID dei thread logicamente collegati (figli o padre, per esempio) e l'ID dell'utente chiamante.

25.3.2 Informazioni di stato del processo

Questa sezione delle informazioni contiene tutte le informazioni relative al processo che possa ripristinarlo dopo una sospensione. Questo include valori di registri generali e speciali, come program counter e registri di stato (i registri che contengono informazioni sul risultato di un'operazione, usati poi, per esempio, per salti condizionati).

Si noti che queste informazioni riguardano lo stato di un processo nell'ultimo istante di esecuzione. Infatti, devono essere modificate solo quando il processo viene sospeso, e non a ogni operazione svolta.

25.3.3 Informazioni di controllo del processo

Possiamo ulteriormente suddividere queste informazioni in sottocategorie:

- Scheduling: stato del processo, priorità, puntatori a eventi in attesa, ecc.

- Gestione della memoria: valori dei registri base *[?]*, puntatori alle tabelle delle pagine.
- Accounting: tempi di esecuzione
- Stato delle risorse: file aperti, device allocati, ecc.
- Inter-process Communication: stato di segnali, semafori.

25.4 Scheduling

È la componente più importante del kernel, poiché gestisce l'avvicendamento dei processi. Lo scheduler decide quale processo deve essere in esecuzione ad ogni istante.

25.4.1 Mode switch e context switch

Quando viene generato un interrupt avviene il *mode switch* da user mode a kernel mode.

Dopo l'esecuzione della prima parte di codice per la gestione dell'interrupt, viene chiamato lo scheduler: questo può decidere eventualmente (in base alle varie necessità) di eseguire un altro processo, nel qual caso avviene un *context switch*. Durante un *context switch* viene salvato lo stato del processo corrente `P1` nel PCB corrispondente, e lo stato del processo `P2`, selezionato per l'esecuzione, viene caricato nel processore dal PCB corrispondente. Al termine di `P2` il controllo torna in mano allo scheduler e, in assenza di altri interrupt, riprende l'esecuzione di `P1`.

Si noti che una system call non è necessariamente bloccante (es: `getpid()`). Non è detto infatti che una richiesta al sistema operativo causi un cambiamento dello stato di un processo. E, in tal caso, il processo corrente al momento dell'interrupt non verrà quindi inserito in una coda di attesa.

Al momento di una chiamata di sistema, il rispettivo codice viene considerato parte di quello del processo corrente. Questa parte tuttavia sarà in kernel mode. Al momento della richiesta di passaggio alla modalità kernel, il processo chiamante cede il controllo ad una routine apposita del sistema operativo.

26 MULTITHREADING - SCHEDULING (10/03/2020)

26.1 Multithreading

Se ogni processo avesse una sola linea di controllo non potrebbe eseguire due diverse attività contemporaneamente, come, ad esempio, scaricare due pagine differenti in un browser web. Per poter eseguire più parti di un processo in maniera concorrente è stato introdotto il concetto di ***thread***. In un processo *multithreaded* esistono molte linee di controllo, ognuna delle quali può eseguire diverse istruzioni. Esse vengono quindi eseguite concorrentemente e condividono parametri e risorse.

26.1.1 Implementazione

[da verificare, guardare su https://en.wikipedia.org/wiki/Thread_(computing)#Processes,_kernel_threads,_user_threads,_and_fibers]

Lo scheduling può avvenire a livello kernel o a livello user: a livello kernel, un processo contiene uno o più kernel thread, che condividono le risorse del processo - in questo senso quindi un processo è l'insieme delle risorse, mentre un thread è l'insieme di scheduling e esecuzione. Anche un processo a livello utente può utilizzare thread multipli di esecuzione: questi sono chiamati *user thread*, e possono essere implementati/eseguiti dal kernel in vari modi. Semplifichiamo quindi questa differenza e esaminiamo i metodi di esecuzione:

- Kernel thread: unità leggera di kernel scheduling, ne esiste almeno uno in ogni processo. Se ne esistono molteplici all'interno dello stesso processo, condividono la memoria e le risorse. Cambiare tra kernel thread dello stesso processo è meno costoso che cambiare processo, e crearli e distruggerli è similmente poco costoso. Il kernel può associare un kernel thread a ciascun core **logico**, e può sostituire i thread che vengono bloccati.
- User thread: i thread possono essere implementati anche a livello user, grazie ad apposite librerie, e quindi in userspace. Gli user thread sono gestiti e schedulati solo in userspace, quindi il kernel non sa della loro esistenza. Gli user thread sono veloci da creare e gestire, ma non possono avvantaggiarsi di multithreading o sistemi multiprocessore; inoltre, se tutti i kernel thread su cui sono basati vengono bloccati, l'user thread corrispondente sarà bloccato anche se altri user thread sono ready. Vi sono vari modi per mappare user thread a kernel thread, soprattutto per poter utilizzare macchine multiprocessore:
  - One-to-one: Ogni user thread è mappato su un kernel thread, può dare problemi di scalabilità
  - Many-to-one: Generalmente usato da sistemi che non supportano più kernel thread, più user thread vengono mappati su un unico kernel thread

– Many-to-many: Riassume i benefici di entrambe le architetture, gestisce vari user thread con un numero diverso di kernel thread

Possiamo riassumere i vantaggi e gli svantaggi che caratterizzano queste implementazioni:

- Thread a livello utente: Implementazione molto efficiente, svantaggio: una system call bloccante bloccherebbe tutti i thread del processo, quindi bisognerebbe evitarle.
- Thread a livello kernel: Implementazione molto più pesante, completamente trasparenti, creiamo il thread e ci penserà lo scheduler del kernel, e se facessimo delle chiamate bloccanti, bloccherebbe solo il singolo thread.

26.2 Scheduling

Nella prima fase della vita di un Sistema Operativo viene creato il processo `init`, che genera il descrittore di un processo. In seguito a ciò, il processo iniziale non tornerà mai più in esecuzione. Le varie funzionalità del kernel saranno infatti portate avanti da appositi processi e dallo scheduler.

Oggi, nella maggior parte dei sistemi, i processi sono organizzati in una gerarchia ad albero. Questa strutturazione permette l'ereditarietà dei parametri per i thread, evitando pesanti ed inutili ripetizioni e conseguente spreco di memoria.

26.2.1 Diagrammi di Gantt

I diagrammi di Gantt permettono la rappresentazione grafica degli schedule, illustrando lo stato d'avanzamento nel tempo. Si possono usare anche nel caso di schedule di più risorse, ad esempio un sistema multiprocessore: il diagramma risulterà composto da più righe parallele.

26.2.2 Tipologie principali di scheduler

Si possono distinguere due categorie principali di scheduler. Per poterle descrivere è necessario prima analizzare gli eventi che possono causare un context switch:

1. quando un processo passa da *running* a *waiting* (per es. in seguito ad una system call bloccante o una procedura di I/O)
2. quando un processo passa da *running* a *ready* (a causa di un interrupt)
3. quando un processo passa da *waiting* a *ready*
4. quando un processo

Si noti quindi che:

- nelle condizioni 1 e 4 l'unica scelta possibile è quella di selezionare un altro processo da eseguire
- nelle condizioni 2 e 3 è possibile continuare ad eseguire il processo corrente

In base a queste situazioni si definiscono le due principali tipologie di scheduler:

- **non-preemptive** o **coperativa**: il context switching avviene solo nelle condizioni 1 e 4, ovvero quando l'assegnatario attuale della risorsa ne cede il controllo volontariamente
- **preemptive**: il context switching può avvenire in qualsiasi situazione. E' la tecnica utilizzata in tutti i sistemi moderni dato che, sebbene più complicata da programmare, permette di utilizzare al meglio le risorse.

26.2.3 Algoritmi di scheduling

I criteri di scelta del processo da eseguire si basano su questi aspetti:

- massimizzazione di
  - utilizzo della CPU
  - throughput (numero di processi completati per unità di tempo)
- minimizzazione di:
  - tempo di turnaround (che intercorre tra la sottomissione di un processo e il suo completamento)
  - tempo di attesa (che un processo impiega nella coda ready)
  - tempo di risposta (che intercorre tra la sottomissione di un processo e la sua prima risposta - reazione ad un input)

Un altro aspetto importante è quello della caratterizzazione. Durante l'esecuzione di un processo si alternano peridodi di attività svolte dalla CPU (*CPU burst*) e periodi di attività di I/O (*I/O burst*). I processi si dicono *CPU bound* o *I/O bound* nel caso siano caratterizzati da lunghi CPU burst o I/O burst.

26.2.3.1 First Come, First Served (FCFS)

[completare, slide 48] Una problematica causata da questa modalità è quella del *convoy effect* (trad. *effetto convoglio*). Si verifica quando ad un complesso processo `P1` devono seguire uno o più processi `Px` molto brevi. Essi sarebbero soggetti a lunghe ed inutili attese.

26.2.3.2 Shortest Job First (SJF)

Assegnando la CPU al processo ready con la minima durata del CPU burst successivo si ottiene un algoritmo ottimale rispetto al tempo di attesa. Siccome è impossibile sapere esattamente la durata del CPU burst successivo, quest'algoritmo è impossibile da implementare in pratica. Si possono fornire delle approssimazioni mediante euristiche. Supponendo che un sistema nel futuro prossimo si comporti in modo simile a quanto ha fatto nel passato prossimo, si può tenere conto dei CPU burst precedenti per prevedere quando ci sarà il successivo CPU burst.

26.2.3.2.1 Calcolo approssimato della durata del CPU burst

Il calcolo si basa sulla media pesata tra il tempo dell'ultimo CPU burst e la previsione della sua durata (detta *media esponenziale*): sia t_n il tempo dell'n-

esimo CPU burst e τ_n la corrispondente previsione. τ_{n+1} può essere calcolato come segue:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Il valore $\alpha \in (0, 1)$ è un parametro regolabile che rappresenta il peso relativo della storia passata e di quella immediatamente recente. Si usa per tarare il meccanismo affinché non sia troppo suscettibile a variazioni improvvise.

26.2.3.3 Round-Robin Scheduling

Quest'algoritmo è basato sul concetto di quanto di tempo (*time slice*). Un processo, quando diventa *running*, non può rimanere in esecuzione per un tempo superiore alla durata di quanto di tempo. Se il processo esaurisce il proprio quanto di tempo senza completare il proprio CPU burst, viene aggiunto in fondo alla coda dei processi *ready*. Un processo può anche lasciare il processore volontariamente, in seguito ad un'operazione di I/O. La durata del quanto di tempo é regolabile. Un quanto di tempo breve causa scarsa efficienza perché il sistema deve cambiare il processo attivo molto spesso. Un quanto di tempo lungo, invece, può causare lunghi periodi di inattività in presenza di numerosi processi pronti. L'algoritmo di Round-Robin fornisce le stesse possibilità di esecuzione a tutti i processi: questo potrebbe rappresentare un problema per processi con priorità alta.

26.2.3.4 Priority Scheduling

Il funzionamento della modalità Round-Robin potrebbe rappresentare un problema per vari processi quali, ad esempio, la riproduzione di un video. *[aggiungere dettagli -slide 61]* Per evitare "imbrogli" da parte dei processi utente, i sistemi Unix-like permettono un numero massimo di priorità. Questo numero, inoltre, può essere solo decrementato. *[aggiungere definizioni di statica e dinamica]* Sostanzialmente la modalità più vantaggiosa è quella dinamica. Questa infatti permette di evitare che processi con priorità bassa cadano in starvation.

26.2.3.4.1 Aging

Questa tecnica consiste nell'incrementare gradualmente la priorità dei processi in attesa, fino ad un limite prefissato. In seguito ad un incremento e ad un successivo CPU-burst, il numero di priorità viene resettato. Questo evita starvation dei processi a priorità minore.

26.2.3.5 Multilvel Scheduling

Consiste nel stabilire una modalità per ogni classe di processo, in base quindi al tipo di performance richiesta al sistema. *[completare - slide 66]*

27 LEZIONE (11/03/2020)

28 RICONOSCIMENTI

Si ringraziano gli studenti dell'Alma Mater - Università di Bologna al corso di Sistemi Operativi dell'a.a. 2019-2020, nonché il titolare del corso professor Renzo Davoli.