Politecnico di Torino
Laurea Magistrale in Ingegneria Informatica

appunti di

Programmazione di sistema

Autori principali: Lorenzo David, Luca Ghio
Docenti: Gianpiero Cabodi, Giovanni Malnati
Anno accademico: 2013/2014
Versione: bozza
Data: 6 agosto 2014

Riferimenti

Le note a margine si riferiscono a questo codice sorgente.

Ringraziamenti

Oltre agli autori precedentemente citati, quest'opera può includere contributi da opere correlate su WikiAppunti e su Wikibooks, perciò grazie anche a tutti gli utenti che hanno apportato contributi agli appunti *Programmazione di sistema* e ai libri *Progetto di sistemi operativi* e *Programmazione di sistema*.

Informazioni su quest'opera

Quest'opera è pubblicata gratuitamente. Puoi scaricare l'ultima versione del documento PDF, insieme al codice sorgente LaTeX, da qui: http://lucaghio.webege.com/redirs/20

Quest'opera non è stata controllata in alcun modo dai professori e quindi potrebbe contenere degli errori. Se ne trovi uno, sei invitato a correggerlo direttamente tu stesso realizzando un commit nel repository Git pubblico o modificando gli appunti *Programmazione di sistema* su WikiAppunti, oppure alternativamente puoi contattare gli autori principali inviando un messaggio di posta elettronica a artghio@tiscali.it o a lorenzodavid91@gmail.com.

Licenza

Indice

Parte I

Progetto di sistemi operativi (Cabodi)

Capitolo 1

Memoria principale

1.0.1 Gestione della contesa tra le pagine

- Global replacement: scelgo il frame da rimpiazzare tra i frame di tutti processi.
  L'esecuzione di un processo varia in funzione di quanti altri processi ci sono nel sistema, dunque risulta non ripetibile.
- Local replacement: un processo può rimpiazzare frame solo tra i frame che gli son stati allocati
  Questo può portare ad un sottoutilizzo delle risorse.

1.0.2 NUMA - Non-Uniform Memory Access

Prevede che non tutti gli indirizzi hanno lo stesso costo di accesso, ma ad un dato core vi sono indirizzi più facilmente raggiungibili.

1.0.3 Thrashing

L'overhead impiegato per scambiare pagine tra memoria e disco per lo swapping risulta inefficiente. Occorre dunque ridurre il livello di multiprogrammazione.

Paginazione a richiesta identifico cosa serve secondo il principio dei riferimenti e lo tengo in memoria principale.

1.1 Working-Set Model

All'istante t_i osservo le ultime pagine su a cui il processo ha fatto accesso. Le pagine del working-set vengono portate in memoria principale (frame), ed è detto resident set. È importante dimensionare bene l'intervallo di tempo δ, con cui si osserva il passato.

Strategie per tenere traccia del working-set:

- aggiorno il working-set solo quando un page-fault occorre
- utilizzo di un timer con tempo fisso

1.1.1 Page fault frequency

Tre casi:

- alto: il processo non ha abbastanza frames
- medio: il processo ha il giusto numero di frames
- basso: al processo è possibile rubare qualche frames

1.1.2 Memory Mapped Files

Far corrispondere delle aree di memoria ad uno o più file su disco. Questo può essere impiegato per permettere la condivisione di risorse tra due o più processi concorrenti (nota: solo i thread hanno risorse condivise).

1.2 Allocatori

1.2.1 Buddy System Allocator

Alloco a multipli di 2 per ridurre la frammentazione.

1.2.2 Slab Allocator

Sistema gerarchico. Vettore di pagine, che raccoglie oggetti di base. Uno slab può essere: full, empty, partial.

Capitolo 2

File-system

2.1 File-system interface

1. File Concept
2. Access Methods
3. Disk and Directory Structure
4. File-System Mounting
5. File Sharing
6. Protection

2.2 File Concept

Accesso contiguo (o sequenziale) ad uno spazio di indirizzo, come se fosse un vettore di byte.

Un file contiene meta-dati (e.g. nome, size, physical pointer, etc.) e su di esso si possono effettuare operazioni (e.g. open, read, close, etc.).

2.2.1 Open Files

Il sistema operativo immagazzina le seguenti informazioni:

- **Open-file table**
- **File-open count** (conta il numero di aperture dello stesso file, per permettere la rimozione di tale file qualora l'ultimo processo lo chiuda)

2.2.2 Open File Locking

Un file è una risorsa condivisa, il sistema operativo garantisce la sincronizzazione e protezione:

- Shared lock
- Exlusive lock

2.2.3 File Structure

Organizzazione dei record:

- Simple record structure (e.g. lines, fixed length, variable length)
- Complex structure (e.g. formatted document, relocatable load file, etc.)
- Mixed approach (può simulare le precedenti due)

2.3 Access Methods

- Sequential access
  - tripartito: begining, current position, end
  - operazioni: rewind, read, write

  Mediante il riposizionamento (e.g. fseek) può diventare ad accesso diretto.
-
- Direct acess
  - operazioni: read n, write n, position to n, rewrite n

  Mediante un puntatore può diventare sequenziale.
-

Un file può diventare un indice per un altro.

2.4 Disk and Directory Structure

- Partizioni
- RAID
- Raw/Formatted (Volume: disco formattato, contenente informazioni sul *device directory* e una *volume table of contents*)

2.4.1 (Logically) Organize the directory

- Efficiency: localizzare un file rapidamente a partire da un naming
- Naming: definito da uno o più utenti
  - Due utenti possono avere lo stesso nome per file diversi
  - Lo stesso file può avere più nomi
- Grouping: raggruppare secondo determinate proprietà i files

2.4.2 Acyclic-Graph Directories

I direttori possono avere più livelli, introducendo un nome composto. Il problema della ricerca non è trascurabile. In realtà un file system moderno non è un albero, ma un DAG (Direct Acyclic-Graph).

Come si garantisce che non ci siano cicli?

- Permettere solo links a file e non sotto-direttori
- Implementare un meccanismo di Garbage Collection (temporaneamente accettare situazioni vietate, e ogni tanto metto a posto)
- Ogni volta che si aggiunge un link si usa un algoritmo per identificare i cicli

2.4.3 File-System Mounting

Un file system che non è "mounted" non è visibile all'utente.

2.4.4 File Sharing

- Client-Server
- Distributed naming services

2.4.5 Features

- Tutti i file systems hanno una **Failure Modes**.
- Consistenza dei dati (e.g. tipo di utente, tipo di accesso).

2.5 File-syste implementation

- Application program
- Logical file system
- ...

Capitolo 3

Dispositivi di IO

- Polling
- Interrupt-request line
- Interrupt handler (può mascherare o ritardare alcuni interrupts)
- Interrupt vector

Le system call sfruttano il meccanismo degli interrupt per risolvere il caso in cui il codice di gestione della system call è trasparente all'user (e.g. può essere riallocato nel kernel).

- Character-stream or block
- Sequential or random-access
- Synchronous or asynchronous
- Shareble or dedicated
- Speed of operation
- read-write, read only, or write only
- Block I/O
- Character I/O
- ...
- Network

seek implica grossomodo accesso diretto.

- Blocking: il processo è sospeso finchè l'I/O non è completato
- Nonblocking: la call di I/O ritorna quando disponibile
- Asynchronous: il processo può girare mentre avviene la procedura di I/O

Chapter 4

OS/161 - Glossary

- thread structure (just the essential informations)

```
/* Thread structure. */
struct thread {
  char *t_name;                   /* Name of this thread */
  const char *t_wchan_name;       /* Name of wait channel, if sleeping */
  threadstate_t t_state;          /* State this thread is in */

  /*
   * Thread subsystem internal fields.
   */
  struct thread_machdep t_machdep;  /* Any machine-dependent goo */
  struct threadlistnode t_listnode; /* Link for run/sleep/zombie lists */
  void *t_stack;                    /* Kernel-level stack */
  struct switchframe *t_context;    /* Saved register context (on stack) */
  struct cpu *t_cpu;                /* CPU thread runs on */

  /*
   * Interrupt state fields.
   *
   * Exercise for the student: why is this material per-thread
   * rather than per-cpu or global?
   */
  bool t_in_interrupt;            /* Are we in an interrupt? */
  int t_curspl;                   /* Current spl*() state */
  int t_iplhigh_count;            /* # of times IPL has been raised */

  /*
   * Public fields
   */

  /* VM */
  struct addrspace *t_addrspace;  /* virtual address space */

  /* VFS */
  struct vnode *t_cwd;            /* current working directory */

  /* add more here as needed */
};
```

- thread state

```
/* States a thread can be in. */
typedef enum {
  S_RUN,          /* running */
  S_READY,        /* ready to run */
  S_SLEEP,        /* sleeping */
  S_ZOMBIE,       /* zombie; exited but not yet deleted */
} threadstate_t;
```

- SPL: set priority level
- IPL: interupt priority level

```
int spl0(void)    /* sets IPL to 0, enabling all interrupts. */
int splhigh(void) /* sets IPL to the highest value, disabling all interrupts. */
int splx(int)     /* sets IPL to S, enabling whatever state S represents. */
```

- wchan: wait channel

```
/* Wait channel. */
struct wchan {
  const char *wc_name;            /* name for this channel */
  struct threadlist wc_threads;   /* list of waiting threads */
  struct spinlock wc_lock;        /* lock for mutual exclusion */
};
```

Chapter 5

OS/161 - Thread fork

5.1 Preparing the new thread

menu.c:171 `common_prog()`: it prepares a new thread and waits for it to close

menu.c:140 1. `thread_fork_sem()`: it prepares a new thread

thread.c:495 1. `thread_create()`: it sets parameters in `struct thread_t` (status `S_READY`)

thread.c:504 2. `kmalloc()`: it allocs the stack (4096 bytes)

thread.c:509 3. `thread_checkstack_init()`: it sets magic numbers on the bottom end of the stack, so that overflows will be able to be checked later

thread.c:535 4. `switchframe_init()`: it initializes some things for the switch context

switchframe.c:91 1. it sets the entry point, which is function `cmd_progthread()`

```
sf->sf_s0 = (uint32_t)entrypoint;
```

switchframe.c:92 2. it sets the command-line arguments

```
sf->sf_s1 = (uint32_t)data1;
```

switchframe.c:93 3. it sets the command-line argument count

```
sf->sf_s2 = (uint32_t)data2;
```

switchframe.c:94 4. it sets the function which instruction `j ra` will jump to, which is function `mips_threadstart()`

```
sf->sf_ra = (uint32_t)mips_threadstart;
```

thread.c:538 5. `thread_make_runnable()`: it inserts the process into the run queue (at tail)

menu.c:149 2. `P()`: it waits for the new thread to close through a semaphore

synch.c:123 1. `wchan_sleep()`: it puts the current thread, which here is the main thread, into the sleep status, then picks the new thread from the run queue (please go to section 7.1)

5.2 Starting the new thread

switch.S:97 `mips_threadstart()`: it is the first assembler instruction which the new thread executes

threadstart.S:65 1. `thread_startup()`: it is the first C instruction which the new thread executes

thread.c:768 1. `exorcise()`: it cleans up dead (zombie) threads, if any (unuseful in this case)

thread.c:785 2. `cmd_progthread()`: it is the entry point (in kernel space, address `0x8002ff00`)

menu.c:104 1. `runprogram()`: it loads the program and starts running it

1. it loads the ELF file (please go to section 5.3)

runprogram.c:99 2. `enter_new_process()`: it switches to user mode by calling a bunch of assembly code (such as `mips_usermode()`, `asm_usermode()`), and it never returns (because the thread will be destroyed by the main thread at the end)

5.3 Loading the ELF file

runprogram.c:62 1. `vfs_open()`: it opens the ELF file, that is it gets a vnode associated to it

vfspath.c:81 1. `vfs_lookup()`: it translates the path name of the user program to a vnode

vfspath.c:96 2. `vnode_incopen()`: it increments the open count of the vnode

runprogram.c:71 2. `as_create()`: it creates a new empty address space

dumbvm.c:203 1. `kmalloc()`: it allocs a new `struct addrspace`

dumbvm.c:208 2. it initializes to 0 all the fields in the `struct addrspace`

runprogram.c:78 3. `as_activate()`: it makes the specified address space the one currently "seen" by the processor (currently not fully implemented: parameter `as` is unused)

dumbvm.c:239 1. `tlb_write()`: it resets TLB entries;

runprogram.c:81 4. `load_elf()`: it loads the ELF file into the current address space

loadelf.c:166 1. `VOP_READ()`: it reads the ELF header, that is the beginning of the file, and puts it into the `eh` buffer

loadelf.c:216 2. first `for` loop: it goes through the list of segment headers to get segment sizes and set up the address space

loadelf.c:220 1. `VOP_READ()`: it reads the next segment header

loadelf.c:235 2. it filters loadable segments from other types

```
case PT_LOAD: break;
```

loadelf.c:242 3. `as_define_region()`: it sets up a region of memory within the address space

dumbvm.c:253 1. it aligns the base with the frame beginning

```
vaddr &= PAGE_FRAME; // PAGE_FRAME = 0xfffff000 =
    11111111111111111111000000000000
```

loadelf.c:256 2. it aligns the limit with the frame size

```
sz = (sz + PAGE_SIZE - 1) & PAGE_FRAME;
```

loadelf.c:258 3. it computes the number of pages

```
npages = sz / PAGE_SIZE; // PAGE_SIZE = 4096
```

loadelf.c:265 4. it assigns the aligned size and the number of pages to the code segment (region 1) or to the data segment (region 2) in the address space

```
if (as->as_vbase1 == 0) { // code segment
   as->as_vbase1 = vaddr; // base virtual address of code segment
   as->as_npages1 = npages; // size (in pages) of code segment
   return 0;
}
if (as->as_vbase2 == 0) { // data segment
   as->as_vbase2 = vaddr; // base virtual address of data segment
   as->as_npages2 = npages; // size (in pages) of data segment
   return 0;
}
```

loadelf.c:252 3. `as_prepare_load()`: it steals some memory (in number of pages) and sets base physical addresses of segments

```
as->as_pbase1 = getppages(as->as_npages1); // base physical address of code
    segment
as->as_pbase2 = getppages(as->as_npages2); // base physical address of data
    segment
as->as_stackpbase = getppages(DUMBVM_STACKPAGES); // base physical address
    of stack
```

loadelf.c:261 4. second `for` loop: it goes through the list of segment headers to actually load segments

loadelf.c:265 1. `VOP_READ()`: it reads the next segment header

loadelf.c:280 2. it filters loadable segments from other types

```
case PT_LOAD: break;
```

loadelf.c:287 3. `load_segment()`: it loads the segment at the specified virtual address of the specified size, located on disk at the specified file offset of the specified length;

loadelf.c:295 5. `as_complete_load()`: it is called when loading from an executable is complete (currently not implemented)

loadelf.c:300 6. it sets the new entrypoint in userspace (in user space, address `0x4000b0`), that is the start address of the user program

runprogram.c:89 5. `vfs_close()`: it closes the vnode associated to the ELF file

runprogram.c:92 6. `as_define_stack()`: it sets up the stack region in the address space

dumbvm.c:332 1. it returns the initial stack pointer for the new process

```
*stackptr = USERSTACK; // USERSTACK = MIPS_KSEG0 = start address of KSEG0
```

Chapter 6

OS/161 - Thread exit

From function `thread_exit()`:

1. decrements the reference counter in the current working directory
2. it calls thread_switch(threadstate_t newstate, struct wchan NULL) passing it the flag "S_ZOMBIE".
3. set interupts off (splhigh)
4. call to function `threadlist_addtail()`: it adds the current thread to the list of zombie threads;
5. the FIFO scheduler picks up a new thread from RUN_QUEUE. In case no process are found, cpu_idle is called.
6. switchframe_switch that performs (by assembly) the context switch
7. thread_switch calls the exorcise that calls: thread_destroy, kfree and eventually free_kpages (not implemented by default).
8. restore the interupts previously setted by mean of splx(spl);
9. **thread_exit does NOT return!** After the call to thread_switch, panic("The zombie walks!");

Chapter 7

OS/161 - Thread sleep

7.1 Voluntary sleep

From function `wchan_sleep()`:

1. it calls thread_switch(threadstate_t newstate, struct wchan *wc) passing it the flag "S_SLEEP".
2. threadlist_addtail(&wc->wc_threads, cur);
3. the FIFO scheduler picks up a new thread from RUN_QUEUE. In case no process are found, cpu_idle is called.

`thread.c:671` 4. `switchframe_switch()`: it performs the context switch

   `switch.S:64` 1. it saves registers of the thread which is going to sleep

   `switch.S:83` 2. it restores registers of the thread which is going to run

   `switch.S:97` 3. it jumps to the address saved in register `ra`, which is function `mips_threadstart()`

      ```
      j ra
      ```

   4. now execution flows are two: one for the thread which is going to sleep (below), the other one for the thread which is going to run (please go to section 5.2)

5. thread_switch calls the exorcise that calls: thread_destroy, kfree and eventually free_kpages (not implemented by default).
6. restore the interupts previously setted by mean of splx(spl);

7.2 ???

From functions `wchan_wakeone()` / `wchan_wakeall()`:

1. threadlist_remhead
2. thread_make_runnable

Chapter 8

OS/161 - Thread yield

8.1 Schedule

The function hardclock() calls the function schedule() and thread_consider_migration() periodically, which is not implemented by default. It may change the curcpu->c_runqueue order. In thread_switch, the first thread in the queue is picked up (FIFO).

```
/* The current cpu is now idle. */
curcpu->c_isidle = true;
do {
  next = threadlist_remhead(&curcpu->c_runqueue);
  if (next == NULL) {
    spinlock_release(&curcpu->c_runqueue_lock);
    cpu_idle();
    spinlock_acquire(&curcpu->c_runqueue_lock);
  }
} while (next == NULL);
curcpu->c_isidle = false;
```

8.2 Voluntary yield

Function `thread_yield()` is called by hardclock().

1. it calls thread_switch(threadstate_t newstate, struct wchan NULL) passing it the flag "S_READY".
2. ...
3. insert the thread in the run queue by mean of thread_make_runnable

Parte II

Programmazione di sistema (Malnati)

Capitolo 9

Introduzione al C++

9.1 Allocazione

L'allocazione può avvenire in tre aree:

- **Memoria globale**: variabili globali. Vengono automaticamente deallocati al termine del main.
- **Stack (local store)**: variabili locali. Vengono automaticamente deallocati al termine del blocco in cui sono definite.
- **Heap (free store)**:
  - new, new[<num_oggetti>]: alloca e inizializza sullo heap.
  - delete, delete[]: invoca il distruttore e rilascia il blocco.

Note:

- Ogni oggetto è memorizzato con un relativo **reference counter**.
- Non invocare mai in modo esplicito il distruttore.
- Costruttore e distruttore possono essere dichiarati privati. In tal caso, la classe non può essere instanziata direttamente (e.g. permette la realizzazione del patter Singleton).
- Gli attributi statici (**static**) vengono sempre allocati come variabili globali.

9.2 Costruttori e operatori

- Se non vengono ridefiniti, il compilatore definisce automaticamente il costruttore di copia e l'operatore di assegnazione.
- Si può evitare la duplicazione di un oggetto dichiarando privati il costruttore di copia e l'operatore di assegnazione, così da impedire l'utilizzo da parte di codice esterno.
- Se una classe dispone di una di queste funzioni membro, per evitare di usare quelli forniti dal compilatori è sempre bene implementare esplicitamente le altre due:
  1. costruttore di copia
  2. operatore di assegnazione
  3. distruttore

Sebbene siano due funzioni differenti, è bene che costruttore di copia e operatore di assegnazione siano semanticamente equivalenti.

9.2.1 Costruttore di copia

L'oggetto destinazione non esiste a priori, dunque il suo contenuto non deve essere rilasciato.

```
CProva(const CProva& originale);
```

9.2.2 Operatore di assegnazione

Prima della riscrittura potrebbe essere necessario effettuare delle operazioni di rilascio. Se si possiede un puntatore ad un blocco di memoria occorre rilasciarlo.

```
CProva& operator = (const CProva& originale);
```

9.3 Copia

9.3.1 Shallow copy

Duplicare il puntatore nell'oggetto destinazione.

9.3.2 Deep copy

Duplicare il blocco a cui il puntatore fa riferimento, separando così i due oggetti.

9.4 Passaggio di parametri

- **per valore**:
  - si effettua una copia sullo stack del parametro originale
  - se è un oggetto si copia l'intero oggetto
    - se non è definito un operatore di copia, vengono copiati i puntatori più esterni
    - se è definito un operatore di copia pubblico, viene invocato
    - se è definito un operatore di copia privato, il compilatore segnala che l'operazione non è possibile
    - se è definito un operatore di copia e un operatore di movimento, il compilatore decide quale dei due invocare, coerentemente con il codice che segue.
  - eventuali modifiche effettuate dalla funzione sul parametro non si ripercuotono sull'originale
- **per indirizzo**:
  - si passa una copia del puntatore all'oggetto (eventualmente NULL)
  - il destinatario deve dereferenziare il puntatore
  - la funzione chiamata può modificare l'oggetto originale
- **per referenza** (TypeName& param):
  - la variabile passata non può mai essere NULL
  - permette la modifica dell'originare, a meno di forzare il passaggio di parametri "const".

9.5 Funzioni e classi friend

I metodi membri di una classe "friend" possono accedere a metodi e variabili private della classe che li ha dichiarati tali.
La relazione di "friendship" non è ereditabile.

9.6 Ereditarietà

La classe eredita le interfacce della classe di base.

```
class Base {
  ...
};
class Der : public Base {
  Der():Base(){
    Base::baz(); // invocare un metodo della classe base
  }
  ...
};
```

9.6.1 Ereditarietà pubblica

Le interfacce ereditate sono note al di fuori della classe derivata.
Il costruttore ed il distruttore usati da new e delete sono quelli definiti nella classe derivata.

9.6.2 Ereditarietà privata

Le interfacce ereditate sono note solo all'interno della classe derivata.
Queste possono essere utilizzate solo all'interno di metodi della classe stessa.

9.6.3 Ereditarietà protetta

Le interfacce ereditate sono note solo all'interno della classe derivata. Queste sono anche note all'interno delle classi eventualmente deriate da quest'ultima.

9.6.4 Ereditarietà Multipla

Una classe può ereditare da più di una classe di base.
La sua interfaccia risulterà l'unione dei metodi contenuti nelle rispettive interfacce delel classi base uniti ai metodi propri della classe derivata.
Occorre evitare di derivare più volte dalla stessa classe.

9.7 Polimorfismo

Se classe A estende in modo pubblico la classe B, è lecito assegnare ad una variabile v di tipo B* un puntatore ad un oggetto di tipo A.
Nel caso in cui un metodo è stato ridefinito nella classe polimorfa, in funzione da come il metodo è stato dichiarato nella classe base si potrà avere:

- **Per default**, il C++ utilizza l'implementazione definita nella classe a cui ritiene appartenga l'oggetto.
- Se è stata solamente anteposta la parola chiave **virtual**, allora avviene l'effettivo polimorfismo.
- Se è stata anteposta la parola chiave **virtual** e si è posposta **= 0** , allora il metodo è un metodo astratto (e sia il metodo sia la classe si dicono astratti).

Note:

- una classe che possiede solo metodi astratti è l'equivalente dell'**interfaccia** Java.
- è opportuno che tutte le classi con funzioni virtuali abbiano un **distruttore virtuale**.

9.8 V-Table

Il C++ utilizza una V-Table per referenziare una tabella statica che contiene tante righe quanti sono i metodi virtuali dell'oggetto creato.
Ogni volta che un oggetto viene creato nella memoria allocata si aggiunge un puntatore per referenziare i metodi virtuali che questo eventualemente utilizzerà.
Se esistono più istanze di una data classe, queste condividono la stessa V-Table.

9.9 Operatori per Type Cast

- **static_cast<T>(p)** converte senza effettuare un controllo run-time.
  Permette di effettuare l'upcast (= cast da una classe derivata alla classe da cui deriva) tenendo conto degli opportuni offset nella V-Table. Permette anche di effettuare il downcast, ma si verifica un errore se si tenta di accedere a ciò che è definito solo nella classe derivata.
- **dynamic_cast<T>(p)** converte effettuando un controllo run-time che si assicura cast sicuri tra tipi di classi. Nel caso in cui si tenta di effettuare il downcast, restituisce NULL.
  - Applicato a un puntatore ritorna 0 se il cast non è valido
  - Applicato a un riferimento genera un'eccezione in caso di incompatibilità
- **reinterpret_cast<T>(p)** interpreta la sequenza di bit di un valore di un tipo come valore di un altro tipo. Questo meccanismo è analogo al C-style cast (che comunque è supportato nativamente).
- **const_cast<T>(p)** elimina la caratteristica di costante dal suo argomento.

Capitolo 10

Programmazione generica e librerie C++

10.1 Exceptions

Se un'eccezione si verifica in un blocco try, lo stack si contrae fino al blocco try (incluso). Questo comporta che tutte le variabili locali vengono distrutte ed eliminate. I blocchi catch sono esaminati nell'odine in cui sono scritit. Se non c'è alcuna corrispondenza, l'eccezione rimane attiva e il programma ritorna alla funzione chiamante, ed eventualmente al chiamante del chiamante, fino alla contrazione completa dello stack, con la terminazione del flusso di esecuzione.

10.2 Template

Il parametro può essere un identificatore per un tipo di dato o un valore costante.

```
template <class T, int size>
```

10.3 Smart pointer

Gli smart pointer possono occuparsi di chiamare la delete sul puntatore dell'oggetto incapsulato quando si esce da un blocco evitando leak di memoria anche in caso di eccezioni. In caso lo smart pointer sia condiviso tra più utilizzatori è necessario definire chi possa elimare tale risorsa:

- l'ultimo a conoscere l'oggetto
- un garbage collector che opera per conto del sistema

A partire dalla versione C++11 sono disponibili:

- std::shared_ptr<BaseType>: implementa un meccanismo di conteggio dei riferimenti
- std::weak_ptr<BaseType>: permette di osservare il contenuto di uno shared_ptr senza partecipare al conteggio dei riferimenti. Questo evita la creazione di cicli, quindi la corretta deallocazione delle risorsse.
- std::unique_ptr<BaseType>: implementa il concetto di proprietà, impedendo la copia
- std::auto_ptr<BaseType>: è stato deprecato perché la copia modifica l'originale.

10.4 Standard Template Library

10.4.1 Contenitori

- set: la chiave deve essere unica
- multiset: la chiave può essere duplicata
- map: basata su coppie chiave (unica), valore
- multimap basata su coppie chiave (anche duplicata), valore

10.4.2 Iteratori

Oggetti che indicano una posizione all'interno di un contenitore. L'accesso avviene deferenziando l'iteratore come se fosse un puntatore.

Capitolo 11

C++: funzionalità avanzate

11.1 Oggetto funzionale (o funtore)

Un oggetto funzionale è un'istanza di una qualsiasi classe che abbia ridefinito la funzione membro operator().

- È possibile includere più definizioni di operator, ma devono avere tipi differenti nell'elenco dei parametri.
- Un oggetto funzionale può contenere variabili membro, in tal caso l'output dipende dallo stato di tali variabili e l'oggetto è detto effettivamente "funzionale".

11.2 Lambda function

Un'espressione lambda è usata per evitare di definire una funzione esplicita e doverle dare un nome che potrebbe collidere con altri nomi.

```
[](int num, int den) -> double {
  if (den==0)
     return std::NaN;
  else
     return (double)num/den;
}
```

- introducono la notazione lambda
  - è possibile elencare eventuali variabili locali il cui valore o il cui riferimento si vuole rendere disponibile all'interno della funzione.

x, y - x e y sono catturate per valore (la funzione λ può essere invocata anche quando tali variabili saranno uscite dallo scope.

&x, &y - x e y sono catturate per riferimento (eventuali cambiamenti influenzano l'originale).

= - cattura tutte le variabili menzionate nell'espressione λ per valore

& - cattura tutte le variabili menzionate nell'espressione λ per riferimento

x, &y - approccio misto

- () - contiene i parametri di ingresso della funzione
- -> - indica il valore di ritorno
- - contiene il corpo della funzione

11.3 Movimento

11.3.1 Costruttore di copia e di movimento

L'operatore di movimento risparmia l'onerosa operazione di allocare un nuovo blocco e inizializzarlo, perchè riusa le componenti dell'oggetto di partenza. A meno di esplicitare la modalità di richiesta, è il compilatore a scegliere quando effettuare copia o movimento, in base all'utilizzo futuro della variabile passata.

```
/* Costruttore di copia */
simple_string(const simple_string& that) {
  size_t size=strlen(that.data)+1;
  data = new char[size];
  memcpy(data,that.data,size);
}
/* Costruttore di movimento */
simple_string(simple_string&& that) {
  data = that.data;
  that.data = NULL;
}
```

Note:

- in alcuni casi, come ad esempio il passaggio di un std::unique_ptr, la copia non è possibile mentre il movimento lo è senza violare l'unicità del puntatore.
- se un oggetto contiene solo valori elementari o tipi valore il movimento equivale alla copia.
- se i dati contenuti rappresentazo una risorsa esterna, il vantaggio può essere notevole (si pensi a un descrittore di un file o a un socket).

11.3.2 Assegnazione per movimento

```
unique_ptr& operator = (unique_ptr&& that){
  if (this!= &that) { // evita l'auto-assegnazione
    delete ptr;
    ptr=that.ptr;
    that.ptr= nullptr;
  }
  return *this;
}
```

11.3.3 Movimento esplicito

La funzione `std::move()` chiama esplicitamente il movimento forzando il compilatore a interpretare l'LValue come un RValue.

11.4 Paradigma Swap&Copy

Per evitare di scrivere delle implementazioni errate dell'operatore di assegnazione, si può utilizzare il paradigma Swap&Copy:

```
friend void swap(intArray& a, intArray& b) {
  std::swap(a.mSize, b.mSize);
  std::swap(a.mArray, b.mArray);
}

// Operatore di assegnazione
intArray& operator=(intArray that) {
  // that e' passato per valore, copiato o mosso a seconda del contesto in cui e'
      usato
  swap(*this, that);
  return *this;
}
```

```
// Costruttore di movimento
intArray(intArray&& that) : mSize(0), mArray(NULL) {
  swap(*this, that);
}
```

Capitolo 12

Programmazione concorrente in C++11

12.1 Introduzione alla programmazione concorrente

Programmazione sequenziale Ogni **processo** contiene un solo flusso di esecuzione che si svolge con un comportamento deterministico.

Processi concorrenti possono comunicare tra loro solo attraverso apposite zone di memoria condivise o primitive per la comunicazione tra processi offerte dal sistema operativo (si rimanda al capitolo 14), che sono delle soluzioni generalmente lente e poco efficienti perché i processi hanno spazi di indirizzamento separati.

Programmazione concorrente Ogni processo contiene più flussi di esecuzione, uno per ogni **thread**, che si svolgono contemporaneamente con un comportamento non deterministico (dipendente dalla politica dello scheduler).

Vantaggi della programmazione concorrente

- stesso spazio di indirizzamento: i thread all'interno di uno stesso processo condividono lo stesso spazio di indirizzamento:
  - inter-comunicazione efficiente: la comunicazione tra thread dello stesso processo avviene in maniera più efficiente rispetto alla comunicazione tra processi, in quanto i dati possono essere condivisi evitando copie onerose da uno spazio di indirizzamento all'altro;
  - commutazione di contesto efficiente: la commutazione di contesto tra thread dello stesso processo avviene in maniera più efficiente rispetto alla commutazione di contesto tra processi, in quanto non richiede la rimappatura dell'area di memoria;
- operazioni di I/O asincrone: mentre un thread si pone in attesa di I/O, un altro thread può eseguire altre parti dell'algoritmo;
- vero parallelismo: se si dispone di CPU multi-core, più flussi di esecuzione possono svolgersi contemporaneamente, riducendo il tempo totale di elaborazione.
  I thread vengono eseguiti effettivamente in parallelo se la CPU dispone di un numero di core sufficiente, altrimenti sono eseguiti in alternanza in modo non deterministico.

Svantaggi della programmazione concorrente L'accesso concorrente a dati in zone di memoria condivise può causare delle **interferenze** ⇒ l'accesso alle zone di memoria condivise deve essere coordinato attraverso opportuni costrutti di sincronizzazione:

- possono crearsi dei deadlock quando due thread si attendono a vicenda;
- gli errori nell'uso dei costrutti di sincronizzazione possono manifestarsi in maniera casuale a causa del comportamento non deterministico, rendendone il debug più complesso.

Costrutti di sincronizzazione

- UNIX:
  - libreria POSIX Threads (Pthreads): mutex, condition;
  - oggetti kernel: semafori, pipe, segnali;
- Win32: (capitoli 14 e 15)
  - CriticalSection
  - ConditionVariable
  - oggetti kernel: Mutex, Event, Semaphore, Pipe, Mailslot, ecc.;
- C++11:
  - approccio di alto livello: basato su `async` e `future`;
  - approccio di basso livello: uso esplicito di thread e costrutti di sincronizzazione.

12.2 Thread

Un oggetto `thread` modella un thread del sistema operativo.
La classe `thread` implementa i seguenti metodi:

- costruttore: crea un nuovo thread e chiama l'oggetto Callable specificato;
- funzione `join()`: il thread chiamante si blocca in attesa che il thread specificato termini;
- funzione `detach()`: il thread chiamante non è interessato ad attendere la terminazione del thread specificato.

Se il thread principale termina normalmente ritornando dal `main()`, l'intero processo termina, insieme a tutti i suoi thread secondari, e tutti i distruttori sono chiamati. In altri casi l'intero processo termina senza che vengano chiamati i distruttori ⇒ ciò può portare a perdite di memoria:

- se in un thread secondario si verifica un'eccezione non gestita;
- se viene distrutto un oggetto `thread` senza che siano state chiamate le funzioni `join()` o `detach()`;
- se il thread principale termina per cause diverse dal ritorno dal `main()`.

Spazio dei nomi `this_thread` Offre delle funzioni relative al thread chiamante:

- funzione `this_thread::get_id()`: restituisce l'identificativo del thread chiamante;
- funzione `this_thread::sleep_for()`: sospende l'esecuzione del thread chiamante per almeno il tempo specificato (`chrono::duration`);
- funzione `this_thread::sleep_until()`: interrompe l'esecuzione del thread chiamante almeno fino al momento specificato (`chrono::time_point`);
- funzione `this_thread::yield()`: il thread chiamante offre volontariamente l'esecuzione a un eventuale altro thread.

12.3 Esecuzione asincrona

12.3.1 Future

Un oggetto `future` rappresenta un risultato che non è disponibile al momento della sua creazione, ma che sarà reso disponibile in futuro.

La classe `future` implementa i seguenti metodi:

- funzione `get()`: aspetta che il risultato sia pronto e ne restituisce il valore (o lancia l'eventuale eccezione), e può essere chiamata una sola volta;
- funzione `wait()`: aspetta che il risultato sia pronto senza prelevarne il valore, e può essere chiamata più volte.
  Le funzioni `get()` e `wait()`:
  - se l'esecuzione è terminata, ritornano subito;
  - se l'esecuzione è ancora in corso, si bloccano in attesa che finisca;
  - se l'esecuzione non è ancora iniziata, ne forzano l'avvio nel thread corrente e si bloccano in attesa che finisca;
- funzioni `wait_for()` e `wait_until()`: aspettano rispettivamente per il tempo specificato e fino al momento specificato:
  - se l'esecuzione è terminata, restituiscono `future_status::ready`;
  - se l'esecuzione è ancora in corso, restituiscono `future_status::timeout`;
  - se l'esecuzione non è ancora iniziata, restituiscono `future_status::deferred`;
- funzione `share()`: permette di ottenere un oggetto `shared_future` a partire da un oggetto `future`, in modo che possa essere valutato più volte (cioè si possano effettuare più `get()` su di esso).
  Un oggetto `shared_future` è copiabile e movibile.

Quando un oggetto `future` viene distrutto, se la computazione è ancora attiva il distruttore ne attende la fine.

12.3.2 Promise

promise<T>

- get_future()

```
void f(promise<string>& p) {
  ...
}

int main() {
  promise<string> p;
  /* Creazione di un thread responsabile dell'elaborazione */
  thread t(f, ref(p)); //forza p ad essere passata per reference t.detach()
      ;
  ...
  /* Attesa del risultato */
  string result = p.get_future().get();
}
```

12.3.3 Async

Il metodo `async()` esegue un oggetto chiamabile in maniera asincrona e ritorna un `future<T>`, dove `T` è il tipo del valore di ritorno dell'oggetto chiamabile:

```
future<<T>> future1 = async(function1, "...", 3.14);
string res1 = future1.get();
```

L'oggetto chiamabile può essere preceduto da una costante che definisce la politica di attivazione:

- `launch::async`: attiva un thread secondario che esegue l'operazione;
- `launch::deferred`: l'operazione non viene eseguita finché non vengono chiamati i metodi `get()` o `wait()` sul future relativo (lazy evalutation).
  Per supportare questo tipo di comportamento, evitando molteplici chiamate su un metodo lazy, è possibile usare la classe `once_flag` e la funzione `call_once()`.

12.4 Sincronizzazione

- mutex
- recursive_mutex
- timed_mutex
- recursive_timed_mutex
- lock_guard<Lockable> (lock_guard<mutex>)
- unique_lock<Lockable> (unique_lock<mutex>)
  - adopt_lock
  - defer_lock
  - try_lock
- condition_variable (richiede l'uso di un unique_lock<mutex> per garantire l'accesso in mutua esclusione alle variabili usate per valutare la condizione di attesa).
  - wait(unique_lock)
  - wait_for()
  - wait_until()
  - notify_one()
  - notify_all()

Nota: attenzione ai risvegli spuri.

12.4.1 Operazioni atomiche

La classe atomic<T> offre la possibilità di accedere in modo atomico al tipo T.

12.5 Singleton pattern e Lazy evaluation

- once_flag: struttura di appoggio per la funzione call_once.
- call_once(flag, function, ...): esegue function una sola volta (se la chiamata è già terminata ritorna immediatamente).

```
#include <mutex>
class Singleton {
  static Singleton *instance; static once_flag inited;
  Singleton() {...} //privato
  public:
  static Singleton *getInstance() {
    call_once( inited, []() {
      instance=new Singleton(); });
    return instance; }
  //altri metodi...
};
```

Capitolo 13

Architettura dei sistemi operativi della famiglia Win32

- contesto per l'esecuzione
- ripartizione delle risorse
- meccanismo di memorizzazione permanente
- sandbox isolata per ogni processo (in modo da evitare collisioni tra processi)
- distinguere le modalità supervisore/utente
  attraverso system calls è possibile passare da una modalità all'altra
  lo switch è molto oneroso perchè richiede di cambiare lo stack

13.1 Handle

- handle opaca che permette l'accesso agli oggetti
  - oggetti user: per la gestione delle finestre (una finestra è un oggetto che ha un suo proprio handle globale di sistema)
  - oggetti grafici (gdi): per la gestione delle primitive grafiche
  - oggetti kernel: accessibili da ogni processo
- Il kernel alloca un vettore di kernel handle che sono lockate in memoria (non possono essere swappate su disco)
- int Closehandle(HObject); Rilascio di un oggetto kernel chiamando tale funzione

13.2 Gestione degli errori

Ogni richiesta al sistema operativo può fallire.

13.3 Processo

Contesto di esecuzione di un programma. Spazio di indirizzamento virtuale.

- codice
- stack dei thread
- heap

13.3.1 Spazio di indirizzamento

In ambiente a 32 bit, la memoria massima accessibile per tutti i threads è circa 2 GB. È possibile dimensionare lo stack di un thread, tenendo conto di sottodimensionamento (che può portare a stack overflow) e a sovradimensionamento (superare il limite massimo di memoria accessibile). In un'architettura a 64 bit, il problema del dimensionamento è meno rilevante, avendo molta più memoria accessibile.

13.4 Cambio di contesto tra User e Kernel

Il passaggio avviene attraverso due system call:

- SYSENTER
- SYSCALL

Tale passaggio richiede di copiare i parametri dello stack.

Le dll accessibili in modalità user sono:

1. Kernel32.dll
2. Gdi32.dll
3. User32.dll

Queste si appoggiano alla NtDll.dll.

13.5 Oggetto e Handle

Gli oggetti kernel sono accessibili solo mediante handle opaca, valida per un singolo processo. Ciascun oggetto è dotato di una propria **access control list** (ACL).

Il rilascio di un Handle avviene per mezzo di:

- int CloseHandle(HObject): tale funzione decrementa il contatore sull'oggetto.

Note:

- un oggetto è distrutto quando il suo contatore è 0
- alla creazione un thread ha reference counter pari a 2 (uno al chiamante, uno al chiamato)
- alcune handle sono ereditabili ma solo ai processi **figli**

1. Oggetti user: per la gestione delle finestre.
2. Oggetti grafici (GDI): per la gestion delle primitive grafiche.
3. Oggetti kernel: per la gestione della memoria, dell'esecuzione dei processi, per la sincronizzazione e la comunicazione tra processi (IPC).
   (a) AccessToken
   (b) Event
   (c) etc.

13.6 Gestione degli errori

- GetLastError()
- FormatMessage()

13.7 Ciclo di vita di un processo

- Creazione: CreateProcess
- Terminazione:
  - il main del primary thread ritorna al loader del S.O
  - un thread del processo esegue ExitProcess
  - un thread di un altro processo esegue una TerminateProcess
  - tutti i thread del processo terminano la loro esecuzione
- Valori di uscita del processo: GetExitProcess

13.8 Informazioni relative ad un processo

- Sicurezza (DACL)
- Risorse utilizzate
- Prioirtà
- Variabili d'ambiente
- Relazioni padre/figlio

13.9 Accesso ad un file

- CreateFile
- ReadFile(Ex)
- WriteFiel(Ex)
- CloseHandle

13.10 Thread

Tutti i thread all'interno di uno stesso processo condividono le aree di memoria, nello stesso spazio di indirizzamento, contenenti:

- il codice macchina;
- le variabili globali;
- le costanti;
- le variabili nello heap.

I thread del processo non condividono lo stack, ma ogni thread ha un proprio stack e un proprio Thread Kernel Object.

13.10.1 Thread Kernel Object

Per ogni thread il sistema operativo alloca nella sua memoria protetta un oggetto, chiamato **Thread Kernel Object**, che memorizza delle informazioni sul thread: proprietà e contesto.

Proprietà Sono delle metainformazioni sul thread, tra cui:

- usage count: è il numero di thread che conoscono il thread:
  - alla creazione del thread lo usage count vale sempre 2: il thread è conosciuto dal sistema operativo e dal thread creante;
  - la memoria del kernel viene rilasciata quando lo usage count scende a 0.
    Attraverso la funzione `CloseHandle()` si può decrementare lo usage count di un thread;
- exit code: memorizza il codice di uscita del thread.

Contesto Rappresenta una fotografia dei registri (user-mode) della CPU, tra cui lo Stack Pointer (SP) e l'Instruction Pointer (IP):

- quando lo scheduler sospende l'esecuzione del thread, vengono salvate le informazioni di contesto;
- quando lo scheduler riprende l'esecuzione del thread, vengono ripristinate le informazioni di contesto.

13.10.2 Schedulazione

La schedulazione è a coda circolare (round-robin), con quanti di tempo (10 ms) assegnati a un thread in funzione della sua priorità. La schedulazione è con diritto di prelazione (pre-emptive): thread a priorità più alta possono sottrarre la CPU a thread a priorità più bassa.

Lo scheduler non è un modulo unico, ma è implementato da un insieme di funzioni del kernel che formano il **message dispatcher**.

Passaggi di stato Un thread può assumere i tre stati running, waiting e ready in diversi modi:

- ready → running: lo scheduler avvia o riprende l'esecuzione del thread;
- running → ready:
  - il thread termina il suo quanto di tempo;
  - avviene la prelazione da parte di un thread a priorità più alta;
  - il thread cede volontariamente l'esecuzione a un altro thread (yield).
    Siccome in ambiente Win32 non esiste una primitiva di yield, essa può essere simulata chiamando la funzione `Sleep()` con il parametro `dwMilliseconds` impostato a 0;
- running → waiting:
  - il thread effettua un'operazione di I/O (da file o dalla rete);
  - il thread chiama una funzione di attesa (`WaitForSingleObject()`, `WaitForMultipleObjects()`) (si rimanda alla sezione 15.2.2);
- waiting → ready:
  - l'operazione di I/O o la funzione di attesa sono terminate;
  - un altro thread chiama la funzione `ResumeThread()`;
- ready → waiting: un altro thread chiama la funzione `SuspendThread()` (sospensione forzata).

13.10.3 Creazione di thread

La funzione `CreateThread()` permette di creare un thread:

```
HANDLE WINAPI CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes, SIZE_T
    dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress, LPVOID lpParameter, DWORD
    dwCreationFlags, LPDWORD lpThreadId);
```

Parametri

- `lpThreadAttributes`: il contesto di sicurezza (predefinito = valore `NULL`);
- `dwStackSize`: la dimensione massima dello stack (predefinito = 1 MB, valore `0`);
- `lpStartAddress`: l'indirizzo della funzione principale (stile C) da cui il thread deve partire;
- `lpParameter`: un eventuale parametro da passare alla funzione principale del thread;
- `dwCreationFlags`:
  - valore `0`: il thread deve essere attivato appena possibile (il thread entra nella coda dello scheduler);
  - valore `CREATE_SUSPENDED`: il thread verrà attivato in seguito con `ResumeThread()`;
- `lpThreadId` [retval]: l'identificativo del thread.

Valore di ritorno L'handle[1] del thread.

Wrapper Le librerie standard del C sono state progettate per un sistema mono-thread ⇒ le variabili globali sarebbero condivise tra tutti i thread e verrebbero accedute senza sincronizzazione. Per la funzione `CreateThread()` sono disponibili due wrapper, `_beginthread()` e `_beginthreadex()`, che garantiscono che le variabili globali delle librerie standard del C non siano condivise tra i thread.

13.10.4 Terminazione di thread

Un thread può terminare in diversi modi:

- la funzione principale del thread ritorna (senza eccezioni);
- funzione `ExitThread()` (wrapper: `_exitthread()`, `_exitthreadex()`): il thread termina se stesso;
- funzione `TerminateThread()` (wrapper: `_endthread()`, `_endthreadex()`): un altro thread all'interno dello stesso processo termina il thread;
- funzione `ExitProcess()`: il processo termina se stesso e tutti i thread al suo interno;
- funzione `TerminateProcess()`: un altro processo termina il processo e tutti i thread al suo interno;
- la funzione principale del thread ritorna con un'eccezione, e il codice di startup (in NTDLL.dll) chiama la funzione `ExitProcess()`.

[1] Si veda la sezione 13.1.

Perdite di memoria Le funzioni `ExitThread()`, `TerminateThread()`, `ExitProcess()` e `TerminateProcess()` non chiamano i distruttori degli oggetti allocati nello stack del thread ⇒ si potrebbero verificare delle perdite di memoria. Quando il thread principale intende terminare l'esecuzione di un thread secondario, occorre perciò evitare perdite di memoria sfruttando ad esempio una **variabile condivisa** di tipo booleano, che deve essere ispezionata periodicamente dal thread secondario:

1. il thread principale imposta la variabile condivisa al valore `false`;
2. il thread principale attende la terminazione del thread secondario con una funzione di attesa (`WaitForSingleObject()`, `WaitForMultipleObjects()`) (si rimanda alla sezione 15.2.2);
3. il thread secondario, quando si accorge che la variabile condivisa è impostata al valore `false`, ritorna dalla funzione principale appena possibile.

Capitolo 14

Comunicazione tra processi

14.1 Perchè è necessario fare IPC?

Perchè i processi vivono in spazi di indirizzamento diversi. Il sistema operativo richiede meccanismi per superare tale barriera.

14.2 Mailslot

Comunicazione asincrona monodirezionale, secondo un modello client-server. Un processo può essere client e server contemporaneamente.

Esiste un broadcast (non a tutte le mailslot di uno stesso elaboratore) tra PC appartenenti a un dominio comune.

14.3 Mailslot

Architettura Client-Server asincrona.

- GetMailslotInfo: restituisce il numero di messaggi accodati e la dimensione del primo da leggere
- CloseHandle: chiude la mailslot e distrugge i messaggi in essa contenuti.
- CreateFile: apre la mailslot.
- WriteFile: scrivono il messaggio.

14.4 Pipe

14.4.1 Anonymous Pipe

- CreatePipe
- ReadFile (operazione bloccante)
- WriteFile (operazione bloccante)

14.4.2 Named Pipe

Occorre fornire un nome (come per le mailslot). Il funzionamento viene gestito come per i files.

- CreateNamedPipe
- OpenFile
- CallNamedPipe

14.5 File Mapping

Meccanismo adatto per condividere ampie zone di memoria e a realizzare aree condivise persistenti. Occorre creare almeno un mutex per gestire la concorrenza su tale file mapping.

- CreateFileMapping: crea un file mapping (con i diritti di accesso voluti), se esiste già un file mapping con il nome passato, ne ritorna l'handle.
- MapViewOfFile: crea una vista del file mapping nello spazio di indirizzamento del processo. Ritorna un puntatore che può essere direttamente usato per accedere all'area di memoria condivisa.
- UnmapViewOfFile: rilascia la vista sul file mapping (ma non distrugge il file).
- CloseHandle

Capitolo 15

Primitive di sincronizzazione Win32

15.1 Programmazione concorrente in win32 - oggetti user

15.1.1 Funzioni Interlocked

Costrutto più semplice di sincronizzazione basato su operazioni assembler atomichee.

- LONG InterlockedIncrement(LONG volatile *Addend): incrementa di un'unità il valore a cui punta il parametro e restituisce il risultato
- LONG InterlockedAdd(LONG volatile *Addend): permette incrementi arbitrari
- LONG InterlockedCompareExchange(LONG volatile *Addend): inserisce il valore New nella cella puntata da Dest se questa contiene il valore Exepcted. Restituisce il valore puntato da Dest.

15.1.2 Sezioni critiche

All'interno dell'oggetto CRITICAL_SECTION, il sistema operativo memorizza un riferimento ad un oggetto kernel. È necessario deallocare la CRITICAL_SECTION per evitare progressivi leak di memoria.

Critical section object

- InitializeCriticalSection()
- InitializeCriticalSectionAndSpinCount(): fa polling sulla critical section per un certo numero di millisecondi, migliorando potenzialmente le prestazioni in un sistema multi-core.
- DeleteCriticalSection()
- WakeConditionVariable()
- WakeAllConditionVariable()
- SleepConditionVariableCS

Slim reader writer lock (SRW)

Permettono **N reader** e **1 writer**.

- SRWLOCK
- SleepConditionVariableSRW

15.2 Programmazione concorrente in win32 - oggetti kernel

Accanto ai costrutti di sincronizzazione a livello utente, win32 offre una ricca serie di meccanismi basati sull'utilizzo di oggetti kernel, che richiedono tempi di attesa significativamente maggiori poichè richiedono un passaggio alla modalità supervisore. Hanno tuttavia alcuni vantaggi:

- una semantica più estesa
- permettono la sincronizzazione anche tra thread di processi differenti

15.2.1 Stato di segnalazione

- segnalato: indica che l'elaborazione è in corso. Non è possibile tornare indietro fino a quando l'elaborazione non sia completamente terminata
- non segnalato: indica che l'elaborazione è terminata

15.2.2 Funzioni di attesa

Funizoni che pongono un thread in attesa fino a che uno o più oggetti kernel assumono lo stato segnalato.

- WaitForSingleObject()
- WaitForMultipleObjects()

```
HANDLE hs[3];
hs[0] = hProcess1;
hs[1] = ...
hs[2] = ...
DWORD ret = WaitForMultipleObjects(
                    3, // number of objs to wait on
                    hs, // pointer to array of objs
                    FALSE, // wait for just one
                    5000); // timeout
switch (ret) {
  case WAIT_FAILED: ...
  case WAIT_TIMEOUT: ...
  case WAIT_OBJECT_0 + 0: ...
  case WAIT_OBJECT_0 + 1: ...
  case WAIT_OBJECT_0 + 2: ...
}
```

Note:

- entrambe le funzioni possono avere un *timeout*
- l'attesa su più oggetti è atomica
- il numero massimo di oggetti su cui si può eseguire un'attesa è MAXIMUM_WAIT_OBJECTS (64)

Alcuni oggetti kernel (thread e processi) mantangono il proprio stato di segnalazione indipendentemente dal fatto che altri thread siano in attesa su di essi. Altri (mutex, semafori, eventi, timer, etc.) possono tornare automaticamente allo stato non segnalato se:

- almeno un thread era in attesa che diventassero segnalati
- sono passati nello stato segnalato
- un thread di quelli in attesa è stato svegliato ed è passato nello stato "ready"

15.3 Eventi

Ci si collega ad un evento tramite:

- CreateEvent()
- OpenEvent()

Oggetti kernel il cui stato può assumere due condizioni:

- segnalato e non segnalato
- auto-reset: possono ritornare automaticamente allo stato non-segnalato quando un thread in attesa si sveglia
- manual-reset: permettono ad un numero indefinito di thread in attesa di svegliarsi

15.4 Waitable Timer Objects

- CreateWaitableTimer(): posto nello stato segnalato ad una data ora od a intervalli regolari.
- OpenWaitableTimer(): per ottenere un handle ad un waitable timer preesistente.

Oggetti kernel che sono posti nello stato segnalato ad una data ora od a intervalli regolari:

- auto-reset: solo uno dei thread in attesa diventa schedulabile
- manual-reset: tutti i thread in attesa diventano schedulabili

15.5 APC - Asynchronous Procedure Call

15.6 Semafori

Oggetti kernel che mantengono al proprio interno un contatore:

- CreateSemaphore
- OpenSemaphore

Stato

- >0: stato segnalato
- =0: stato non segnalato

Le operazioni possibili sono:

- WaitForSingleObject: decrementa il contatore se >0, altrimenti blocca il thread in attesa finchè un altro thread incrementi il contatore
- ReleaseSemaphore: incrementa il contatore

Note:

- i semafori vengono usati tipicamente quando si hanno più copie disponibili di una risorsa
- un semaforo è bene che sia associato ad un mutex
- un semaforo non sostituisce un mutex

15.7 Mutex

Assicurano l'accesso in mutua esclusione ad una singola risorsa da parte di più thread. Si comportano allo stesso modo delle sezioni critiche, ma sono kernel object.

- CreateMutex
- OpenMutex
- WaitForSingleObject
- WaitForMultipleObjects
- ReleaseMutex: decrementa il contatore e se il questo diventa zero, segnala il mutex

Capitolo 16

WinSock 2

16.1 Definizione

Un socket è identificato univocamente da:

- indirizzo rete
- tipo di protocollo utilizzato
- indirizzo trasporto e porta

16.2 Creazione di un socket

```
WSADATA wsaData;
WORD wVersionRequested = MAKEWORD(2,0); //Version 2,0
/*Inizio ad utilizzare i socket*/
WSAStartup(wVersionRequested, &wsaData);
...
SOCKET s = socket(AF_INET, SOCK_STREAM, 0);
...
/*Ho finito di usare i socket*/
WSACleanup();
```

Una volta che si è controllato che il socket funziona, è possibile:

- assegnare una porta
- attendere e accetare connesioni da clients

È possibile implementare il polimorfismo in C utilizzando il cast da un dato più generico a uno più specifico.

16.2.1 Gestion degli errori

Attraverso il valore di ritorno delle funzioni si possono ricevere idetificatori di errori quali:

- INVALID_SOCKET
- SOCKET_ERROR

In entrambi i casi si ottiene un'indicazione del tipo di errore tramite la chiamata

```
int iErrorCode = WSAGetLastError();
```

16.2.2 Il modello non bloccante

I socket offrono anche richieste non bloccanti.

Questo richiede una modifica strutturale al programma, impiegando un modello reattivo.

Le operazioni sui socket possono introdurre un'attesa indefinita: non si possono trattare protocolli full-duplex o connessioni multiple con un unico thread!

La gestion multi-thread richiede: gestione della sincronizzazione, overhead per la creazione dei threads.

Windows offre tre modelli non bloccanti

1. **Invio di messaggi ad una finestra.**
   Analogamente a come è implementata la reazione ad un evento in seguito ad un click del mouse, posso reagire ad un evento in un socket.

   ```
   int PASCAL FAR WSAAsyncSelect(SOCKET s, HWND hWnd, unsigned int wMsg,
       long lEvent);
   ```

   - s: socket
   - hWnd: la finestra a cui devno essere inviati i messagi
   - wMsg: il messaggio da inviare
   - lEvent: una bitmask che indica a quale combinazione di eventi l'applicazione sia interessata (FD_READ, FD_WRITE, FD_OOB, etc.)

2. **Segnalazione di un oggetto kernel (evento)**

3. **Invocazione di una APC (Overlapped I/O)**
   Quando hai finito un'azione, chiama una callback function.

   ```
   WSASend(...);
   WSARecv(...);
   ```

   È possibile specificare la modalità overlapped con eventualmente segnalazione di evento e/o callback functions.

   Si può determinare il ocmpletamento di un evento tramite:

   - polling (WSAOverlappedResult())
   - wait (WSAWaitForMultipleEvents())

   La callback può essere usata per la gestione asincrona degli eventi di rete basati su un **singolo thread**.

Capitolo 17

Introduzione al linguaggio C#

17.1 Linguaggio a Componenti

Voglio nascondere i dettagli *low level* dell'esecuzione.

17.1.1 Autoboxing

Migliora l'interoperabilità dei tipi fondamentali, senza le classi "wrapper" del java.

17.1.2 Garbage Collector

Tutto risiede nello heap, dunque la copia è sempre per puntatore.

17.1.3 Architettura .NET

- Windows Presentation Framework, Web Forms, Windows Forms
- Classi e dati e XML (e.g. ADO.NET, SQL, XSLT, XML, etc.)
- Classi Base del Framework
- Common Language Runtime (alternative open source: MONO)
- Piattaforma Windows

Il bytecode java è un CIL per la macchina stack based CLR. Il CIL viene compilato una seconda volta in runtime (inizialmente è un po' più lento ma poi va più veloce di un linguaggio interpretato).

17.2 Internal details

17.2.1 Indicizzatori

Array virtuali associati ad un oggetto, per i quali si esplicitano le operazioni di accesso in lettura e scrittura alle singole celle:

1. gli indici possono essere non numerici
2. sono possibili versioni *overloaded* dello stesso indicizzatore
3. sono possibili indici multidimensionali

```
public class ListBox: control{
  private string[] items;
  public string this[int index]{
    get{
      return items[index];
    }
    set{
      items[index] = value:
      Repaint();
    }
  }
}
```