Sistemi Operativi Windows - Architettura di Sistema.

Capitolo 1 - Sezione 1: Processi e thread

Capitolo 1 - Sezione 2: L'autenticazione nel quadro architetturale

Capitolo 2 - Utente e Dominio

Capitolo 1

Sezione 1 - Processi e thread

1.2 Il modello a processi

Per “modello a processi” s'intende un modello che descrive il modo in cui il sistema operativo gestisce l'esecuzione dei programmi (in generale sequenzialmente o parallelamente). In particolare nei moderni sistemi operativi, tra cui Windows NT/XP/.NET/2000/7/10/11, si utilizza di norma un modello a processi che permette l'esecuzione parallela di più programmi. Quando i calcolatori erano utilizzati solamente per scopi di ricerca ed i tipici programmi che vi venivano eseguiti dovevano sostanzialmente svolgere più che altro calcoli di natura matematica, l'esecuzione dei processi avveniva in modo sequenziale e la CPU veniva sfruttata pienamente. Con l'introduzione dei computer negli ambienti commerciali la maggior parte dei programmi eseguiti, leggendo e scrivendo continuamente dati dalle periferiche, effettuava una forte attività di I/O. In una situazione come questa, dato che i processi venivano eseguiti in modo

sequenziale, durante una qualunque operazione di I/O la CPU rimaneva in stato IDLE (attesa) per rilevanti percentuali di tempo. In questo scenario i tempi d'esecuzione dei programmi si allungavano enormemente. Dato che, con questo tipo di modello di esecuzione di processi, i costi per unità di tempo risultavano piuttosto rilevanti, era poco conveniente per le imprese investire in questo tipo di tecnologie. Ciò indusse i progettisti dei sistemi operativi a ricercare soluzioni architetturali idonee alla esecuzione di processi fortemente orientati all'I/O; si ritenne utile creare un meccanismo software che permettesse alla CPU di non sprecare tempo in stato di IDLE. Questo meccanismo prevedeva di assegnare la CPU ad ogni programma in esecuzione per un certo intervallo di tempo, simulando così l'esecuzione parallela; inoltre ogni volta che un processo iniziava un'operazione di I/O, la CPU passava ad eseguire un altro processo anziché sprecare tempo in attesa di completamento delle operazioni di I/O. Questo meccanismo, se da un lato ha permesso di ottimizzare l'uso della CPU, dall'altro ha introdotto nuove e più complesse problematiche legate all'esecuzione parallela dei programmi.

1.3 Prerequisiti

Prima di addentrarci nella definizione di modello a processi utilizzato in Windows introdurremo i concetti di handle, user mode, kernel mode e subsystem Win32 fondamentali per la comprensione del seguito di questo capitolo.

1.3.1 Handle

Un handle di fatto è un riferimento ad una risorsa di cui il processo può disporre: tali risorse possono essere file, socket, mutex, semafori, eventi ed altre ancora; sono raggiungibili grazie agli handle che sono inseriti in una speciale tabella, unica per processo, chiamata handle table. Ogni handle quindi rappresenta un elemento di questa tabella, e di conseguenza identifica una risorsa ben precisa, e può essere usato e manipolato solamente nelle modalità previste per l'oggetto cui fa riferimento (l'handle di una mutex si usa in modo diverso da quello di un file).

1.3.2 User mode

User mode e kernel mode sono termini che si riferiscono ai contesti di sicurezza esecutivi. La modalità user indica che il processo sta girando in un contesto di sicurezza stretto. Generalmente sono le applicazioni dell'utente che vengono eseguite in questa modalità; dato che non è possibile sapere a priori se queste applicazioni si comporteranno sempre in modo corretto, il sistema operativo non consente loro di effettuare operazioni critiche o privilegiate; ad esempio gli nega l'accesso diretto all'hardware. L'unico modo per ottenere l'accesso diretto alle risorse è mediante funzioni messe a disposizione dal sistema operativo. Queste funzioni del S.O. sono state implementate in modo assolutamente corretto.

1.3.3 Kernel mode

Al contrario della modalità user, quella kernel ha accesso diretto alle risorse, in questa modalità possono girare solamente il sistema operativo e i driver delle periferiche. La sicurezza della modalità kernel e data da due fattori:

- le applicazioni in user mode non sono in grado di cambiare contesto di sicurezza autonomamente, in quanto possono farlo solamente usando alcune funzioni del S.O.;
- le routine del sistema operativo possono eseguire solamente il codice contenuto al loro interno e non quello dell'applicazione che le ha richiamate; ciò impedisce che possa essere eseguito in kernel mode del codice considerato non sicuro.

In entrambi i casi il codice delle applicazioni user mode non verrà mai eseguito in kernel mode.

1.3.4 Subsystem Win32

Il subsystem Win32 è un ambiente logicamente situato al di sopra dell'EXECUTIVE (cfr. fig. 1.0 e 1.19), cioè al di sopra del kernel Windows col quale le applicazioni possono parlare per mezzo delle API messe loro a disposizione dal subsystem stesso. La struttura del subsystem Win32 è composta di due parti: una che gira in user mode (CSRSS.EXE) e l'altra che gira in kernel mode (WIN32K.SYS). Questa separazione è dovuta a motivi di prestazioni: il WIN32K.SYS è il responsabile del windowing, ovvero della gestione grafica delle finestre e girando in kernel mode è in grado di funzionare in modo ottimale e veloce fornendo performance migliori.

1.4 I processi

In ambiente Windows la definizione di processo non differisce da quella di molti altri sistemi operativi. Varia come viene implementato il modello di concorrenza. Un processo è un'istanza di esecuzione di un programma: è formato da almeno un thread e da una serie di strutture dati, utilizzate per contenere tutte le informazioni relative al funzionamento del processo stesso. Per chiarire meglio il concetto, si può immaginare il processo come una scatola inizialmente vuota il cui spazio viene utilizzato per contenere tutti i dati e le informazioni che gli serviranno per tutta la durata del processo stesso. All'interno di questo spazio risiedono la parte codice e la parte dati del processo, le DLL di cui fa uso e la sua **handle table** che contiene i riferimenti alle risorse. Il thread, contenuto in questa scatola, è un

elemento che tiene traccia del contesto d'esecuzione del processo ed è manipolato dal sistema operativo. Un processo Win32 fondamentalmente è composto da:

- uno spazio d'indirizzamento virtuale:
  a) ogni processo ha a sua disposizione uno spazio di memoria riservato (nessun altro processo vi può accedere); in questo modo viene garantita al sistema operativo maggiore protezione e al processo una maggiore stabilità e sicurezza;
  b) lo spazio d'indirizzamento massimo corrisponde a 4 Gbyte, di cui 2 sono riservati per il codice che viene eseguito in modalità user e 2 per quello in kernel;
- un insieme di handle raggruppati in una handle table;
- un access token che rappresenta il contesto di sicurezza (autorizzazioni) di un processo (cfr. par. 2.6);
- un ID (identifier) unico;
- un programma eseguibile che ne definisce il codice ed i dati (immagine del processo);
- almeno un thread d'esecuzione.

Come accennato prima i processi, per motivi di sicurezza e stabilità, non possono accedere direttamente a zone di memoria appartenenti ad altri processi. Al fine di poter comunicare e scambiare dati tra loro devono quindi utilizzare appositi strumenti detti di inter-process communication.

1.5 Process working set

Ogni processo fa riferimento ad uno spazio d'indirizzamento privato e virtuale; lo spazio d'indirizzamento è frazionato in unità minime dette pagine in cui sono contenute sia le istruzioni da eseguire sia i dati locali che quelli globali dei var thread. L'insieme di queste pagine, che il sistema operativo cerca di tenere presenti in RAM costituisce il Working Set del processo. Quest'insieme è gestito dal Virtual Memory Manager (VMM) che si preoccupa di caricare in memoria le pagine richieste

e scaricare quelle non più necessarie; il VMM determina la dimensione di RAM necessaria al working set per il corretto funzionamento del processo Il working set ha una dimensione minima e una massima, per vedere ed impostare tali valori è possibile utilizzare le seguenti funzioni:

- *GetProcess WorkingSetSize();*
- *SetProcess WorkingSetSize().*

1.6 Thread

Un thread è - in estrema sintesi - un contesto di esecuzione, è un insieme di informazioni che permettono al S.O. di tenere traccia dello stato d'esecuzione di un processo. Tra i vari elementi di cui è composto trovate lo stack, un'area di memoria riservata ed alcuni registri, tra cui il program counter. Questi elementi definiscono il *thread context* e sono strettamente legati all'architettura hardware in uso[1]. Un thread Win32 è composto da:

- un insieme di registri che rappresentano lo stato della CPU
- uno stack per il codice eseguito in user mode ed uno per quello eseguito in kernel mode;
- un access token ereditato dal processo cui appartiene;
- un ID univoco;
- un'area di memoria particolare chiamata Thread Local Storage (TLS) utilizzata dai subsystem.

Un processo può avere più di un thread, ciò significa che in un processo possono esistere più contesti d'esecuzione. I thread di un processo possono avere un access token diverso da quello del processo a cui appartengono, questo permette ad un processo multithread di impersonare più di un utente. I thread di un processo condividono in lettura e scrittura lo spazio d'indirizzamento e tutte le risorse assegnate al processo stesso, perciò i thread possono comunicare direttamente tra loro, utilizzando le risorse condivise senza dover ricorrere a metodi di comunicazione tra processi. Ciò permette al programmatore di realizzare più facilmente la comunicazione tra entità di esecuzione (thread). In fig.

[1] *Per ulteriori riformazioni vedere GetThreadContext() su MSDN online*

1.1 è rappresentato concettualmente un processo multithread. Il codice del processo esemplificato in figura è composto da più funzioni, ognuna delle quali viene eseguita da un thread differente, generato da uno stesso processo.

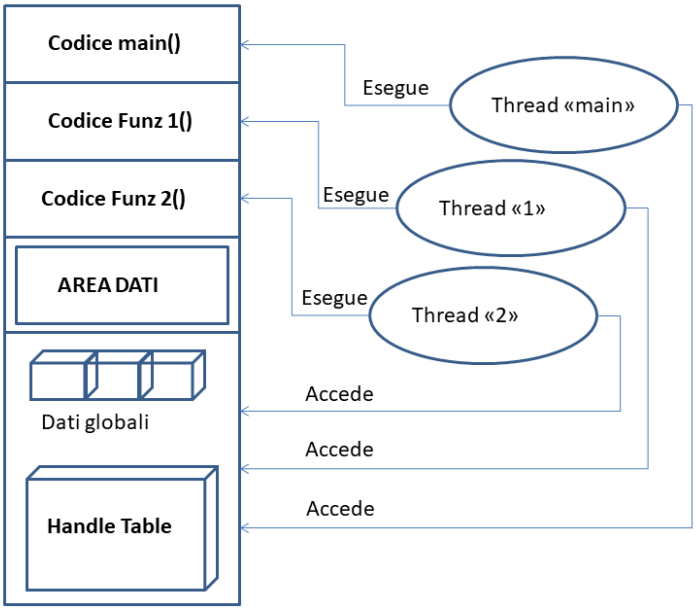

Fig. 1.1 La condivisione delle risorse di un processo tra i vari thread

Ogni programma ha un entry point, un punto da cui iniziare l'esecuzione, normalmente la funzione main() in linguaggio C/C++. Il thread generato in fase di creazione del processo, che si incarica dell'esecuzione del main(), è detto "thread principale" o "main thread" e così potrebbe essere menzionato talvolta.

1.7 La gestione dei processi

Con gestione dei processi s'intende definire l'insieme di tutte quelle azioni che vengono intraprese per manipolare un processo, dal momento della sua creazione fino alla sua terminazione. In particolare la gestione di un processo si divide in tre fasi

- creazione di un processo:

  insieme di azioni necessarie per mandare in esecuzione un programma,

- gestione di un processo in esecuzione:

  lo scheduling dei thread, la gestione delle priorità, ecc.

- gestione della terminazione di un processo:

  il rilascio delle risorse allocate per il processo.

1.3 La creazione di un processo

La creazione di un processo e caratterizzata dai seguenti passi:

- caricamento di un'applicazione Win32 valida da eseguire,
- creazione ed inizializzazione di un process object,
- creazione ed inizializzazione di un thread object;
- creazione ed inizializzazione delle strutture dati del subsystem win32;
- ultime inizializzazioni ed avvio dell'esecuzione del nuovo processo.

1.9 Loading di un'applicazione Win32 valida

Per prima cosa un programma deve essere caricato in memoria per poter essere eseguito, ed in ambiente Windows questo programma deve essere una applicazione Win32 valida: se non lo è, come ad esempio capita per le vecchie applicazioni MS-DOS, o per quelle LINUX, allora il S.O. deve "incapsulare" l'applicativo in un contesto di esecuzione Win32. A questo scopo cerca di eseguire il processo all'interno di un "*Win32 image support"* , e cerca quello più idoneo per l'applicazione richiesta. In altre parole mediante un'applicazione Win32 (appunto il Win32 image support) viene eseguito codice **non** Win32. Le varie informazioni necessarie a capire con che tipo di applicazione abbiamo a che fare, sono contenute nell'header del programma da eseguire e le si possono visualizzare utilizzando appositi tool[2] .

Gli applicativi elencati di seguito svolgono funzione di "Win32 image support":

- WoW64 (Windows over Windows "Virtual Dos"):

[2] *Tra questi: dependency walker ed exetype*

per retrocombatibilità con MS-DOS per programmi con estensione .exe, .com e .pif;

- WSL (Window Subsytem for Linux)

  per tutti gli applicativi che richiedono il subsystem Linux / Posix ;

- CMD.EXE:

  per gli applicativi “DOS” (ad es .bat e .cmd)

- Powershell.exe:

  la Power Shell consente di eseguire codice dot NET Framework da linea di comando estendo in modo importante le possibilità di intervento rispetto al CMD.exe

Se l'applicativo è già una applicazione Win32 non è necessario l’ image support, può essere direttamente eseguito il codice dell'applicazione.

Se non è un applicazione Win32 e non è possibile trovare un Win32 image support idoneo, la creazione del processo non andrà a buon fine.

> ***Nota:*** *È utile segnalare che cmd.exe non è un emulatore del sistema operativo MS-DOS. L'image support cmd.exe è una shell che permette di eseguire programmi da riga di comando; in questa shell rimangono valide tutte le regole di gestione dei processi di Windows (kernel mode, user mode, ecc) che andiamo a esaminare.*

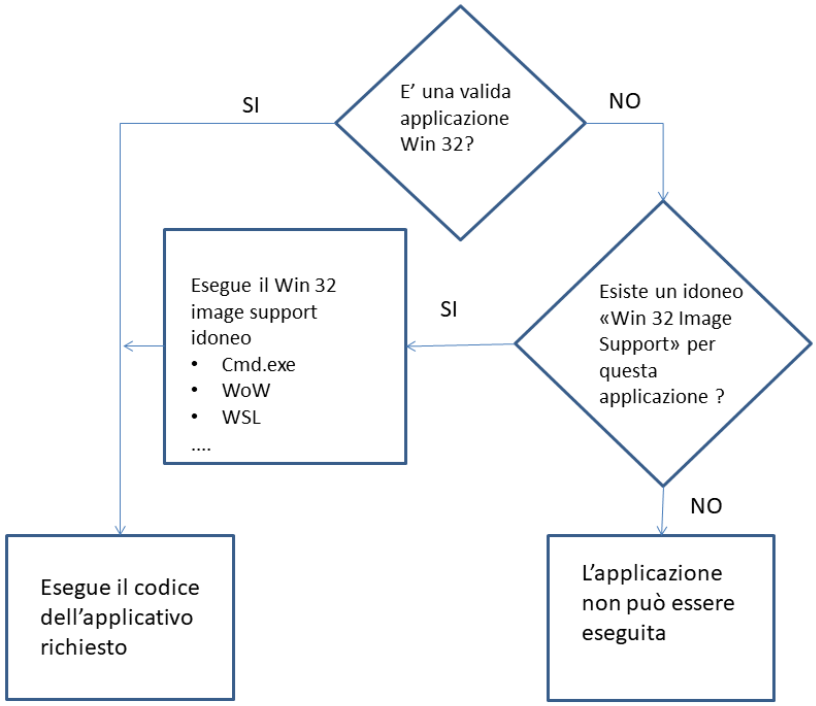

Fig. 1.3 Il meccanismo di attivazione del codice di un'applicazione

La creazione di un processo avviene sempre ad opera di un altro programma già in esecuzione. Per questo motivo si parla di relazione "padre - figlio" tra il processo creante ed il processo creato. In ambiente Windows questa relazione non ha un particolare significato in ambiente Windows; dopo che il processo figlio è stato correttamente creato i due processi procedono slegati tra loro. L'esistenza dell'uno non implica quella dell'altro. Il tool tlist.exe (support tools) mostra la lista di processi in esecuzione; con l'opzione "/t" evidenzia le relazioni padre-figlio esistenti tra i processi indentando opportunamente l'output.

1.10 La creazione di un processo all'interno dell'EXECUTIVE

Ogni processo corrisponde una struttura EXECUTIVE PROCESS (EPROCESS block); all'interno della struttura ritroviamo tutte le informazioni che riguardano il processo come l'handle table, i thread, l'access token, la priorità ed il PID (Process Identifier). Si può accedere a questa struttura solo quando

ci si trova in modalità kernel; una parte di questa struttura, chiamata PROCESS ENVIRONMENT BLOCK (PEB), è accessibile anche in user mode. Creare un process object significa allocare una nuova struttura EPROCESS ed inizializzarne i campi; in questa fase viene creato il security context del processo. Per default il processo creato eredita il security context del processo creante.

1.11 La creazione di thread all'interno dell'EXECUTIVE

Un thread è identificato da una struttura chiamata EXECUTIVE THREAD (ETHREAD) che contiene tutte le informazioni necessarie alla gestione del thread stesso. Creare un thread object significa allocare una nuova struttura dati ETHREAD ed inizializzarne i campi, in particolare il suo access token.

1.12 Strutture dati del subsystem Win32

Poiché le applicazioni dipendono totalmente dal subsystem Win32, questo deve allocare delle apposite strutture dati per poterle gestire. Dato che il subsystem è diviso in due parti le strutture dati devono essere allocate a coppie, una per la parte che gira in user mode (csrss.exe) e l'altra che gira in kernel mode (win32k.sys).

1.13 Caricamento delle librerie dinamiche (DLL)

Un processo può essere eseguito se sono presenti le librerie dinamiche di cui ha bisogno, quindi è necessario verificare se sono tutte reperibili e disponibili. Nel caso in cui le librerie non siano disponibili, il processo termina immediatamente; in caso contrario le librerie vanno inserite all'interno dello spazio d'indirizzamento del processo, perché possa utilizzarle. Quest'operazione viene effettuata nel momento in cui un processo viene creato ("load-time linking",). L'operazione di controllo e caricamento delle librerie è a carico del thread del processo creato.

> ***Nota.*** *Un processo può essere creato in modalità "sospesa", ciò implica che il suo thread non può eseguire alcun'operazione, inclusa la verifica della presenza delle librerie richieste. Non appena il thread verrà riattivato, provvederà ad effettuare il controllo ed il caricamento delle DLL.*

A questo punto, dopo aver eseguito le inizializzazioni ed i controlli descritti, un processo Win32 può essere eseguito.

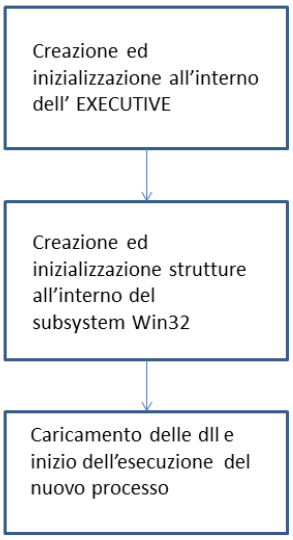

Fig. 1.6 Le fasi principali all'avvio di un processo

1.14 Multasking

Come già accennato nel par. 1.2 i moderni sistemi operativi sono in grado di gestire più applicazioni in esecuzione in un dato momento. In questo tipo di gestione dei processi la CPU è considerata una risorsa da condividere tra tutte le applicazioni in esecuzione. Questa condivisione si può realizzare in due modi:

- mediante una cessione volontaria del processore: cooperative multitasking;
- mediante prelazione del processore: preemptive multitasking.

1.15 Cooperative multitasking

Questo modello di multitasking lascia al programmatore il compito di preoccuparsi di cedere la CPU ad altri processi, ad intervalli regolari. Questo modo di realizzare il multitasking ha importanti

controindicazioni; se l'applicazione non è in grado di raggiungere il segmento di codice in cui in cui è previsto il rilascio della CPU viene impedita l'esecuzione di altri processi.

1.16 Preemptive multitasking

In questo tipo di multitasking il programmatore non deve preoccuparsi di gestire in maniera esplicita le operazioni di rilascio della CPU dato che non è sotto il controllo diretto del programmatore, una componente del sistema operativo (scheduler) si preoccupa di assegnare ad intervalli regolari (detti quanti) la CPU ai processi. Questa operazione viene effettuata in modo imperativo (preemption): il S.O. allo scadere del quanto di tempo, distoglie la CPU da un processo per assegnarla ad un altro. Grazie al preemptive multitasking la condivisione della CPU tra i thread, non può essere bloccata né per un errore di programmazione né per altre cause analoghe.

Windows come già detto implementa un preemptive multitasking; nei seguenti paragrafi vedremo come creare e gestire processi e thread tramite le funzioni del subsystem Win32.

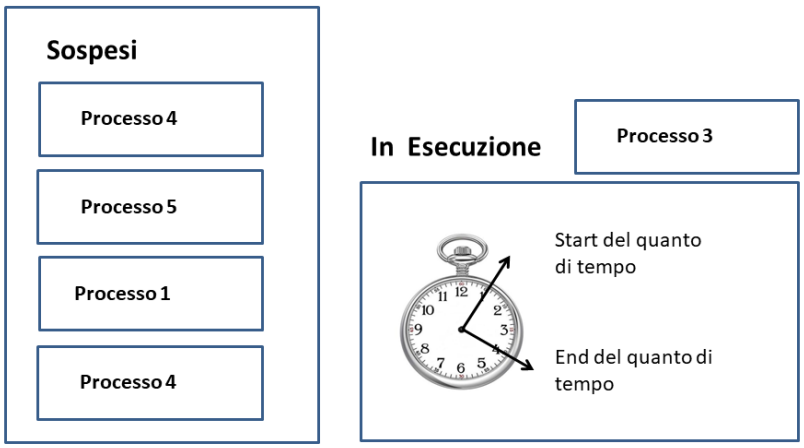

Fig. 1.7 Il Multitasking "round robin" e la rappresentazione del quanto di tempo di un processo

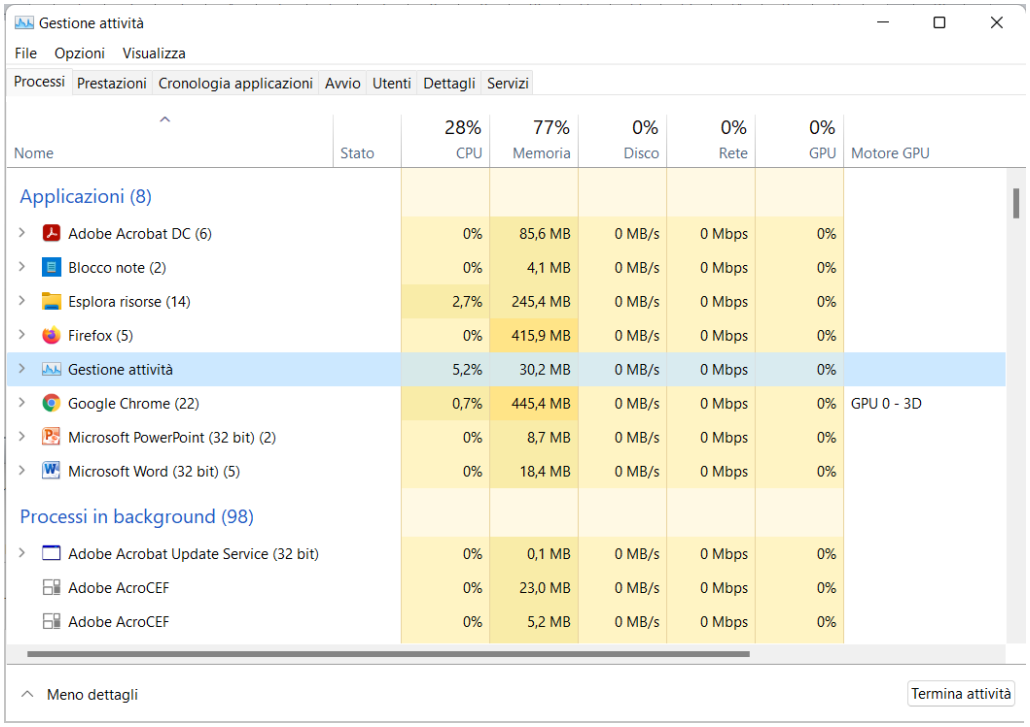

Fig 1.8 Il task manager (gestione risorse) mostra informazioni sui processi in esecuzione

1.17 La creazione di un processo

La funzione che si deve usare in C per creare un nuovo processo è la CreateProcess():

```
BOOL CreateProcess(
        LPCTSTR lpApplicationName,
        LPTSTR lpCommandLine,
        LPSECURITY_ATTRIBUTES lpProcessAttributes,
        LPSECURITY_ATTRIBUTES lpThreadAttributes,
        BOOL blnheritHandles,
        DWORD dwCreationFlags,
        LPVOID lpEnvironment,
        LPCTSTR lpCurrentDirectory,
        LPSTARTUPINFO lpStartuplnfo,
        LPPROCESS_INFORMATION lpProcessInformation [dato in output]);
```

Nel corso di questo scritto spiegheremo meglio il significato di questi parametri. Partiamo con un esempio d'uso: il seguente codice fa uso della funzione CreateProcess() per creare un'istanza del processo "notepad.exe". A differenza della funzione fork() di Unix questa funzione **non duplica** un processo esistente, ma crea l'istanza di un processo per l'immagine di un programma. Il programmatore Unix troverà che esistono **maggiori similitudini** tra la funzione fork() e la Create Thread() (cfr. par. 1.18).

```
#include <windows.h>
int main(int argc, char* argv[]){
        BOOL result;
        STARTUPINFO startuplnfo;
        PROCESS_INFORMATION plnfo;
        char path[] = "c:\\winnt|\notepad.exe",
        15arametric[] = "c:\\winnt\\notepad.exe c:\\testo.txt";
        ZeroMemory(&startuplnfo, sizeof(startuplnfo));
```

```
startupInfo.cb = sizeof(startupInfo);
result = CreateProcess(NULL,// percorso dell'eseguibile
                       parametri,// eseguibile + parametri
                       NULL,// Security del nuovo processo
                       NULL,// Security del thread del nuovo processo
                       FALSE,// Eredita gli handle
                       0,11 Crea il processo in stato running
                       NULL,// Environment
                       NULL,// directory di partenza
                       &startupInfo, // Info. del processo da creare
                       &pInfo);// Informazioni sul nuovo processo
                       Close Handle(pInfo.hThread);
                       CloseHandle(pinfo.hProcess);
return 0;}
```

È possibile utilizzare indifferentemente il primo o il secondo parametro della funzione CreateProcess() per indicare quale programma debba essere eseguito; nel secondo parametro possiamo anche specificare i vari valori di avvio del nuovo processo da creare. Come si vede, nell'esempio abbiamo usato direttamente il secondo parametro per indicare quale programma eseguire. Il sesto parametro di questa funzione ci consente di indicare se vogliamo creare un nuovo processo in stato suspended o running. Nel seguente elenco vediamo alcuni flag relativi:

- CREATE_SUSPENDED: crea il processo in modalità *sospeso*;
- 0 (zero): indica "nessun flag in particolare" e crea il processo in modo che sia "running";
  CREATE_NEW_CONSOLE: crea una nuova console a caratteri per il nuovo processo.

Il nono parametro specifica informazioni che hanno a che fare con l'aspetto grafico del processo. L'aspetto più interessante di questo parametro è che ci permette di manipolare i tre canali di

comunicazione standard (stdin, stdout, stderr). L'ultimo parametro è il puntatore ad una struttura il cui contenuto viene riempito dalla funzione CreateProcess() in fase run-time; in particolare questa struttura verrà riempita con i valori dell'handle del nuovo processo e del suo thread e con i rispettivi valori di ID.

1.18 Creazione di un thread

La funzione utilizzata per creare i thread è (la terminologia già lo suggerisce) la Create Thread() di cui diamo di seguito un esempio. Con questa funzione è possibile associare ad un contesto di esecuzione ad una specifica funzione o procedura.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
        DWORD WINAPI FunzioneDelNuovo Thread(LPVOID lparam) {
        MessageBox(NULL, (char* )lparam, "Nuovo thread", MB_OK);
        return 0;
        }
        int main(int argc, char* argv[]){
                DWORD dwThld;
                HANDLE hNewThread;
                char parametri[] = "Codice eseguito dal nuovo thread";
                hNewThread = Create Thread(NULL, O, FunzioneDelNuovo Thread, parametri, 0, &dwThld);
                MessageBox(NULL, "Codice eseguito dal thread principale", "Thread principale", MB_OK);
                Close Handle(hNewThread);
                return 0;
        }
```

La funzione FunzioneDelNuovoThread() dell'esempio è il segmento di codice che vogliamo che funzioni in un contesto di esecuzione separato (thread); per raggiungere questo scopo occorre specificare nel

terzo parametro della Create Thread() il nome della funzione da eseguire. Nel quarto parametro possono essere specificati i parametri *che vanno passati alla funzione invocata* dalla Create Thread().

La funzione Create Thread() restituisce due valori: uno mediante l'istruzione return che rappresenta l'handle del nuovo thread; l'altro, che rappresenta l'ID del nuovo thread, viene restituito tramite il sesto parametro della funzione Create Thread().

Il primo parametro rappresenta un puntatore ad una struttura contenente informazioni relative alla security ed alla possibilità di ereditare l'handle del nuovo thread e può anche essere posto a NULL. Il secondo parametro rappresenta la dimensione dello stack a disposizione del nuovo thread e solitamente è posto a 0 (zero). Il quinto parametro rappresenta lo stato in cui il thread viene creato (0 per running e CREATE_SUSPENDED per sospeso).

In questo esempio esistono due contesti di esecuzione, quello del main thread e quello della funzione *FunzioneDelNuovoThread()* generato dalla CreateThread().

Entrambi i thread eseguono una MessageBox(): cliccando sul relativo bottone della message box il thread principale viene terminato, cosa che comporta la terminazione del thread secondario, dato che l'intero processo termina quando termina il thread principale.

1.19 Lo scheduling dei thread

Nei sistemi *multithread* occorre decidere quando assegnare la CPU ad un thread in esecuzione e per quanto tempo lasciargliela in uso. Questo procedimento prende il nome di “thread scheduling”. Windows implementa un o *scheduling dei thread a priorità* (cfr. par. 1.24). È necessario precisare che l'attività di *scheduling è svolta a livello di thread e non di processo.* Come abbiamo già detto in precedenza, un processo è solamente un insieme di informazioni e di risorse, mentre un thread è la sua istanza di esecuzione, quindi è su di questa che vengono applicate le politiche di scheduling.

1.20 Il quanto di tempo

Ogni thread, che viene selezionato per essere eseguito, ha a propria disposizione la CPU per una certa quantità di tempo, al termine del quale il processore deve eseguire un altro thread avente priorità uguale o superiore. Questo intervallo di tempo è il cosiddetto quanto di tempo o semplicemente *quanto*.

1.21 Preemption (prelazione)

I thread, esaurito il loro quanto di tempo, sono "obbligati" a rilasciare la CPU, in altre parole non appena il quanto termina, lo stato del thread attualmente in esecuzione viene salvato in memoria, ed i valori dei registri della CPU vengono sostituiti con i valori di stato di un altro thread: il thread non si accorge di essere stato interrotto (nessuna variabile di stato del thread tiene traccia di questo) e l'esecuzione passa quindi ad un altro thread per opera dello scheduler (preemption).
In base alla sua priorità (cfr. par. 1.24) un thread può essere obbligato a rilasciare la CPU a favore di un thread con priorità più alta, in stato "pronto per essere eseguito" (ready).

1.22 Tempo d'esecuzione di un thread

Il tempo d'utilizzo della CPU per un thread è calcolato mediante un contatore numerico anziché un valore temporale. Ad intervalli regolari lo scheduler entra in azione e decrementa questo valore. Non appena questo contatore raggiunge lo zero il quanto di tempo si considera scaduto e viene mandato in esecuzione un nuovo thread. Di *default* il contatore può non essere identico per versione di Windows, ad es. è inizializzato a 6 su alcune versioni Workstation e a 36 su alcune versioni Server. Questa differenza esiste perché Workstation e Server svolgono ruoli differenti in un dominio: un valore più elevato del quanto di tempo permette ai server di risolvere più rapidamente le richieste dei clienti a discapito del tempo di reazione (masggiore latency). Ogni qualvolta termina un ciclo di clock[3] il valore del contatore viene decrementato di 3 unità, quindi un thread rimane in esecuzione per 2 cicli di clock

[3] *col termine clock si intende il clock di sistema, ovvero il "millisec clock" e non il clock hardware*

su certe Workstation e 12 sui Server. Al termine del quanto di tempo il thread viene messo in *ready* (pronto per essere selezionato nuovamente per l'esecuzione). La Fig. 1.9 illustra i diagrammi a stati del thread. Il decremento del quanto di tempo di 3 unità avviene quando il thread rimane in esecuzione per un tempo pari a quello dell'intero clock di sistema. Quando invece un thread si mette in wait, ad esempio quando è in attesa di un evento, il suo quanto viene decrementato di 1 unità perché si intende convenzionalmente che abbia consumato solo una porzione del tempo a sua disposizione.

3

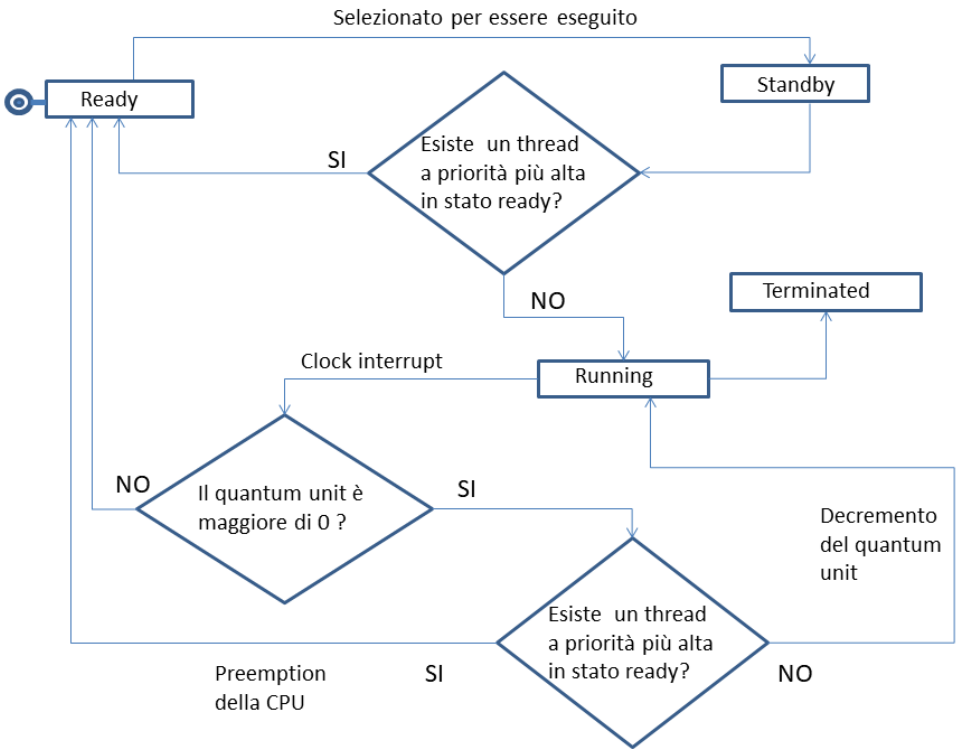

Fig 1.9 Diagramma di stato dei thread in relazione al quanto di tempo

1.23 Gli stati di un thread

Il modello a processi simula l'esecuzione parallela di più processi. In realtà si tratta di un parallelismo “virtuale” dato che una CPU è in grado di eseguire un solo thread per volta; i thread non sono sempre in esecuzione, ma passano di volta in volta da uno stato di running ad uno di sospensione (standby). Gli stati di un thread sono rappresentati nello schema di **fig. 1.10** e descritti di seguito.

Initialized

È lo stato iniziale di un thread non appena viene creato, tutte le sue strutture dati vengono inizializzate.

Ready

Una volta che un thread viene creato ed inizializzato è pronto per essere eseguito, ovvero diventa ready. In base alla sua priorità viene inserito in una apposita coda di attesa.

Standby

Questa fase coinvolge il thread selezionato per essere eseguito. Prima di dar luogo alla sua esecuzione di questo thread viene effettuato un ulteriore controllo per verificare se nel frattempo un thread con priorità più alta si sia risvegliato dallo stato di waiting. Se l'esito di questa verifica è negativo il thread selezionato passa in stato running, in caso contrario il thread selezionato viene posto - in stato ready - in cima alla coda d'attesa per la sua priorità (cfr. fig. 1.10 e fig 1.15).

Running

Lo stato running è quello in cui si trova il thread in esecuzione. Superato lo stato standby un thread il suo stato diventa running e può venire materialmente eseguito. Esiste un (solo) thread in stato running per *ogni* processore presente.

Terminated

Quando un thread ha terminato il suo tempo di vita viene eliminato. Tutti i suoi dati, precedentemente allocati ed inizializzati vengono cancellati dalla memoria.

Waiting

Questo stato indica che un thread è in attesa di un determinato evento, come il completamento di un'operazione di I/O o il rilascio di un oggetto di *mutua esclusione*, per sincronizzare la propria esecuzione con quella di altri thread.

Transition

Questo è uno stato piuttosto particolare: indica che un thread è pronto per essere eseguito, ma che non tutti i suoi dati sono caricati in memoria e quindi si sta attendendo che vengano ripristinati in RAM.

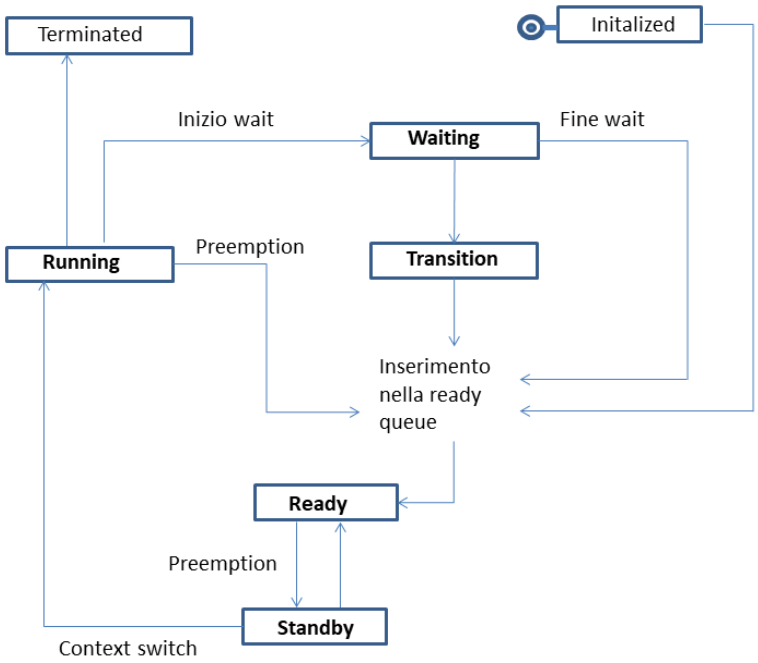

Fig. 1.10 Diagramma a stati dei thread

1.24 La priorità

Lo scheduling con priorità prevede che ad ogni thread venga assegnato un valore detto – appunto - di "priorità". In base a questo valore un thread può essere eseguito prima (o dopo) rispetto ad un thread avente priorità più bassa (più alta). Per motivi di ottimizzazione è buona norma assegnare una priorità alta a quei processi che fanno molta attività di I/O ed una bassa a quelli che fanno molto uso di CPU. In

questo modo si garantisce una risposta più sollecita da parte di un'applicazione server. I valori di priorità dei thread variano da 0 a 31, come riportato nel seguente schema (Fig. 1.11).

Ogni **processo** possiede un unico livello di priorità chiamato *base priority* che rappresenta la priorità iniziale con cui i suoi **thread** vengono eseguiti. Ai thread sono invece assegnati **due livelli di priorità** una base priority ereditata dal processo e una **current priority.**

La possibilità di cambiare priorità ad un processo è ovviamente consentita, e si può effettuaresia in via programmativa sia tramite pannellino del task manager o con programmi di analisi dei processi come il Process Explorer (cfr. fig. 1.14).

Qui di seguito è riportato un esempio di cambio di priorità per via programmativa. Anche in questo esempio, adattamento dell'esempio precedente, esistono due thread rappresentati dal *main()* e *FunzioneDelNuovoThread().* Il comportamento del thread *FunzioneDelNuovo Thread()* è quello di occupare totalmente la CPU finché non gli viene segnalato di rilasciarla. Il thread principale (ovvero quello che esegue la funzione *main()*) crea un oggetto “EVENTO” mediante la funzione *CreateEvent().* Quest'oggetto è associato logicamente ad una determinata condizione, come ad esempio “terminare l'esecuzione" (TerminateThread()).

L'evento è in stato “segnalato” (signaled) quando questa condizione si manifesta, in caso contrario rimane in stato non segnalato.

Per segnalare un evento, e indicare che una determinata condizione si è verificata, viene usata la funzione *SetEvent().* Il thread continua a “ciclare” e quindi a consumare CPU, finché l'evento associato all'oggetto *hExit* (variabile globale di tipo HANDLE) non diventa “segnalato”.

La funzione *WaitForSingleObject(),* utilizzata da un thread, controlla lo stato dell'evento riportando o meno il manifestarsi di questa condizione. Queste tre funzioni vengono spiegate meglio nel par. 1.28 Concorrenza e Sincronizzazione.

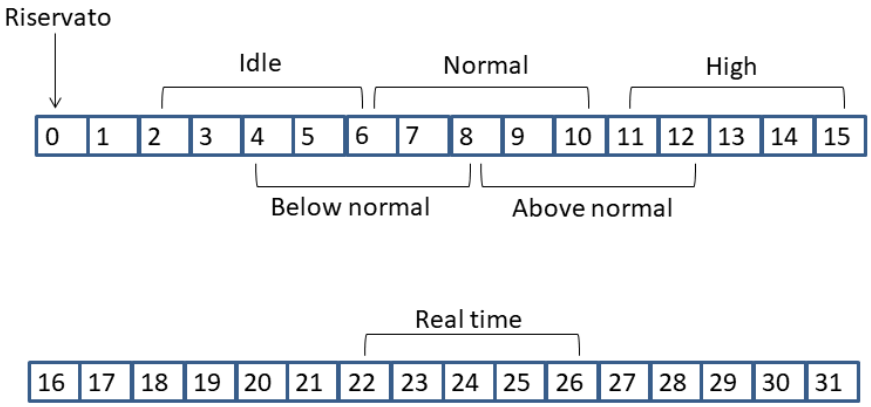

Fig. 1.11 I livelli di priorità dei thread in Windows

Vediamo come cambiare **la priorità di un thread** con un piccolo esempio programmativo**:**

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
HANDLE hExit;
DWORD WINAPI FunzioneDelNuovo Thread(LPVOID lparam){
        MessageBox(NULL, (char*)lparam, "Nuovo thread", MB_OK);
        SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_IDLE);
        while( WaitForSingleObject(hExit, 0) ) )
        return 0;
}
int main(int argc, char* argv[]){
        DWORD dwThId;
        HANDLE hNewThread;
        char parametri[] = "Codice eseguito dal nuovo thread";
        hExit = CreateEvent(NULL, TRUE, FALSE, NULL);
        hNewThread = Create Thread(NULL, 0, FunzioneDelNuovo Thread, parametri, 0, &dwThId);
        MessageBox(NULL, "Codice del thread principale", "Thread principale", MB_OK);
```

```
        SetEvent(hExit);
        WaitForSingleObject(hNewThread, INFINITE);
        CloseHandle(hExit);
        CloseHandle(hNewThread);
        return 0;
}
```

La funzione SetThreadPriority() è quella che ci permette di cambiare la priorità del thread indicato dall'handle che viene passato come primo parametro (nota: la funzione GetCurrentThread() ci consente di reperire l'handle del thread). Il secondo parametro indica quale priorità assegnare. Nell'esempio è stata usata la priorità IDLE, altri valori possibili sono:

```
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_IDLE
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_TIME_CRITICAL
```

In questo esempio il thread main genera il thread *"FunzioneDelNuovoThread",* che imposta la *propria priorità* al valore più basso *possibile* (IDLE). Occorre premere il pulsante della message box del "Nuovo Thread" (il titolo della finestra è "Nuovo Thread") perché l'esempio venga eseguito correttamente. Durante la sua esecuzione il nuovo thread occupa la CPU, ma di fatto non blocca l'esecuzione di altri processi o thread perché la sua priorità è molto bassa. L'esempio termina cliccando il pulsante "Ok" della message box mostrata dal thread principale (finestra "Thread principale" nell'esempio

precedente). Il tool Process Explorer[4] ci permette di esaminare alcuni dettagli analitici relativi ai processi e thread, tra i quali la loro priorità. In fig. 1.13 vediamo quanto appena descritto sulle priorità: utilizzando Process Explorer si notano (banalmente) che all'interno di uno stesso processo esistono thread a differenti priorità.

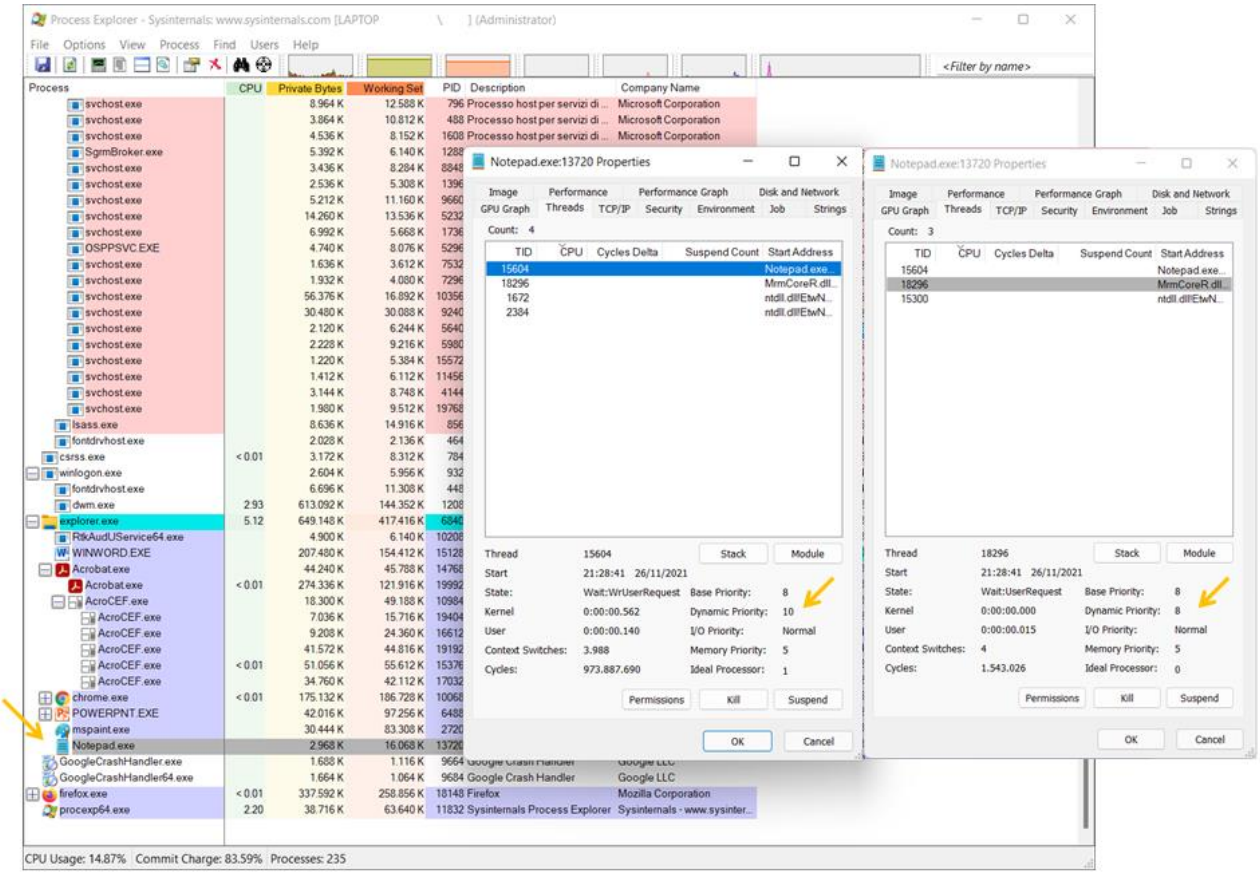

Fig. 1.13 Differenze di priorità tra i thread di un processo in Process Explorer

[4] *Process Explorer.exe (sysinternals).*

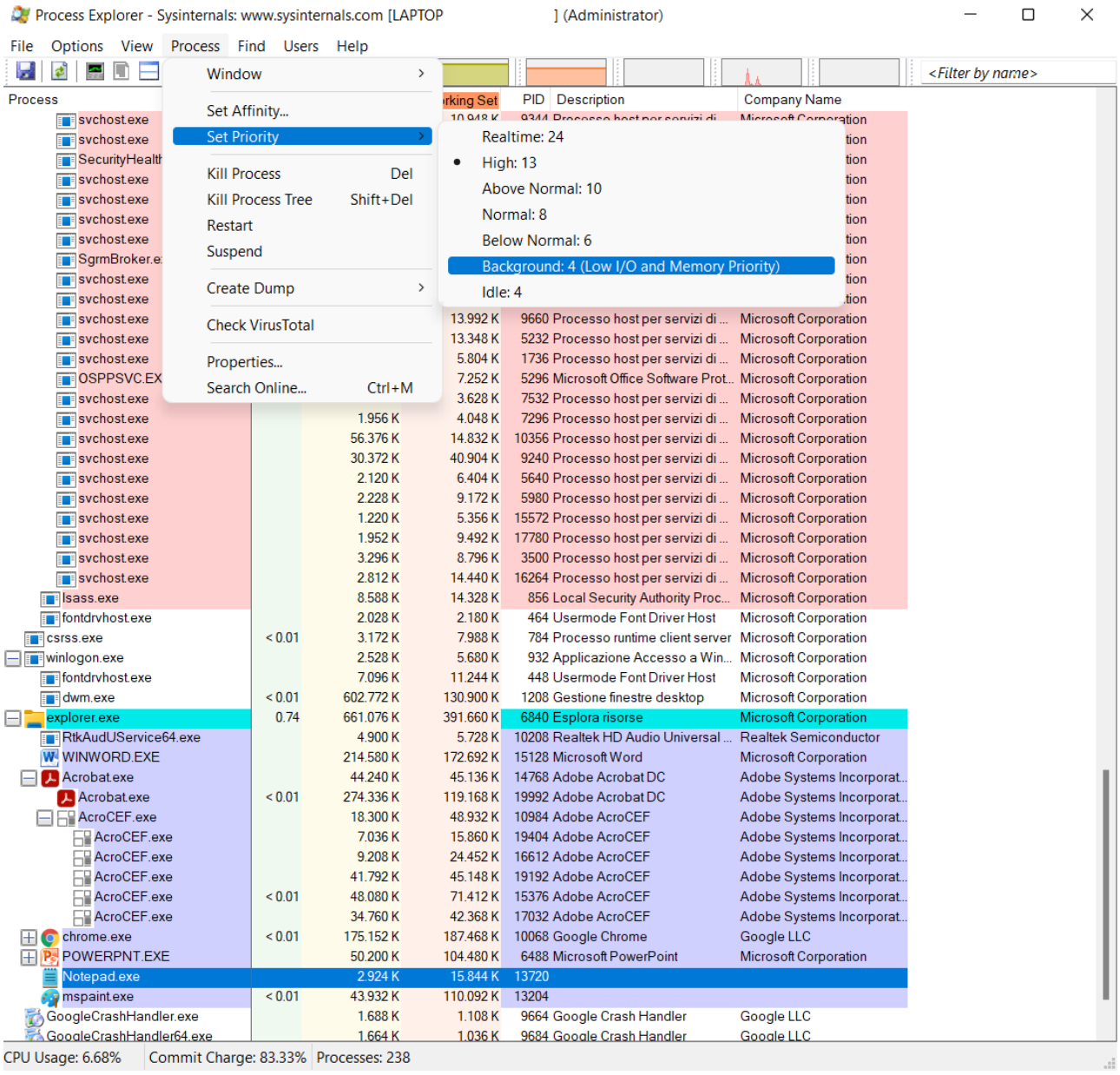

Fig 1.14 Le priorità di base un processo sono dinamicamente modificabili

1.25 Le priorità real time

Come si può vedere nella fig. 1.14 esiste la possibilità di impostare la priorità di un Processo a “real time”. Ricordiamo, tuttavia, che non è possibile fare assunzioni sui **tempi** di esecuzione dei processi o dei thread, per cui il termine *real time* non va inteso in senso letterale, e va inteso solo come etichetta che indica una priorità molto alta.

1.26 Le code dei thread ready

In un dato momento è possibile che più thread si trovino nello stato ready (pronti per essere eseguiti). Il sistema operativo inserisce questi thread in un'apposita lista chiamata "coda dei ready' (*ready queue)* e li organizza in base alla loro priorità; ciò vuol dire che questa lista ha più indici, e ognuno identifica una priorità.

Come si vede nella fig. 1.15 la *ready queue* è un array di liste di thread. Per ogni livello (corrispondente ad una diversa priorità) esiste un'apposita coda, ovvero una lista di puntatori alle strutture dei threa. Il primo thread nella lista a priorità più alta è il primo che verrà eseguito.

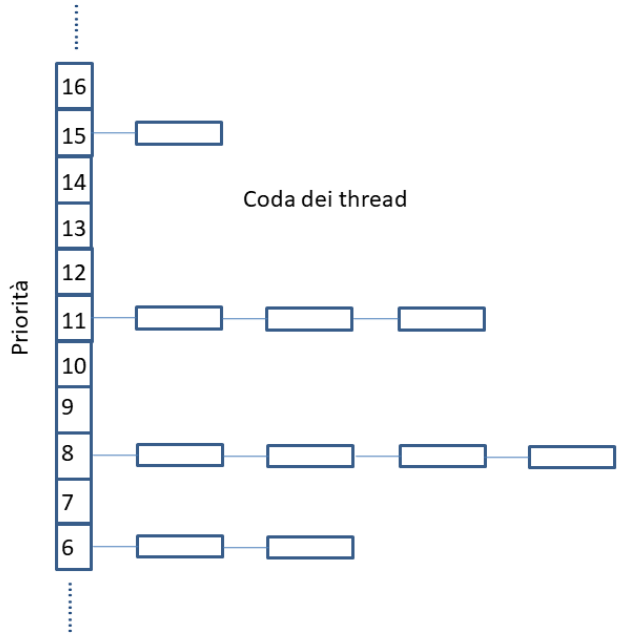

Fig. 1.15 La coda con priorità dei thread in stato ready

1.27 Priority boost

In situazioni particolarmente “perverse” esiste l'eventualità che un thread o un gruppo di thread a priorità alta monopolizzi la CPU senza rilasciarla ai processi a bassa priorità creando un problema noto

in letteratura come “starvation”. Per impedire che questo accada, periodicamente viene effettuato un controllo sulle code dei thread pronti (*ready*) per verificare lo stato di attesa dei thread. Se qualche thread è *in stato di attesa da troppo tempo* viene innalzata "artificiosamente” la sua priorità (*priority boost*) per un certo intervallo di tempo affinché possa continuare la sua esecuzione. Questa attività di controllo viene svolta da un apposito thread di sistema chiamato *balance set manager.*

1.28 Concorrenza e sincronizzazione

Nei sistemi operativi multiprocess può capitare che thread e processi cerchino di accedere a risorse **condivise** come ad esempio una stessa area dati aperta in lettura e **scrittura**. Se queste operazioni avvengono in parallelo si possono verificare delle inconsistenze. Se un thread cerca di leggere una zona di memoria condivisa mentre un altro thread non ha ancora finito di scrivere, il thread in lettura potrebbe trovare i dati in uno stato non previsto o non consono, e quindi incorrere in un errore. Servono criteri e metodi per regolare l'accesso alle zone condivise.

Il sistema operativo Windows fornisce al programmatore gli strumenti (*mutex, semafori, eventi*, ecc.) atti a regolamentare l'accesso alle zone condivise. Questi sono manipolabili mediante *handle* ed opportune funzioni, tra le quali:

- CreateMutex();
- WaitForSingleObject();
- ReleaseMutex();
- CreateEvent();
- SetEvent();
- ResetEvent().

L'oggetto *mutex* (mutua esclusione) è una risorsa fornita dal sistema operativo che permette di regolare l'accesso ad una zona di memoria condivisa. Il suo funzionamento è del tutto simile a quello di un *semaforo*: concede solamente ad un thread alla volta la possibilità di accedere alla zona condivisa.

Un oggetto mutex può essere in stato *libero* od *occupato*. Quando un thread occupa un oggetto mutex, quest'ultimo impedisce l'ingresso ad altri thread nella zona critica. Conseguentemente nessun altro thread p otrà accedere alla zona di memoria condivisa e quindi non si possono generare inconsistenze.

Le funzioni di sistema per gestire un oggetto di mutua esclusione **in Windows sono fondamentalmente quattro**:

- CreateMutex();
- Wait For SingleObject();
- Release Mutex();
- Close Handle()

Prima di tutto occorre creare l'oggetto *mutex* mediante la funzione *Create Mutex()* per poterne disporre, quest'operazione può essere svolta da un qualunque thread, e in generale viene effettuata in fase d'inizializzazione.

In secondo luogo occorre delimitare la zona di codice "critica" con le funzioni *WaitFor SingleObject()* e *Release Mutex();* sarà cura del programmatore **delimitare la zona crítica** con queste due istruzioni.

Il thread deve invocare la funzione *WaitForSingleObject()* prima di accedere alla zona condivisa; la *WaitForSingleObject()* controlla lo stato dell'oggetto mutex verificando se sia libero od occupato, nel caso in cui sia libero l'oggetto mutex permette al thread che ha invocato la *WaitForSingleObject()* di accedere alla zona critica impedendo l'ingresso a tutti gli altri.

Non appena il thread ha finito di percorrere la zona critica deve **rilasciare** l'oggetto mutex in modo da **permettere** ad un altro thread di potervi entrare, quest'operazione viene effettuata invocando la funzione *ReleaseMutex().*

Quando un oggetto mutex non è più necessario lo si deve eliminare utilizzando la funzione CloseHandle(). In figura sotto viene schematizzato l'uso di queste quattro funzioni di controllo della concorrenza,

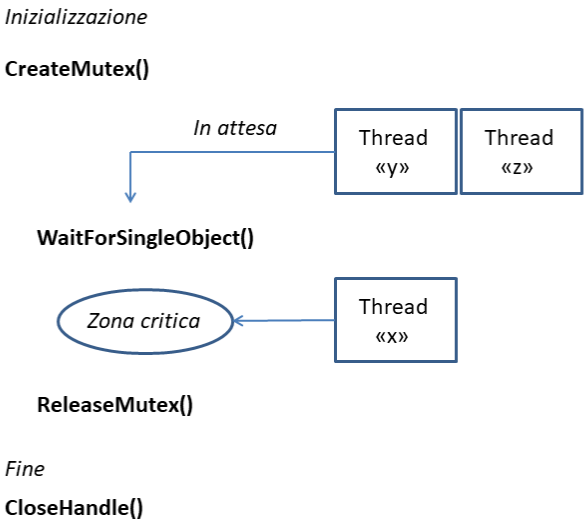

Fig. 1.16 Thread multipli che condividono un segmento di codice con una zona crítica