

Processi e thread 5 Windows

martedì 14 ottobre 2025 18:16

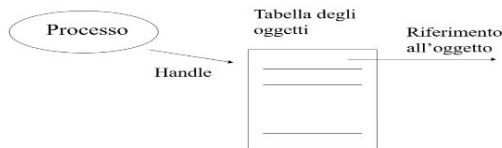
Vediamo ora i processi in ambiente windows : cominciamo a parlare di **oggetti windows** , ovvero tutte le entità che gestisce , tra cui le applicazioni attive e processi. Ogni entità è oggetto e sono caratterizzati dal tipo (tipi diversi), attributo (attributi in base al tipo dell'oggetto-> mantiene opportune strutture dati) e servizi (systemcall che windows fornisce in base al tipo). Vediamo gli attributi:

1. Id del processo
 - a. Identificatore numerico
2. Descrittore della sicurezza
 - a. Collezione delle info che ci permettono di capire il ruolo che applicazione ha all'interno del sistema: cosa quindi può fare
3. Priorità di base
 - a. Importanza dell'applicazione attiva
 - b. Si parla di **thread**
4. Affinità processore
 - a. Insieme di processori dove può essere mandata in esecuzione l'applicazione
5. Limiti di quota
 - a. Utilizzo delle risorse
6. Tempo esecuzione
 - a. Quanto tempo per eseguirlo
7. Contatori i/o
 - a. Operazione di i/o
8. Contatori memoria virtuale
 - a. Operazione all'interno dell'address space
9. Porte per eccezioni
 - a. Cosa succede se vi sono eccezioni, con corretta gestione
10. Stato di uscita
 - a. Codice di ritorno dell'applicazione terminata

Mentre per quello che riguarda i servizi:

1. Creazione processi
 - a. Creare nuovi processi: nuove istanze processi attivi nel sistema
2. Apertura processi
 - a. Ritorna la chiave per ulteriore richiesta di servizi riguardanti quel processo
3. Richiesta/modifica di informazioni di gestione dei processi
 - a. Possibile modifica di attributi
4. Terminazione di processo
 - a. Finisco l'esecuzione del processo attivo
 - b. Chiamo un servizio di sistema

In windows per "lavorare" con un processo , il quale deve essere relazionato con gli oggetti , si ha una la **tabella degli oggetti** ovvero una struttura dati (a livello kernel) che permette di mettere in relazione un processo attivo con altri oggetti :



In caso di system call che crea altro processo, si aggiunge ulteriore entry della tabella, la quale viene relazionata al processo tramite maniglia(handle). Vediamo alcune strutture importanti utilizzate come parametri per le system call in windows :

```

typedef struct _SECURITY_ATTRIBUTES {
    DWORD nLength;
    LPVOID lpSecurityDescriptor;
    BOOL bInheritHandle;
} SECURITY_ATTRIBUTES

```

Descrizione

- struttura dati che specifica permessi

Campi

- nLength: va settato SEMPRE alla dimensione della struttura
- lpSecurityDescriptor: puntatore a una struttura SECURITY_DESCRIPTOR
- bInheritHandle: se uguale a TRUE un nuovo processo puo' ereditare l'handle a cui fa riferimento questa struttura

Questa struttura serve per gestire la sicurezza ; in dettaglio se ne occupa il secondo campo che è un puntatore : di solito passiamo NULL come valore -> valore di default. Invece l'ultimo parametro serve a specificare se la maniglia può essere ereditata . Vediamo ora quindi come creare un oggetto di tipo processo (processo) :

```

BOOL CreateProcess( LPCTSTR lpApplicationName,
                    LPTSTR lpCommandLine,
                    LPSECURITY_ATTRIBUTES lpProcessAttributes,
                    LPSECURITY_ATTRIBUTES lpThreadAttributes,
                    BOOL bInheritHandles,
                    DWORD dwCreationFlags,
                    LPVOID lpEnvironment,
                    LPCTSTR lpCurrentDirectory,
                    LPSTARTUPINFO lpStartupInfo,
                    LPPROCESS_INFORMATION lpProcessInformation)

```

Descrizione

- invoca la creazione di un nuovo processo (creazione di un figlio)

Restituzione

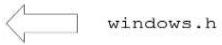
- nel chiamante: un valore diverso da zero in caso di successo, 0 in caso di fallimento.

Il primo parametro (puntatore costante a stringa) rappresenta il nome del programma eseguibile che deve essere eseguito all'interno del nuovo processo ; il secondo (puntatore a stringa) rappresenta la linea di comando (parametri del main) ; il settimo rappresenta l'informazione ambientale : puntatore unico ; l'ottavo rappresenta un puntatore che specifica una directory di lavoro; il nono (info di startup del processo che stiamo lanciando) ed il decimo sono due strutture dati per scambiare dati con il kernel (zona di memoria dove voglio il kernel mi ritorni il valore della chiamata); il sesto invece ci servono per dare dei flag . Vediamo in dettaglio le strutture dati :

```

typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION;

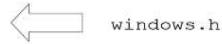
```




```

typedef struct _STARTUPINFO {
    DWORD cb;
    .....
    .....
} STARTUPINFO;

```



Vediamo un esempio di creazione di un processo :

```

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

int main(int argc, char *argv[]) {
    BOOL newprocess; STARTUPINFO si; PROCESS_INFORMATION pi;
    memset(&si, 0, sizeof(si)); memset(&pi, 0, sizeof(pi)); si.cb = sizeof(si);
    newprocess = CreateProcess(".\\figlio.exe",
                            ".\\figlio.exe pippo pluto",
                            NULL,
                            NULL,
                            FALSE,
                            NORMAL_PRIORITY_CLASS,
                            NULL,
                            NULL,
                            &si,
                            &pi);
    if (newprocess == FALSE) { printf("Chiamata fallita\n"); }
}

```

Memset (standard C) serve a settare la memoria : in questo caso setto tutti i campi a 0 (in questo caso) ; lo stesso per la seconda tabella . Vediamo ora un esempio :

```

// echoing-process.c : Defines the entry point for the console application.
// #define _CRT_SECURE_NO_WARNINGS
#include <windows.h>
#include <stdio.h> // actual stuff included will depend on macros such as _CRT_SECURE_NO_WARNINGS - you can set it manually or via IDE
#include <stdlib.h>

#define ECHOING_EXIT_CODE 2

int main(int argc, char *argv[]) {
    BOOL newprocess;
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    DWORD exitcode;
    char buff[128];
    memset(&si, 0, sizeof(si));
    memset(&pi, 0, sizeof(pi));
    si.cb = sizeof(si);

    if (argc == 2) {
        printf("Starting up echoing process ....\n");
        fflush(stdout);
        newprocess = CreateProcessA((LPCSTR)argv[1], (LPSTR)argv[1], NULL, NULL, FALSE, NORMAL_PRIORITY_CLASS, NULL, NULL, (LPSTARTUPINFO)&si, &pi);
        // controllo creazione del nuovo processo
        if (newprocess == FALSE) {
            printf(" failure\n");
            // siamo noi stessi che chiamiamo su del so per uscire
            ExitProcess(1);
        };
        printf(" done\n");
        /* attendo che lo stato del processo generato sia disponibile
        per il processo che lo ha creato
        INFINITE è macro di windows.h-> attesa infinita
        */
        WaitForSingleObject(pi.hProcess, INFINITE);
        /* ritorna il codice di terminazione di un processo
        quello della maniglia
        */
        GetExitCodeProcess(pi.hProcess, &exitcode);
        printf("Echoing process exited with code %d\n", exitcode);
        fflush(stdout);
    }
    else{
        printf("\nHere I'm for echoing\n");
        fflush(stdout);
        while (1) {
            scanf("%s", buff);
            if (strcmp(buff, "bye") == 0) ExitProcess(ECHOING_EXIT_CODE);
            printf("%s\n", buff);
            fflush(stdout);
        }
    }
    // se non passo parametri va direttamente nel ramo else
}

```

Quindi l'output senza parametri è direttamente l'echo in stdout di quello che viene immesso, fino ad immissione della stringa "bye"; altrimenti se passo un parametro si ha che viene creato il nuovo processo. Tornando ai puntatori a caratteri : **In C si ha la codifica ascii ovvero per ogni carattere della stringa del nome si ha 1 byte in memoria (ambito UNIX) ; per quanto riguarda WINDOWS i nomi (rappresentabili tramite stringhe) sono rappresentati tramite codifica unicode (ogni carattere viene espresso con 2 byte)**. **In generale quindi ogni system call ha 3 possibili forme** : una anonima (non specifico se l'oggetto è ascii o unicode) , una ASCII e una UNICODE . Quindi aspetta all'ambiente di compilazione scegliere quale versione usare. Quindi la create process (versione anonima) sarà CreateProcessA (ASCII) oppure sarà CreateProcessW(UNICODE). MA come specifichiamo al compilatore se la stringa è ASCII o UNICODE ? Attraverso delle macro :

```

#ifndef _UNICODE
typedef wchar_t TCHAR;
#else
typedef char TCHAR;
#endif

```

Il quale comporta ad avere la seguente rappresentazione:

```

char oneChar = 'x';
wchar_t oneChar = L'x'

```

Dove la 'L' sta per LARGE. Vediamo un esempio :

```

_tcslen (TCHAR*)
funzione per determinare la lunghezza di una stringa di TCHAR

wprintf(wchar_t*)
funzione per l'output con stringa di formato specificata tramite wchar_t

wscanf(wchar_t*, ...)
funzione per l'input con stringa di formato specificata tramite wchar_t

```

Quindi in generale in WINDOWS si hanno questi tipi di dato/rappresentazione:

TypeDef	Definition
CHAR	char
PSTR or LPSTR	char*
PCSTR or LPCSTR	const char*
PWSTR or LPWSTR	wchar_t*
PCWSTR or LPCWSTR	const wchar_t*

Tornando al discorso di scanf e gets , visto che sono soggette a buffer overflow, per poterle utilizzare si deve usare la macro **_CRT_SECURE_NO_WARNINGS**. tornando al discorso dei processi possiamo utilizzare le seguenti funzioni :

```

VOID ExitProcess(UINT uExitCode)
Descrizione
• Richiede la terminazione del processo chiamante
Argomenti
• uExitCode: valore di uscita del processo e di tutti i thread terminati
da questa chiamata

```

```

DWORD WaitForSingleObject(HANDLE hHandle,
                         DWORD dwMilliseconds)

Descrizione
• permette di entrare in attesa fino a che un oggetto sia disponibile
Parametri
• hHandle: handle all'oggetto target
• dwMilliseconds: timeout
Restituzione
• WAIT_FAILED in caso di fallimento

```

```

int GetExitCodeProcess(
    HANDLE hProcess,
    LPDWORD lpExitCode
)

```

Descrizione
• richiede lo stato di terminazione di un processo
Parametri
• hProcess: handle al processo
• lpExitCode: puntatore all'area dove viene scritto il codice di uscita

Questa system call è **non bloccante**, e ritorna il valore **STILL_ACTIVE** nel caso in cui il processo target sia ancora attivo

```

C++

BOOL WINAPI TerminateProcess(
    _In_ HANDLE hProcess,
    _In_ UINT uExitCode
);

```

Parameters

hProcess [in]
A handle to the process to be terminated.

The handle must have the **PROCESS_TERMINATE** access right. For more information, see [Process Security and Access Rights](#).

uExitCode [in]
The exit code to be used by the process and threads terminated as a result of this call. Use the [GetExitCodeProcess](#) function to retrieve a process's exit value. Use the [GetExitCodeThread](#) function to retrieve a thread's exit value.

Vediamo ora degli esempi:

```

// ASCII-vs-UNICODE.c : Defines the entry point for the console application.
//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <tchar.h>

TCHAR *p = TEXT("francesco");
TCHAR *format = TEXT("%ls\n");

int main(int argc, _TCHAR* argv[])
{
    /*
        numero dei caratteri della stringa
        prima del terminatore
    */
    int size;
    size = _tcslen(p);
    printf("size is %d\n", size);
    printf("size of TCHAR is %d\n", sizeof(TCHAR));

    wprintf(format, p);
    wprintf(L"%d\n", 33);
    wprintf(L"cioa a tutti\n");

    return 0;
}

```

Dove la macro TEXT indica che la stringa che indichiamo è **TCHAR**. Vediamo ora le variabili d'ambiente :

```

LPTCH WINAPI GetEnvironmentStrings(void);

Descrizione
• acquisizione del valore delle variabili di ambiente
Parametri
• nessuno
Ritorno
• puntatore al blocco (sequenza di stringhe) di variabili d'ambiente

DWORD WINAPI GetEnvironmentVariable(_In_opt_ LPCTSTR lpName,
                                    _Out_opt_ LPTSTR lpBuffer, _In_ DWORD nSize );

BOOL WINAPI SetEnvironmentVariable(_In_ LPCTSTR lpName,
                                    _In_opt_ LPCTSTR lpValue );

BOOL WINAPI FreeEnvironmentStrings(_In_ LPTCH lpszEnvironmentBlock );

```

Dove la prima restituisce l'informazione sul puntatore dove ci sono le stringhe ambientali ; analogamente per una singola variabile (seconda sys call). Nota : a differenza di UNIX , in WINDOWS le informazioni ambientali vengono rappresentate come un'unica stringa in un'unica area : ogni valore è separato da '\0'. Ritorna un solo indirizzo quindi. L'ultima invece permette di "liberarci" dall'ambiente : elimino quindi le stringhe . Vediamo in dettaglio :

```

// environment audit on parent or child (if any)
// please compile with ASCII settings

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

int main(int argc, char *argv[]) {
    BOOL newprocess;
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    int i = 0;
    char b = *(char*)&argc;
    char path[MAX_PATH];
    LPVOID lpEnv;
    LPCTSTR lpszVariable;
    GetCurrentDirectory(MAX_PATH, pwd);
    printf("Process is %d - current directory is: %s\n", GetCurrentProcessId(), pwd);
    /* print argv elements */
    while (i < argc) {
        printf("arg %d is: %s\n", i, argv[i]);
        i++;
    }
    if (argc>1){
        memset(&si, 0, sizeof(si));
        memset(&pi, 0, sizeof(pi));
        si.cb = sizeof(si);
        printf("trying running the child command: %s\n", argv[1]);
        fflush(stdout);
        newprocess = CreateProcess(argv[1], "", NULL, NULL, FALSE, NORMAL_PRIORITY_CLASS, /*"A=10|B=20|0|0"*/, NULL, /*"MY_VARIABLE=3410|0"*/ NULL, &si, &pi);
        if (newprocess == FALSE) {
            printf("CreateProcess failed - no child command has been run\n");
            ExitProcess(-1);
        }
        //attendo processo child
        WaitForSingleObject(pi.hProcess, INFINITE);
        ExitProcess(0);
    }
    else{
        //ritorna un puntatore alla zona
        lpEnv = GetEnvironmentStrings();
        if (lpEnv == NULL) printf("GetEnvironmentStrings() failed.\n");
        else printf("GetEnvironmentStrings() is OK.\n");
        for (lpszVariable = (LPSTR)lpEnv; *lpszVariable; ) {
            printf("%s\n", lpszVariable);
            lpszVariable += strlen(lpszVariable) + 1;
        }
    }
}

```

Con output il seguente : se non ho ulteriori parametri (quindi ho solo argv[0] -> il nome dell'eseguibile) stampa solo il pid del processo eseguibile e mostra tutto l'elenco delle variabili; se invece specifico anche un parametro (argv[1]) : crea processo figlio il quale dopo aver mostrato il duo pid fa il retrieve delle variabili e le inserisce come parametri da inserirli come parametri nell'address space del figlio . Se invece quando creiamo il processo decommentiamo A=10 e B=20 , e se passiamo anche il secondo parametro (argv[1]) , oltre a stampare il pid del parent, parte il processo figlio e ti ritorna un nuovo ambiente : stampa solo le due stringhe A=10 e B=10