Sistemi Operativi

Laurea in Ingegneria Informatica
Università di Roma Tor Vergata
Docente: Francesco Quaglia

Gestione degli eventi

1. Nozioni di base: eventi e segnalazioni
2. Segnali UNIX
3. Messaggi/eventi Windows

Eventi nei sistemi operativi

- Sono particolari condizioni/situazioni rilevabili dal software del sistema operativo
- Tali condizioni sono però (o possono essere) di interesse anche per il software applicativo
- Il sistema operativo opera da ponte tra tali condizioni ed il software applicativo, per permettere a quest'ultimo di poter "reagire" all'accadimento di un evento
- Il meccanismo di basa su **segnalazione (o notifica) dell'evento**
- Gli eventi possono essere causati dall'esecuzione della stessa applicazione "destinataria" della segnalazione
- Oppure possono essere generati da componenti software esterni a tale applicazione (ad esempio da un componente di sistema)

Gestione degli eventi a livello applicativo

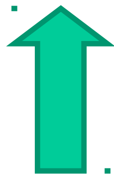
- Reagire all’accadimento di un evento implica che una applicazione esegua un modulo software denominato “gestore dell’evento”
- Le modalità secondo cui l’attivazione del gestore avviene sono differenti a seconda della tipologia di sistema operativo in cui ci troviamo
- Questo deriva dai diversi percorsi di sviluppo delle varie famiglie di sistemi
- Tra questi un fattore primario risulta essere il supporto (o il non supporto) per il multi-threading
- Ricordiamo che molte versioni di sistemi UNIX originariamente non supportavano il multi-threading
- Sistemi Windows sono invece nativamente multi-thread

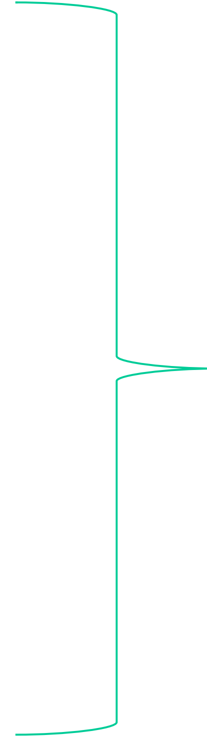
Un primo schema di gestione degli eventi

Thread execution

Execution flow variation to a routine that handles the event

Event *e* occurrence





Baseline UNIX style relying on the concept of software interrupt

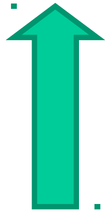
Kernel needs to be notified that we need the control flow variation

Uno secondo schema

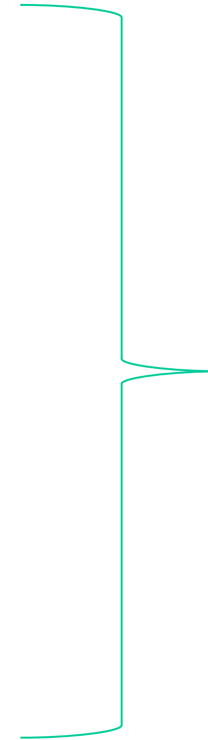
Spawn of a new thread running a routine that handles the event

New thread execution

Event ***e*** occurrence

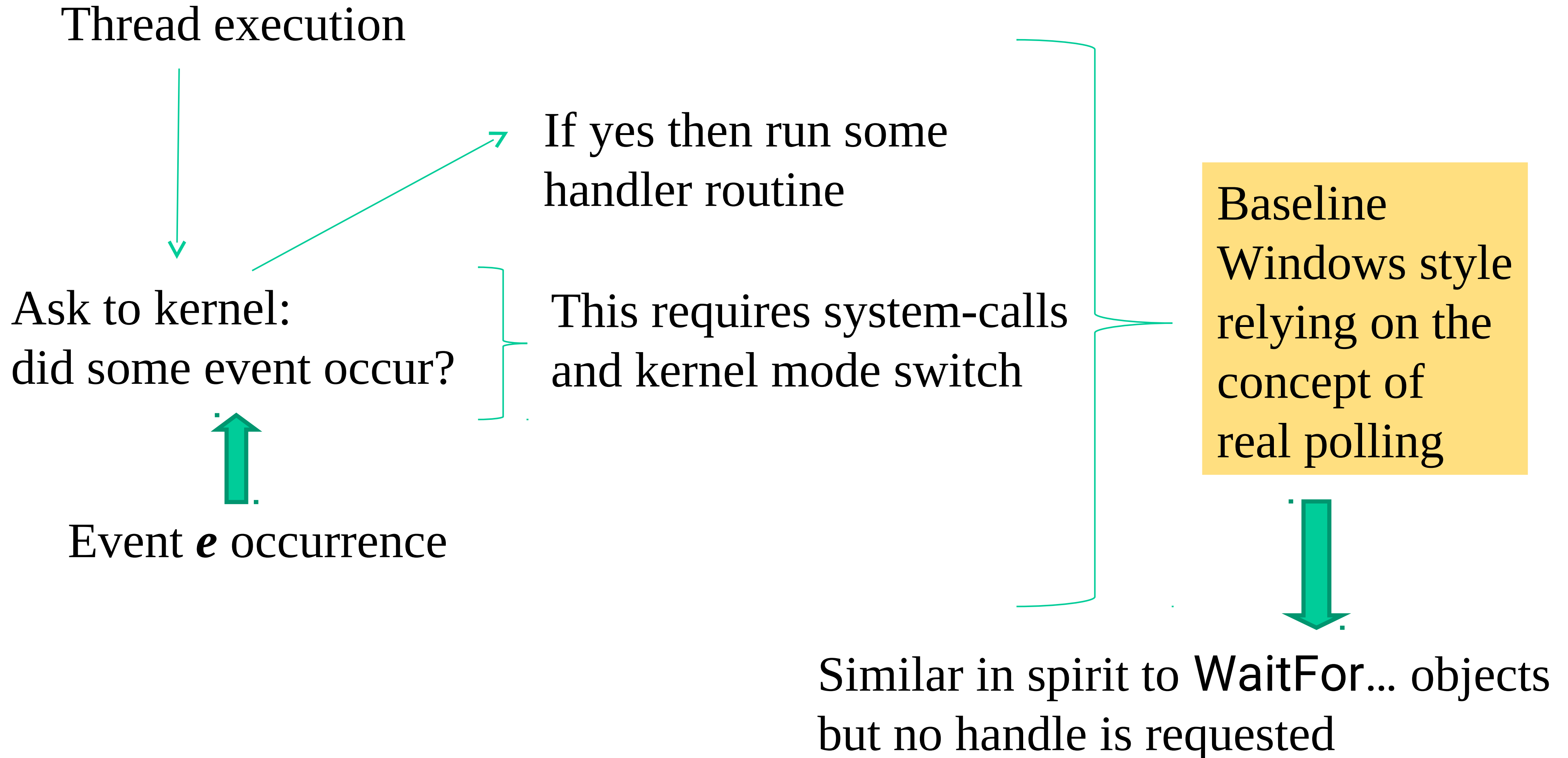


Kernel needs to be notified that we need to spawn the thread



Baseline Windows style relying on the concept of virtual polling

Uno terzo schema



Segnalazione di evento in sistemi UNIX

- Esiste un set predeterminato di eventi possibili, <u>il cui accadimento può quindi essere segnalato dal kernel</u>

- A ogni istante di tempo un processo risiede in uno dei seguenti tre stati di relazione verso l'evento, e quindi verso la relativa segnalazione
  - ✓ evento/segnalazione <u>ignorata implicitamente</u>
  - ✓ evento/segnalazione <u>ignorata esplicitamente</u>
  - ✓ evento/segnalazione <u>catturata</u>

- Il cambio di stato è ottenuto tramite system-call

- Si può cambiare lo stato di relazione tra processo ed evento/segnalazione in modo arbitrario durante il tempo di vita del processo stesso

Segnali UNIX più comuni

- SIGHUP (1): Hangup. Il processo riceve questo segnale quando il terminale a cui era associato viene chiuso (ad esempio nel caso di un xterm)

- SIGINT (2): Interrupt. Il processo riceve questo segnale quando l'utente preme la combinazione di tasti di interrupt (solitamente Control+C)

- SIGQUIT (3): Quit. Simile a SIGINT, ma in più, **in caso di terminazione del processo**, il sistema genera un "core dump" , ovvero un file che contiene lo stato della memoria al momento in cui il segnale SIGQUIT è stato ricevuto. Solitamente SIGQUIT viene generato premendo i tasti Control+\

- SIGILL (4): Illegal Instruction. Il processo riceve questo segnale nel caso in cui tenti di eseguire un'istruzione proibita (o inesistente)

- SIGKILL (9): Kill. Questo segnale non può essere catturato in nessun modo dal processo ricevente, **che non può fare altro che terminare**. Mandare questo segnale è il modo per ``uccidere'' un processo

- SIGSEGV (11): Segmentation violation. Il processo riceve questo segnale quando tenta di eseguire un accesso non supportato all'interno dello spazio di indirizzamento

- SIGTERM (15): Termination. Inviato come richiesta non forzata di terminazione

- SIGALRM (14): Alarm. Inviato allo scadere del conteggio dell'orologio di allarme

- SIGUSR1, SIGUSR2 (10, 12): User defined. Non hanno un significato preciso, e possono essere utilizzati per implementare un qualsiasi protocollo di comunicazione e/o sincronizzazione

- SIGCHLD (17): Child death. Inviato ad un processo quando uno dei suoi figli termina

Richiesta esplicita di emissione di segnale

int kill(int pid, int segnale)

Descrizione richiede l'invio di un segnale

Argomenti 1) pid: identificatore di processo destinatario del segnale (**ovvero il suo main thread**)

2) segnale: specifica del numero del segnale da inviare

Restituzione -1 in caso di fallimento

Default al lato destinazione

- tutti i segnali tranne SIGKILL e SIGCHLD sono ignorati implicitamente, e al loro arrivo il processo destinatario termina
- SIGCHLD è ignorato implicitamente, ma il suo arrivo non provoca la terminazione del processo destinatario
- SIGKILL non è ignorabile, esso provoca la terminazione del processo destinatario

Richiesta di emissione di segnale per il thread corrente

```
int raise(int segnale)
```

Descrizione richiede l'invio di un segnale al **thread** corrente

Argomenti segnale: specifica del numero del segnale da inviare

Restituzione 0 in caso di successo

Segnali temporizzati

- un processo può programmare il sistema operativo ad inviargli il segnale SIGALRM dopo lo scadere di un certo tempo (in questo caso si dice che il conteggio dell'orologio di allarme è terminato)

```
unsigned alarm(unsigned time)
```

Descrizione invoca l'invio del segnale SIGALRM a se stessi

Argomenti time: tempo allo scadere del quale il segnale SIGALRM deve essere inviato

Restituzione tempo restante prima che un segnale SIGALRM invocato da una chiamata precedente arrivi

le impostazioni di `alarm()` non vengono ereditate dopo una `fork()` da un processo figlio, e modifiche successive sulle impostazioni dell'orologio di allarme sono del tutto indipendenti le une dalle altre

Catturare/ignorare segnali

```
void (*signal(int sig, void (*ptr)(int)))(int)
```

Descrizione specifica il comportamento di ricezione di un segnale

Argomenti 1) sig: specifica il numero del segnale da trattare

2) ptr: puntatore alla funzione di gestione del segnale o SIG_DFL o SIG_IGN

Restituzione SIG_ERR in caso di errore altrimenti il valore della precedente funzione di gestione del segnale che viene sovrascritto con ptr

- SIG_DFL setta il comportamento di default
- SIG_IGN ignora il segnale esplicitamente (importante per il segnale SIGCHLD)

Eredità delle impostazioni

- il comportamento associato alla ricezione di un segnale viene ereditato da processi figli

- per quanto riguarda la famiglia execX(), solo le impostazioni di SIG_IGN e SIG_DFL vengono mantenute, mentre per ogni segnale armato con una specifica funzione di gestione viene automaticamente settato il comportamento di default

- infatti il codice della funzione di gestione potrebbe non essere più presente dopo la execX()

- ovviamente, il nuovo programma può definire una sua nuova gestione dei segnali

Meccanismo di consegna dei segnali

- basato sul concetto di interruzione del flusso di esecuzione corrente
- l’interruzione viene attuata dal kernel al prossimo re-schedule del thread (interruzione software) in user mode
- una applicazione (un thread) in stato non-ready viene rimessa ready dal kernel in caso di arrivo di segnalazione
- questo porta all’aborto (fallimento) della system-call bloccante correntemente in atto (errno viene settato a EINTR)
- d‘altro canto l’aborto della system-call bloccante permettere gestioni “timely” dei segnali stessi
- system-call abortite per effetto di segnalazioni non sono ripristinate automaticamente

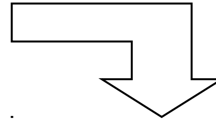
errno vs multi-threading

- Nello standard POSIX 1 **errno** è definita come una variabile `extern`
- Essa è quindi ad istanza singola nell'addresss space e la sua gestione non è ``thread-safe"
- Tale approccio era adeguato per sistemi a processi, non per sistemi multi-thread
- Attualmente (a partire dallo standard POSIX 1.c) **errno** è definita come una macro di invocazione ad una funzione di lettura del codice di errore registrato in una variabile "thread local"
- Il tutto rende la gestione di **errno** thread-safe anche in applicazioni multithread

Corretta invocazione di system-call bloccanti

```
while( systemcall_XX() == -1 )
        if ( errno != EINTR) {
            printf("Errore");
            exit(EXIT_FAILURE);
        }
```

Il classico schema sottostante non basta



```
if( systemcall_XX() == -1 ){
        printf("Errore");
            exit(EXIT_FAILURE);
        }
```

Attesa di un qualsiasi segnale

int pause(void)

Descrizione blocca il **thread** in attesa di un qualsiasi segnale

Restituzione sempre –1 (poiché è una system call interotta da segnale)

- `pause()` ritorna sempre al completamento della funzione di handling del segnale

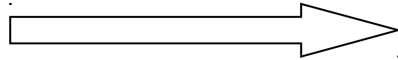
- non è possibile sapere direttamente da `pause()` (ovvero tramite valore di ritorno) quale segnale ha provocato lo sblocco; si può rimediare facendo si che l'handler del segnale modifichi il valore di qualche variabile globale o thread_local

Un semplice esempio

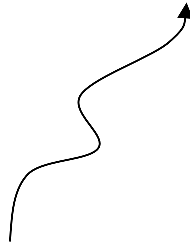
```
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>


 void gestione_timeout() {
     printf(“I’m alive!\n");
     alarm(5);
 }

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

     alarm(5);
     signal(SIGALRM, gestione_timeout);

     while(1) pause();
}
```

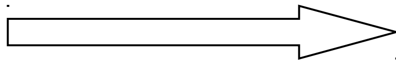
Corse critiche

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

char c;
int x, y, i;

void gestione_timeout() {
   printf(“I’m alive! x = %d, y = %d\n“,x,y);
   alarm(5);
}

int main(int argc, char *argv[]) {
   signal(SIGALRM, gestione_timeout);
   alarm(5);

   while(1)  x = y = i++ % 1000;
}
```



i valori di x e y possono essere diversi

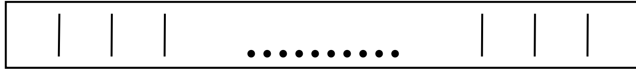




lo statement del C può essere eseguita in modo non atomico

Stuttura dati per la consegna dei segnali

Signal mask



Il sistema operativo aggiorna la maschera dei segnali in modo autonomo durante l'esecuzione di un suo modulo o su richiesta di altri processi (ovvero per effetto della chiamata alla system call `kill()`)

- se il thread è in stato di blocco e il segnale richiede l'esecuzione di una funzione di gestione (sia essa di default oppure no), allora il thread viene passato nello stato ready

- il punto di ritorno effettivo (valore da assegnare al PC) quando la maschera dei segnali indica la presenza di almeno un segnale richiede l'esecuzione di una funzione di gestione avviene al momento del ritorno in modo utente

- consegne multiple dello stesso segnale possono essere perse

Non atomicità dei gestori

- Un gestore di segnale impostato con `signal()` può essere a sua volta interrotto
- Questo implica la non atomicità dell’esecuzione del gestore rispetto alle possibili segnalazioni (di qualsiasi tipo)
- Chiaramente gli effetti possono essere molteplici e negativi
  - ✓ stack overflow
  - ✓ deadlock in caso di accesso a risorse quali mutex e semafori
  - ✓ Impossibilità di azioni complesse atomiche sui segnalazioni critiche

Set di segnali

- il tipo **sigset_t** rappresenta un insieme di segnali (signal set)
- funzioni (definite in signal.h) per la gestione dei signal set:

int sigemptyset (sigset_t *set) svuota il set

int sigfillset (sigset_t *set) inserisce tutti i segnali in set

int sigaddset (sigset_t *set, int signo) aggiunge il segnale signo a set

int sigdelset (sigset_t *set, int signo) toglie il segnale signo da set

int sigismember (sigset_t *set, int signo) controlla se signo è in set

Gestione della signal mask

int sigprocmask(int *how*, const sigset_t **set*, sigset_t **oset*)

Descrizione imposta la gestione della signal mask

Argomenti 1) how: indica in che modo intervenire sulla signal mask e può valere:

SIG_BLOCK: i segnali indicati in set sono aggiunti alla signal mask,

SIG_UNBLOCK: i segnali indicati in set sono rimossi dalla signal mask;

SIG_SETMASK: La nuova signal mask diventa quella specificata da set

2) set: il signal set sulla base del quale verranno effettuate le modifiche

3) oset: se non è NULL, nella relativa locazione verrà scritto il valore della signal mask PRIMA di effettuare le modifiche richieste

Restituzione –1 in caso di errore

int sigpending(sigset_t **set*);

Descrizione restituisce l’insieme dei segnali che sono pendenti

Argomenti set: il signal set in cui verrà scritto l’insieme dei segnali pendenti

Restituzione –1 in caso di errore

Permette l’implementazione di
schemi di polling, in particolare
nel caso di applicazioni multi-thread

Gestione “affidabile” dei segnali

int sigaction(int *sig*, const struct sigaction * *act*, struct sigaction * *oact*);

Descrizione permette di esaminare e/o modificare l’azione associata ad un segnale

Argomenti 1) sig: il segnale interessato dalla modifica;

2) act: indica come modificare la gestione del segnale

3) oact: in questa struttura vengono memorizzate le impostazioni precedenti per la gestione del segnale

Restituzione –1 in caso di errore

La struttura sigaction

The sigaction structure is defined as something like

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

Supporti per l'atomicità

On some architectures a union is involved - do not assign to both sa_handler and sa_sigaction.

The sa_restorer element is obsolete and should not be used. POSIX does not specify a sa_restorer element.

sa_handler specifies the action to be associated with signum and may be SIG_DFL for the default action, SIG_IGN to ignore this signal, or a pointer to a signal handling function.

sa_mask gives a mask of signals which should be blocked during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the SA_NODEFER or SA_NOMASK flags are used.

sa_flags specifies a set of flags which modify the behaviour of the signal handling process. It is formed by the bitwise OR of zero or more of the following:

Informazioni per la gestione del segnale

```
siginfo_t {
    int       si_signo;  /* Signal number */
    int       si_errno;  /* An errno value */
    int       si_code;   /* Signal code */
    pid_t     si_pid;    /* Sending process ID */
    uid_t     si_uid;    /* Real user ID of sending process */
    int       si_status; /* Exit value or signal */
    clock_t   si_utime;  /* User time consumed */
    clock_t   si_stime;  /* System time consumed */
    sigval_t  si_value;  /* Signal value */
    int       si_int;    /* POSIX.1b signal */
    void *    si_ptr;    /* POSIX.1b signal */
    void *    si_addr;   /* Memory location which caused fault */
    int       si_band;   /* Band event */
    int       si_fd;     /* File descriptor */
}
```

Un esempio

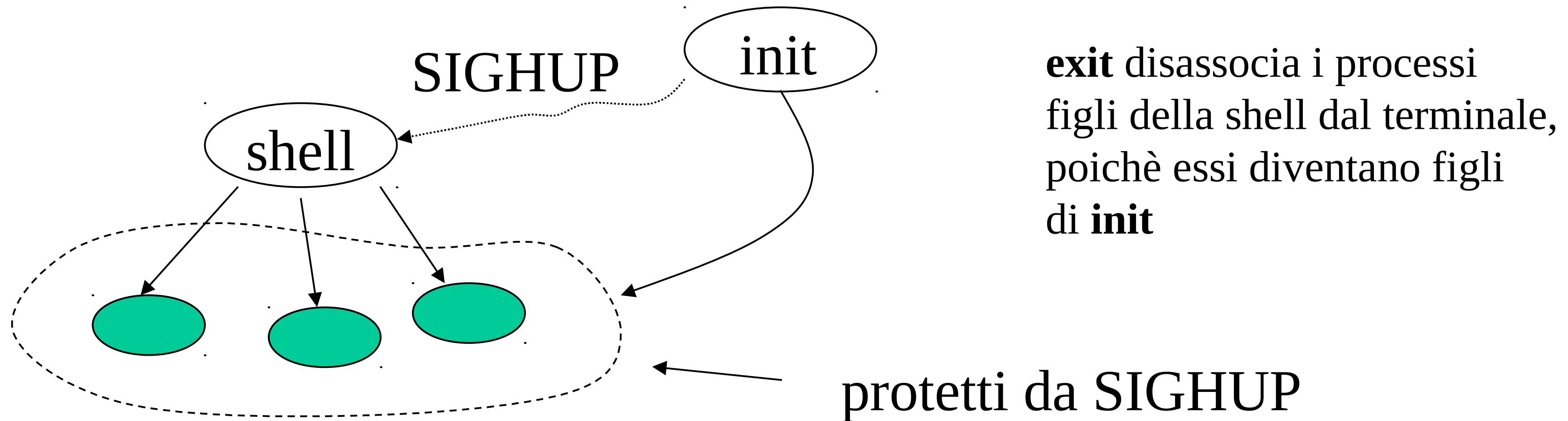
```
void gestione_sigsegv(int dummy1, siginfo_t *info, void *dummy2){
    unsigned int address;
    address = (unsigned int) info->si_addr;
    printf("segfault occurred (address is %x)\n",address);
    fflush(stdout);
}
```

```
act.sa_sigaction = gestione_sigsegv;
act.sa_mask =  set;
act.sa_flags = SA_SIGINFO;
act.sa_restorer = NULL;
sigaction(SIGSEGV,&act,NULL);
```

Trattamento del segnale SIGHUP sulle shell

- comandi eseguiti in background vengono terminati o non alla chiusura del terminale associato alla shell dipendendo dalle impostazioni sul trattamento di SIGHUP

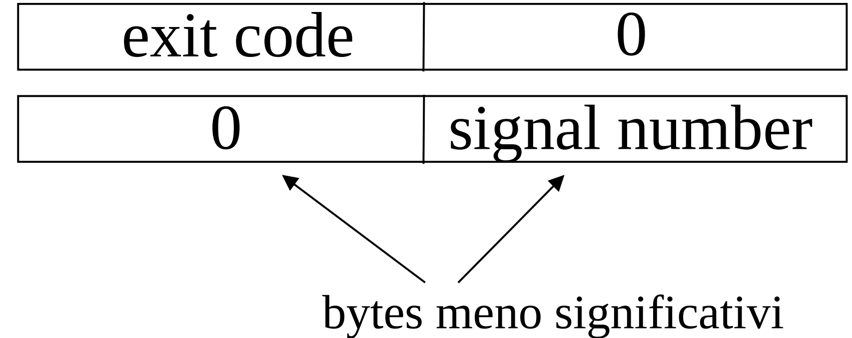
1. Non terminano se il costrutto fork/exec imposta il trattamento a SIG_IGN (argomento **nohup** sulla linea di comando)
2. Terminano in ogni altro caso a meno che il terminale sia chiuso per effetto della system call **exit** eseguita dalla shell



exit disassocia i processi figli della shell dal terminale, poichè essi diventano figli di **init**

Determinare il modo di terminazione di un processo

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <errno.h>
#include <sys/wait.h>
#define COMMAND_LENGTH 1024

int main (int argc, char *argv[]){
  int i, status;
  if (argc<2) {  printf("Need at least a program-name\n"); exit(EXIT_FAILURE); }
  if ((i=fork()) == 0) { execvp(argv[1],&argv[1]);
                                   printf("Can't execute program %s\n",argv[1]);
                        exit(EXIT_FAILURE);
  } else if (i<0) { printf("Can't spawn process for error %d\n", errno);   exit(EXIT_FAILURE); }
  wait(&status);

   if ((status & 255) == 0) {
        printf("\nProcess regularly exited with exit status %d\n\n", (status>>8) & 255);  }
   else if ( ((status>>8) & 255) == 0) {
            printf("\nProcess abnormally terminated by signal: %d\n\n",  status & 255);
          }
}
```



Sistemi Windows

In Windows si distinguono due categorie di eventi

- **Eventi di sistema**, simili ai segnali UNIX, eccetto per la modalità con la quale l'evento stesso viene ad essere processato

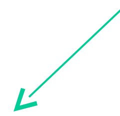
- **Messaggi-evento**, utilizzati come strumento di notifica e comunicazione, specialmente in contesti dove i processi gestiscono reali "finestre"

Windows vs UNIX - considerazioni di base

- I segnali UNIX sono nati in contesto di sistemi a processi (single-thread)
- Essi quindi operano secondo schemi basati su interruzione del flusso di esecuzione
- I sistemi Windows sono nativamente multi-thread
- Eventi/messaggi-evento Windows vengono quindi gestiti secondo schemi basati su polling
- Il polling per gli eventi di sistema è virtuale (spawn dinamico di thread per la gestione degli eventi in callback), eccetto che in alcuni casi, in cui lo schema UNIX viene adottato
- Quello per i messaggi-evento è reale (basato su thread applicativi dedicati)

Eventi di sistema Windows

```
BOOL WINAPI HandlerRoutine( _In_  DWORD dwCtrlType )
```



Value	Meaning
CTRL_C_EVENT 0	A CTRL+C signal was received, either from keyboard input or from a signal generated by the **GenerateConsoleCtrlEvent** function.
CTRL_BREAK_EVENT 1	A CTRL+BREAK signal was received, either from keyboard input or from a signal generated by **GenerateConsoleCtrlEvent**.
CTRL_CLOSE_EVENT 2	A signal that the system sends to all processes attached to a console when the user closes the console (either by clicking **Close** on the console window's window menu, or by clicking the **End Task** button command from Task Manager).
CTRL_LOGOFF_EVENT 5	A signal that the system sends to all console processes when a user is logging off. This signal does not indicate which user is logging off, so no assumptions can be made. Note that this signal is received only by services. Interactive applications are terminated at logoff, so they are not present when the system sends this signal.
CTRL_SHUTDOWN_EVENT 6	A signal that the system sends when the system is shutting down. Interactive applications are not present by the time the system sends this signal, therefore it can be received only be services in this situation. Services also have their own notification mechanism for shutdown events. For more information, see **Handler**. This signal can also be generated by an application using **GenerateConsoleCtrlEvent**.

Organizzazione degli handler per eventi di sistema

- Essi vengono attivati dal kernel a cascata secondo uno schema basato su ``callback’’
- L’ordine di attivazione rispetta la pila delle registrazioni
- Il primo handler invocato che ritorna TRUE interrompe la catena di attivazione
- L’ultimo handler di tale catena è per default la system-call `ExitProcess()`
- Questo è il motivo per cui un processo Windows senza gestori di eventi di sistema termina qualora tali eventi vengano inoltrati verso di lui

Registrazione di gestori di evento

Aggiunta o rimozione

C++

```
BOOL WINAPI SetConsoleCtrlHandler(
  _In_opt_  PHANDLER_ROUTINE HandlerRoutine,
  _In_      BOOL Add
);
```

Parameters

HandlerRoutine [in, optional]

A pointer to the application-defined **HandlerRoutine** function to be added or removed. This parameter can be **NULL**.

Add [in]

If this parameter is **TRUE**, the handler is added; if it is **FALSE**, the handler is removed.

If the *HandlerRoutine* parameter is **NULL**, a **TRUE** value causes the calling process to ignore CTRL+C input, and a **FALSE** value restores normal processing of CTRL input. This attribute of ignoring or processing CTRL+C is inherited by child processes.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Inoltro di eventi di sistema on-demand

```
BOOL WINAPI GenerateConsoleCtrlEvent( _In_  DWORD dwCtrlEvent,
                                      _In_  DWORD dwProcessGroupId );
```

Nel caso un processo sia attivato tramite `CreateProcess()` con parametro
CREATE_NEW_PROCESS_GROUP
il suo identificatore corrisponde ad un identificatore di gruppo

Compliance con sistemi UNIX - il servizio signal

Tali segnali possono essere inoltrati
Tramite il servizio `race()` anche su Windows

Note

La funzione `signal` permette ad un processo di scegliere uno dei vari modi per gestire un segnale di interrupt proveniente dal sistema operativo. L'argomento `sig` è l'interrupt al quale risponde `signal`; deve essere una delle seguenti costanti manifesto, che sono definite in SIGNAL.H.

Valore `sig`	Descrizione
`SIGABRT`	Terminazione anomala
`SIGFPE`	Errore a virgola mobile
`SIGILL`	Istruzione non valida
`SIGINT`	Segnale CTRL+C
`SIGSEGV`	Accesso alla memoria non valido
`SIGTERM`	Richiesta di terminazione

Compatibilità solo nominale

Messaggi evento in Windows

- i messaggi evento sono una forma di comunicazione/notifica asincrona

- tuttavia tali messaggi non possono interrompere un flusso di esecuzione

- pertanto il relativo handler non può essere eseguito finchè un thread non decide esplicitamente di processare la notifica

- l'approccio non utilizza quindi la nozione di interrupt, bensì quella di polling

- i messaggi evento di Windows sono caratterizzati da un numero che ne identifica il tipo e da due valori numerici (parametri)

“Armare” messaggi evento

- per poter eseguire l’handler di un messaggio evento, un processo deve necessariamente creare un oggetto di tipo ``finestra”
- non è necessario che tale finestra venga effettivamente visualizzata

```
ATOM RegisterClass(const WNDCLASS *lpWndClass)
```

Descrizione
- crea un nuovo tipo di finestra

Argomenti
- `lpWndClass`: indirizzo di una struttura di tipo WNDCLASS contenente le informazioni relative al nuovo tipo di finestra. Tra esse si trova anche il puntatore all'handler che gestisce i messaggi evento

Restituzione
- 0 in caso di fallimento

La struttura WNDCLASS

```
typedef struct _WNDCLASS {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HANDLE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS;
```

- style: informazioni di stile sulla finestra; un valore tipico è ``CS_HREDRAW | CS_VREDRAW''
- lpfnWndProc: indirizzo della funzione che fungerà da handler di tutti i messaggi evento

- cbClsExtra: byte extra da allocare per esigenze del programmatore; tipicamente è 0
- cbWndExtra: altri byte extra da allocare per esigenze del programmatore; tipicamente è 0
- **hInstance: handler all'istanza del processo che ospita la procedura di finestra. NULL indica il processo corrente**
- hIcon: handler ad un'icona da usare per la finestra; come default usare il

  valore restituito dalla system call ``LoadIcon(NULL, IDI_APPLICATION)”
- hCursor: handler ad un cursore da usare nella finestra; come default usare il valore restituito dalla system call ``LoadCursor(NULL, IDC_ARROW)”
- hbrBackground: handle al pennello di background
- lpszMenuName: stringa che specifica il nome del menu di default da usare. NULL se non ci sono menu
- **lpszClassName: stringa indicante il <u>nome associato a questo tipo di finestra</u>**

Struttura dell’handler

- come si vede dal campo ``lpfnWndProc'' un solo handler gestisce tutti i messaggi evento
- dunque all'interno di questo handler si dovrà usare il costrutto ``switch{}'' per gestire i diversi tipi di messaggi evento
- l'handler di un messaggio evento ha il seguente prototipo (naturalmente il nome effettivo dell'handler può variare secondo le scelte del programmatore)

```
LRESULT CALLBACK WindowProcedure(
      HWND hwnd,
      UINT uMsg,
      WPARAM wParam,
      LPARAM lParam
)
```

Descrizione
- viene chiamata per gestire messaggi evento

Parametri
- hwnd: handle alla finestra che ha ricevuto il messaggio evento

- uMsg: tipo del messaggio evento ricevuto

- wParam: primo parametro del messaggio evento

- lParam: secondo parametro del messaggio evento

Restituzione
- il valore ritornato da questo handler dipende dal messaggio evento ricevuto

Creazione di in un oggetto finestra

```
HWND CreateWindow(LPCTSTR lpClassName,
             LPTSTR lpWindowName,
             DWORD dwStyle,
             int x,
             int y,
             int nWidth,
             int nHeight,
             HWND hWndParent,
             HMENU hMenu,
             HANDLE hInstance,
             PVOID lpParam)
```

Descrizione

- crea un nuovo oggetto finestra; non necessariamente la visualizza sullo schermo

Restituzione

- handle alla nuova finestra in caso di successo, NULL in caso di fallimento

Paramteri

- lpClassName: una stringa contenente il nome del tipo di finestra, precedentemente definito tramite RegisterClass()
- lpWindowName: una stringa contenente l'intestazione della finestra
- dwStyle: stile della finestra (default WS_OVERLAPPEDWINDOW)
- x: posizione iniziale della finestra (coordinata x); usare CW_USEDEFAULT
- y: posizione iniziale della finestra (coordinata y); usare CW_USEDEFAULT
- nWidth: dimensione della finestra (coordinata x); usare CW_USEDEFAULT
- nHeight: dimensione della finestra (coordinata y); usare CW_USEDEFAULT
- hWndParent: handle alla finestra genitrice; NULL è il default
- hMenu: handle ad un menu; se non ci sono menu usare NULL
- hInstance: handle ad una istanza del processo di riferimento; NULL è il default
- lpParam: puntatore a parametri di creazione; NULL è il default

Polling di messaggi evento

- dopo aver eseguito `RegisterClass()` e `CreateWindow()` il thread può cominciare a ricevere i messaggi evento entranti con il seguente loop:

```
while(GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
```

Da chiavi virtuali a WM_CHAR per messaggi evento relativi ai devices

- `msg` è una struttura di tipo `MSG` e `GetMessage()` è definita come:

```
INT GetMessage(LPMSG lpMsg,
      HWND hWnd,
      UINT wMsgFilterMin,
      UINT wMsgFilterMax)
```

Descrizione

- riceve un messaggio nuovo. Ritorna solo se c‘è un nuovo messaggio pendente o se viene ricevuto un messaggio di tipo WM_QUIT

Parametri

- lpMsg: indirizzo ad una struttura di tipo MSG
- hWnd: handle della finestra di cui si vogliono ricevere i messaggi; NULL per ricevere messaggi da tutte le finestre associate al processo
- wMsgFilterMin: valore più basso del tipo di messaggi evento da ricevere; 0 non pone limiti inferiori
- wMsgFilterMax: valore più alto del tipo di messaggi evento da ricevere; 0 non pone limiti superiori

Restituzione

- -1 se c‘è un errore, 0 se viene ricevuto un messaggio di tipo WM_QUIT, un valore diverso da 0 e -1 se viene ricevuto un altro messaggio

Invio di messaggi evento

```
LRESULT SendMessage(HWND hWnd, UINT Msg, WPARAM wParam,
                                    LPARAM lParam)
```

Descrizione

- invia un messaggio ad una finestra ; il messaggio verrà posto <u>in testa alla coda</u> dei messaggi-evento

Parametri

- hWnd: handle alla finestra che deve ricevere il messaggio; HWND_BROADCAST per mandare il messaggio a tutte le finestre prive di genitore (top level)
- Msg: intero che identifica il tipo di messaggio
- wParam: primo parametro del messaggio
- lParam: secondo parametro del messaggio

Restituzione

- ritorna il risultato del processamento del messaggio (true/false), ritorna quindi soltanto quando il messaggio evento è stato processato

Notifica non bloccante

```
BOOL PostMessage(HWND hWnd, UINT Msg, WPARAM wParam,
                      LPARAM lParam)
```

Descrizione
- invia un messaggio evento ad una finestra; il messaggio verrà posto <u>in fondo alla coda</u> dei messaggi evento

Parametri
- hWnd: Handle alla finestra che deve ricevere il messaggio HWND_BROADCAST per mandare il messaggio a tutte le finestre prive di genitore (top level)
- Msg: intero che identifica il tipo di messaggio
- wParam: parametro del messaggio
- lParam: secondo parametro del messaggio

Restituzione
- 0 in caso di fallimento, un valore diverso da 0 in caso di successo; non attende il processamento del messaggio evento

Tipi di messaggi evento

```
UINT RegisterWindowMessage(LPCTSTR lpString)
```

Descrizione

- crea un nuovo tipo di messaggio

Parametri

- lpString: stringa che assegna un nome al tipo di messaggio

Restituzione

- 0 indica un errore, ogni altro valore rappresenta il nuovo tipo di messaggio creato

Messaggi evento e default

- i messaggi evento che vengono inviati dal sistema operativo hanno un **tipo ben definito** all'interno dei file header di Windows

- ad esempio quando viene eseguita la system call `CreateWindow()` il sistema operativo invia alla finestra vari messaggi evento tra cui uno di tipo WM_CREATE

- verrà quindi eseguita la window procedure usando in input questo messaggio

- visto che i tipi di messaggi evento di sistema sono nell'ordine delle centinaia non ci si aspetta che il programmatore debba gestirli tutti

- Windows fornisce un gestore di default `DefWindowProc()` che prende in ingresso gli stessi parametri della window procedure e restituisce lo stesso tipo di valore restituito dalla window procedure

NOTA: messaggi evento ad alta priorità (ad esempio generati dal Sistema o con chiamata sincrona di spedizione da parte di un thread) non rispettano la disciplina di accodamento

Classi di finestre predefinite

- come ausilio di programmazione, WINAPI offre classi di finestre precodificate, quali ad esempio:
  - ✓ Button
  - ✓ Boxes
- queste incorporano funzioni predefinite per processare messaggi evento, la cui esecuzione può essere attivata inoltranto appunto specifici messaggi evento verso la finestra
- ad esempio, la classe di finestre “EDIT” processa il messaggio evento WM_GETTEXT fornendo su un buffer (identificato dal messaggio evento tramite parametri) il contenuto del testo inoltrato sulla finestra
- la stessa classe processa il messaggio evento WM_SETTEXT mostrando sulla finestra il testo attualmente presente in un buffer (identificato dal messaggio evento tramite parametri)

Ulteriori API basiche

```
HMENU WINAPI CreateMenu(void);

BOOL WINAPI AppendMenu( _In_ HMENU hMenu,
                        _In_ UINT uFlags,
                        _In_ UINT_PTR uIDNewItem,
                        _In_opt_ LPCTSTR lpNewItem );
```

Tipo di rappresentazione

Codice o handle identificativo

Nome della entry

Handle alla finestra da mostrare e modalità di visualizzazione

```
BOOL WINAPI ShowWindow( _In_ HWND hWnd, _In_ int  nCmdShow );
```