

Sistemi Operativi II

Corso di Laurea in Ingegneria Informatica

Facolta' di Ingegneria, Universita' "La Sapienza"

Docente: Francesco Quaglia

## **Gestione di eventi asincroni:**

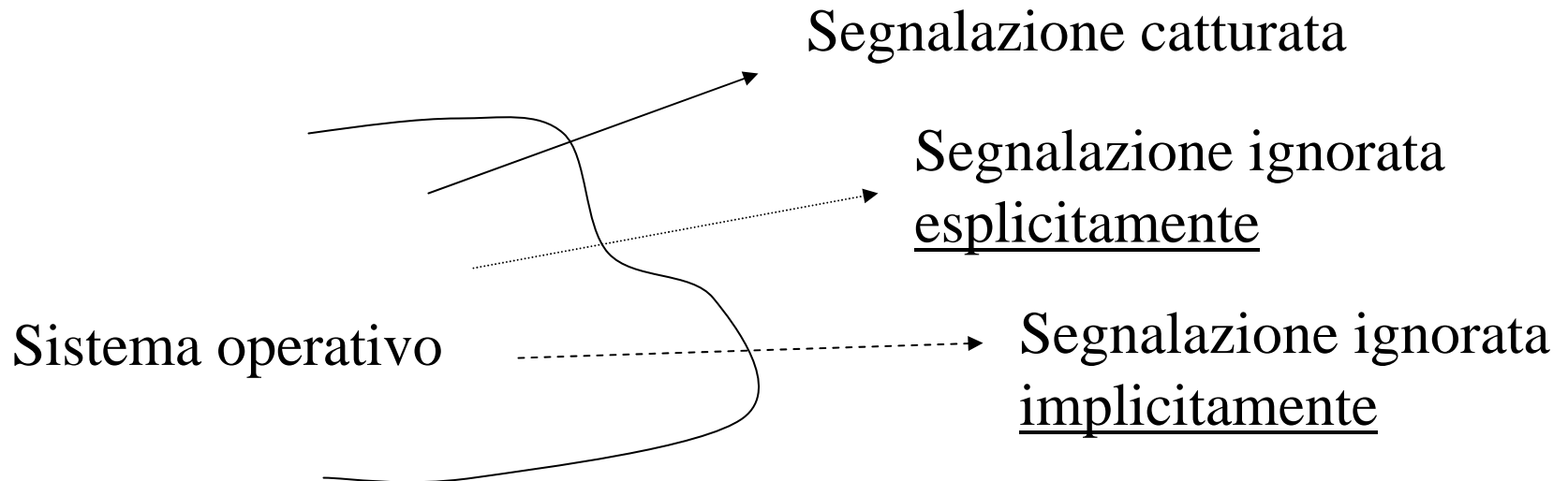
1. Nozioni di base
2. Segnali UNIX
3. Messaggi evento Windows

# Nozioni di base

- i meccanismi di segnalazione sono un semplice mezzo tramite il quale l'accadimento di un evento può essere notificato ad un processo
- il processo può eventualmente eseguire specifiche azioni associate a tale accadimento
- una segnalazione è talvolta classificata come un semplice messaggio, tuttavia, essa differisce da un messaggio in diversi modi, quali
  1. una segnalazione viene in genere inviata occasionalmente da un processo, molto più spesso dal sistema operativo come risultato di un evento eccezionale (es. errore di calcolo in virgola mobile)
  2. una segnalazione può non avere contenuto informativo, in particolare, il ricevente può non arrivare a conoscere l'identità di chi emette la segnalazione

# Trattamento delle segnalazioni

- trattare le segnalazioni è molto importante per i programmatori perchè il sistema operativo invia segnalazioni indipendentemente dalla “volontà” dei processi
- se i processi decidono di “non ignorarle esplicitamente” o di non “catturarle”, essi possono venir automaticamente terminati

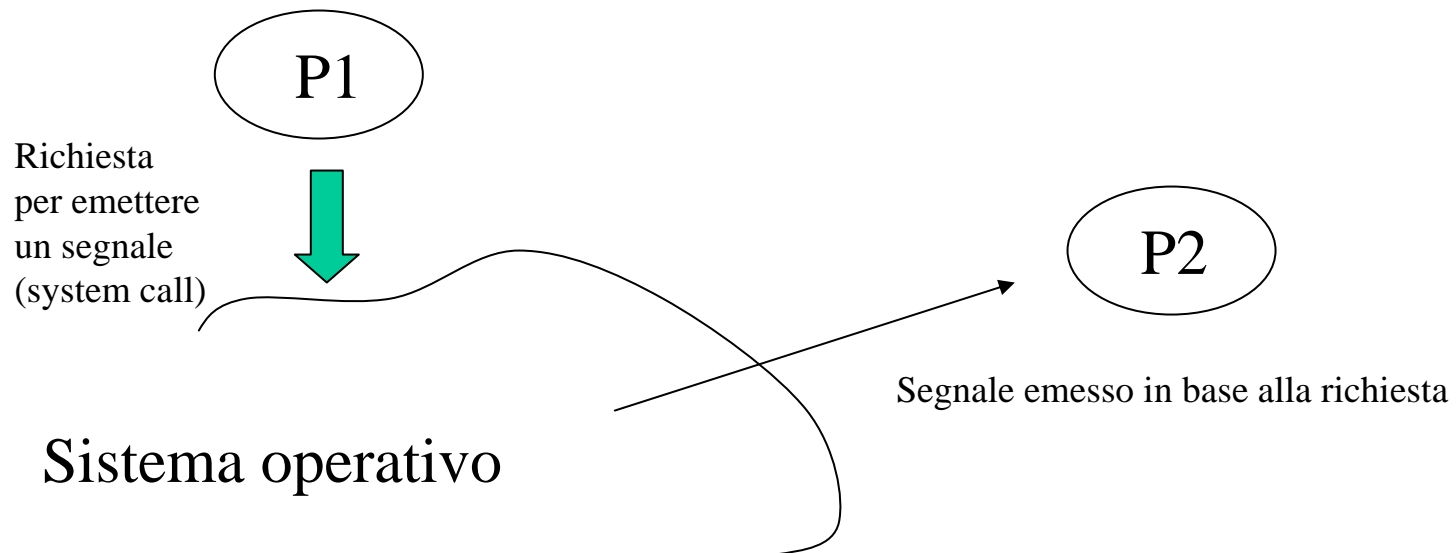


# Segnali UNIX più frequenti

- **SIGHUP (1): Hangup**. Il processo riceve questo segnale quando il terminale a cui era associato viene chiuso (ed esempio nel caso di un xterm) oppure scollegato (ad esempio nel caso di una connessione via modem o via telnet)
- **SIGINT (2): Interrupt**. Ricevuto da un processo 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 ha tentato 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 più sicuro e brutale per ``uccidere" un processo
- SIGSEGV (11): Segmentation violation. Generato quando il processo tenta di accedere ad un indirizzo di memoria al di fuori del proprio spazio
- SIGTERM (15): Termination. Inviato ad un processo come richiesta non forzata di terminazione.
- SIGALRM (14): Alarm. Inviato ad un processo allo scadere del conteggio dell'orologio di allarme.

- SIGUSR1, SIGUSR2 (10, 12): User defined. Non hanno un significato preciso, e possono essere utilizzati dai processi utente per implementare un rudimentale protocollo di comunicazione e/o sincronizzazione
- SIGCHLD (17): Child death. Inviato ad un processo quando uno dei suoi figli termina





# Richiesta di emissione di segnale per il processo corrente

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

---

**Descrizione**      richiede l'invio di un segnale al processo 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() vengono ereditate dopo una fork() da un processo figlio, tuttavia 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

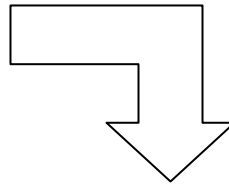
- il comportamento associato alla ricezione di un segnale viene ereditato da processi figli
- per quanto riguarda la famiglia `execX()`, solo le impostazioni di `SIG_IGN` e `SIG_DFL` vengono mantenute, mentre per ogni segnale armato con una specifica funzione di gestione viene automaticamente settato il comportamento di default
- infatti il codice della funzione di gestione potrebbe non essere più presente dopo la `exec()`
- ovviamente, il nuovo programma può definire una sua nuova gestione dei segnali

# Computazione interrotta dall'arrivo di segnale

- se la computazione interrotta era l'esecuzione di una generica istruzione in modo utente, essa riprende da dove era stata interrotta (istruzione macchina puntata dal program counter prima della gestione del segnale)
- se la computazione interrotta era una system call su cui il processo era bloccato, la system call viene abortita (fallisce) e la variabile globale **errno** (codice di errore) viene settata al valore EINTR (si veda l'header file "errno.h")
- si noti che la system call abortita non viene ripristinata automaticamente
- system call che non bloccano il processo sono eseguite in modo atomico e non vengono mai interrotte da alcun segnale

# Corretta invocazione di system call bloccanti

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



Il classico schema  
sottostante non basta

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

# Attesa di un qualsiasi segnale

```
int pause(void)
```

---

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

---

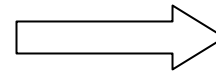
**Restituzione**      sempre -1 (poiche' e' una system call interrotta da  
segnale

- `pause()` ritorna sempre al completamento della funzione di handling del segnale
- non è possibile sapere direttamente da `pause()` quale segnale ha provocato lo sblocco; si può rimediare facendo sì che l'handler del segnale modifichi il valore di una variabile globale

# Un semplice esempio

```
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
char c;
```

```
void gestione_timeout() {
    printf("I'm alive!\n");
    signal(SIGALRM, gestione_timeout);
    alarm(5);
}
```



Ogni segnale va “riarmato”  
se vi è necessità di ricatturarlo

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

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

    while(1) read(0, &c, 1);
}
```

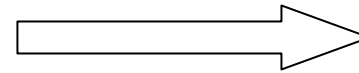
# Corse critiche

```
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
char c;
int x, y,i;
```

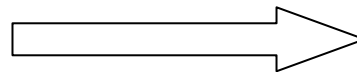
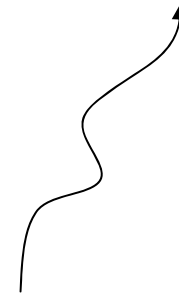
```
void gestione_timeout() {
    printf("I'm alive! x = %d, y = %d\n",x,y);
    signal(SIGALRM, gestione_timeout);
    alarm(5);
}
```

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

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



i valori di x e y  
possono essere  
diversi



l'istruzione può essere  
eseguita in modo non  
atomico

# 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 processo (ovvero per effetto della chiamata alla system call **kill**)

- 
- se il processo è in stato di blocco e il segnale richiede l'esecuzione di una funzione di gestione (sia essa di default oppure no), allora il processo 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 (inaffidabilità dei segnali)

# Un esempio

## Server

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

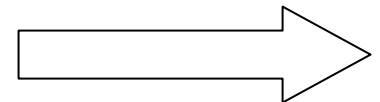
void gestione_segnoale_terminazione() {
    printf("SHUTDOWN del server in corso (eliminazione FIFO)\n");
    unlink("serv");
    printf("SHUTDOWN del server completato\n");
    exit(0);
}

typedef struct {
    long type;
    char fifo_response[20];
} request;

int main(int argc, char *argv[]) {
    char *nome, *response = "fatto";
    int pid, status, fd, fdc, i, ret, my_pid; request r;

    my_pid = getpid();
    ret = mkfifo("serv", O_CREAT|0666);
    if ( ret == -1 ) {
        printf("Errore nella chiamata mkfifo\n");
        exit(1);
    }
}
```

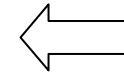
**continua**



```
fd = open("serv",O_RDWR);
```

```
signal(SIGCHLD, SIG_IGN);
```

```
signal(SIGTERM, gestione_segnalet_terminazione);
```



- ignora esplicitamente SIGCHLD
- cattura SIGTERM

```
while(1) {
```

```
    ret = read(fd, &r, sizeof(request));
```

```
    if (ret > 0) {
```

```
        pid = fork();
```

```
        if (pid == 0) {
```

```
            printf("richiesto un servizio (fifo di restituzione = %s)\n",  
                  r.fifo_response);
```

```
            sleep(10);      /* emulazione di un ritardo per il servizio */
```

```
            fdc = open(r.fifo_response,O_WRONLY);
```

```
            ret = write(fdc, response, 20);
```

```
            ret = close(fdc);
```

```
            exit(0);
```

```
        }
```

```
    }
```

```
}/* end while */
```

```
}
```

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

```
typedef struct {
    long type;
    char fifo_response[20];
} request;
```

```
request r; int fdc, ret; char response[20];
```

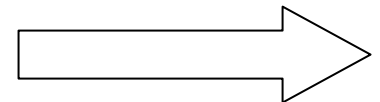
```
void gestione_timeout() {
    char c[2];

    printf("Service timeout. Attendere ancora (y/n): ");
    scanf("%s", c);

    if (c[0] == 'n') {
        unlink(r.fifo_response);
        exit(0);
    }
    else {
        alarm(5);
        signal(SIGALRM, gestione_timeout);
        ret = read(fdc, response, 20);
    }
}
```

## Client

**continua**



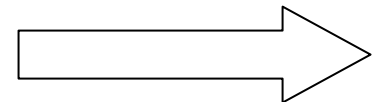
```
int main(int argc, char *argv[]) {
    int fd;

    printf("Selezionare un carattere alfabetico minuscolo: ");
    scanf("%s", r.fifo_response);
    if (r.fifo_response[0] > 'z' | r.fifo_response[0] < 'a' ) {
        printf("Selezionato non valido, ricominciare perazione\n");
        exit(1);
    }

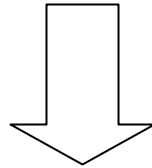
    r.fifo_response[1] = '\0';
    ret = mkfifo(r.fifo_response, O_CREAT|0666);
    if ( ret == -1 ) {
        printf("Servente sovraccarico - riprovare \n");
        exit(1);
    }

    alarm(5);
    signal(SIGALRM, gestione_timeout);
    fd = open("serv", 1);
    if ( fd == -1 ) {
        printf("Servizio non disponibile \n");
        ret = unlink(r.fifo_response);
        exit(1);
    }
}
```

**continua**



```
ret = write(fd, &r, 24);  
ret = close(fd);  
fdc = open(r.fifo_response, O_RDWR);  
ret = read(fdc, response, 20);  
printf("risposta = %s\n", response);  
ret = close(fdc);  
ret = unlink(r.fifo_response);  
}
```

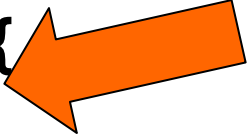


**Livelock prone**

# Problemi legati all'inaffidabilità dei segnali (1)

- terminazione dovuta al reset del gestore dopo una notifica del segnale:

```
...  
signal(SIGINT, sig_int);  
...  
void sig_int() {  
...  
signal(SIGINT, sig_int);  
...  
}
```

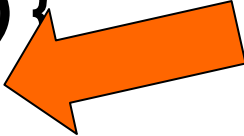


- esiste una finestra temporale all'interno della quale la notifica di un secondo segnale può far terminare il processo (prima della nuova signal() è attiva la gestione di default!)

# Problemi legati all'inaffidabilità dei segnali (2)

- Attesa indefinita di un segnale:

```
...
int segnale_arrivato=0;
...
signal(SIGINT, sig_int);
...
if (! segnale_arrivato) {
...
pause(); /*Il processo attenderà un ulteriore segnale */
}
void sig_int() {
signal(SIGINT, sig_int);
segnale_arrivato = 1;
}
```



- Esiste una finestra temporale all'interno della quale la notifica di un segnale può provocare un deadlock

# 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

# 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);
}
```

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 */
}
```

Tipico per la gestione di SIGSEGV



# 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);
```

# Implementazione dell'affidabilità in LINUX

SYNOPSIS                    **int sigreturn(unsigned long \_\_unused);**

DESCRIPTION: When the Linux kernel creates the stack frame for a signal handler, a call to sigreturn is inserted into the stack frame so that the signal handler will call sigreturn upon return. This inserted call to sigreturn cleans up the stack so that the process can restart from where it was interrupted by the signal

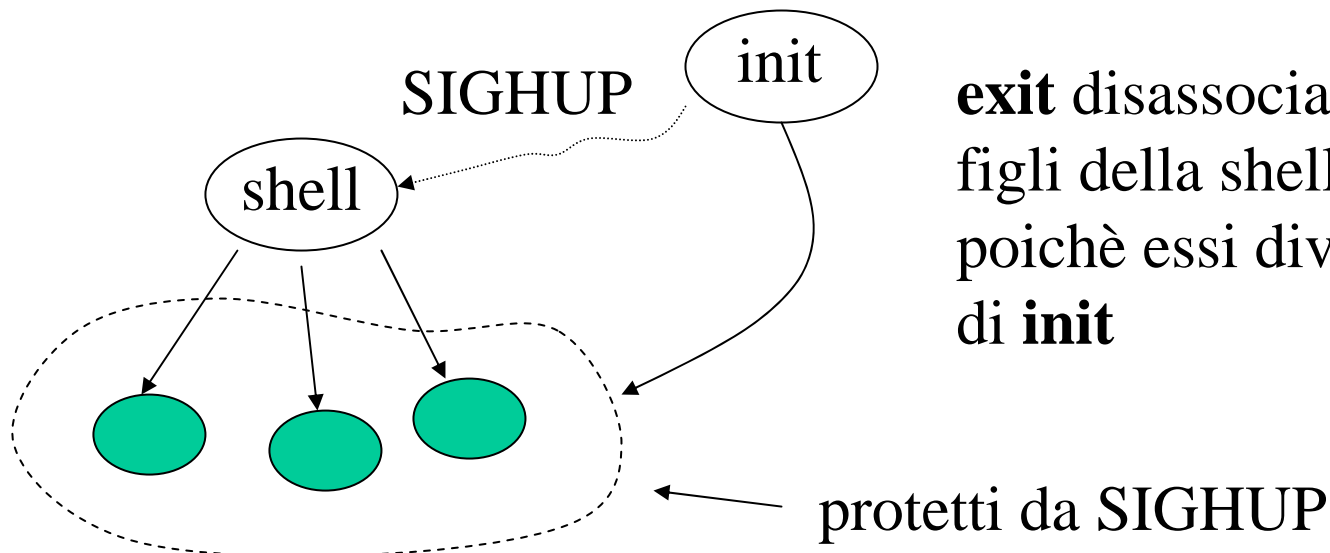
RETURN VALUE: sigreturn never returns.

WARNING: The sigreturn call is used by the kernel to implement signal handlers. It should never be called directly. Better yet, the specific use of the \_\_unused argument varies depending on the architecture.

CONFORMING TO: sigreturn is specific to Linux and should not be used in programs intended to be portable.

# 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

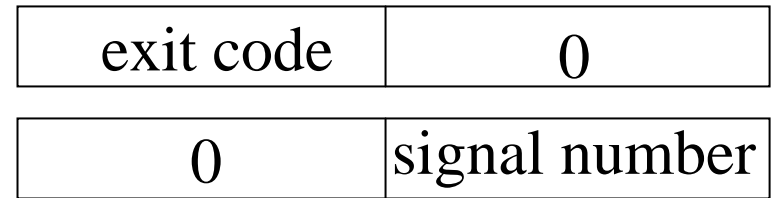


# Determinare il modo di terminazione di un processo

```
#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 parameter\n"); exit(-1); }
    if ((i=fork()) == 0) { execvp(argv[1],&argv[1]);
        printf("Can't execute file %s\n",argv[1]);
        exit(-1);
    } else if (i<0) { printf("Can't spawn process for error %d\n", errno); exit(-1); }
    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);
    }
}
```



bytes meno significativi

# Messaggi evento in Windows

- anche una applicazione Windows, in maniera simile ad una UNIX ha la possibilità di notificare eventi in modo asincrono
- questo avviene tramite la spedizione di messaggi evento
- tuttavia tali messaggi non possono interrompere un flusso di esecuzione
- pertanto il relativo handler non può essere eseguito finchè il thread non decide esplicitamente di processare la notifica
- l'approccio non utilizza quindi la nozione di interrupt, bensì quella di polling
- l'unica eccezione a questa regola è quando messaggi evento vengono inviati dallo stesso thread che li deve ricevere (in questo caso l'handler del messaggio evento viene eseguito immediatamente)
- 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 lanciato. In ambiente "Win32 console" va messo `NULL`
- `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; come default usare il valore restituito alla system call `(HBRUSH)GetStockObject(WHITEBRUSH)`
- `lpszMenuName`: stringa che specifica il nome del menu da usare. `NULL` se non ci sono menu
- `lpszClassName`: stringa indicante il nome associato a questo tipo di finestra

# 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 del C ``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 i gusti 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
- 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** 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; in contesto ``Win32 console`` usare NULL
- 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);  
}
```

- msg è una struttura di tipo MSG (gestita dal sistema) e la system call 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 thread
- wParamFilterMin: valore più basso del tipo di messaggi evento da ricevere; 0 non pone limiti inferiori
- wParamFilterMax: 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 in testa alla coda 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; 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 in fondo alla coda 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 un messaggio 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`

# Un esempio

**Processo di acquisizione di risorse**

**Gestisce un messaggio evento di terminazione**

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

UINT my_mex_type = 0;
char *termination_message="my_termination";
DWORD tid;
int shutting_down=0;

LRESULT CALLBACK WndProc (HWND hWnd, UINT message, WPARAM wParam,
                          LPARAM lParam) {
    switch(message) {
        case WM_CREATE:
            return 0;

        default:
            if (message == my_mex_type) {
                printf("Shutting down process\n");
                fflush(stdout);
                shutting_down = 1;
                PostQuitMessage(0);
                return(0);
            }
            else return(DefWindowProc(hWnd, message, wParam, lParam));
    }
}
```

```

DWORD WINAPI do_work (void * no_value) {
    // Occupa delle risorse (ES: crea dei file)
    while(!shutting_down);
    return(0);
}

void main() {
    char nome_applicazione[] = "test_eventi"; HWND hWindow;
    WNDCLASS wndclass; MSG msg; HANDLE work_thread;

    my_mex_type = RegisterWindowMessage(termination_message);
    if (!my_mex_type) {
        printf("Can't create message for error %d\n", GetLastError());
        fflush(stdout);
        ExitProcess(-1);
    }

    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = NULL;
    wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadIcon (NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = nome_applicazione;
}

```

```
if (!RegisterClass(&wndclass)) {
    printf("Can't register class");
    fflush(stdout);
    ExitProcess(-1);
}

hWindow = CreateWindow (nome_applicazione, "Test sugli eventi",
                        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, NULL, NULL, NULL, NULL);

if (hWindow == INVALID_HANDLE_VALUE) {
    printf("Can't create window for error %d\n", GetLastError());
    fflush(stdout);
    ExitProcess(-1);
}

if ((work_thread = CreateThread (NULL, 0, do_work, NULL, 0, &tid)) ==
INVALID_HANDLE_VALUE) {
    printf("Can't start working thread! Aborting....\n");
    fflush(stdout);
    ExitProcess(0);
}

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

WaitForSingleObject(work_thread, INFINITE);
}
```

## Processo di lancio del messaggio evento di terminazione

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

```
UINT my_mex_type = 0;
char *termination_message="my_termination";
```

```
void main(int argc, char ** argv) {
    char nome_applicazione[] = "test_eventi";

    my_mex_type = RegisterWindowMessage(termination_message);
    if (!my_mex_type) {
        printf("Can't create message for error %d\n",
            GetLastError());fflush(stdout);
        ExitProcess(-1);
    }
    SendMessage(HWND_BROADCAST, my_mex_type, 0, 0);
    printf("Ending NOW\n");
    fflush(stdout);
    ExitProcess(0);
}
```