SEGNALI

- Sono interruzioni software.
- Un processo all'arrivo di un segnale può decidere di:
    - ignorarlo.
    - lasciarlo gestire al kernel con l'azione di default.
    - specificare una funzione (signal handler) che viene mandata in esecuzione appena il segnale viene rilevato.
- Da chi sono inviati?
    - da un processo/thread all'altro con kill()/pthread_kill().
    - dall'utente al processo in foreground con combinazioni di tasti.
    - dall'utente con l'utility kill della shell.
    - dal SO per comunicare al processo il verificarsi di particolari eventi (SIGFPE errore float, SIGSEGV segmentation fault).

Nota:
SIGPIPE è generato dal kernel quando si scrive su una pipe o su un socket che non ha più nessun lettore perché il lettore ha chiuso la connessione. SIGCHLD inviato dal kernel al padre quando si è verificato un cambiamento in un processo figlio (in particolare quando il figlio termina). In questo caso il comportamento di default è ignorare, con gli altri segnali invece è terminare il processo che riceve quel segnale.

SD del kernel relative ai segnali
- Signal handler array: descrive cosa fare quando arriva un segnale di un certo tipo ovvero ignorare oppure trattare + puntatore al codice della funzione da eseguire (handler). Unico per tutto il processo (condiviso tra i thread).
- Pending signal bitmap: un bit per ogni tipo di segnale.
  Il bit X è a 1 se c'è un segnale pendente di tipo X. Una per thread.
- Signal mask: un bit per ogni tipo di segnale.
  Il bit X è a 1 se non voglio ricevere segnali di tipo X. Una per thread.

Cosa succede quando arriva un segnale X a un processo con un solo thread?
- I segnali vengono inseriti nella *pending signal bitmap* e controllati al ritorno da SC. Se la *signal mask* non blocca X il processo è interrotto.
- Il kernel stabilisce quale comportamento adottare controllando il contenuto del *signal handler array* (default, ignora, sig handler).
- Se deve essere eseguito un signal handler safun:
    - Si crea un frame fittizio sullo stack e si salva l'indirizzo di ritorno (quello dell'istruzione successiva a quella interrotta).
    - Si esegue safun e il processo riprende l'esecuzione dall'istruzione successiva a quella interrotta.

Cosa succede quando arriva un segnale X a un processo con più thread?

- Se il segnale è destinato a un thread particolare T nel processo:
  - Se la *signal mask* di T non blocca il segnale X il thread viene interrotto.
  - Il kernel stabilisce quale comportamento adottare controllando il contenuto del *signal handler array* (globale nel processo).
  - Se deve essere eseguito un signal handler si procede come sopra.
- Se il segnale è destinato al processo:
  - Viene scelto un thread T a caso e si procede come prima.

- Tipicamente se il segnale è dovuto ad un errore viene inviato al thread che ha fatto l'errore, tutti gli altri vanno al processo.

SC per segnali

```
int sigaction(
    int signum, [1]
    const struct sigaction *act, [2]
    struct sigaction *oldact [3]
);
```

[1] segnale.
[2] struttura che definisce il nuovo trattamento del segnale.
[3] ritorna il contenuto precedente del *signal handler array* (può servire per ristabilire il comportamento precedente).

```
struct sigaction {
    void (*sa_handler) (int); [4]
    sigset_t sa_mask; [5]
    int sa_flags;
    ...
} ;
```

[4] SIG_IGN ignora, SIG_DFL default o puntatore a funzione che prende come argomento un intero che corrisponde al segnale che si vuole gestire.
[5] segnali da mascherare durante l'esecuzione del gestore. sigaction() blocca di default solo il segnale che stiamo ricevendo, ma se uso lo stesso gestore per due o più segnali devo bloccarli tutti esplicitamente per evitare stati inconsistenti. (*)

(*) Se ho installato una funzione signal handler che gestisce SIGINT e SIGQUIT, quando mi arriva il segnale SIGINT, SIGINT sarà temporaneamente mascherato quindi un altro segnale SIGINT non lo sento (rimarrà pendente) però potrebbe arrivare un SIGQUIT e a quel punto l'handler viene eseguito di nuovo, questo potrebbe causare inconsistenze.
Si risolve passando sa_mask.

Personalizzare la gestione

- SIGKILL e SIGSTOP non possono essere gestiti (solo default).
- SIGKILL uccide tutto il processo (non solo un thread).
- SIGSTOP blocca tutto il processo (non solo un thread).
- Se il segnale è gestito, l'handler è eseguito solo da un thread.
- I segnali SIGCHLD sono gli unici ad essere accumulati (stacked), negli altri casi se arriva un segnale dello stesso tipo di uno già arrivato viene perso (al più un segnale è pendente).

Cosa mettere nell'handler

- Deve essere breve per evitare di perdere i segnali (non sono stacked).
- Solo chiamate alle funzioni contenute in 'man signal-safety'.
- L'accesso safe a variabili GLOBALI è garantito solo se definite di tipo volatile sig_atomic_t.
    - volatile: modificatore che dice al compilatore di non cachare in nessun modo la variabile, per cui gli accessi avvengono solo in memoria.
    - sig_atomic_t: è un long (lungo quanto una parola di memoria).

- Di solito gestire SIGINT, SIGTERM e simili per ripulire l'ambiente in caso si richieda la terminazione dell'applicazione.
  Ignorare SIGPIPE (e.g. in modo da non far terminare il server se un client ha riattaccato).

Cosa succede con fork, exec, pthread create

- Con la fork()
    - il figlio eredita la gestione dei segnali del padre quindi eredita il *signal handler array*.
    - il figlio eredita la *signal mask* del padre.
    - la *pending signal bitmap* messa a 0 (nessun segnale pendente).
- Dopo la exec()
    - la gestione ritorna quella di default, ma i segnali ignorati continuano ad essere ignorati.
    - la *signal mask* rimane la stessa.
    - la *pending signal bitmap* rimane la stessa del thread che fa exec.
- pthread_create()
    - siccome le funzioni di gestione interessano tutto il processo non vengono alterate.
    - la *signal mask* del nuovo thread viene ereditata dal thread che invoca pthread_create().
    - la *pending signal bitmap* messa a 0.

Mascherare

```
int pthread_sigmask(
    int how, [1]
    const sigset_t set, [2]
    sigset_t *oldset [3]
);
```

Ritorna 0 in caso di successo, codice di errore in caso di errore (NON setta errno).

[1] SIG_BLOCK: la nuova *signal mask* diventa l'or di set e della vecchia *signal mask* (i segnali in set sono aggiunti alla maschera).
SIG_SETMASK: la nuova *signal mask* diventa set.
SIG_UNBLOCK: la nuova *signal mask* rimuove i segnali presenti in set dalla vecchia *signal mask*.
[2] bitmap.
[3] se non è NULL restituisce il valore della vecchia *signal mask* prima di effettuare la modifica.

- Funzioni da passare come set:
  - `int sigemptyset(sigset_t *pset);`
    - Azzera la maschera.
  - `int sigfillset(sigset_t *pset);`
    - Mette a 1 tutte le posizioni della maschera.
  - `int sigaddset(sigset_t *pset, int signum);`
    - Mette a 1 la posizione signum.
  - `int sigdelset(sigset_t *pset, int signum);`
    - Mette a 0 la posizione signum.
  - `int sigismember(const sigset_t *pset, int signum);`
    - Ritorna 1 se signum è membro della maschera, 0 se non lo è, -1 in caso di errore.

Quando è utile mascherare segnali?
- Per indirizzare i segnali verso un thread specifico.
- Per non essere interrotti durante l'esecuzione di un handler.
- All'inizio quando non abbiamo ancora registrato tutta la gestione con sigaction().

Inviare un segnale

```
int kill(pid_t pid, int signum);
```

Genera un segnale di tipo signum e lo invia a:
- se pid > 0 al processo di pid pid.
- se pid = 0 ai processi dello stesso gruppo di chi ha invocato kill.

- se pid < 0 ai processi del gruppo -pid.
- se pid = -1 a tutti i processi per cui l'utente ha il permesso (se siamo root si fa casino).

```
int pthread_kill(pthread_t tid, int signum);          (NON setta errno)
```

Mandare segnali tra thread è pericoloso, meglio usare variabili condivise.

Attendere un segnale

```
int sigwait(                                          (NON setta errno)
    const sigset_t *set,
    int *signum
);
```

set permette di specificare i segnali da attendere con una maschera (come per la *signal mask*).
Quando ritorna, signum contiene il segnale effettivamente ricevuto.

Nota: i segnali che voglio attendere li devo avere prima MASCHERATI con pthread_sigmask() perché la sigwait() atomicamente smaschera i segnali e si mette in attesa. Fa così per evitare race condition.

La sigwait() ci permette di attendere in modo sincrono un qualunque segnale ma non siamo in un signal handler. Quindi se ad esempio progetto il mio codice in modo tale da avere un thread specifico in attesa dei segnali che voglio gestire sulla sigwait() non ho più tutti i problemi dei signal handler (è normale codice C) perché non ho usato sigaction() per installare un signal handler.

- È possibile usare sigwait() in ambiente multi-thread per gestire i segnali destinati al processo senza installare gestori.
    - usiamo un thread handler come gestore dei segnali.
    - tutti i thread mascherano tutti i segnali da gestire all'inizio.
    - handler fa continuamente sigwait() e ogni volta che ritorna gestisce il segnale corrispondente nel codice normale del thread.
    - non funziona se il segnale è destinato direttamente a un thread.