5. PRIMITIVE DI COMMUNICAZIONE FRA PROCESSI

CLASSI DI PRIMITIVE

Le varie versioni di UNIX mettono a disposizione diversi meccanismi di comunicazione fra processi:

- **pipes**
- **segnali**
- [code di messaggi]
- [semafori]
- [memoria condivisa]
- **sockets**

5.1 Pipes

PIPE

Struttura della memoria

Una pipe è un file di dimensione limitata gestito come una coda FIFO:

- un processo produttore deposita dati (e resta in attesa se la pipe è piena)
- un processo consumatore legge dati (e resta in attesa se la coda è vuota)

Posso accedere sia in lettura che in scrittura con 2 descriptor diversi.

Esistono due tipi di pipes:

- **pipe con nome**, create da `mknod` (devono essere aperte con `open`)
- **pipe senza nome**, create e aperte da `pipe` su un "pipe device" definito al momento della configurazione

Creando una pipe vengono restituiti 2 file descriptor. Uno per read, uno per write. Come li rendo disponibili a 2 processi diversi? L'amministratore della pipe, perché le info si hanno solo tra padre e figlio. (Tabella file è la stessa).

CREAZIONE DI PIPE SENZA NOME: `pipe`

```
int pipe(int fildes[2]);
```

- crea una "pipe", la apre in lettura e scrittura
- restituisce l'esito dell'operazione (`0` o `-1`)
- assegna a `fildes[0]` il file descriptor del lato aperto in lettura, e a `fildes[1]` quello del lato aperto in scrittura

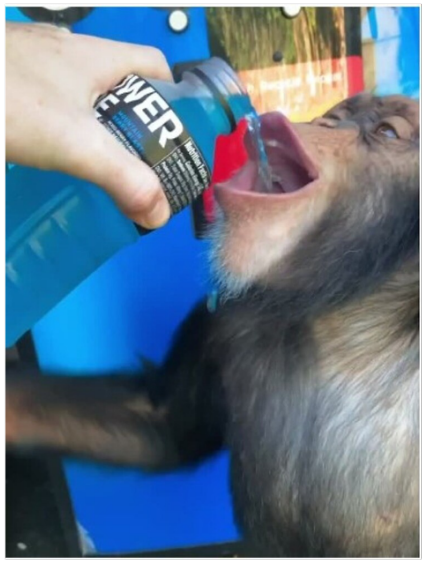
PIPE E FILE ORDINARI

Creazione e uso di un file ordinario:

```
int fd;
...
if ((fd=creat(filename,...)<0)err();
...
write(fd, ...);
...
```

Creazione e uso di una pipe (senza nome):

```
int fd[2];
...
if (pipe(fd) < 0) err();
...
write(fd[1], ...);
...
read(fd[0], ...);
...
```



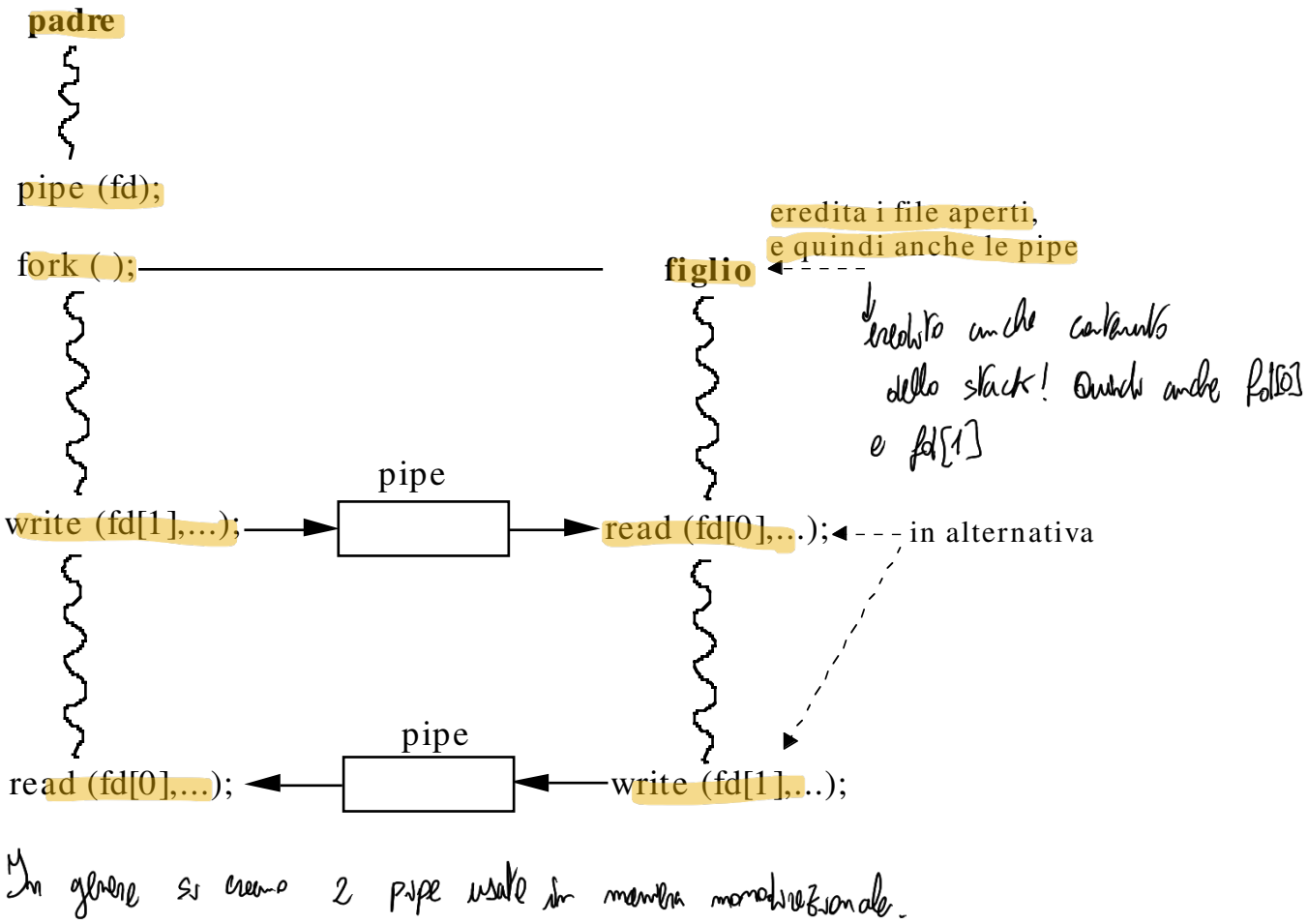
Note:

- la `pipe` è analoga a `creat`, ma non specifica il nome e restituisce due file descriptor
- `read` e `write` sono le stesse nei due casi

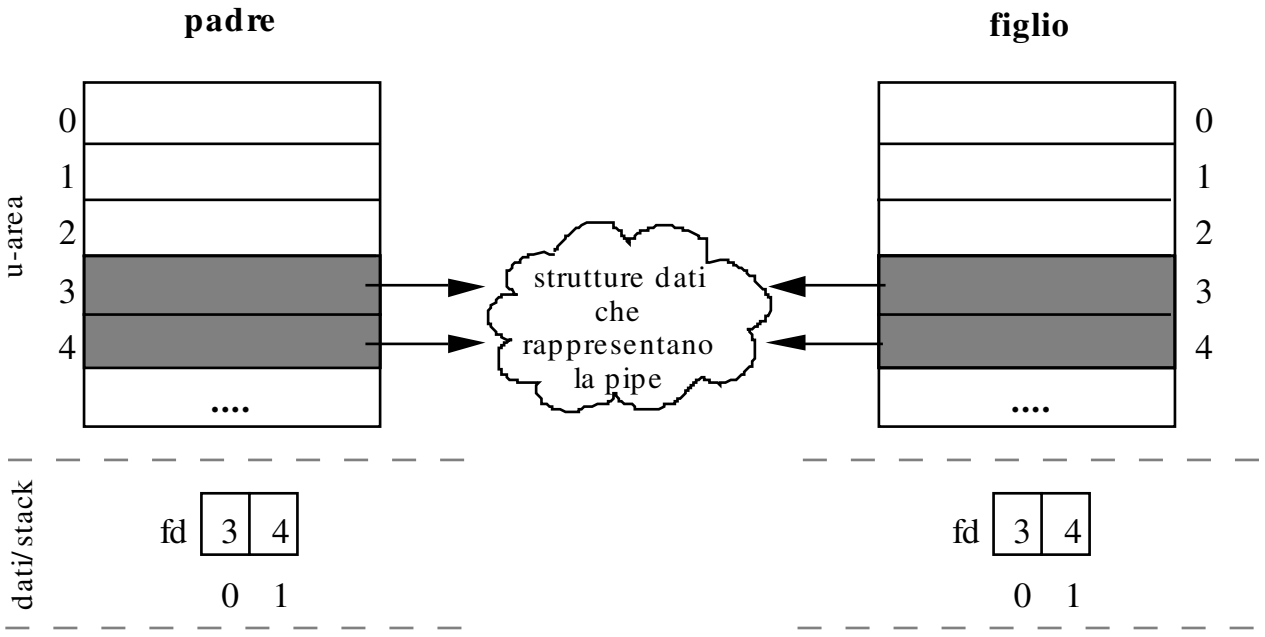
PIPE: USO TIPICO

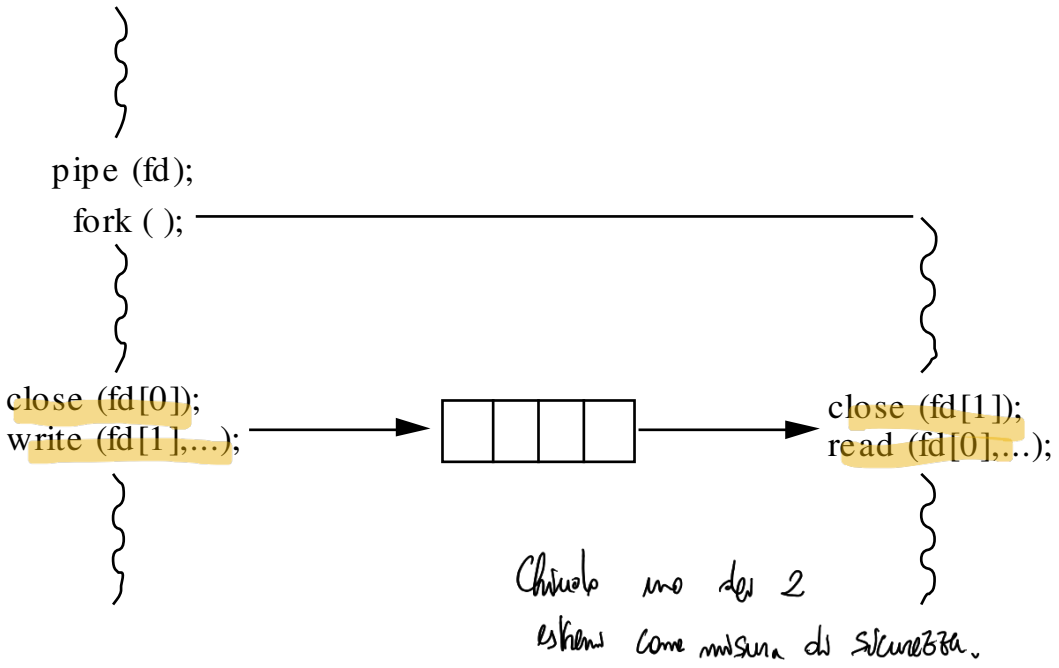
Generalmente una pipe viene usata per far comunicare un processo padre con un suo figlio

Lo schema tipico:



PIPE: USO TIPICO (SEGUE)





Comunicazione padre-figlio con pipe

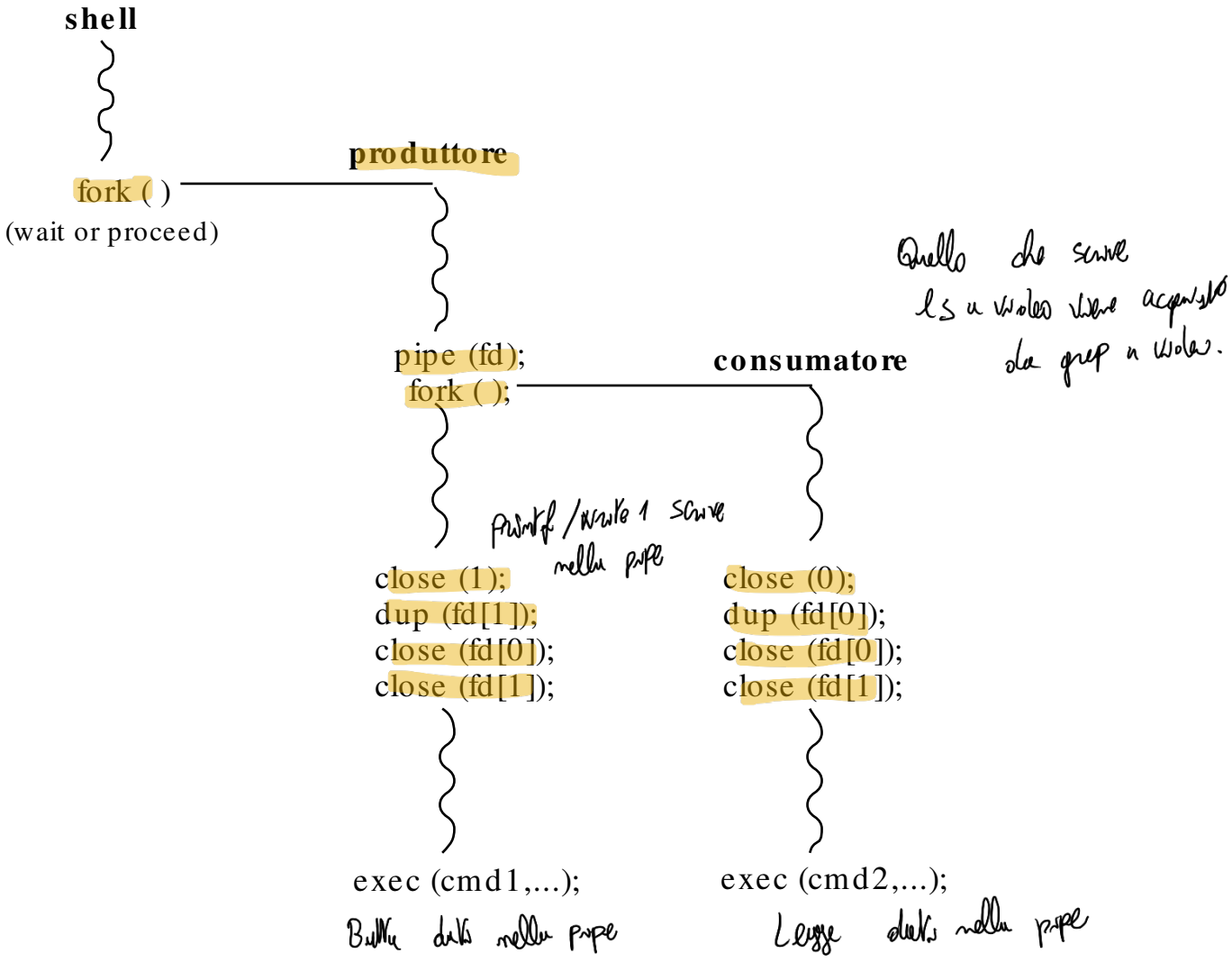
```
#include <stdio.h>
#include <stdlib.h>
int main() {
int pid, j, c;
int piped[2];

/*Apre la pipe creando due file descriptor,
uno per la lettura e l'altro per la scrittura
(vengono memorizzati nell'array piped[])*/
if (pipe(piped) < 0)
   exit(1);

if ((pid = fork()) < 0)
    exit(2);
else if (pid == 0) /* figlio, che ha una copia di piped[] */
{
close(piped[1]);
for (j = 1; j <= 10; j++)
   {
read(piped[0], &c, sizeof (int));
printf("Figlio: ho letto dalla pipe il numero %d\n", c);
    }
exit(0);
}
else /* padre */
{
   close(piped[0]);
   for (j = 1; j <= 10; j++)
     {
      write(piped[1], &j, sizeof (int));
      printf("Padre: ho scritto nella pipe il numero %d\n",j);
      sleep(1);
      }
   exit(0);
}
}
```

REALIZZAZIONE DI UNA PIPELINE

$cmd1 | cmd2



REALIZZAZIONE DI UNA PIPELINE (SEGUE)

$cmd1 | cmd2

```
if ((pid=fork()) == 0)  {
    /* processo figlio */
    if (pipeline)  {
        pipe (fd_pipe);
        if (fork()==0) { /* produttore */
            close(1);
            dup(fd_pipe[1]);
            close(fd_pipe[0]);
            close(fd_pipe[1]);
            exec (cmd1, ...);
        }                    /* consumatore */
        close (0);
        dup (fd_pipe[0]);
        close (fd_pipe[0]);
        close(fd_pipe[1]);
        exec (cmd2, ...);
    }
...
}
```

Facile comunicare 2 processi che non si conoscono proprio

5.2 Segnali

SEGNALI

Un **segnale** è una notifica a un processo che è occorso un particolare "evento":

- un errore di floating point
- la notifica della morte di un figlio
- una richiesta di terminazione
- ...

I segnali possono essere pensati come degli "interrupts software"

I segnali possono essere inviati:

- da un processo a un altro processo
- da un processo a se stesso
- dal kernel a un processo

Ogni segnale é identificato da un numero intero (associato a un nome simbolico in `<signal.h>`)

Segnali standard

Molti numeri di segnale dipendono dall'architettura, come indicato nella colonna "Valore" (dove sono indicati tre valori, il primo è normalmente valido per alpha e sparc, quello in mezzo per i386, ppc e sh, e l'ultimo per mips. A - denota che un segnale è assente sulla corrispondente architettura).

Prima i segnali descritti nello standard POSIX.1-1990 originale.

Segnale	Valore	Azione	Commento
			agganciata o il processo che ha il controllo è morto
SIGINT	2	Term	Interrupt da tastiera
SIGQUIT	3	Core	Segnale d'uscita della tastiera
SIGILL	4	Core	Istruzione illegale
SIGABRT	6	Core	Segnale d'abbandono di *abort*(3)
SIGFPE	8	Core	Eccezione di virgola mobile
SIGKILL	9	Term	Termina il processo
SIGSEGV	11	Core	Riferimento di memoria non valido
SIGPIPE	13	Term	Pipe rotta: scrittura su una pipe priva di lettori
SIGALRM	14	Term	Segnale del timer da **alarm**(2)
SIGTERM	15	Term	Segnale di termine
SIGUSR1	30,10,16	Term	Segnale 1 definito dall'utente
SIGUSR2	31,12,17	Term	Segnale 2 definito dall'utente
SIGCHLD	20,17,18	Ign	Figlio fermato o terminato
SIGCONT	19,18,25	Cont	Continua se fermato
SIGSTOP	17,19,23	Stop	Ferma il processo
SIGTSTP	18,20,24	Stop	Stop digitato da tty
SIGTTIN	21,21,26	Stop	Input da tty per un processo in background
SIGTTOU	22,22,27	Stop	Output da tty per un processo in background

I segnali **SIGKILL** e **SIGSTOP** non possono essere intercettati, bloccati o ignorati.

ESEMPIO

Per terminare o sospendere un processo in foreground, l'utente può premere i tasti `CTRL-C` o `CTRL-Z` (rispettivamente)

Tale carattere viene acquisito dal driver del terminale, che notifica al processo il segnale `SIGINT` o `SIGTSTP` (rispettivamente)

Per default, `SIGINT` termina il processo e `SIGTSTP` lo sospende

Nota:

In realtà, tali segnali vengono inviati a tutto il gruppo di processi

GESTIONE DEI SEGNALI

- per inviare un segnale:

`kill`

- per specificare come trattare un segnale:

`signal`

Nota:

La primitiva `signal` non é POSIX: la corrispondente primitiva POSIX si chiama `sigaction`

manda segnale

LA PRIMITIVA `kill`

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

- notifica il segnale `sig` al processo/gruppo di processi specificato con `pid` :

  `pid>0`: `pid` indica il process-id

  `pid<0`: |`pid`| indica il process-group-id

- restituisce il risultato dell'operazione (0 se è stato inviato almeno un segnale, −1 se errore)

Che succede: il processo che fa la kill è il running. Il SO segna da qualche parte che ho mandato segnale. Lo fa con maschera dei segnali, 1 bit x segnale. Quindi se mando segnale a altro processo mette 1 per la maschera dei segnali. Quando il destinatario si sveglia consulta la mask (no SO) e se c'è evento attivo funzione associata a quell'evento. Controllo e attivazione funzione fa l'OS.

PROTEZIONE

Per motivi di protezione, deve valere almeno una delle seguenti condizioni:

- Il processo che riceve e il processo che invia il segnale devono avere lo stesso owner[2]
- L'owner[3] del processo che invia il segnale éil superuser

[2] Ricevente e inviante devono avere gli stessi Effective-user-id e real-user-id

[3] Effective-user-id

LA PRIMITIVA `signal`

Permette di specificare come dovrà essere trattata, dal processo ricevente, la "prossima" occorrenza di uno specificato segnale

Tre possibilità:

1. il segnale dovrà essere **trattato** da uno specificata funzione ("handler")
2. il segnale dovrà essere **ignorato**
3. il segnale dovrà innescare l'azione di **default** associata al segnale stesso.

Note:

- Le possibili azioni di default sono fisse per ogni segnale e possono essere: terminare il processo (con o senza core dump); ignorare il segnale; sospendere il processo; riattivare il processo
- `SIGKILL` e `SIGSTOP` non possono essere riprogrammati: si attiva sempre l'azione di default

LA PRIMITIVA `signal`: ESEMPIO

Indica come vuole gestita, ma non sovrascrivere
NOTA: signal non è altresì di segnale. Dice: nel caso riceva segnale usa questa funzione come ricezione.

```
...
int sig;

void func(int signo)

{ /* corpo dell'handler del segnale.*/ };

signal(sig, func);      /* al ricevimento del
                           segnale sig!=SIGKILL, viene
                           chiamata func, passandole sig come
                           argomento. Al termine di func, il
                           controllo ritorna al punto di
                           interruzione */

...

signal(sig, SIG_IGN); /* sig dovrà essere
                         ignorato */

...

signal(sig, SIG_DFL); /* all'occorrenza di
                         sig, dovrà essere
                         attivata l'azione di
                         default */
```

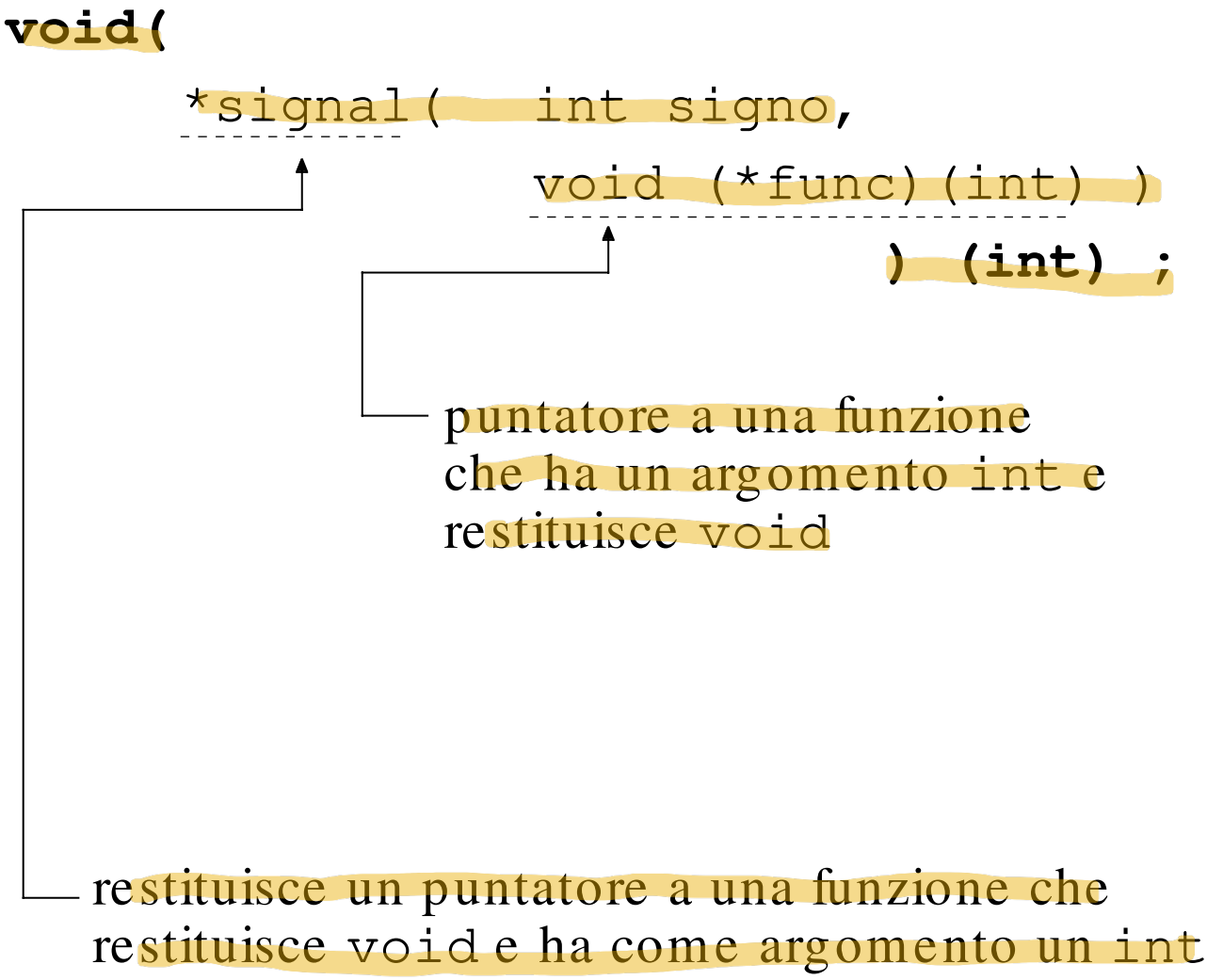
→ Un puntatore a una funzione.

N.B. Se ok, `signal` restituisce il puntatore alla funzione precedentemente associata a `sig`

Un solo handler che gestisce più segnali.

NOTA: Mask segnali in memoria residente.

LA PRIMITIVA `signal`: PROTOTIPO



Esempio sospensione di un segnale

- **Dopo aver trattato un segnale, l'azione associata viene "dimenticata".**

- **Al successivo ricevimento dello stesso segnale verrà eseguita l'azione di default**

Esempio:

```
void (*oldHandler) (int);
...
oldHandler=signal(SIGINT, SIG_IGN);
...
/* qui CTRL-C viene ignorato */
...

signal(SIGINT, oldHandler);
...
/* qui CTRL-C viene gestito da
oldHandler, ma solo la prima volta;
poi causa terminazione (default) */
```

LA PRIMITIVA `alarm`

```
unsigned int alarm (unsigned int
                    seconds);
```

- ritorna al chiamante, e dopo `seconds` secondi gli invia il segnale `SIGALRM`

- ci può essere una sola richiesta pendente: la funzione ritorna i secondi mancanti alla terminazione di eventuali `alarm` precedenti

LA PRIMITIVA `sleep`

```
unsigned int sleep (unsigned int
                    seconds);
```

- sospende il chiamante per `seconds` secondi
- restituisce il numero di secondi mancanti a soddisfare la richiesta:

  la sospensione può essere infatti più breve di quanto richiesto, per vari motivi (ad es., un segnale che deve essere trattato)

LA PRIMITIVA `pause`

Chiamo prima la signal e poi pause. Sospendo fino a quando il processo ha ricevuto segnale.

```
int pause (void);
```

- sospende il chiamante fino a quando esso riceve un segnale (che non deve essere ignorato)
- restituisce −1 in caso di errore