Ivan De Marino
matr. *50/529*

prof. *Verzino*

Progetto Laboratorio di Sistemi Operativi
a.a. 2002/2003

Patty Shell
PSH

A mia madre.
Per aver mostrato Forza e Coraggio quando la tempesta imperversava...
Per aver combattutto quando i fragili petali dei tuoi fiori si erano sgualciti,
e la tua bellezza si dissipava...
Per aver deciso di Rifiorire ancor più bella nonostante tutto.

OBIETTIVI E CONTENUTI

Ciò che è richiesto

Scopo del progetto è realizzare in ambiente Unix/Linux una shell.
La shell deve essere realizzata utilizzando il linguaggio C e le primitive POSIX.

La shell deve disporre delle seguenti funzionalità:

1. Esecuzione di tutti i programmi il cui eseguibile si trova in una delle directory listate nella variabile di ambiente PATH.
2. Esecuzione di processi sia in Foreground, sia in Background.
3. Redirezione dei segnali da tastiera dalla shell al processo in Foreground (se presente) o ignorati.
4. Nel caso particolare dei processi da eseguire in Backgound, implementazione di un comando "built-in[1]": `BACK <eseguibile> <arg1> <arg2> ... <argn>`
5. Nel caso dell'intercettazione del segnale SIGINT e della mancanza di un processo in FG (Foreground), la shell chiederà l'identificativo del processo a cui inviare il segnale.
6. Esecuzione di programmi "in pipeline[2]".
7. Esecuzione di un programma ricorsivamente attraversando il sottoalbero radicato nella directory corrente: `RECO <eseguibile> <arg1> <arg2> ... <argn>`

Ciò che è stato realizzato

Il nome della shell è "Patty Shell" (anche detta PSH).
Quello che si è cercato di realizzare è una shell quanto il più possibile (nei limiti del tempo e delle conoscenze) conforme a quelle che sono le richieste, tenendo sempre d'occhio la possibilità di una possibile espansione.
Le strutture dati realizzate e la forte modularità che si è cercata nella progettazione, dovrebbero (almeno nelle intenzioni del realizzatore) consentire una facile modifica delle singole componenti della PSH, con la possibilità di espansioni delle funzionalità e miglioramenti che non richiedano una completa ristesura del codice.
Il tutto è stato programmato (come richiesto) utilizzando il linguaggio C e (nella quasi totalità dei casi) System Call[3] conformi allo standard POSIX. In alcuni casi si è anche ricorso alle moderne e sicure estensioni GNU_SOURCE.
Oltre ai comandi "built-in" richiesti nella traccia, sono stati implementati pochi altri comandi "standard" di qualsiasi altra shell. Ad esempio "`CD`".

In più è mia intenzione rilasciare il codice sotto licenza GPL così da essere, se non un grande applicativo, almeno una fonte di studio e comprensione di alcuni di quelli che sono i meccanismi basilari di funzionamento di un sistema Unix.

---

[1] Si intendono per comandi "built-in" quei comandi che non sono eseguibili, ma stringhe che la shell interpreta.

[2] Cioé lo STANDARD OUTPUT di un processo, diventa STANDARD INPUT del processo successivo e così via.

[3] Una System Call è una "chiamata" alle funzionalità del sistema: il kernel si incarica di eseguire le operazioni per cui solo lui ha i privilegi

MANUALE

Estrazione tgz, compilazione, installazione

Per estrarre i sorgenti di PSH dal file "`psh_sources.tar.gz`" digitare.
`# tar -xzf psh_sources.tar.gz`

Dopo, entrare nella directory dei sorgenti ed eseguire lo script "`./compile.sh`"

Lo script creerà il file eseguibile "pattysh". Copiare il file in una delle cartelle elencate da PATH o, meglio, nella cartella "`/bin/`".

Cosa può fare

PSH può eseguire qualunque programma si trovi nella variabile d'ambiente PATH[4]. Baterà digitare il *pathname* dell'eseguibile per farlo partire.

PSH redirige i segnali da tastiera[5]:
- CTRL+C corrisponde a SIGINT. Se un processo è in esecuzione in FG gli viene rimbalzato il segnale da PSH. Se non ci sono processi in FG, PSH stampa una lista dei processi e chiede di digitare il PID del processo a cui inviare il segnale.
- CTRL+Z corrispnde a SIGTSTP. Se un processo è in esecuzione in FG riceve il segnale e viene stoppato. Altrimenti PSH lo ignora.
- CTRL+\ corrisponde a SIGQUIT. Se un processo è in esecuzione in FG riceve il segnale, viene terminato ed in più viene eseguito il DUMP della sua immagine in memoria. Altrimenti PSH lo ignora.

PSH monitora tutti i processi da essa eseguiti[6]. Ne controlla lo stato di esecuzione e verifica se questo è attivo o stoppato. Data la natura asincrona dei processi eseguiti in BG, controlla se ci sono processi figli terminati[7].

PSH implementa un semplice PARSER della riga di comando[8]: il suo scopo è quello di evidenziare la presenza di eventuali simboli "`|`" (pipe) tra un eseguibile e l'altro. Così sarà poi possibile collegarne sequenzialmente STANDARD INPUT e STANDARD OUTPUT.

PSH disponde di alcuni semplici comandi "built-in" per eseguire le operazioni più semplici[9]:

- *`exit`* : permette di terminare PSH
- *`cd`* : permette di cambiare directory corrente
- *`env`* : lista tutte le variabili d'ambiente
- *`cwd`* : visualizza il percorso completo della directory corrente
- *`PS`* : visualizza tutti i processi in esecuzione
- *`back`* : esegue in background un programma
- *`reco`* : esegue ricorsivamente sul sottoalbero della directory corrente un programma

---

4 Fare riferimento al codice del file "`signal_and_exec.c`"
5 Fare riferimento al codice del file "`signal_and_exec.c`"
6 Fare riferimento al codice dei file "`pprocs_struct.c`" e "`pprocs_struct_function.c`"
7 Tratteremo, successivamente, tutto in maniera più dettagliata
8 Fare riferimento al codice dei file "`cmd_line_parser.c`"
9 Fare riferimento al codice del file "`built_in.c`"

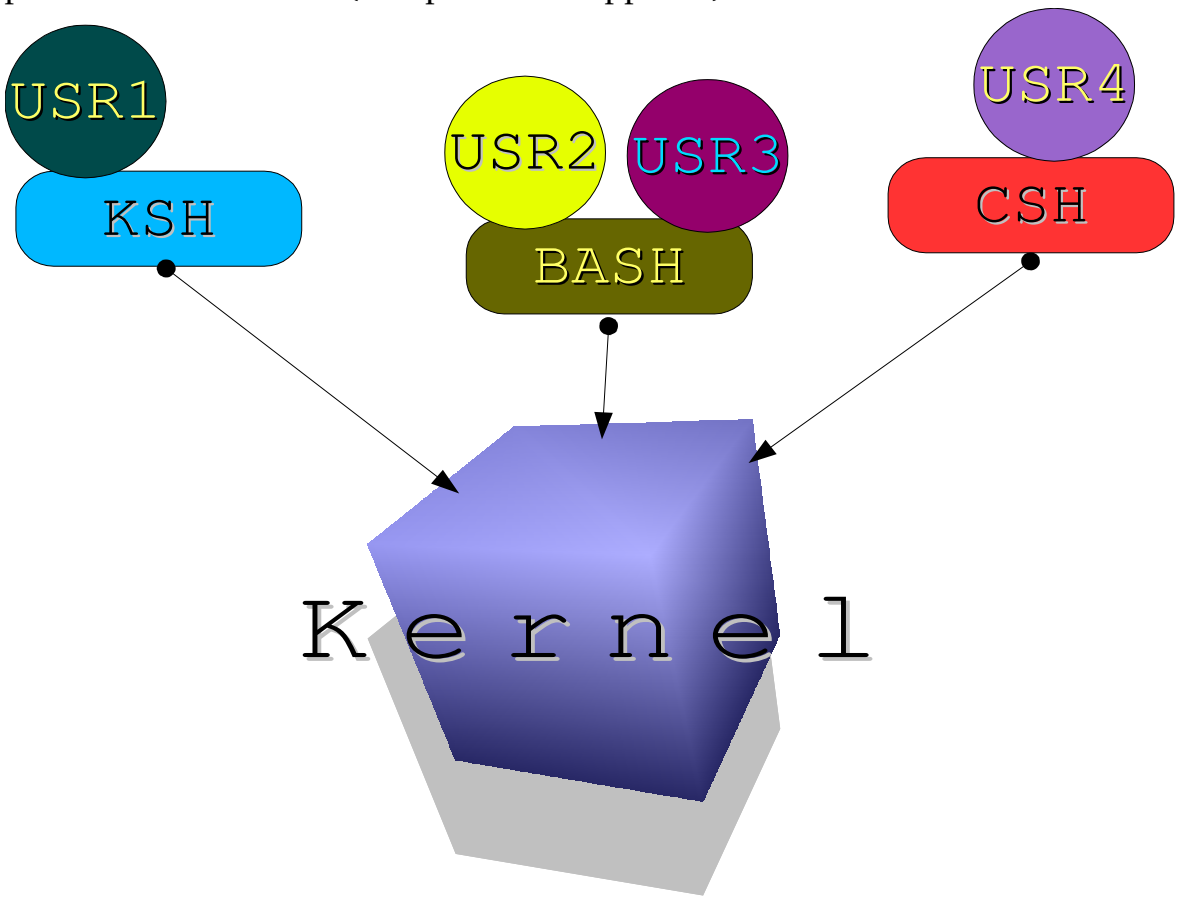
COS'È UNA SHELL E A COSA SERVE...

La SHELL è un interprete della linea di comando.
Ogni utente ha una sua shell di default. Famose sono "SH[10]", "KSH[11]", "CSH[12]" e, la più diffusa, "BASH[13]".

La shell è l'interfaccia tra l'utente e il kernel vero e proprio. Permette quindi una "comunicazione semplificata" con il sistema operativo stesso, mettendo a disposizione numerosi comandi "nativi" (built-in) e funzionalità spesso molto complesse. Vedasi ad esempio la grammatica dei linguaggi di scripting come quello implementato in BASH, o funzionalità come il completamento automatico dei comandi, o l'avanzato "Job-Controlling" e non solo.

La shell viene eseguita, in genere, in modalità interattiva al momento del login, e resta attiva per l'intera sessione di lavoro.

Le shell standard eseguono, oltre che i comandi built-in e gli eseguibili dei programmi (ovvero il programma in formato oggetto compilato), anche file di testo che hanno attivi i permessi di esecuzione ("script di shell" appunto).



[10] Bourne Shell
[11] Korne Shell
[12] C Shell. Ha una "grammatica" simile al linguaggio C
[13] "Bourne Again" Shell. E' la versione "riscritta" di SH, e la sua prima versione è opera di Richard Stallman

I Processi: Creazione e Terminazione

Il Processo è "l'unità di lavoro atomica" di un sistema operativo multi-task[14].
E' costituito da diverse parti residenti in memoria:

- Environment (ambiente del processo)
- User Stack (Stack delle funzioni e variabili automatiche)
- User Heap (Memoria allocata dinamicamente)
- Dati inizializzati
- Dati non inizializzati (BSS – Block Started by Symbol)
- User Text (Codice macchina del programma)
- PCB
- Kernel Stack
- Shared Memory(Memoria condivisa tra processi allocata dal kernel)

Ogni processo ha un padre (parent) che lo genera. Ogni programma che eseguiamo dalla shell ha come padre, appunto, la shell. Esiste ovviamente un processo che non ha padre e che è padre di tutti i processi. Prende il nome di INIT.

Ogni processo ha associato un identificativo (PID=Process Identifier) univoco. Sono generati sequenzialmente e associati ad ogni processo dal Kernel. INIT ha PID=1. Ogni processo ha anche un PPID, ovvero l'identificativo del processo padre che l'ha generato.

Inoltre ci sono tutta una serie di identificativi e permessi legati all'utente che ha generato il processo:
- Real User ID (RUID)
- Real Group ID (RGID)
- Effective User ID (EUID)
- Effective Group ID (EGID)

RUID e RGID sono permessi relativi all'utente che ha "generato" il processo.
EUID e EGID sono, di norma, uguali ai precedenti e specificano i permessi "effettivi" di quel processo. Possono cambiare nel caso sia attivo sull'eseguibile del processo il Set-UID bit[15].

CREAZIONE PROCESSI: FORK ed EXEC

Per creare un nuovo processo esistono apposite System Call (**`unistd.h`**):
- `fork`
- `vfork` (ormai praticamente inutile[16])

Con queste System Call si fa in modo che il kernel crei **un processo del tutto identico** al processo chiamante, tranne che per il PID, il PPID e qualche altra informazione.
Per far ciò esegue le seguenti operazioni:
- Allocare spazio nella tabella dei processi per il nuovo processo
- Assegnare un nuovo PID al processo creato
- Esegue una copia dell'immagine in memoria del genitore tranne che per i segmenti di

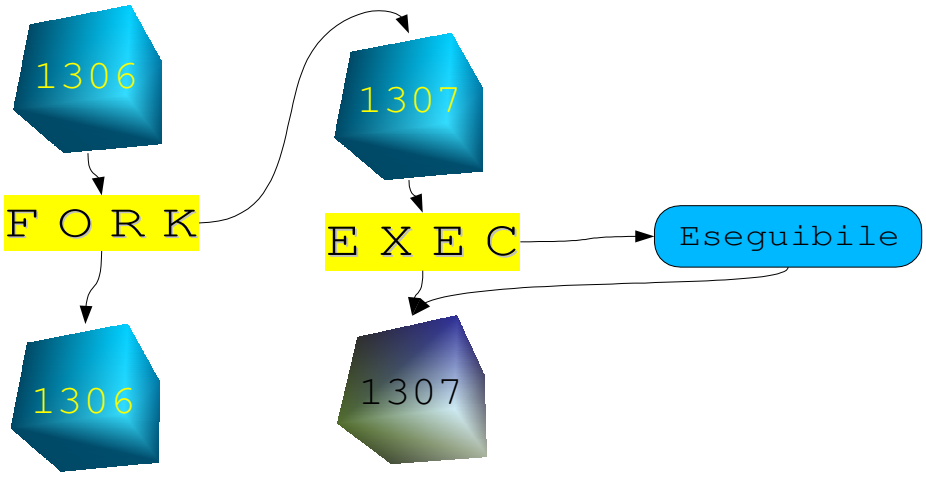
[14] Per differenziare dai sistemi operativi che gestiscono anche i kernel thread
[15] In sostanza, il processo viene eseguito con i permessi del proprietario
[16] Spiegheremo tra un po' il perché

memoria condivisi
- Duplica i descrittori dei file aperti dal padre al figlio
- Assegna lo stato di READY al processo figlio
- Ritorna al padre il PID del figlio, mentre al figlio 0[17]

Si è accennato all'ormai inutilità di `vfork`: questa funzione, a differenza di `fork`, esegue una copia "minimale" del processo padre, in prospettiva di una "sostituzione" dell'immagine del processo con una nuova esecuzione[18]. Le moderne implementazioni di `fork` e i sistemi moderni adottano la cosiddetta tecnica di "copy-on-write": in sostanza l'immagine di memoria del processo figlio viene effettivamente copiata dal padre, solo nel momento in cui dovesse essere eseguita su di essa una modifica. Quindi l'uso di `vfork` per minimizzare il carico di lavoro è ormai inutile e deprecato. Resta la system call solo per retrocompatibilità.

Accanto alla `fork` troviamo una famiglia di System Call utili all'esecuzione di un nuovo processo: EXEC.
Queste System Call servono a "sostituire" l'immagine in memoria di un processo con quella di un nuovo processo (in sostanza, ad eseguirlo).
Le System Call sono (**`unistd.h`**):
- `execl`
- `execlp`
- `execle`
- `execv`
- `execvp`
- `execve`

I suffissi "l", "v", "p" ed "e" determinano particolari comportamenti di exec:
- "l" indica che accetta in input una lista di parametri
- "v" indica che accetta in input un array di stringhe
- "p" indica che cercherà l'eseguibile all'interno delle directory listate dalla variabile PATH
- "e" indica che accetta in ingresso un array di stringhe che contiene una nuova definizione di variabili d'ambiente



Al processo che esegue un EXEC vengono sostituiti:
- Text Segment (codice del programma)
- Data Segment (dati)
- Heap
- Stack

Inoltre subiscono delle modifiche anche il PCB, se viene eseguito un programma con Set-UID attivo, e Environment, se si eseguono `execve` o `execle`.

Restano invece invariati:

---

[17] Così da renderci conto se è in esecuzione il padre o il figlio
[18] Tutto sarà più chiaro quando tratteremo la famiglia delle System Call EXEC

- Terminale di controllo
- Current Work Directory
- Root Directory
- Maschera di creazione file (umask)
- Eventuali lock sui file
- Coda dei segnali non gestiti
- Segnali mascherati
- Descrittori dei file aperti (a meno che non si setti diversamente un opportuno FLAG)

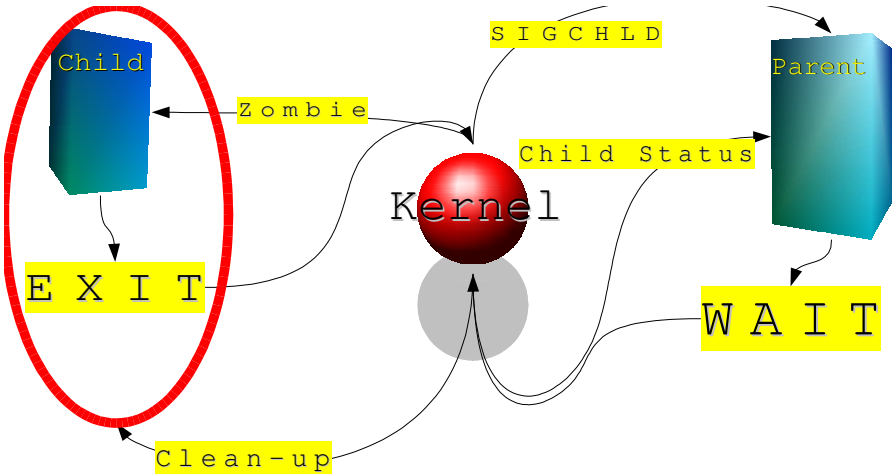
TERMINAZIONE: EXIT e WAIT

Si potrebbe definire "processo terminato" un processo che ha concluso il suo flusso di istruzioni. Forse però sarebbe più corretto definire un processo "terminato", solo dopo che il kernel ha liberato la memoria dalla sua immagine. Questo serve a mettere in evidenza il fatto che il compito di terminare realmente un processo lo ha il kernel. Questi infatti attende o che il processo faccia "richiesta di terminare" (attraverso apposite System Call), o che questo venga terminato da un qualche segnale (generato sempre dal Kernel) per motivi che possono essere la pressione di una sequenza di tasti, l'esecuzione di una istruzione illegale, l'accesso ad un area di memoria non consentita...

Le System Call adibite alla "terminazione" di un processo sono (**`stdlib.h`**):
- `exit` (ANSI C: permette la chiamata a funzioni opzionali, provvede allo svuotamento dei buffer, chiude gli stream)
- `_exit` (POSIX: clean-up della memoria, chiusura effettiva, immediato ritorno del controllo al kernel)
- `return` (richiamato dal main)

La S.C.[19] "`exit`" permette di associare alla chiusura l'esecuzione di funzioni. Per farlo si usa "`_at_exit`". Questo, ad esempio, potrebbe permettere di implementare eventuali funzioni di "pulizia finale" della memoria.

Un processo invece non termina normalmente se riceve determinati segnali:
- `abort`, `raise`... (richiamati dal processo stesso)
- `kill` (richiamata da un altro processo che ne ha la facoltà)
- segnali inviati dal kernel (istruzioni illegali, violazioni di memoria, divisioni per zero...)



Inoltre, quando un processo termina, il kernel ne avverte il processo padre inviandogli il segnale SIGCHLD. Sarà poi compito del padre "raccogliere" lo stato di terminazione del figlio con apposite S.C. (**`sys/wait.h`**):

[19] D'ora in avanti sarà utilizzato scambievolmente con System Call

- `wait`
- `waitpid`
- `waitpid3`
- `waitpid4`

Queste permettono di attendere che un processo termini e di analizzarne lo stato di terminazione. Si differenziano solo per alcune "feature"[20].
Inoltre esistono delle MACRO[21] utili per la valutazione dello stato di uscita di un processo figlio: servono a determinare se è terminato normalmente, se per via di un segnale, se è stoppato e via dicendo. Si può addirittura ricavare quale segnale ha terminato il processo.

Con questi strumenti un processo padre può quindi tenere sotto controllo l'esecuzione dei suoi processi figli.

PSH: GESTIONE DEI PROCESSI

Il primo aspetto affrontato nella realizzazione di PSH è stata la creazione di una struttura dati che permettesse di conservare almeno le informazioni essenziali sui processi eseguiti. Si è quindi realizzata una struttura che prende il nome di PPROCS.
In sostanza è una lista concatenata di cui ogni elemento è un l'insieme di informazioni che identificano un processo:

filename: `<path_to_psh_sources>/include/pprocs_struct.c`

```
struct pproc_str{
     char proc_name[MAX_PROG_NAME_LENGTH];  /*Nome processo*/
     pid_t pid;                             /*PID processo*/
     pid_t ppid;                            /*PPID processo*/
     struct pproc_str *prec_pproc;          /*Processo precedente nella
struttura PPROCS*/
     struct pproc_str *next_pproc;          /*Processo successivo...*/
     int status;                            /*Stato del processo: Attivo o
Bloccato*/
};

struct PPROCS{
     int pprocs_num;                        /*Numero di processi presenti*/
     struct pproc_str *first_pproc;         /*Primo processo*/
     struct pproc_str *last_pproc;          /*Ultimo processo*/
};
```

In più esistono apposite funzioni di gestione:

filename: `<path_to_psh_sources>/include/pprocs_struct_function.c`

Crea una struttura PPROCS:

```
PPROCS *create_PPROCS(void);
```

Aggiunge le informazioni di un processo alla struttura :

```
void add_pprocs(char *nname, pid_t npid, pid_t nppid, int stat, PPROCS *HEAD);
```

Mostra i processi presenti nella struttura:

```
int show_pprocs(PPROCS *HEAD, const int to_vis);
```

Cerca un processo nella struttura:

```
pproc_str* find_pprocs(pid_t pid, PPROCS *HEAD);
```

Rimuove un processo dalla struttura:

```
int remove_pprocs(pid_t rpid, PPROCS *HEAD);
```

[20] `wait` attende la terminazione di qualunque processo figlio, mentre `waitpid` permette di specificare un PID preciso. Inoltre `waitpid3` e `waitpid4` permettono di recuperare le risorse rilasciate dal processo figlio terminato per analizzarle.

[21] `WIFEXITED(status)`, `WIFSIGNALED(STATUS)`, `WIFSTOPPED(STATUS)`. Ce ne sono altre ancora che permettono di risalire anche ai segnali.

Cambia il flag di stato di un processo presente:

```
int ch_status_pprocs(pid_t pid, PPROCS* HEAD);
```

Questi strumenti permettono (almeno nelle intenzioni) una buona riutilizzabilità e la possibilità di migliorare ancora il progetto, espandendolo e rendendolo più complesso.

Dopo aver quindi preparato una la struttura PPROCS, si è passati alla realizzazione delle funzioni che permettessero di gestire l'esecuzione di nuovi processi e il monitoraggio del loro stato:

Filename: `<path_to_psh_sources>/include/signal_and_execs.c`

Aggiunge un processo alla struttura PPROCS, si mette in attesa per la terminazione del processo, ne analizza lo stato di terminazione e infine, a seconda se terminato o stoppato, lo elimina da PPROCS oppure ne modifica il flag di stato:

```
void add_and_wait(char *proc_name, pid_t pid);
```

Esegue il "`fork->exec`" di un processo e richiama "`add_and_wait`":

```
int execute_and_wait(char **argv_p);
```

Esegue il "`fork->exec`" di un processo ma non ne attende la terminazione:

```
int execute_background(char **argv_p);
```

Valuta lo stato di terminazione di un processo:

```
int exit_status_evaluate(pid_t pid, int exit_status);
```

Verifica se esistono processi in background terminati asincronamente e ne analizza lo stato di uscita:

```
int wait_childs(void);
```

Queste funzioni si occupano quindi di operare direttamente sulla struttura PPROCS (dichiarata variabile globale), rendendo l'implementazione di nuovi comandi o il miglioramento di quelli presenti abbastanza semplice.

I Segnali

La più semplice e vecchia tecnica di comunicazione tra i processi sono i segnali.
Si possono definire come degli "interrupt software" inviati dal Kernel ai processi: permettono una comunicazione sia Kernel->processo sia processo->processo.
Vengono generati da:

- Eccezioni hardware che il Kernel rileva e gestisce (es. divisione per 0, fault di memoria)
- Esecuzione di operazioni per cui non si hanno i permessi (es. scrittura su un file aperto in sola lettura)
- Pressione di una sequenza da terminale (tipo CTRL+C)

Caratteristiche dei segnali rispetto ai processi:

- Sono eventi asincroni
- Ad ognuno di essi è associato un Signal Handler[22] di default. Questo è modificabile con le S.C "`signal`" o (meglio) "`sigaction`"
- Ogni segnale è associato ad un intero maggiore di 0

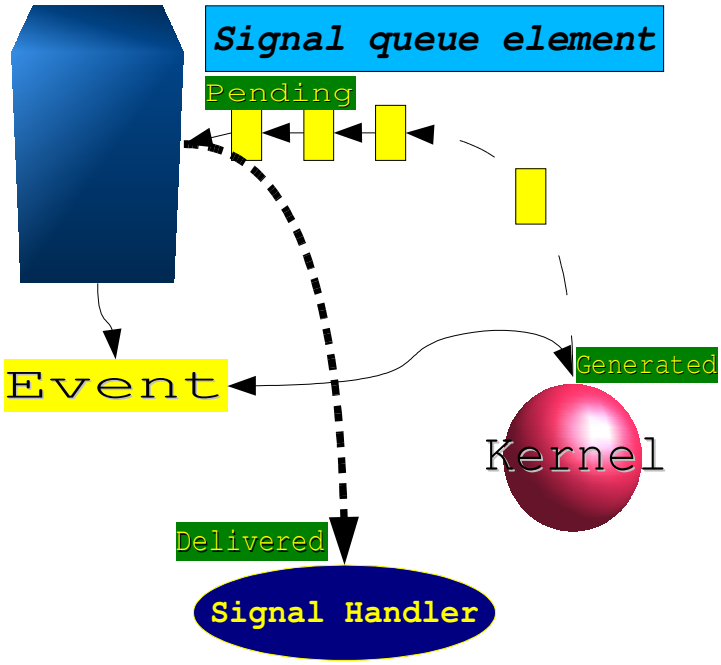
Caratteristiche dei processi rispetto ai segnali:

- Ogni processo ha una coda associata ad ogni segnale
- Alla ricezione di un segnale, il processo esegue il Signal Handler associato

[22] E' in sostanza una funzione che viene eseguita dal processo quando questi riceve il rispettivo segnale. Si può dire anche "gestore del segnale".

- Il processo può mascherare i suoi segnali così da ignorarli

Per quanto riguarda il mascheramento, questo risulta impossibile per 2 segnali particolari: `SIGKILL` e `SIGSTOP`. Sono infatti segnali riservati all'utente ROOT così da poter (in ogni momento) terminare o stoppare processi dannosi per la stabilità del sistema.



Quando un evento determina la generazione di un segnale, questo passa attrraverso tre fasi:

- segnale generato (GENERATED)
- segnale pendente (accodato alla coda del segnale) (PENDING)
- segnale consegnato (il processo ha ricevuto il segnale) (DELIVERED)

Non sono fasi che vengono "attraversate atomicamente": è quindi possibile che, ad esempio, un segnale sia GENERATED nell'istante T ma diventi DELIVERED in un istante S, con S > T. Questo può quindi creare qualche problema nella sequenzialità degli eventi[23].

Come già detto, i segnali sono associati ad interi positivi distiniti, che possono variare di numero da implementazione a implementazione. Lo standard POSIX però ha definito delle costanti mnemoniche per rendere più leggibile e portabile il codice.

SEGNALI e SIGNAL HANDLER: FUNZIONI DI GESTIONE

Esistono apposite S.C per generare segnali (**`signal.h`**):

- `kill` : invia un segnale ad un processo
- `raise` : invia un segnale al processo stesso
- `abort` : invia un segnale di terminazione al processo stesso
- `alarm` : invia il segnale SIGALARM allo scadere di un timer

S.C. per l'associazione ai segnali di particolari Signal Handler (**`signal.h`**) :

- `signal` : assegna il Signal Handler
- `sigaction` : assegna il Signal Handler, attiva i flag per il mascheramento del segnale durante la sua gestione, permette di associare maschere dei segnali complesse.

Ovviamente risulta molto più corretto l'utilizzo della seconda S.C. per la sua completezza e configurabilità.

Tipi di Signal Handler:

- funzione personalizzata
- `SIG_IGN` (fa in modo di ignorare il segnale)[24]
- `SIG_DFL` (reimposta il Signal Handler di default)

---

[23] Insomma che un evento A, che dovrebbe verificarsi prima degli eventi B e C, si interfogli con essi.

[24] E' molto più giusto però utilizzare le maschere dei processi

SEGNALI e GENERAZIONE DI PROCESSI

Poiché le informazioni circa i Signal Handler e le maschere dei segnali sono memorizzati nello User Stack di un processo, al momento dell'esecuzione di una `fork` il processo figlio le eredita.
Invece, dopo un `exec`, la sostituzione dello User Stack comporta che gli Handler vengano risettati a `SIG_DFL` (a parte i segnali impostati a `SIG_IGN`).

PSH: REDIREZIONE SEGNALI

La progettazione della PSH ha richiesto la gestione di 3 segnali particolari generati da terminale:

- `SIGINT` : generato da CTRL+C
- `SIGTSTP` : generato da CTRL+Z
- `SIGQUIT` : generato da CTRL+\

PSH deve intercettare questi segnali e, o "rimbalzarli" al processo in esecuzione in Foreground, o ignorarli[25].

Per far ciò, in prima analisi, si era intenzionati ad utilizzare la S.C. "`signal`" ma, documentandosi meglio, ci si è resi conto che la mancanza di un mascheramento dei segnali durante la gestione degli stessi sarebbe risultata pericolosa. Per questo si è optato per una gestione più complessa ma efficace: si è ricorsi alla funzione "`sigaction`", conforme allo standard POSIX. Non a caso lo standard definisce sigaction come uno "specificatore di azione" e non, come nel caso di `signal`, "specificatore di gestore". Infatti, al prezzo di una maggiore complessità di gestione, permette un **settaggio capillare dell'azione** da intraprendere alla ricezione di un segnale (maschere, flags, ecc.) e **non solo il settaggio di una funzione.**

Filename: `<path_to_psh_sources>/include/signal_and_execs.c`

E' utilizzata per gestire `SIGINT`, `SIGTSTP` e `SIGQUIT`. Redirige questi segnali da PSH al processo in FG, se presente. Altrimenti, nel caso di `SIGINT` visualizza la lista dei processi in esecuzione in BG e chiede a quale di questi reinviare il segnale, mentre per `SIGTSTP` e `SIGQUIT` li ignora:

```
static void psh_signal_handler(int Signal);
```

Si occupa di associare ai segnali l'handler di sopra. Utilizza (come già anticipato) "`sigaction`":

```
void psh_signal_redirection(void);
```

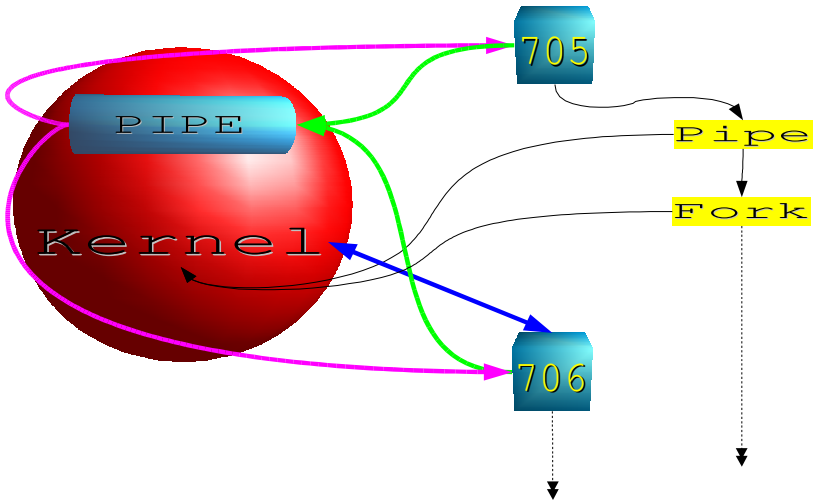
PIPE

Sono uno strumento di comunicazione unidirezionale tra processi. Una PIPE non è altro che un buffer allocato dal Kernel, che i processi utilizzatori vedono come un normale file. Concettualmente si può vedere come un "tubo[26]" a cui sono associati 2 descrittori: uno per il capo in scrittura (ingresso del tubo) e uno per il capo in lettura (uscita dal tubo). Quindi è sufficiente per i due (o più) processi avere uno il descrittore del capo in scrittura e uno il descrittore del capo in lettura per far si che le informazioni scritte dal primo nella PIPE vengano lette dal secondo.

---

[25] Nel caso di SIGINT, se non ci sono processi in FG deve listare tutti i processi in Background
[26] In inglese appunto "PIPE"

La PIPE, come già detto, è un buffer e come tale deve sottostare a determinate condizioni: prima di tutto la necessità, per avere scritture/letture atomiche, di non superare la dimensione del buffer (`PIPE_BUF`). Sono di semplice utilizzo perché permettono di usare le normali S.C. per la lettura/scrittura da/su file. Questo però non è vero "completamente": infatti una PIPE non ha un nome associato sul filesystem, quindi non è possibile eseguire una normale "`open`" sul file. Questo comporta che, per permettere a due processi di condividere i descrittori della PIPE, questi debbano essere, per forza di cose, legati da una relazione di parentela.



Le System Call:

- `pipe` : crea un buffer di comunicazione nel kernel e ritorna 2 descrittori: uno per il lato in scrittura, uno per quello in lettura.
- `popen` : crea una pipe, esegue il `fork-exec` di un processo, ritorna il capo in lettura (o in scrittura) del processo creato.
- `pclose` : chiude la pipe aperta con `popen`.

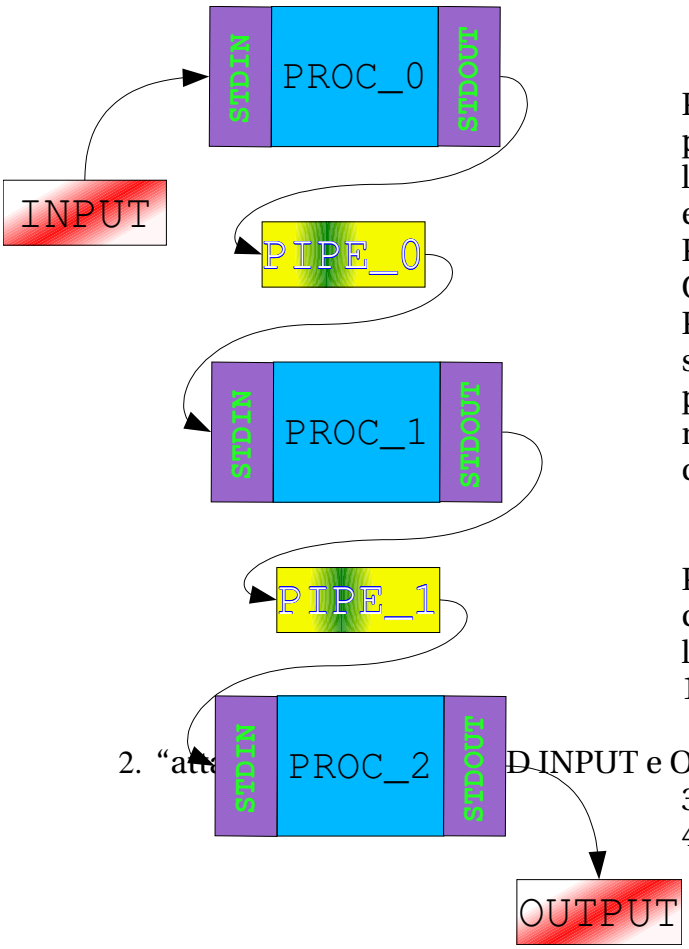
Esiste anche un altro tipo di PIPE detta FIFO (o "PIPE con nome"). Questa si comporta come la PIPE, con l'unica differenza che è associata al nome di un file, e quindi permette la comunicazione anche tra processi che non sono "parenti".

In più, per permettere ai processi di comunicare, esistono altre tecniche:

- Socket : per processi che sono in esecuzione su macchine diverse (condizione non necessaria)
- Code di messaggi : (System V)
- Semafori : (System V)
- Memoria Condivisa : (System V)

Ie prime (Socket) permettono una semplice comunicazione bidirezionale tra processi remoti, basandosi su una logica molto simile a quella delle PIPE.
Le rimanenti tre sono state introdotte da System V, ma sono ormai presenti su tutti i sistemi "Unix Like" (e non solo probabilmente). Il loro funzionamento non viene quì trattato.

PSH: PIPELINE



PSH gestisce anche il “pipelining”: due o più processi possono essere “collegati” attraverso la linea di comando. Per far ciò si utilizza (tra un eseguibile e l'altro) il simbolo delimitatore “ | ”. PSH non fa altro che collegare lo STANDARD OUTPUT di un processo al lato in scrittura di una PIPE, e lo STANDARD INPUT del processo successivo al lato in lettura della stessa PIPE. E' possibile quindi creare “catene di processi”. Il risultato sarà l'Output dell'ultimo processo della catena.

Per la creazione di una “pipeline” PSH si comporta in maniera abbastanza semplice e lineare:

1. Se i processi in pipeline sono N, PSH crea N-1 PIPE
2. “attaccato” INPUT e OUTPUT dei processi alle rispettive PIPE
3. esegue i processi
4. attende la terminazione dell'ultimo processo in pipelining.

Filename: `<path_to_psh_sources>/include/built_in.c`

Si occupa della gestione di tutto il pipelining:

- Crea le pipe
- Crea i processi
- Collega opportunamente i processi alle rispettive pipe
- Esegue i processi

Per una descrizione completa riferirsi al codice fornito:

```
void exec_pipe(char** parsed_processes);
```

Analisi della Linea di Comando

Dato che la più importante “via di comunicazione” tra l'utente e la shell è la linea di comando[28], è basilare un corretto trattamento per l'input da terminale.
Fondamentalmente ciò che la shell deve fare tramite la linea di comando è:

- eseguire processi
- gestire comandi speciali

Ovviamente le moderne shell non si fermano a queste semplici funzionalità, ma queste sono quelle che a noi interessano maggiormente.

Per rispondere a queste esigenze, la shell deve “distinguere” tra tutte le strighe alfanumeriche che legge dallo STANDARD INPUT, quali sono gli eseguibili, quali i comandi built-in, quali i costrutti complessi (come le pipeline o la redirezione su file), quali gli errori di digitazione.

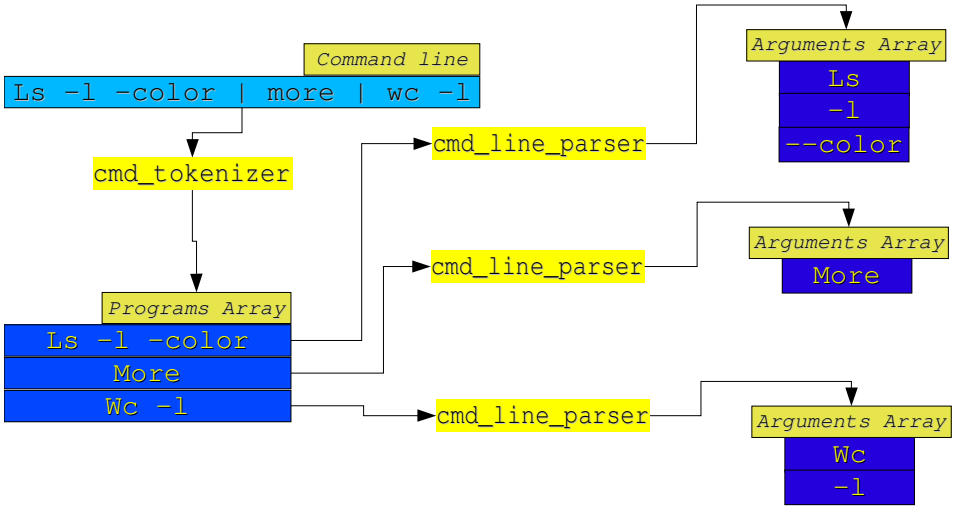
---

[27] Per far ciò utilizza le funzioni di “duplicazione dei descrittori”: `dup` e `dup2`

[28] Cioè tutto ciò che viene scritto sullo STANDARD INPUT dalla tastiera del terminale

Per questo la linea di comendo deve essere scandita e analizzata ad ogni lettura.

PSH: COMMAND LINE



Per l'esecuzione di un programma è necessario richiamare una exec. Tra tutte, quello che si è preferito utilizzare è la `execvp`: prende in ingresso un "array di stringhe" e si occupa della ricerca dell'eseguibile nelle directory listate da PATH.

Per far ciò però è necessario suddividere l'intera riga di comando in tante stringhe, ognuna delle quali è un elemento di questo array.

Filename: `<path_to_psh_sources>/include/cmd_line_parser.c`

Suddivide la stringa in input in un "array di stringhe" che poi verrà passato a `exevp`. L'ultimo elemento (come richiesto da `execvp`) punterà a NULL. Il carattere che "delimita" ogni stringa è lo spazio:

```
char **cmd_line_parser(char *string_cmd);
```

Opera come "`cmd_line_parser`" con l'unica differenza che il carattere delimitatore è "`|`". Viene utilizzata per suddividere e contare i processi in pipeline:

```
char **cmd_tokenizer(char *string_cmd);
```

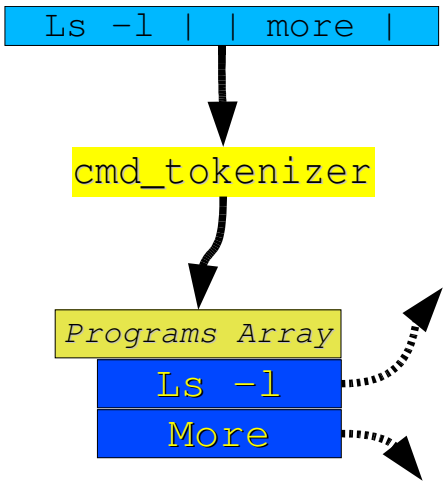
PSH: GLI ERRORI UMANI

Aspetto da non sottovalutare nella progettazione di qualunque applicativo è l'errore umano. Nel caso di una shell questa è una cosa ancora più sentita.
E' facile rendersi conto di quanto sia alta la possibilità di digitare caratteri errati dal terminale. Immettere sequenze sbagliate. Richiamare eseguibili inesistenti. Queste situazioni potrebbero minare alla stabilità stessa dell'applicativo, se questo è scritto con poco criterio. Quindi è fondamentale una "previsione" degli errori.

PSH opera in questa maniera:

1. Suddivide la linea di comando in base al carattere delimitatore "`|`"
2. Controlla la presenza di stringhe vuote tra un delimitatore e l'altro e le elimina
3. Se c'è un solo comando, va al passo 4, altrimenti al passo 6
4. Controlla se è un "built-in": se si lo esegue, se no va al passo 5
5. Esegue il programma
6. Richiama "`exec_pipe`"

Se però i caratteri immessi non sono ne comandi “built-in”, ne eseguibili di programmi, PSH ritorna opportuni messaggi di errore e ristampa il prompt. Queste condizioni di errore sono controllate e “gestite” in maniera lineare, basandosi su una semplice condizione: se non si esegue nulla, non si è immessa una sequenza di caratteri corretta.



Uno degli errori più temuti era l'erroneo utilizzo del simbolo “`|`”: ciò avrebbe potuto portare a risultati imprevedibili. Questi problemi sono stati risolti a monte, implementando la funzione “`cmt_tokenizer`” che, semplicemente, raccoglie solo le sequenza di caratteri corrette, e ignora quelle vuote.

I comandi di PSH

PSH implementa, oltre che i comandi richiesti, anche pochi altri comandi che si ritenevano essenziali. Partiamo da questi ultimi:

- `cd <path_name>` : permette di cambiare directory corrente. Se non si inserisce alcun argomento, la directory correnti diventa la HOME_DIRECTORY dell'utente
- `cwd` : visualizza il percorso completo della Current Work Directory. In realtà è superfluo, perché PSH visualizza nel prompt la stessa informazione
- `env` : visualizza tutte le variabili d'ambiente
- `exit` : termina la sessione di lavoro con PSH
- `PS` : visualizza tutti i processi che PSH ha mandato in esecuzione e che lo sono ancora. In più informa se sono attivi (ACTIVE) o stoppati (BLOCKED)

Ci sono poi quelli previsti dal progetto:

- `back <eseguibile> <arg1> <arg2> ... <argn>` : esegue un processo in Background
- `reco <eseguibile> <arg1> <arg2> ... <argn>` : esegue un processo ricorsivamente in tutte le directory del sottoalbero radicato nella directory corrente.

Filename: `<path_to_psh_sources>/include/built_in.c e signal_and_execs.c`

```
int execute_background(char **argv_p);

int reco(char **argv_p, char *path_dir);
```

Aspetti della programmazione

Il codice è suddiviso in 6 file:

- *`patty_shell_1.0.c`* - è il file principale: contiene il main del programma.
- *`built_in.c`* – contiene tutte le funzioni per i comandi built-in
- *`pprocs_struct.c`* – contiene le dichiarazioni della struttura dati PPROCS
- *`pprocs_struct_function.c`* – contiene tutte le primitive per il trattamento della struttura dati PPROCS
- *`cmd_line_parser.c`* – contiene le funzioni per il parsing delle riga di comando

- `signal_and_execs.c` – contiene le funzioni per la redirezione dei segnali, l'esecuzione sincrona ed asincrona dei processi, la valutazione degli stati di uscita ecc.

Come già accennato, durante il lavoro di programmazione è stato dato un occhio di riguardo alla "riutilizzabilità" di questo codice. Non a caso gran parte delle funzioni ha funzionalità che vanno oltre quello che poi si è effettivamente realizzato per PSH. Per esempio la differenziazione tra un processo in stato "active" ed uno in stato "blocked". O il parser della linea di comando che risolve problemi di digitazione.
O anche il fatto che la struttura PPROCS si preoccupa di conservare anche "il nome" dei processi in esecuzione, per permettere, ad esempio, una successiva implementazione di un comando che opera sui processi referenziandoli per nome e non solo per PID.

Inoltre si è cercato di implementare un minimo di tecniche di pulizia della memoria utilizzata nel caso del parsing della riga di comando, per evitare che, a lungo andare, la memoria utilizzata dal processo cresca smisuratamente[29].

Per quanto riguarda i segnali, l'aspetto che ha creato maggiori grattacapo è stata l'atomicità delle operazioni, ed è proprio questa che ha poi creato non pochi problemi per la gestione della terminazione dei processi: infatti, utilizzando la funzione "*waitpid*" ci si è accorti del fatto che, nel caso si utilizzi l'opzione WUNTRACED, l'aggiornamento della struttura "*task_struct*" all'interno del kernel non era atomica, e così a volte poteva capitare che la valutazione dello stato di uscita di un processo, non mettesse in evidenza il suo essere stoppato e non terminato. La soluzione ha questo problema è stato l'utilizzo di una variabile logica globale di controllo che cambiava nel caso fosse digitata la sequenza CTRL+Z. Ovviamente questo preclude la possibilità anche di monitorare processi che risultano stoppati da altri segnali come SIGSTOP, SIGTTOUT e SIGTTIN.

Altro problema, rimasto inrisolto, è il non perfetto funzionamento del pipelining: il collegamento di STDIN e STDOUT funziona a dovere; unico problema è che l'ultimo processo della pipe (a volte) resta bloccato dopo aver eseguito tutti i suoi compiti. Il problema è stato risolto invitando l'utente alla pressione della sequenza CTRL+C.

Limiti e "TO DO"

Elenchiamo ora tutti i problemi noti e che saranno successivamente risolti:
1. L'ultimo processo di una pipeline a volte si blocca dopo aver eseguito tutti i suoi compiti.
2. Risultano processi "stoppati" solo quelli che lo sono per la pressione della sequenza CTRL+Z
3. Il comando built-in "reco" esegue una volta in più la sua ultima chiamata
4. Non è ancora possibile utilizzare comandi built-in in pipe. Ma la modifica sarà molto semplice.
5. Il job-controlling implementato è troppo semplice. Ad esempio non è possibile riportare in ForeGround che è in esecuzione in BackGround.

[29]Consultare il file "`cmd_line_parser.c`" per chiarimenti su queste funzioni.