Sistema Operativo Unix/Linux

Nella seconda metà degli anni sessanta Ken Thompson ricercatore della Bell Labs della AT&T realizzò un gioco del nome Space Travel.

L'esecuzione del gioco sul GE645 (mainframe della General Electrics su cui la Bell Labs doveva realizzazione di un sistema operativo chiamato Multics) non avveniva in maniera soddisfacente anche a causa del Sistema Operativi.

Di qui l'esigenza di sviluppare un supporto diverso e Thompson e alcuni suoi colleghi (fra cui Dennis Ritchie) svilupparono un nuovo sistema operativo multi-tasking, che sfruttava una gestione del file system innovativa e comprendeva un interprete di comandi ed alcune utility, per un altro tipo di computer: il DEC PDP-7.

Brian Kernighan lo chiamò UNICS (Uniplexed Information and Computing System), poco dopo sintetizzato in Unix.

Nel 1973 Thompson e Ritchie riscrissero il kernel in C (ideato dallo stesso Ritchie e da Kernighan) e questo Unix facilmente mantenibile e soprattutto ampiamente portabile.

Nel 1982 Unix divenne un prodotto commerciale, regolarmente distribuito da At&T anche se a quel punto esistevano diverse versioni di Unix, sviluppate a partire dal codice originario da centri di ricerca indipendenti.

Nel 1983 At&T sviluppò Unix System V Release 1 impegnandosi a mantenere la compatibilità.

Fra le realizzazioni più importanti citiamo BSD (Berckley Software Distribution) e XENIX.

BSD un figlio di Unix nato all'Università di Berckley, su richiesta del DARPA (Dipartimento della Difesa).

William Joy, uno degli autori di BSD, fondò la Sun Microsystems, dove venne realizzata la versione di Unix nota come Sun OS e poi sviluppatasi in Solaris.

XENIX nacque nel 1980 ad opera di Microsoft ma non ebbe molto successo e il suo maggior contributo fu l'introduzione di Unix nel mondo dei desktop.

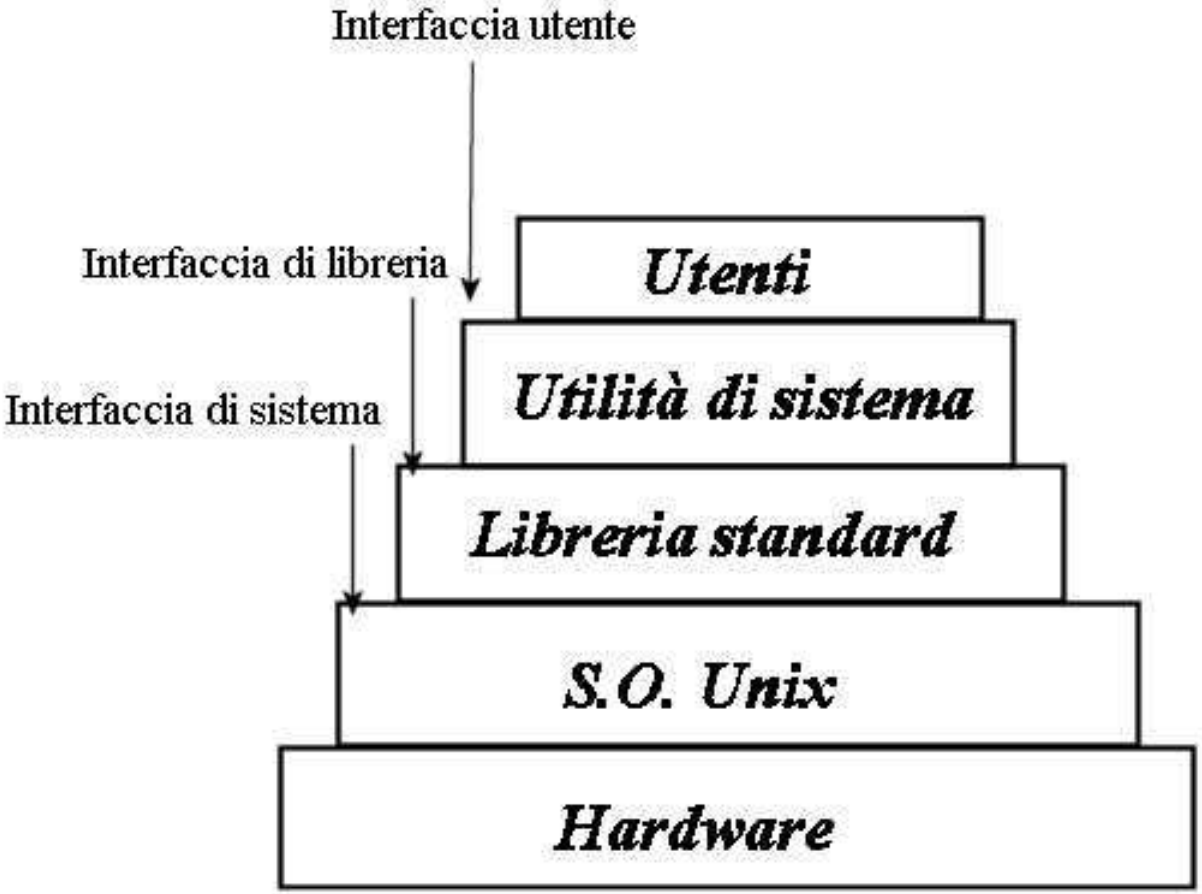
Fra i molti che hanno contribuito a Unix oltre i già citati Thompson, Ritchie, Kernighan e Joy si devono citare R.Canady (coautore del file system), J.Ossanna (autore di troff), D. Korn e S. Bourne (autore della kornshell –ksh– e della bourne shell –bsh– rispettivamente), e R. Stallman (autore di emacs e fondatore della Free Software Foundation).

Nel 1984 dopo la cessione ad AT&T, Unix divenne un prodotto proprietario,e Richard Stallman, rifiutandosi di sottostare a questa logica, decise di dare vita ad un progetto per realizzare ed assemblare da zero un nuovo O.S. di tipo Unix il cui codice sorgente fosse libero e nacque così GNU.

All'inizio degli anni '90, Linus Torvalds, studente finlandese, iniziò ad apportare variazioni a Minix, il SO di tipo Unix per pc sviluppato da Andrew S. Tanenbaum, per puri fini didattici.

Alla fine del 1991 Torvalds pubblicò la prima versione di Linux, chiamato così ancora per un gioco di parole tra Minix e il suo nome.

Unix/Linux è un S.O multiutente e multiprogrammato strutturato a livelli ad ognuno dei quali corrisponde una diversa funzionalità.



Unix è suddiviso in

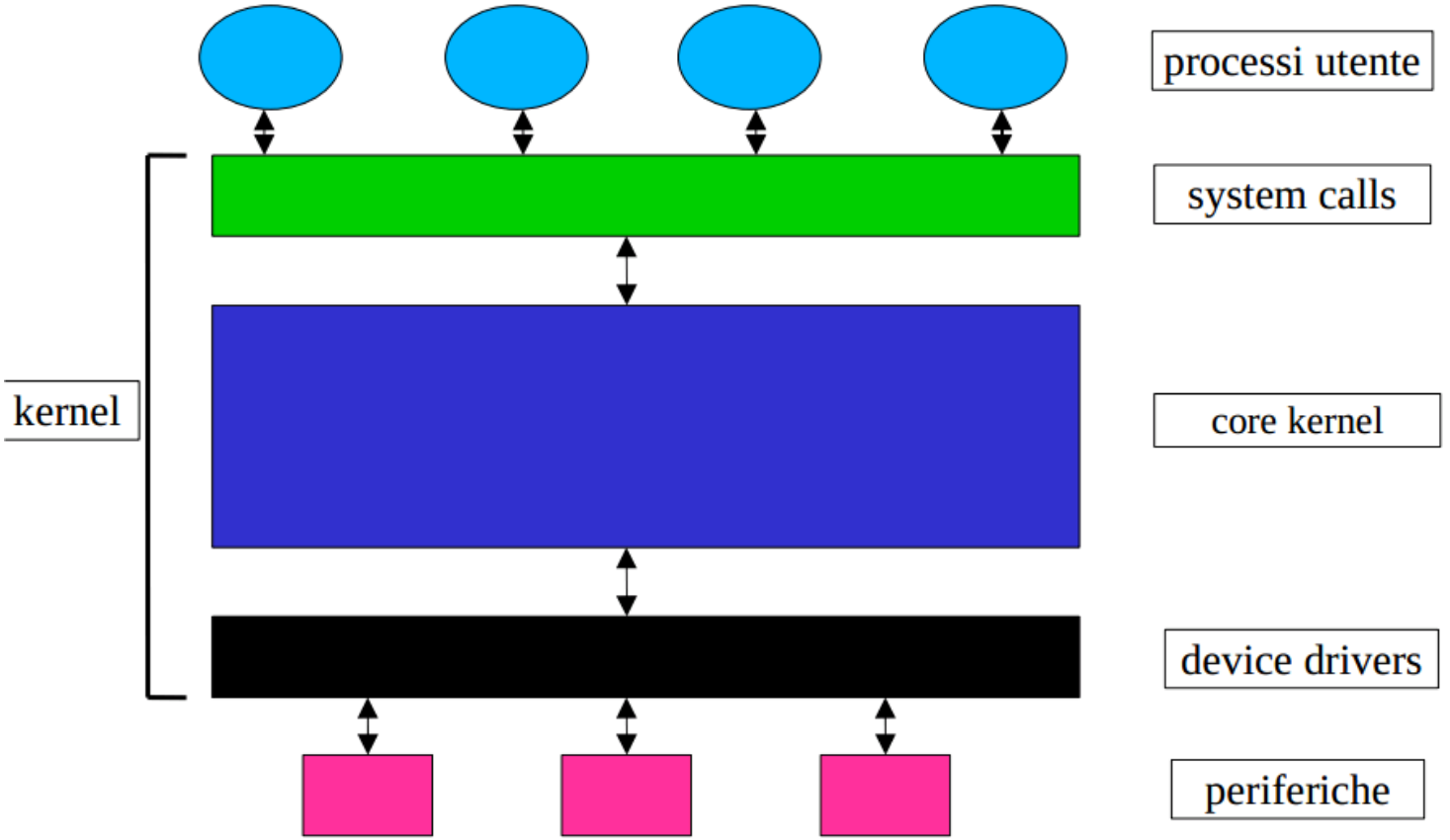
Core Kernel in cui è implementato solo l'essenziale tutto il resto è implementato a livello utente (inclusa la shell)

- inizializzazione del sistema
- gestione dei processi
- gestione delle risorse
  - scheduling della CPU
  - allocazione e protezione della memoria
  - etc.
- gestione dei filesystems
- gestione dei meccanismi di protezione
- gestione "astratta" delle periferiche

Due strati di interfaccia, ben separati dal core kernel

System calls verso i processi utente

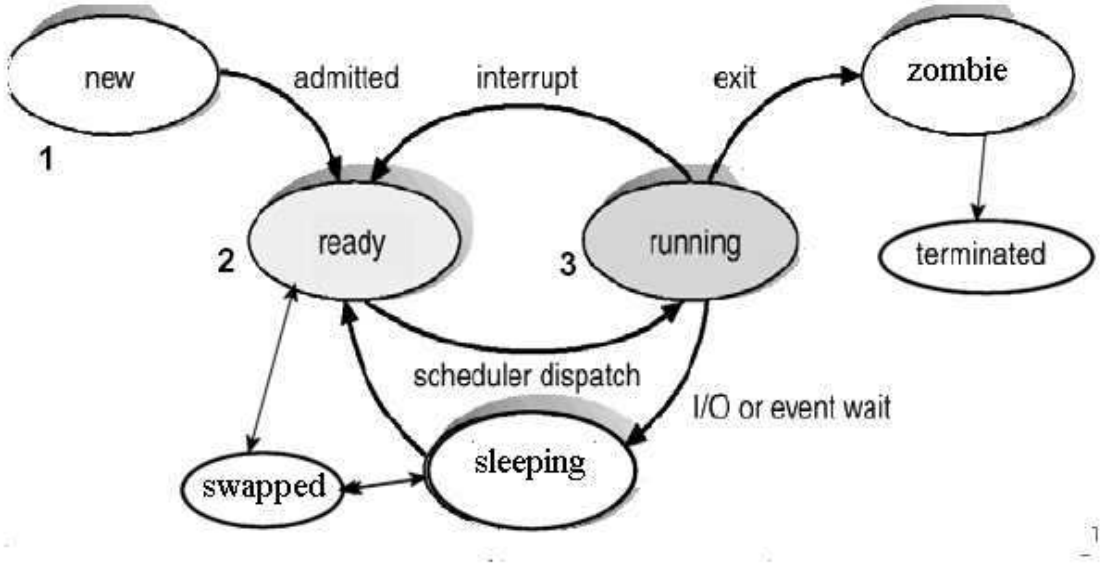
Device drivers verso il mondo esterno (periferiche fisiche)



Processi Unix/Linux

Nelle maggior parte delle realizzazioni, Unix/Linux è basato su processi e non prevede il multitheading ossia il processo Unix prevede un solo thread.

Il diagramma degli stati dei processi è il seguente.



Nei sistemi operativi Unix/Linux, un processo zombie o processo defunto è un processo che nonostante abbia terminato la propria esecuzione, possiede ancora un PID ed un process control block, necessario per permettere al proprio processo padre di leggerne il valore di uscita.

Quando un processo termina, tutta la memoria e le risorse ad esso associate vengono liberate così da poter essere utilizzate da altri processi.

Ciò nonostante, il process control block del processo resta nella tabella dei processi (process table) affinché il processo padre possa leggerne il valore di uscita eseguendo la chiamata di sistema wait(), al seguito della quale il processo zombie viene definitivamente rimosso e i relativi PID e process control block possono essere riutilizzati.

Un processo si trova in stato di Zombi quando ad esempio il processo padre che lo ha generato è terminato senza attendere la terminazione dei suoi figli.

La funzione che crea un nuovo processo in Unix/Linux è fork, una system call che crea un nuovo processo uguale a quello del padre in cui l'unica differenza tra processo padre e processo figlio è il valore restituito dalla funzione fork:

= Pid del processo figlio nel padre,

= zero nel figlio.

Esempio di Fork

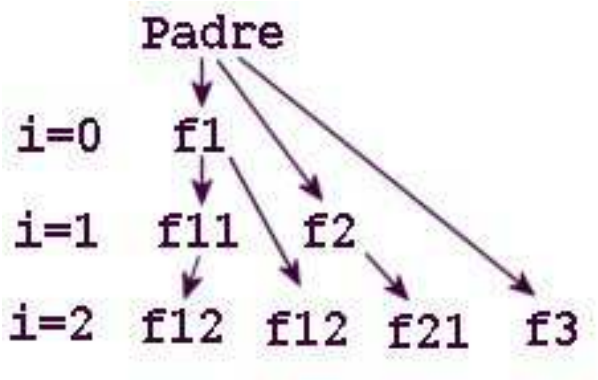
```
#include<stdio.h>
#include<sys/types.h>
main()
 {int n;
  n = fork();
  if (n == -1)
  {fprintf(stderr,"fork fallita\n");
   fflush(stdout);exit(1);}
  else
  if ( n == 0) /* processo figlio */
   {printf("\nsono il figlio; risultato della fork =%d\n",n);
    printf("\n(figlio) il mio id= %d\n",getpid());
    printf("\n(figlio) id di mio padre = %d\n",getppid());
    exit(0);}
   else /* processo padre */
    {printf("\nsono il padre; risultato della fork =%d\n",n);
      printf("\n(padre) mio id= %d\n",getpid());
      exit(0);}
  }
```

Supponiamo di voler creare n figli di un processo padre con fork.
Il seguente codice sembra essere corretto

```
#include<stdio.h>
#include<sys/types.h>
#define N 2
main()
{int id,i;
 for(i=0;i<N;i++)
 {id = fork();
  if (id == 0) /* processo figlio */
  {printf("\n(figlio %d) il mio id= %d\n",i,getpid());
   printf("\n(figlio) id di mio padre =%d\n"i,getppid());}
  else /* processo padre */
  {printf("\nsono il padre; risultato della fork =%d\n",n);
   printf("\n(padre) mio id= %d\n",getpid());}
}
```

In realtà poiché il processo viene clonato ad ogni chiamata di fork il padre effettivamente crea n figli, ma anche i figli creeranno n-1 figli, ed per ognuno di essi il processo “nipote” crea n-2 figli ect.

Alla fine creeremo n*n-1*n-2…=n! Figli.



Per fare in modo che i processi generati siano solo i figli del padre si deve terminare tutti i processi figli ad esempio con un con un edita().

Sospensione dei Processi

Uno degli usi più comuni del multitasking e la creazione di programmi di tipo server, in cui un processo principale attende le richieste che vengono poi soddisfatte da una serie di processi figli. In questo caso è necessario gestire esplicitamente la conclusione dei figli onde evitare di riempire di zombie la tabella dei processi.

Le funzioni deputate a questo compito sono sostanzialmente due, wait e waitpid.

wait

La funzione wait ha la seguente sintassi:

int wait(int *status)

La funzione wait() sospende il processo corrente finché un figlio (child) termina o finché il processo corrente riceve un segnale di terminazione o un segnale che sia gestito da una funzione.

Quando un child termina il processo, senza che il parent abbia atteso la sua terminazione attraverso la funzione di wait() , allora il child assume lo stato di "zombie" ossia di processo "defunto".

Se il processo corrente esegue la funzione di wait() , mentre ci sono uno o più child in stato di zombie, allora la funzione ritorna immediatamente e ciascuna risorsa del child verrà liberata.

La funzione wait restituisce

il pid del figlio in caso di successo,

-1 in caso di errore.

Nel caso un processo abbia più figli il valore di ritorno (il pid del figlio) permette di identificare quello che è terminato.

Se il parametro status non è NULL, la funzione memorizza l'informazione dello stato nell'area di memoria puntata da status.

Ossia se il figlio ha eseguito un exit(9); l'indirizzo di memoria puntata da status avrà valore 9.

waitpid

La funzione waitpid ha la seguente sintassi:

int waitpid(pid_t pid, int *status, int options)

La funzione waitpid() sospende il processo corrente finchè il figlio (child) corrispobndente al pid passato in argomento termina o finché il processo corrente riceve un segnale di terminazione o un segnale che sia gestito da una funzione.

Se il processo corrente esegue la funzione di waitpid() e il child identificato dal pid è in stato di zombie, allora la funzione ritorna immediatamente e ciascuna risorsa del child viene liberata.

Il valore del pid può essere

- <-1 la funzione attende ciascun child avente il process group ID uguale al valore assoluto del pid,
- 0 la funzione attende ciascun child avente il *process group ID* uguale a quello del processo corrente
- >0 la funzione attende il child avente il *process ID* corrispondente al valore del pid

Se status non e' NULL, la funzione memorizza l'informazione dello stato nell'area di memoria puntata da questo argomento.

Le seguenti macro sono utilizzate per valutare lo stato:

- WIFEXITED(status)

  risulta vera se il child è uscito normalmente

- WEXITSTATUS(status)

  riporta gli 8 bit meno significativi del codice di ritorno del child. Il child può comunicare il codice di ritorno al parent, attraverso l'argomento passato alla funzione exit() o return della funzione main(). Questa macro può essere valutata solamente se la macro WIFEXITED e' risultata vera.

- WIFSIGNALED(status)

  risulta vera (diversa da zero) se il child e' uscito per mezzo di un segnale non gestito.

- WTERMSIG(status)

  riporta il segnale (il suo numero) che ha causato il termine del processo figlio. Questa macro può essere valutata solamente se la macro WIFSIGNALED e' risultata.

- WIFSTOPPED(status) risulta vera se il child è fermo (stop). Questa possibilità è condizionata dall'impiego del flag WUNTRACED nell'argomento delle options .

- WSTOPSIG(status)

  ritorna il numero del segnale che ha causato il child a fermarsi (stop). Questa macro può essere valutata solamente se la macro WIFSTOPPED e' risultata vera

Il parametro options può essere zero o i seguenti valori costanti anche messi in OR:

- NOHANG ritorno immediato se i child non sono usciti.
- WUNTRACED ritornare anche se i child sono fermati (stop), e lo stato non deve venire riportato.

la funzioni waitpid ritorna:

- il process ID del child che termina in caso di sucesso.
- -1 in caso di errore e errno viene settato in modo appropriato.
- 0 se e' stato impiegato WNOHANG e non ci sono figli che hanno terminato.

Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{pid_t ok_fork; // status di generazione del figlio
 int w; // informazioni sullo status del figlio
 // prova a generare il figlio
 if ((ok_fork=fork())<0)
 {printf("Error !!! File %s, line %d\n",__FILE__,__LINE__);
  exit(EXIT_FAILURE);}
  if (!ok_fork)
    printf("My pid is %d, my parent pid is %d, I'm the forked
child\n",getpid(),getppid());
 else
  printf("My pid is %d, my parent pid is %d, my child pid is
%d\n",getpid(),getppid(),ok_fork);
  // il figlio attende 3 caratteri da terminare (invio
incluso)
 if (!ok_fork)
  {getchar();getchar();
```

```
    if (getchar()==EOF) printf("EOF\n");
   }
  if(!ok_fork)// il figlio termina
   printf("Bye bye from the child\n");
  else
   {while (1)
    {// il padre attende informazioni sullo stato del figlio
     wait(&w);
     // se il figlio ha terminato attende un carattere da
     // terminale e poi conclude altrimenti aspetta nuove
     // informazioni sullo status del figlio
     if (WIFEXITED(w))
      {printf("Child cleaned\n");
       getchar();
       printf("Bye bye from the parent\n");
                  break;
              }
          }
    }
   exit(EXIT_SUCCESS);
}
```