ARISO 2

13: GESTIÓ DE PROCESSOS

- int fork()
  retorn → Pare : PID fill o -1 si error
  Fill : Ø

  Crea un fill del procès, les variables, TDVA, etc - del fill són una copia de les del pare.
  El fill comença l'execució en el punt on s'ha fet el fork().

- void exit (int codi_finalització)

  Termina l'execució del procès. El pare pot obtenir codi_fin.
  Es destrueix la TDVA, i per tant es decrementa el nombre de links de cada entrada de la TFA.

- int wait (int *codi_finalització_fill)
  retorn → PID procès que ha acabat o -1 si error

  - Si existeixen fills actius es bloqueja fins que un termina i retorna el PID d'aquest. i el codi_fin
  - Si no hi ha fills actius retorna -1.
  - Si hi ha zombies retorna el PID i codi_fin i destrueix el zombie.

- int waitpid(int pid, int codi_fin, int flags)

  - Idem anterior però podem especificar esperar un procés en concret (pid = PID) o qualsevol fill (pid = -1)
  - Si flags = WNOHANG el pare no es bloqueja.

- int execv (char *path, char *arg0, ... , null);
  int execl (char *path, char * arg[]);
  retorn : Ø si OK, -1 si error

  Executa un programa en l'espai del procès que fa la crida, de forma que aquest mor, però es conserva l'estructura de la TDVA pel nou procès.

- int getpid() / int getppid()

  retorn : el primer retorna el propi pid i el segon el pid del pare.

T4: GESTIÓ DE FITXERS

- ✱ int open (char *path, int flags, [int mode])
  return : # entrada a la TDVA, -1 si error
  flags:
  mode d'apertura: O_RDONLY, O_WRONLY, O_RDWR
  mode de creació (opcional):
  O_CREAT → Si no existeix el crea.
  O_CREAT | O_EXCL → " " , sino error
  O_APPEND → Quan es faci el primer write serà sobre EOF
  O_TRUNC → Borra el contingut del fitxer
  mode: permissos per O_CREAT p.e.: 0644

  - Cada crida a open() crea una nova entrada a la TFA i busca la primera posició lliure a la TDVA perque apunti a ella.

- ✱ int close (int tdva_pos)
  return: 0 si OK, -1 si error

  - Allibera l'entrada tdva_pos i decrementa les referencies del fitxer a la entrada de la TFA associada i si arriba a 0 l'allibera.

- ✱ int dup (int tdva_pos)
  return: # nova entrada TDVA, o -1 si error

  - Duplica l'entrada tdva_pos i incrementa el número de referències al fitxer, a la TFA.

- ✱ int dup2 (int tdva_src, int tdva_dst)

  - Idem anterior però permet especificar la posició de destí a la TDVA, si aquesta està ocupada l'allibera abans

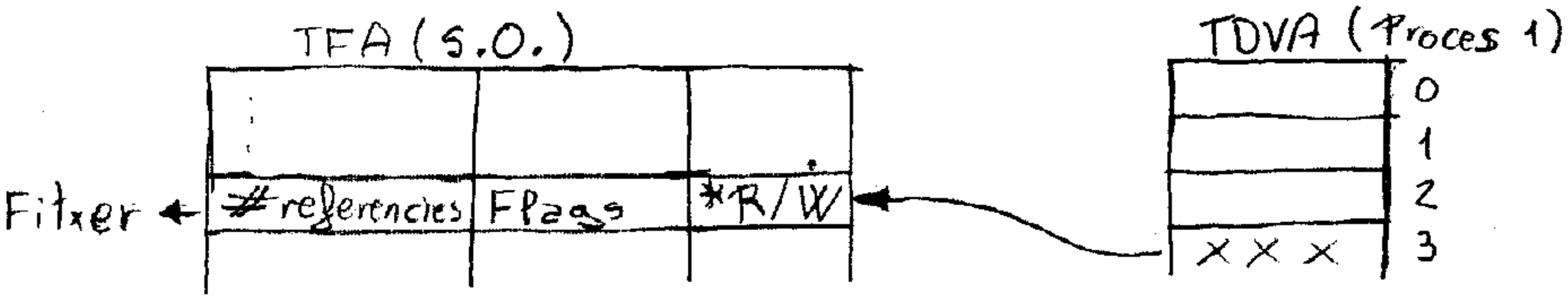
- ✱ int read (int tdva_pos, void *buffer, int nbytes)
  return: -1 error, 0 EOF, # bytes llegits

- ✱ int write (int tdva_pos, void *buffer, int nbytes)
  return: -1 error, # bytes escrits

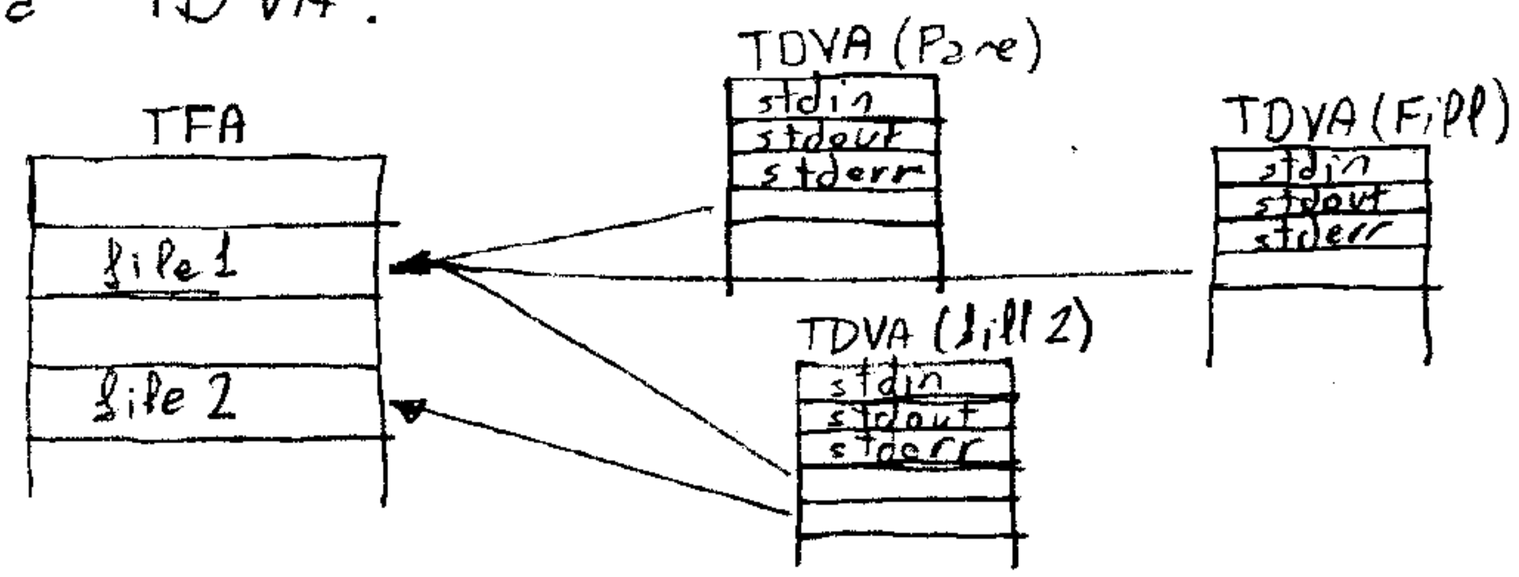
- ✱ int lseek (int tdva_pos, long desplaçament, int reference_pos)
  return: Nova posició dins el fitxer, -1 si error
  reference_pos:
  SEEK_SET → Desplaçament des de la posició inici.
  SEEK_CUR → " " " " actual.
  SEEK_END → " " " " EOF.

- ESTRUCTURA DE LA TFA, TDVA

TFA → Tabla de Ficheros Abiertos
TDVA → Tabla de Dispositivos virtuales abiertos



- Quan un procès crea fills aquests copien la TDVA i s'incrementa #ref a la TFA
- Cada cop que es fa un open es crea una nova entrada a la TFA i s'ocupa la primera pos lliure de la TDVA.



- Redireccionar stdin (teclat) pq sigui un fitxer

```
close (0);
open("file", O_RDONLY|O_CREAT|O_EXCL ,0640);
```

- Redireccionar stdout (pantalla) a un fitxer

```
close (1);
open ("file", O_WRONLY|O_CREAT ,0640);
```

T5: GESTIÓ DE PIPES

* int pipe(int fd[2])
  return: 0 OK, -1 error

  Crea una pipe. fd[0]: canal de lectura
  fd[1]: canal d'escriptura

- Per llegir ⇒ read(fd[0], ...)
- Per escriure ⇒ write(fd[1], ...)
- Per tancar s'utilitza close() sobre el canal a tancar.

READ | ○ Si pipe buida & ∄ escriptors oberts ⇒ retorna 0
○ " " ∃ " " ⇒ bloqueig

WRITE | ○ Pipe plena & ∄ lectors oberts ⇒ retorna -1
○ " " ∃ " " ⇒ bloqueig

⚠ Per evitar bloquejos cada proces ha de tancar les pipes que no farà servir.

16: GESTIÓ DE SIGNALS

Signal: Interrupció software que pot ser enviada a un procés per informar-lo d'algún event.

✱ struct sigset_t : contenidor de signals

int sigemptyset(sigset_t *set)
⇒ set = cap signal

int sigfillset(sigset_t *set)
⇒ set = tots els signals

int sigaddset(sigset_t *set, int signal)
⇒ set = set + signal

int sigdelset(sigset_t *set, int signal)
⇒ set = set - signal

int sigismember(sigset_t *set, int signal)
return: 1 set conté signal, 0 en altre cas

✱ int sigprocmask(int action, sigset_t *set, sigset_t *old)

Modifica o consulta signals bloquejats per un procés.

action: SIG_BLOCK : pmask += set
SIG_UNBLOCK : pmask -= set
SIG_SETMASK : pmask = set

| set != NULL ⇒ modificar | old != NULL ⇒ consultar
| set == NULL ⇒ consultar | old == NULL ⇒ ---

⚠ Quan un signal arriba i es troba bloquejat, no s'executa fins que es desbloqueja i s'executa un sol cop independentment de quantes vegades hagi arribat

⚠ Un signal es bloqueja durant l'execució de la seva RAS.

⚠ Les variables usades a la RAS i al main han de ser globals, i la seva modificació al main s'ha de fer dins d'una zona crítica.

- int Kill(int pid, int signal)

  Genera un signal per al proces especificat.

- int alarm(int seconds)
  retorn: segons que queden fins exhaurir la petició anterior

  Programa el temporitzador del procés per generar una alarma.

⚠ alarm(0) desactiva la alarma.

- struct sigaction
  {
  void *sa_handler; // direcció de la RAS
  int sa_flags; // ho posem a Ø sempre
  sigset_t sa_mask;
  }

  sa_handler: SIG_DFL → comportament per defecte
  SIG_IGN → ignorar el signal
  rutina d'atenció → void ras(int signal)

  sa_mask: conjunt de signals a bloquejar mentre s'executa la ras, a part dels que ja ho estén.

- int sigaction(int signal, struct sigaction *act, struct sigaction * old)

  Modifica o consulta la rutina de tractament d'un signal.

  set != NULL ⇒ nou tractament associat al signal
  set == NULL ⇒ consultar
  old != NULL ⇒ consultar

- int sigsuspend(sigset_t *set)

  Espera que arribi algún signal NO contingut en set.
  set: és la nova màscara del procés mentre estigvem bloquejats en aquesta crida.

17: SOCKETS

* int socket (int family , int type , int protocol)
  return : >0 socket file descriptor.

  family : PF_INET. (IPv4)
  type : SOCK-STREAM (tcp), SOCK_DGRAM (udp)
  protocol = 0

* struct sockaddr_in {
    sin_len = sizeof (struct sockaddr_in)
    sin_family = AF_INET
    sin_port = htons (<port>)
    sin_addr . s_addr = inet_addr(<dirIP>)
  } . s_addr = htonl (INADDR_ANY)

* Conversions network byte ordes ⇔ host byte order

  ushort htons (ushort) } Per port
  ushort ntohs (ushort) }
  ulong htonl (ulong) } Per adreces IP
  ulong ntohl (ulong) }

* Conversió adreces IP : array ⇔ unsigned long

  ulong inet_addr(char *dirIP)
  char *inet_ntoa (struct in_addr in)

* Conversió de noms a @IP

  struct hostent gethostbyname (char *hostname)
  struct hostent gethostbyaddr (ulong *addr, int a_len, int type)

* int bind(int sockfd , struct sockadrr *addr, int addrlen)
  sockfd : valor retornat per socket()
  addr : estructura amb les dades per configurar el socket
  addrlen : tamany de l'estructura addr.

  S'utilitza per configurar un socket com a servidor

* int listen (int socketfd, int backlog)
  backlog : numero màxim de conexions incompletes (5)

  Posa el socket en estat de listen

* int accept(int socketfd, struct sockaddr cliaddr, int *addrlen)
  socketfd: socket del server
  cliaddr, addrlen: info del nou client conectat

  Retorna el descriptor d'un nou client conectat.

* int connect(int socketfd, struct sockaddr srvaddr, int addrlen)
  socketfd: socket del client
  srvaddr, addrlen: info del server al que volem conectar.

  Retorna socketfd si conecta i -1 si error

* read(), write(), close()
  Funcionen igual que per a fitxers