Fonaments dels Sistemes Operatius

Departament d'Informàtica de Sistemas i ComputAdores (DISCA)
Universitat Politècnica de València

fSO

Pràctica 4
Creació de processos en UNIX
Versió 2.0

Contingut

1. Objectius

La **creació de processos** és un dels serveis més importants que ha de proporcionar un sistema operatiu multiprogramat. En aquesta pràctica estudiem la interfície d'aplicació que ofereix UNIX. Els objectius són:

- Adquirir destresa en la utilització de les crides POSIX, **fork(), exec(), exit() i wait().**
- Analitzar mitjançant exemples de programes el comportament de les crides en termes de **valores de retom, atributs heredats entre processos.**
- Visualitzar els diferents **estats dels processos, execució(R), suspendido (S), zombie (Z)**

En aquesta pràctica es treballen els conceptes i crides impartides en el Seminari 3 titulat "*Crides al sistema UNIX per a processos*", de l'assignatura Fonaments de Sistemas Operatius.

2. Introducció

UNIX utilitza un mecanisme de clonació per a la creació de processos nous. Es crea una nova estructura PCB que da suporte a la còpia del procés que invoca la **crida al sistema fork()**. Ambdós processos, creat (fill) i creador (pare) són idèntics en l'instant de creació però amb **identificadors PID i PPID distints**. El **procés fill hereta** una còpia dels **descriptors d'arxiu, el codi i les variables del procés pare**. Cada procés té el seu propi espai de memòria, les variables del procés fill estan ubicades en posicions de memòria diferents a les del pare, per això quan es modifica una variable en un dels processos no es reflexa en el altre. Resumint amb la crida a *fork()* se crea un nou procés "fill":

- Que és una còpia exacta del creador "pare" excepte en el PID i PPID
- Que hereta les variables del procés pare i evolucionen independentment del pare
- Que hereta del procés pare la tabla de descriptors podent compartir arxius amb ell

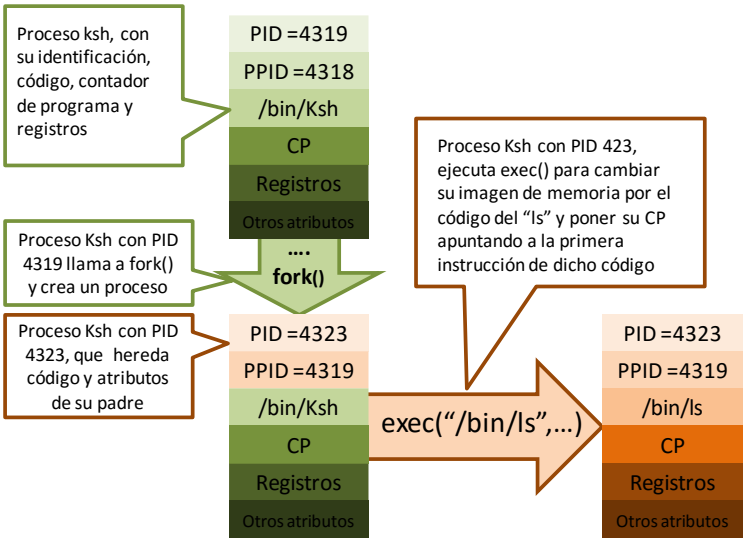
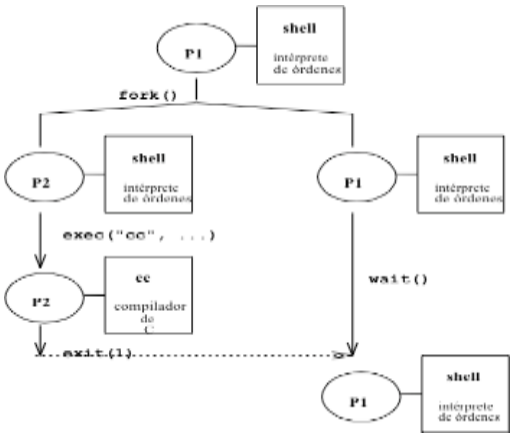


Figura-1: Crides fork() i exec()

La crida exec() canvia el mapa de memòria d'un procés, és a dir, "allò que va a executar", les seues variables. La crida a exec() canvia el contingut de la memòria del procés que la invoca carregant noves regions de dades, codi etc. i ficant el comptador de programa del procés apuntant a la primera instrucció. El codi se carrega a partir d'un arxiu de disc que conté codi executable. Interpretant l'arxiu executable (en format ELF) el sistema operatiu crea les regions corresponents i carrega els seus valors en les dades.



La creació de processos genera dins del sistema un arbre de parentiu entre ells que vincula als processos en termes de sincronització. Un cas d'aplicació típic de la sincronització entre pare i fills és el Shell.
El Shell de Unix és un procés que crea un procés fill per cada ordre invocada des del terminal. El procés Shell espera a que el procés que executa el comando finalitze i imprimeix el *prompt* en pantalla (excepte en l'execució en background) i queda a la espera d'un nou comando. Aquesta sincronització del Shell amb el seu fill es realitza mitjançant les crides *wait()* o *waitpid()* i *exit()*.

Figura-2: El Shell de UNIX

En el Annex de la pràctica se detallen, la crida *exit() i wait()*. exit() finalitza l'execució d'un procés i wait() suspèn l'execució del procés que la invoca fins que algun dels seus processos fills acabe. Utilitza també el manual del sistema per a consultar detalls d'aquestes crides

```
$man fork
```

3. Creació de processos en UNIX amb fork()

UNIX crea processos nous amb la crida al sistema `fork()`.
Crea un nou directori per a emmagatzemar tots els arxius d'aquesta pràctica amb:

```
$mkdir $HOME/pract4
```

El codi de la figura-3 crea un procés fill mitjançant la crida fork().

```
/***code of my_child.c***/
#include<stdio.h>
#include<stdlib.h>
int main(intargc, char *argv[])
{    printf("Parent process %ld \n", (long)getpid());
     fork();
     printf("I am %ld process, my parent is
%ld\n",(long)getpid(),(long)getppid());
     sleep(15);
     return0;
}
```

Figura-3: Codi de my_child.c

Exercici 1: Creant un procés fill amb la crida fork() "my_child.c"

Crea un arxiu denominat *my_child.c i* escriu en ell el codi de la figura-3, compila-ho i executa-ho en background. Ompli la tabla 1 amb els PID, PPID i l'ordre que executen els processos creats.

```
$ gcc –o my_child my_child.c
$./my_child&
$ps -la
```

	PID	PPID	COMMAND
Procés Pare	3348	3028	./my_child&
Procés fill	3349	3348	fork()

Taula-1: Exercici-1, creant processos amb la crida fork()

4. Valores de retorn del fork()

El valor de retorn de la crida `fork()` permet identificar al procés pare e fill i decidir que codi executen cadascun. `fork()` retorna un 0 a els fills i un valor positiu, el PID del fill, al procés pare. La figura-4 il·lustra el concepte dels valores retornats per `fork()` al procés pare e fill.

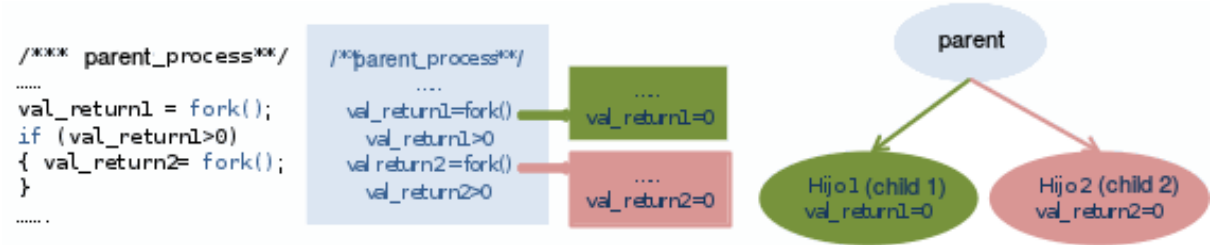


Figura-4: D'esquerra a dreta es mostra, el codi en c que ens permet diferenciar el valor retornat per `fork()` a els processos pare e fill. L'arbre de processos que es genera.

L'esquema general de codi en C a utilitzar per a que un procés pare cree un nombre n de processos fills en ventall, és el mostrat en la figura-5.

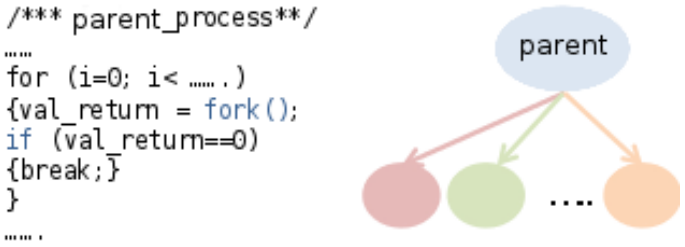


Figura-5: Esquema de processos creats en ventall, estructura de codi i arbre de parentiu.

Exercici 2: Valor de retorn de la crida fork() "range_process.c"

Crea un arxiu denominat *range_process.c* i escriu en ell el codi necessari per a la creació de 5 processos fills en ventall. Introdueix les crides i instruccions necessàries per a que:

- Mitjançant `i` iteracions d'un bucle for, se creen 5 processos en ventall.
- Cada fill escriga "`printf( "Fill creat en iteració=%d", i)`" i després acabe la seua execució amb una crida a `exit(i)`.
- El procés pare realitze un `sleep(10)` després de crear a els fills i a continuació faça `exit(0)`.

Compila el programa i executa-ho en background. Immediatament execute "`$ps l`". Ompli la tabla-2.

	PID	PPID	COMMAND	ESTAT
Procés Pare	3531	3028	ejecutable	T
Procés fill 1	3532	3531	ejecutable <defunct>	Z
Procés fill 2	3533	3531	ejecutable "	Z
Procés fill 3	3534	3531	" "	Z
Procés fill 4	3535	3531	" "	Z
Procés fill 5	3536	3531	" "	Z

Taula-2: Exercici-2, creant processos en ventall

Pregunta-1:Justifica perquè estan en eixe estat els processos fill generats per *range_process.c*

Perque el proces pare no espera la finalitzacio dels fills portant els fills queden zombies (Z) i el pare terminado (T)

5. El procés init()

En els sistemes Unix, *init*(*initialization*) és el primer procés en execució rere la càrrega del nucli del sistema i el responsable d'establir l'estructura de processos. Init s'executa com a dimoni (daemon) i normalment té el seu PID igual a 1. Tots els processos en el sistema, excepte el procés *swapper* , descendeixen del procés *init*.

A l'iniciar un sistema UNIX, es realitza una seqüència de passos coneguts per *boot strap* l'objectiu dels quals és carregar una còpia del sistema operatiu en memòria principal i iniciar la seua execució. Una vegada carregat el nucli, es transfereix el control a l'adreça d`inici del nucli, i comença a executar-se. El nucli inicialitza les seues estructures de dades internes, munta el sistema d'arxius principal i crea el procés 0 denominat *swapper* .
El procés 0 crea amb una crida fork() el procés 1 denominat init. El procés *init*, llig l'arxiu /etc/ttytab i crea un procés login per cada terminal connectat al sistema. Després de que un usuari introdueixi un login i un pasword (arxiu /etc/passwd) correcte, el procés login llança un intèrpret d'ordres o Shell.
A partir d'aquest moment el responsable d'atendre a l'usuari en eixe terminal és el intèrpret Shell, mentre que el procés login es queda dormit fins que es realitza un logout.
Nota: Els processos dimonis (daemon) fan funcions del sistema però s'executen en mode usuari.

Exercici 3: Procesos adoptats per INIT "adopted_process.c"

Comprovarem que aquells processos el procés pare dels quals ha finalitzat són adoptats per *init* el patriarca de tots els processos en un sistema UNIX. Para fer-ho hem d'assegurar-nos de que els fills estan el sistema després de que el seu pare haja acabat.
Copia l'arxiu *range_process.c* en *adopted_process.c* utilizant l'ordre cp:

```
$cp range_process.c adopted_process.c
```

Edita l'arxiu *adopted_process.c* i añada un `sleep(20)` antes del `exit(i)` de cada procés fill. Asegúresede que els fills estan suspesos en `sleep()` durant més temps que el pare. Compila *adopted_process.c* i executa-ho en background, a continuació executa"`$ps l`" diverses vegades seguides fins que finalitzen els processos. Fixat en el PPID dels processos fills a l'inici i al final i anota-ho en la tabla-3.

	PID	PPID		COMMAND	ESTAT
Procés Pare	4102	3028		ejecutable	T
Procés fill 1	4101	4102	1658	"	Z
Procés fill 2	4102	4100	1658	"	Z
Procés fill 3	4103	4100	1658	"	Z
Procés fill 4	4104	4100	1658	"	Z
Procés fill 5	4105	4100	1658	"	Z

Taula-3: Exercici-3, processos adoptats

Pregunta 2: Dels processos creats a l'executar *adopted_process* ¿Quin procés finalitza en primer lloc i com afecta aquesta finalització a la resta de processos?

Primer finalitza el pare i ja que el PPID dels fill canvie ya que el pare ya no existeix, per lo tan el PPID dels fills deuria ser el 1 que es el PID del process init

6. Espera de processos amb wait()

Cal que evitar que es generen processos zombis ja que sobrecarreguen el sistema. Un procés passa a l'estat zombi quan ell invoca la crida `exit()` i el seu pare encara no ha executat la crida `wait()`. Per a evitar processos zombis, un procés pare ha d'esperar sempre als seus fills invocant `wait()` o `waitpid()`.
La crida wait() es bloquejant ja que suspèn l'execució del procés que la invoca, fins que finalitza algun dels seus fills. En l'annex es descriu amb detall aquesta crida.

Exercici 4: Pare ha d'esperar "parent_wait.c"

En aquest exercici modificarem el programa *range_process.c* de manera que no genere processos zombis a l'executar-ho. Copia l'arxiu *range_process.c* en un altre denomina *parent_wait.c*

```
$cp range_process.c parent_wait.c
```

Edita *parent_wait.c* afegeix les instruccions i crides necessàries per a que el procés pare espere als seus fills. El procés pare ha d'esperar amb `wait()` a tots els fills.
Recorda que has de compilar i executar

```
$ gcc parent_wait.c –o parent_wait
$./parent_wait&
$ps -la
```

Pregunta3: ¿Es generen processos zombis a l'executar parent_wait? Justifica la teua resposta.

No, perque aquest proces pare no finalitza la seua execucio fins que no acaben els fills.

Exercici 5: Comunicant l'estat de finalització al pare "final_state.c"

La crida exit() permet que el fill li passe informació sobre el seu estat de finalització al pare mitjançant el paràmetre stat_loc de la crida wait(int *stat_loc). Copia *parent_wait.c* en l'arxiu *final_state.c*

```
$cp parent_wait.c final_state.c
```

Modifica el codi de *final_state.c* per a que genere la seqüència de processos de la figura-6, ésa a dir:

- Utilitzant un bucle for i una variable i, crea 4 processos que mostren un missatge indicant la iteració i en la qual es creen.
- Tots els processos han d'esperar 10 segons abans d'invocar exit().
- Tots els pares han d'esperar la finalització dels seus fills amb wait (&stat_loc), i imprimir el seu PID i el valor del paràmetre stat_loc.

¡AVISO!: El valor màxim amb el que es pot invocar la crida a exit és 255 (8 bits). Per a obtenir els 8 bits més significatius, s'utilitza WEXITSTATUS(stat_loc).

```
/*** final_state **/
int  final_state;
pid_t  val_return;
……
for (i=0; i<MAX_PROC; i++)
{val_return= fork();
  if (val_return==0)
   {printf("Son %ld created in iteration %d\n",(long) getpid(),i);
    }
  else
  { /*father out of the loop for */
   printf("Father %ld, iteration %d\n", (long)getpid(),i);
   printf("I have created a son %ld\n",(long) val_return);
   break;
  }
}

/*wait all my sons and make exit*/
While(wait(&final_state)>0)
{printf("Father %ld  iteration %d",(long) getpid(),i);
 printf( "My son said %d\n",WEXITSTATUS(final_state));
 printf( "My son said %d\n", final_state/256);
}
exit(i);
….
```

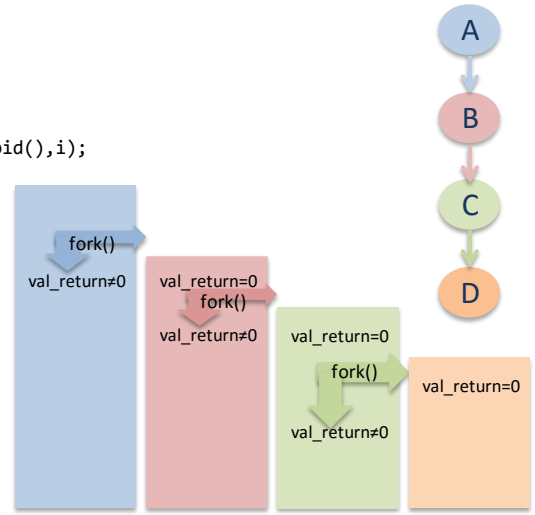


Figura-6: Esquema de processos en vertical.

Compila *final_state.c* i executa-ho. Anotar en la taula els resultats:

	PID	Valor de finalització
Procés Pare	6570	47
Procés creat amb i=0	6571	47
Procés creat amb i=1	6572	47
Procés creat amb i=2	6573	47
Procés creat amb i=3	6574	47

Taula-4: Exercici-5

Pregunta 4: ¿quina és la primera instrucció que executen els processos fills al ser creats?¿Quantes iteracions del bucle realitza cada procés?

If(res == 0)
Cada proces fa 1 iteració

7. La crida exec()

La crida exec() permet canviar la imatge de memòria o el mapa de memòria del procés que la invoca.
En el annex d'aquesta pràctica es troben detallades les 6 variants que existeixen d'aquesta crida.
La figura-7 il·lustra l'ús de la crida fork i exec per a crear un nou procés que execute el comando "ls".

```
#include<stdio.h>
#include<stdlib.h>


#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
intmain(intargc, char *argv[])
```

Variantes:
```
execvp(arguments[0],arguments);
execvp(argv[1], &argv[1]);
```

```
{pid_tchildpid;
intstatus, x;
char* arguments [] = { "ls", "-R", "/", 0 };

chilpid=fork();
 if ( childpid == -1)
     { printf("fork failed\ n");
exit(1);
     } else if (childpid ==0)
{  if(execl("/bin/ls", "ls", "-1", NULL) <0 ) {
     printf("Could not execute: ls\n");
exit(1);}
 }
x=wait(&status);
if ( x!= childpid)
printf("Child has been interrupted by a signal\n");

exit(0);
}
```

Figura-7: Codi per a crear un nou procés que execute el "ls"

Exercici 6: Creant un procés per a executar el "ls"

Sense executar el codi de la figura-7, indica quin procés/s pare, fill o ambdós invoquen els diferents fprintf i si lo mostren per pantalla sempre que s'executa el codi o només quan es compleixen certes condiciones.

	Pare ¿sempre? ¿Quina condició?	Fill ¿sempre? ¿Quina condició?
"fork failed\n"	Si: falla el fork() i no crea fill	No
"Could not execute: ls\n"	No	Si: falla el execl
"Child has been interrupted by a signal\n"	Si: el fill falla	No

Taula-5: Exercici-6

Exercici 7: Intercambio de memòria d'un procés "change_memory1.c"

El arxiu suministrat *change_memory1.c* conté el codi de la figura-7, on el fill executa l'ordre:

```
$ ls -R /
```

Mentre s'executa *change_memory1.c*, obri una altra consola i executa un comando "$ps -l" i anota en la tabla-6 els resultats associats a ell.

	PID	PPID	COMMAND
Procés Pare	6220	6128	executable
Procés fill	6221	6220	ls

Taula-6: Exercici-7

Pregunta 5: ¿Quina és la principal diferència entre els resultats d'aquesta taula i els obtinguts en l'exercici 1?

El comando no es el mateix, antes no feia canvi de memoria: ara si ya que hacian execl, además que ara el pare espera als fills.

Exercici 8: Ordre *execl()* en *"change_memory1.c"*

Practica les versions del *exec*(), modifica en*change_memory1.c*, i utilitza l'ordre execl() on s'ha d'especificar com strings els literals que formen part de la mateixa

execl("/bin/ls","ls","-R","/",NULL);

Exercici 9: Pas d'arguments *"change_memory2.c"*

Per a augmentar la funcionalitat del programa *change_memory1.c* i poder executar l'ordre *ls* amb diferents arguments que se li passaran al programa mitjançant la línia de comandos, utilitzarem els dos arguments predefinits en la funció *main (intargc,char *argv)*. Com ja coneix de la pràctica de C, el paràmetre *argc* és un sencer que conté el nombre de arguments amb que s'invoca l'ordre. Com a mínim sempre existeix un argument el argv[0] o nom del programa. El paràmetre *argv* és un vector de punters a caràcters que apunten a les cadenes dels arguments que se passen en la línia d'ordres.

A partir de *change_memory1.c* crea *change_memory2.c* per a que execute les ordres passades com a arguments, utilitze les variables *argc* i *argv*.

Comprova amb les següents execucions:

```
$ ./change_memory2 ps
$ ./change_memory2 ps -l
$ ./change_memory2 ls -lh
$ ./change_memory2 ls -lht
```

Exercici Opcional:

En la figura-8 se mostra un codi que realitza la suma de tots els elements d'una matriu. La matriu te de dimensions NUMROWS i DIMROWS. Es realitza un doble bucle, un per a obtenir la suma de totes les columnes i d'una mateixa fila i altre per a realitzar la suma resultant de cada fila.

```
#include<stdio.h>
#include<math.h>
#define DIMROW100
#define NUMROWS20

typedefstructrow
{ intvector [DIMROW];
     longadd;
} row;

matrixrow [NUMROWS];
intmain()
{
  Int i, j, k;
  Long total_add = 0 ;
  // Initializing to 1 all the elements of the vector
  for (i =0; i< NUMROWS; i ++) {
    for (j =0; j < DIMROW; j ++) {
    matrix [i] [j] .vector = 1 ;
array[i].add = 0 ;
    }
  }
```

```
  for (i =0; i< NUMROWS; i ++) {
    for (k =0; k < DIMROW; k ++) {

array[i].add += array[i][k].vector;
   }
  }
  for (i =0; i< NUMROWS; i ++)
  total_add += array[i].add;
printf ("The total addition is %ld\n", total_add);
}
```

Figura–8: Código de addrows.c per a sumar les files d'una matriu

Descarregue el codi de *addrows.c* del poliformat (), compila-ho i executa-ho. Per a la seua execució utilitze l'ordre $*time* que ens permet esbrinar el temps que tarda en executar-se un programa. Per a fer-ho invoque en el shell

```
$time ./addrows
```

Anota el temps retornat pel comando i el resultat de total de la suma.

Es demana:
Crea un programa anomentat addrows*pro.c* en el que se creen tants processos com files, cada procés s'encarregarà de la suma d'una fila (calcular `matrix[i].add`). Cada procés fill finalitzarà amb un `exit(matrix[i].add)`. El procés pare ha d'esperar a tots els processos fills per a acumular tots els valors en la variable `total_add`, que finalment imprimirà per pantalla.

Execute el programa amb l'ordre *$time*. El valor de la suma total ha de ser el mateix que en la versió sense processos fill. Anota el valor de temps que retorna i compara-ho amb la versió anterior

NOTA: El mecanisme de retorn mitjançant la crida exit() està orientat per a que els fills comuniquen al seu pare el seu estat codificable en 8 bits i no el valor resultant d'un càlcul. Per tant, l'ús de exit() que es fa en l'exercici anterior recau fora del que seria la guia de programació, utilitzant-se en aquest exemple només per a suplir la carència d'altres mecanismes de comunicació entre processos.

8. Annex

8.1 Sintaxi crides al sistema

Una crida al sistema s'invoca mitjançant una funció que sempre retorna un valor amb la informació al voltant del servei proporcionat o de l'error que s'ha produit en la seua execució. Aquest retorn pot ser ignorat, però es recomanable testejar-lo.

Valors de retorn de les crides:

- Retornen un valor de -1 o punter a NULL quan es produeix un error en l'execució de la crida
- Existeix una variable externa *errno,* que indica un codi sobre el tipus d'error que s'ha produit. En l'arxiu

de *errno.h*(#include<errno.h>) hi ha declaracions referents a aquesta variable.

- La variable *errno* no canvia de valor si una crida al sistema retorna amb èxit, per tant es important prendre el valor just després de la crida al sistema i únicament quan aquestes retornen error, és a dir, quan la crida retorna -1.

8.2 Crida fork()

```
#include <stdio>
pid_t    fork(void)
```

- Descripció:
  - Crea un procés fill que és un "clon" del pare: hereta gran part dels seus atributs.
  - Atributs heretables: tots excepte PID, PPID, senyals pendents, temps/comptabilitat.
- Valor de retorn:
  - 0 al fill
  - PID del fill al pare
  - -1 al pare si error.
- Error

  Se produeix error quan hi ha mancança de recursos per a crear el procés

8.3 Crida exec()

Existeixen 6 variants de la crida exec(), que se diferencien en la forma en que són passats els arguments (l=llista o v=array) i l'entorn (e), i si es necessari proporciona la ruta d'accés i el nom de l'arxiu executable.

```
#include <unistd.h>
void  execl (const char *path, const char *arg0, ... , const char *argn, (char*) 0 )
void  execle(const char *path, const char *arg0, ... ,const char *argn, (char*) 0, char *constenvp[])
void  execlp(const char *file, const char *arg0, ... , const char *argn, (char*) 0)

void  execv (const char *path, char *constargv[])
void  execve(const char *path, char *constargv[], char *constenvp[])
 void  execvp(const char *file,  char *constargv[])
```

Les crides execl (execl, execlp, execle) passen els arguments en una llista i són útils si se coneixen els arguments en el moment de la compilació.

Exemples de execl: execl("/usr/bin/ls","ls","-l",NULL); execlp("ls","ls","-l",NULL);

Les crides execv (execv, execve, execvp) passen els arguments en un array.

Exemple de execv: execvp(argv[1], &argv[1]);

- Descripció de exec():
  - Canvia la imatge de memòria d'un procés per la definida en un fitxer executable.
  - El fitxer executable es pot expressar donant el seu nom file o la seua ruta completa path.
  - Alguns atributs del procés se conserven i, en particular:
    - Maneig de senyals, excepte els senyals capturades l'acció de les quals serà la que hi ha per defecte.
    - PID i PPID, donat que no se crea un procés nou només canvia la seua imatge
    - Temps de CPU(comptabilitat)
    - Descriptors d'arxius
    - Directori de treball, el directori arrel, la màscara del mode de creació de arxius,
  - Si bit SETUID del arxiu executable està activat, exec fica com UID efectiu del procés al UID del propietari de l'arxiu executable.

        - Ídem amb el bit SETGID
- Errors:
    - Fitxer no existent o no executable
    - Permisos
    - Arguments incorrectes
    - Memòria o recursos insuficients
- Valor de retorn:
        - –1 si exec() retorna al programa que va fer la crida això indica que ha ocorregut un error.

- Descripció de paràmetres:
    - path: Nom del arxiu executable s'ha d'indicar la seua trajectòria. Exemple: "/bin/cp" .
    - file: Nom del arxiu executable, s'utilitza la variable d'entorn PATH per a localitzar-lo.
    - arg0: Primer argument, correspon al nom del programa sense la trajectòria. Exemple: "cp"
    - arg1 ...argN: Conjunt de paràmetres que rep el programa per a la seua execució.
    - argv: Matriu de punters a cadenes de caràcters. Aquestes cadenes de caràcters constitueixen la llista de arguments disponibles per al nou programa. El últim dels punters ha de ser NULL. Este array conté almenys un element en argv[0], nom del programa.
    - envp: Matriu de punters a cadenes de caràcters, constitueixen l'entorn d'execució del nou programa.

8.4 Crida exit()

```
#include <stdio.h>
void exit (int status)
```

- Descripció:
    - Termina "normalment" la execució del procés que la invoca.
    - Si no se invoca explícitament, es fa de forma implícita al finalitzar tot procés.
    - El estat de terminació *status* es transfereix al pare que executa *wait(&status)*.
    - Si el pare del procés no està executant *wait*, se transforma en un zombi.
    - Quan un procés executa *exit*, tots els seus fills són adoptats per init i el seus PPID passen a ser 1.
- Valor de retorn:
    - Cap
- Errors:
    - Cap

8.5 Crida wait()

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc)
pid_twaitpid (pid_tpid, int *stat_loc, int options)
```

- Descripció:
    - Suspèn l'execució del procés que la invoca, fins que algun dels fills (wait) o un fill en concret (waitpid) finalitze.
    - Si existeix un fill zombi, wait finalitza immediatament, sinó, es detén.
    - Quan `stat_loc` no es el punter nul, conté:
        - Si **Fill acaba amb exit**:
          MSB: status definit per exit, LSB: 0
        - Si **Fill acaba per senyal**:
          MSB: 0, LSB: nombre de senyal (bit mes alt 1: coredump)

- Valor de retorn:
  - El PID del fill que ha finalitzat excepte:
  - -1: se rep un senyal o error (no existeixen fills).
- Errors
  - El procés no té fills
- Descripció waitpid
  - Argumento `pid`
    - pid<-1 esperar qualsevol fill el grup de processos del qual siga igual al pid
    - pid = -1: esperar qualsevol fill (com wait)
    - pid=0: esperar qualsevol fill amb el mateix grup de processos
    - pid>0: esperar al fill el pid del qual és l'indicat
  - Argument `options`
    - WNOHANG: retornar immediatament si el fill no ha acabat

La finalització del procés fill amb exit (status) permet emmagatzemar l'estat de finalització en la'adreça apuntada pel paràmetre stat_loc. Estàndard POSIX defineix macros per a analitzar l'estat retornat pel procés fill:

- WIFEXITED, terminació normal.
- WIFSIGNALED, terminació per senyal no atrapada.
- WTERMSIG, terminació per senyal SIGTERM.
- WIFSTOPPED, el procés està aturat.
- WSTOPSIG, terminació per senyal SIGSTOP (que no es pot atrapar ni ignorar).