Criação de Processos
O modelo LINUX/UNIX

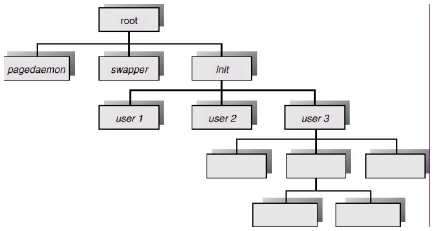
• O *processo progenitor* (*parent*/pai) cria *processos progénitos* (*child*/filhos), os quais, por sua vez, criam outros processos, formando uma árvore de processos.

• O *progénito* criado pela chamada ao sistema fork() duplica o espaço de memória do *progenitor* .

• Parent e Child procesos executam concorrentemente a partir da instrução a seguir o fork()

• A função *fork()* devolve valores diferentes para o processo *pai* e processo *filho* permitindo assim o programa pode tomar varias linhas de acção através duma instrução de controlo (if)

. O processo *pai* pode esperar a terminação do filho usando a chamada *wait()*



Win 32 api – ver "spawn" etc

1

$ man fork

NAME

fork - create a new process

SYNOPSIS

#include <sys/types.h>

#include <unistd.h>

pid_t fork(void);

DESCRIPTION

fork() causes creation of a new process. The new process (child process) is an **exact copy** of the calling process (*parent* process) **except** for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the *parent*).
- The child process has its own ***copy*** of the parent's descriptors.
- The child processes resource utilizations are set to 0;

RETURN VALUES Upon successful completion, fork()
returns a value of 0 to the child process and
returns the process ID of the child process to the parent process.

Otherwise on error, a value of -1 is returned to the parent process, no child process is created, and the global variable errno is set to indicate the error.

2

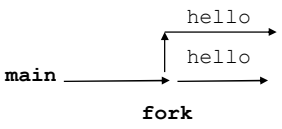
```
#include <stdio.h> #include <sys/types.h> #include <unistd.h>
main()
{
  int pid;

  pid=fork();

  if ( pid < 0 ) {  fprintf(stderr,"erro\n"); exit(1);  }

  if ( 0 ==pid )
    printf("FILHO: \t id is %d, pid (valor)is %d\n", getpid(), pid);
  else
     printf("PAI: \t id is %d, pid (filho)is %d\n",      getpid(), pid);

  /* este comando executado duas vezes..*/
   system("date");
}
```

```
alunos:~/so/eprogs/forks crocker$ ./fork1x
PAI:      id is 22571, pid (filho)  is 22572
FILHO:  id is 22572, pid (valor) is 0
Tue Mar 29 12:19:44 WEST 2005
Tue Mar 29 12:19:44 WEST 2005
```

3

Exemplo 1

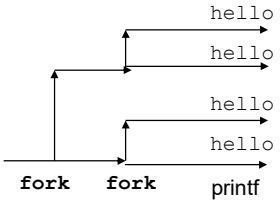
```
main()
{
  fork()
  printf("hello\n");
}
```



4

Exemplo 2

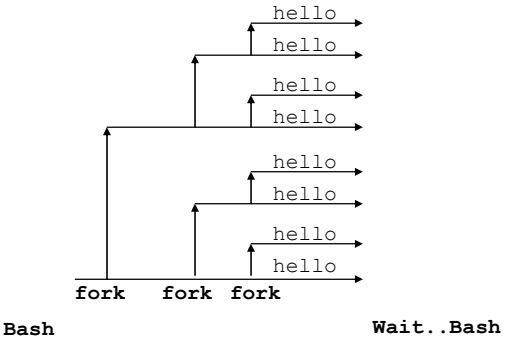
```
main()
{
  fork()
  fork()
  printf("hello\n");
}
```



5

Exemplo 3

```
main()
{
  fork()
  fork()
  fork()
  printf("hello\n");
}
```



6

Exemplo 3N

Quantos processos são criados ?

```
main()
{
   int i;
   for (i=0; i<n; i++)
            fork();
}
```

7

Exercício 4

O que será escrito no ficheiro e porque ?

```
main()
{
   FILE *fp=fopen("out.txt","w");

   fprintf (fp,"Bom Dia");

   if (0 == fork() )
   {
      fprintf( fp , " Child Process\n");
      flcose(fp);
    } else
   {
      fprintf( fp," Parent Process\n");
  }
}
```

Output bufferizado (FILE *).

É escrito para ficheiro quando ?

A limpeza da buffer do apontador para o ficheiro é feito usando

i) fflush()

ou

i) quando a buffer estiver cheia

ou

i) usando um \n (normalmente!)

8

Exercício 5

O que será escrito na ecrã e em qual ordem e porque ?

```
main()
{
   printf ("Bom Dia  ");

   if ( 0 == fork() )
      printf(" Child Process\n");
  else
      printf(" Parent Process\n");
}
```

9

Exercício 6

```
main()
{
   for (i=0;i<2;i++)
      fork()
   printf("Ola\n")
}
```

Exercício 7

```
doit()
{
   fork()
   fork()
   printf("Fim\n")
}
main()
{
    doit()
    printf("Ola\n")
}
```

10

Exercício - Exame 2007

(a) Explique a chamada ao sistema fork() ?

(b) Qual é o output do seguinte programa?

```
int main()
{
  int  pid, x = 4;
  pid = fork();
  if ( 0 == pid ) { fork(); x=x+2; }
  else            { x--; }
  printf("x=%d\n",x);
}
```

11

Substituição do Programa Velho pelo Novo

- Um processo pode carregar um novo programa usando a chamada ao sistema **exec**().
- A chamada ao execve necessita o nome dum novo programo.

```
If (0==fork())  exec("ls", ...);
```

- O texto e variáveis do programa velho são substituído pelo novo.
- O programa novo herde o identificador do processo (PID) e outras informações como p.ex acesso aos ficheiros que anteriormente tinham sido abertos.

12

exec()

```
NAME
       execl, execlp, execle, execv, execvp, execvpe - execute a file

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       int execl(const char *pathname, const char *arg, ...
                       /* (char  *) NULL */);
       int execlp(const char *file, const char *arg, ...
                       /* (char  *) NULL */);
       int execle(const char *pathname, const char *arg, ...
                       /*, (char *) NULL, char *const envp[] */);
       int execv(const char *pathname, char *const argv[]);
       int execvp(const char *file, char *const argv[]);
       int execvpe(const char *file, char *const argv[],
                       char *const envp[]);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       execvpe(): _GNU_SOURCE

DESCRIPTION
       The exec() family of functions replaces the current process image with a new process image.  The functions de-
       scribed in this manual page are layered on top of execve(2).  (See the manual page for execve(2)  for  further
       details about the replacement of the current process image.)

       The initial argument for these functions is the name of a file that is to be executed.

       The functions can be grouped based on the letters following the "exec" prefix.
```

13

Encontrando um executável

Recordar PATH

Variações do exec() -- tipo de pesquisa, formato de argumentos etc

14

Exemplo Com Exec

```
main()
{
    if (0 == fork() )
        printf(" Child Process\n")
  else
  {
        printf("Ficheiros na directoria:\n" );
        execl( "/bin/ls", "ls", "-l", NULL );
        printf(" Parent Process\n");
  }
  printf("Fim\n");
}
```

15

Exercício

```
int main (int argc, char **argv)
{
  pid_t pids[4], pid;
  int i, status;
  for (i = 0; i < 4; i++)
    {
      if ((pid = fork ()) == 0)
            execlp ("./jogos", "jogos", NULL);
      else if (pid > 0)
            pids[i] = pid;
      else
            perror(NULL);
      sleep(1);
    }
  waitpid (-1, &status, 0);
  printf("Fim\n");
  exit (0);
}
```

```
Jogos.c
main(){
   sleep(rand()%5);
   printf("jogos\n");
}
```

Versus

```
for (i=0;i<4;i++)
      waitpid (pids[i], &status, 0);
```

16

Frequência I, 14 de Abril de 2015,

1. Qual é o output do seguinte programa? (Deverá mostrar o *trace* do funcionamento do programa) **(2)**

```
{
   int  pid, x = 2;
   pid = fork();
   if (pid == 0) {
       fork();
       fork();
       x--;
   }
   else {
      execl("/bin/date", "date", NULL );
      x=x+2;
   }
   printf("x=%d\n",x);
}
```

17

Exercício (win32)

O programa "teste.exe" e "casa.exe" serão os executáveis criados a partir dos programas em baixo.

Explique as chamados a spawn() no programa "teste.c" e os (dois ?) outputs possíveis de execução do programa "teste.exe"

```
//teste.c
main(int argc, char **argv) {
  int N, pid;

  pid = spawnl( WAIT , "casa.exe", "casa.exe", NULL);
  printf("Fundao\n");

  pid = spawnl(  NOWAIT , "casa.exe", "casa.exe", NULL);
  printf("Guarda\n");
}

//casa.c
main(int argc, char **argv)  {
  printf("Covilha\n");
  getchar();
}
```

18

Terminação

- Um processo pode terminar a si próprio com uma chamada a exit()
  - Neste caso os seus filhos são herdados pelo processo init (PID=1).
- Processos progenitores podem esperar pela terminação dos seus filhos usando a chamada ao sistema wait().
  - Se um filho já tinha terminada (i.e., tornou-se 'zombie') quando wait() foi chamada então wait() retorne imediatamente.
  - Caso contrario o processo progenitor "bloqueia", esperando um sinal do OS a indicar que o filho terminou.

Nota: *Zombie – The Undead* – a situação onde um processo não pode morrer devido o facto de está a espera de enviar um sinal para o processo progenitor.

Onde é que está no diagrama de 5 estados ?

19

Terminação

exit() terminates the calling process. Before termination, all files are closed, buffered output (waiting to be output) is written, and registered "exit functions" (posted with atexit) are called. This is a library function (stdlib.h)

_exit() terminates execution without closing any files, flushing any output, or calling any exit functions. (see also abort), This is a system cal (unistd.h)

```
void exit_fn1(void){
   printf("função fn-1 do exit chamada\n");
}
int main() {
   /* post exit function #1 */
   atexit(exit_fn1);
   printf("ola ");
   exit(1);
   printf("Adeus");
}
```

```
$a.out
ola função fn-1 do exit chamada
$
```

```
void exit_fn1(void){
   printf("função fn-1 do exit chamada\n");
}
int main() {
   /* post exit function #1 */
   atexit(exit_fn1);
   printf("ola ");
   _exit(1);
   printf("Adeus");
}
```

```
$a.out
$
```

20

Comportamento típico dum Interpretador de Comandos (bash shell)

```
while (1)
{
   fprintf( stdout, "%s", prompt );        //escrever um prompt
   gets(buf) ;                             //esperar e ler input
   processarLinhaLida( buf, &args);   //processar linha
   pid = fork()                            //fork
   if (0 == pid)   execvp(*args, args);  //executar pedido num novo processo
   //progenitor
   if ( Execução em foreground )     //esperar terminação do processo filho
         while ( wait(&status) != pid )   //progenitor executa a espera
               /*ciclo infinito */  ;
}
```

21

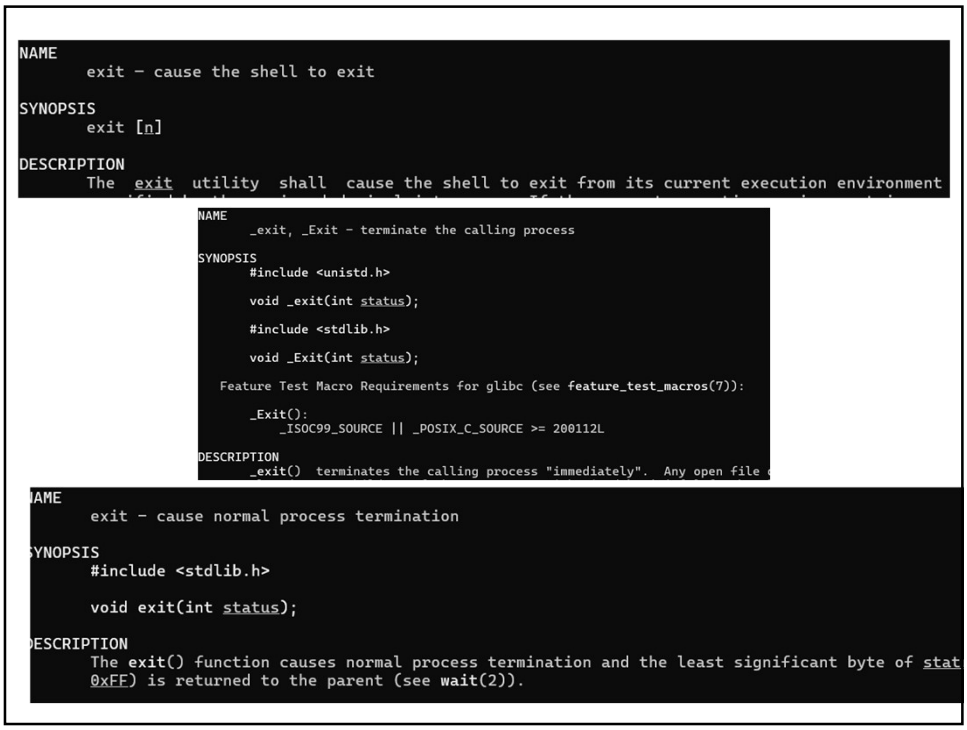
Man(ual) pages

man 1 Bash Shell and System Commands
man 2 System Calls
man 3 Standard C Libraries

Exemplos
man exit (system and shell commands)
man 2 (syscalls) exit → the system call man _exit
man 3 (std. Library) exit → exit() of stdlib.h

Ver slide seguinte

22



23

Man(ual) pages

man 1 POSIX Programmer's Manual
man 2 Linux Programmer's Manual
man 3 POSIX Programmer's Manual

Exemplos

man printf → Shell format and print data
man 2 printf .. Nada
man 3 printf → standard library formatted output conversion

man read → read from standard input into shell variables
man 2 read → read from a file descriptor - system call

24