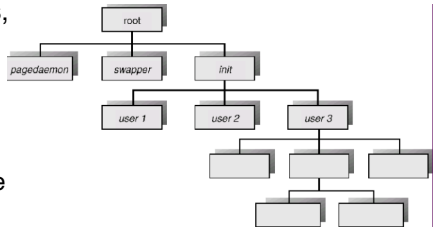
Criação de Processos – O modelo UNIX

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



• O filho criado pela chamada ao sistema fork() duplica o espaço de memória do pai.

• Pai e filho(s) 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

alunos:~ darwin-crocker$ 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 process).
- The child process has its own *copy* of the parent's descriptors.

etc.

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, 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.

```
#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/cprogs/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
```

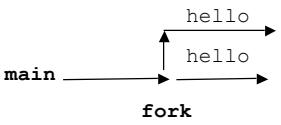
Substituição do Programa Velho pelo Novo

- O filho (ou pai) pode carregar um novo programa usando a chamada ao sistema exec().
- A chamada ao exec necessita o nome dum novo programo. 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.

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

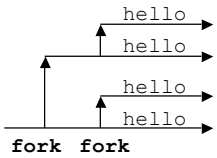
Exemplo 1

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



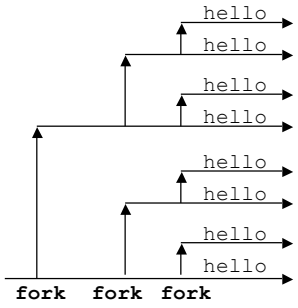
Exemplo 2

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



Exemplo 3

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




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

Exercício 1

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

Exercício 2

Exercício 3 - 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);
}
```

Exemplo 4 : com Exec

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

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 , " Filho\n");
      flcose(fp);
    } else
   {
      fprintf( fp," Pai\n");
  }
}
```

Output bufferizado escrito para ficheiro quando ?

A limpeza da buffer do apontador para o ficheiro é feito usando fflush() ou quando a buffer estiver cheia ou usando um \n (normalemente!)

Exercício 5

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

```
main()
{

   printf ("Bom Dia  ");

   if (0 == fork() )
      printf(" Filho\n")
  else
      printf(" Pai\n");
}
```

Exercício (win32)

O programa “teste.exe” e “casa.exe” são os executáveis criados a partir dos programas em baixo. Explique o programa “teste.c“ e os dois outputs possíveis de execução do programa “test.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(“Canhoso\n”);
 }

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

Terminação

• Um processo pode terminar a si próprio com uma chamada a exit() – os seus filhos são depois herdados pelo processo init (PID=1).

• Processos pais podem esperar para a terminação dos seus filhos usando a chamada ao sistema wait().

  • Se um filho já tinha terminada (i.e., tornou-se ‘zombie’) quando wait() for chamada então wait() retorne imediatamente…

  • Caso contrario o processo pai “bloqueia”, esperando um sinal do OS a indicar que o filho terminou.

Zombie – The Undead – a situação onde um processo não pode morrer devido o facto de está a espera de enviar um sinal para o prpcesso pai.

Onde é que está no diagrama de 5 estados ?

Terminação

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

_exit() terminates execution without closing any files, flushing any output, or calling any exit functions. abort() == _exit(3); Signals normally raise an _exit(x)

```
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
$
```

```
int main(void)  {
   printf(“Programa com _exit\n”);
   fflush(stdout);
   printf(“ola”);
   _exit() ;
   printf(“Adeus”);
}

$a.out
Programa com –exit
$
```