1. Identificação de processos

Um processo é a entidade lógica que suporta a execução de um programa. O SO mantém uma lista dos processos, identificando cada um deles através de um número inteiro designado por PID (*Process identifier*). A função de sistema 'getpid()' devolve o PID do processo (+ info em man 2 getpid).
Considere o programa 'mypid.c':

```
#include <stdio.h>
main()
{
 printf("PID:%d\n", getpid());
 while (1);
}
```

Coloque o programa em execução em *background* com o comando:

```
$ mypid &
```

Liste todos os processos com o comando:

```
$ ps
```

Identifique o processo 'mypid' e o respectivo PID na lista de processos. Para interromper a execução deste processo envie o sinal SIGTERM (kill -SIGTERM <pid>).

2. Hierarquia de processos

Qualquer processo em execução encontra-se hierarquicamente relacionado com outro processo que o colocou em execução. Ao digitar o comando 'mypid' na linha de comandos, o respectivo programa é colocado em execução pela *shell* (mypid é o processo filho). O processo correspondente à *shell* é o processo pai. A função 'getppid()' devolve o PID do processo pai.
Crie o seguinte programa ('myppid.c'):

```
#include <stdio.h>
main()
{
 printf("PID:%d  Parent PID:%d\n", getpid(),getppid());
 while (1);
}
```

Coloque o programa em execução em *background* com:

```
$ myppid &
```

Liste todos os processos com o comando:

```
$ ps -fH
```

Identifique o PID da *shell* e a respectiva hierarquia entre processos. Para interromper a execução do processo 'myppid' envie o sinal SIGTERM (kill -SIGTERM <pid>).

3. Invocar ou transferir?

Considere o seguinte programa ('sysexec.c'):

```
#include <stdio.h>
#include <unistd.h>
main()
{
 printf("PID:%d\n", getpid());
 printf("Invoca uma nova shell, com o programa ps\n");
 system("ps -fH");
 printf("Transfere para o programa ps\n");
 execlp("ps","ps","-fH", NULL);
 printf("Isto nunca vai ser executado");
}
```

Execute o programa e analise os resultados. A função 'system()' invoca uma nova shell que, por sua vez, coloca em execução o processo 'ps -fH'. A função 'execlp()' substitui a actual imagem do processo por uma nova imagem correspondente ao programa 'ps -fH'.

4. Criação de processos - a função fork()

A função 'fork()' permite criar um processo filho, cuja imagem é exactamente igual à imagem do processo pai (+ info em man 2 fork).
Teste os seguintes programas:
'fork1.c':

```
#include <stdio.h>
main()
{ printf("E vai um\n");
  fork();
  printf("E vão dois\n");
}
```

'fork2.c':

```
#include <stdio.h>
main()
{
  printf("Valor devolvido pelo fork: %d\n", fork());
}
```

'fork3.c':

```
#include <stdio.h>
main()
{
 int pid;
 printf("E vai um\n");
 pid=fork();
 if (pid!=0) /* processo pai*/
   printf("Sou o processo pai. Criei um processo filho\n");
 else /* processo filho*/
   printf("Sou um novo processo. Chamem-me filho\n");
 printf("E vão dois\n");
}
```

Para o processo que invoca a função fork() (processo pai), é devolvido o PID do processo filho criado. Para o processo filho criado, é devolvido o valor 0. Desta forma é possível identificar o código a ser executado por cada um dos processos, conforme se evidencia em 'fork3.c'. A ordem pela qual as mensagens são visualizadas dependem do escalonamento dos processos realizado pelo SO. Ambos os processos partilham o *stdout*.

5. Espaço de endereçamento

Na sequência da invocação da função 'fork()', o processo filho é criado com uma imagem igual à imagem do processo pai. Trata-se de uma cópia e, por conseguinte, os espaços de endereçamento dos dois processos são completamente distintos. Considere o programa seguinte ('pvar.c'):

```
#include <stdio.h>
main()
{
 int i=10;
 if (fork()!=0) /* processo pai*/
   {i=i+1;
    printf("PP:i=%d\n",i);}
  else /* processo filho*/
```

```
   {i=i+2;
   printf("PF:i=%d\n",i);}
 i++;
 printf("i=%d\n",i);
}
```

A variável i, depois do fork(), é distinta para cada um dos processos (pertencem a imagens diferentes). Teste o programa e valide esta informação.

6. Partilha do stdout e stdin

Conforme se pode verificar nos exercícios anteriores, ambos os processos partilham o *stdout*. Qual dos processos fica com o *stdin*? Teste o seguinte programa ('pfstdin.c')e tire conclusões:

```
#include <stdio.h>
main()
{
 char s[100];
 if (fork()) /* processo pai*/
   {scanf("%[^\n]",s);
    printf("PP: s=%s\nVou terminar",s);}
  else /* processo filho*/
   {scanf("%[^\n]",s);
    printf("PF: s=%s\nVou terminar",s);}
}
```

7. Sincronização entre processos na fase de terminação

Cada um dos processos pode aguardar pela terminação do outro processo hierarquicamente relacionado. Para o processo filho, basta verificar se o PID do processo pai passou a ser igual a 1. Isto acontece quando o processo pai termina, o processo filho é adoptado pelo processo 'init' (PID=1).
Teste o seguinte programa ('fwait.c').

```
#include <stdio.h>
main()
{
 if(fork()) /* pai */
  {sleep(2); /* para simular actividade */
   printf("PP: Vou terminar\n");}
 else
  {printf("PF:À espera que o pai termine\n");
   while (getppid()!=1);
   printf("PF:Já terminou, fui adoptado pelo init\n");}
}
```

A função 'sleep(2)' coloca o processo sem realizar qualquer tarefa durante um período de 2 segundos (+ info em man 3 sleep).

Para o processo criador (processo pai), este deve recorrer à função 'wait()' ou 'waitpid()' para aguardar pela terminação do processo filho (+ info em man 2 wait). Adicionalmente, é possível identificar o estado em que a terminação ocorreu ('return' ou 'exit()'). Considere o seguinte programa ('pwait1.c'):

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

```
if(fork()) /* pai */
 {printf("À espera...\n");
  wait(&status); /* bloqueio até que o filho termine */
  printf("O meu filho já terminou\n");
  printf("Terminou Status: exit:%d sinal:%d\n", status>>8, status &
0xFF);}
 else     /* filho */
 {printf("PF: PID=%d\n",getpid());
  sleep(15);/* para simular actividade */
  return 5;}/* valor a ser devolvido em status*/
}
```

O valor devolvido em status é um inteiro (16 bits). Os 8 bits + significativos representam o valor devolvido pela função 'return' ou 'exit()'. Quando a terminação ocorre devido a um envio de um sinal, o número do sinal é colocado nos 8 bits menos significativos.
Teste o programa com a seguinte sequência de comandos:

```
$ pwait1
```

Aguarde o tempo definido em 'sleep()' e verifique o valor da variável 'status' (8 bits + significativos).

```
$ pwait1 &
```

Antes de terminar o tempo definido em 'sleep()', envie o sinal SIGTERM ou SIGKILL ao processo filho (kill -SIGTERM <pid>). Verifique que o processo termina de imediato e o número do sinal é colocado na variável status (8 bits - significativos).

A função 'wait()' é bloqueante, isto é, não retorna enquanto um processo filho não terminar. Em algumas situações, isto pode ser limitador. A função waitpid() permite verificar se um processo já terminou, retornando de imediato mesmo que nenhum processo tenha terminado. Considere o programa ('pwait2.c'):

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
main()
{int pid, status;
 if(!(pid=fork())) /* filho */
   sleep(1); /* assumir que chega */
 else /* pai */
   {
     while ( waitpid(pid, &status, WNOHANG) != pid)
          printf("Ainda não terminou !!\n");
    printf("Terminou (PID=%d) (Status=%d)\n", pid, status);}
}
```

A constante WNOHANG indica o modo não bloqueante para a função 'waitpid()'. Deste modo é possível realizar outras tarefas (neste caso printf("Ainda...")), enquanto aguarda pela terminação do processo. Teste o programa e valide esta informação.

8. Estado "zombie"

Conforme foi verificado anteriormente, ao processo pai é possível verificar o estado em que um processo filho terminou. Para o efeito, e para o caso em que o processo filho termine antes do processo pai fazer 'wait()', 'waitpid()' ou terminar, o SO mantém o processo filho na lista de processos, assinalando-o com o estado <defunct>.
Considere o programa 'zombie.c':

```
main()
{
  if(fork()) /*pai*/
```

```
    while(1); /* em execução */
}
```

O processo filho termina, mas o processo pai continua em execução e não fez o 'wait()'. Teste o programa com os seguintes comandos:

```
$ zombie &
```

Liste os processos com o comando:

```
$ ps -fH
```

Verifique que a lista apresenta o processo filho em estado <defunct> ("zombie"). Este processo não pode ser eliminado da lista, mesmo enviando o sinal SIGKILL. Apenas será retirado da lista quando o processo pai terminar.

Termine o processo pai com o envio do sinal SIGKILL (kill -SIGKILL <pid>). Verifique que o processo "zombie" foi retirado da lista (ps -fH).

9. Criação de vários processos

Determine quantos processos são criados pelo seguinte programa ('nprocf1.c'):

```
#include <stdio.h>
main()
{
 fork();
 fork();
 printf("Proc. %d (filho de %d)\n", getpid(), getppid());
 sleep(1); /*admitir que chega para nenhum pai
           termine antes do filho fazer getppid() */
}
```

Como criar N processos? No programa seguinte 'nprocf2.c', procurou-se criar N=3 processos: Quantos processos são criados na realidade?

```
#include <stdio.h>
main()
{
 int i;
 for (i=1; i<=3; i++)
   fork();
 printf("Proc. PID=(%d)\n", getpid());
}
```

Teste o programa 'nprocf3.c':

```
#include <stdio.h>
main()
{
 int i; int pidpai=getpid();
 for (i=1; i<=3; i++)
   if (pidpai==getpid()) fork(); /* só o pai é que faz fork()*/
 if (pidpai!=getpid()) printf("Proc. filho PID=(%d)\n", getpid());
}
```

Assim está melhor!

O exemplo anterior cria N processos filho, todos descendentes de um único processo pai. Como criar N processos (i=0..N-1), sendo o processo i+1 descendente do processo i? Para validar esta sequência, cada processo deve imprimir o seu PID e o PID do processo pai. O programa deve receber o número N através da linha de comandos (ex. nprocd 5).

Escreva um programa que receba uma lista de ficheiros como argumentos da linha de comandos. Por cada ficheiro deve ser criado um processo filho que procede à abertura desse ficheiro para leitura. No caso de não ocorrer um erro na abertura do ficheiro, o respectivo processo deve terminar com o valor 0.

Caso contrário o processo deve terminar com o valor 1. O processo pai deve contabilizar e visualizar o número de ocorrências de erros.