Processos

- Um SO executa uma multiplicidade de programas, em batch ou time-sharing, que se designam por:

  processos ou tarefas (processes/tasks/jobs).

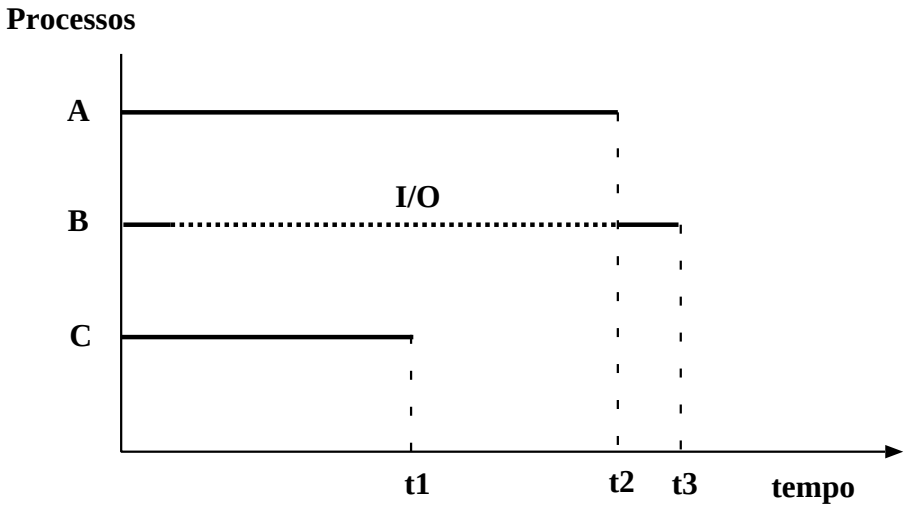
- Processo – é um programa em execução.

  A execução de um processo é sequencial, no sentido em que num dado instante, apenas uma instrução do processo é executada.

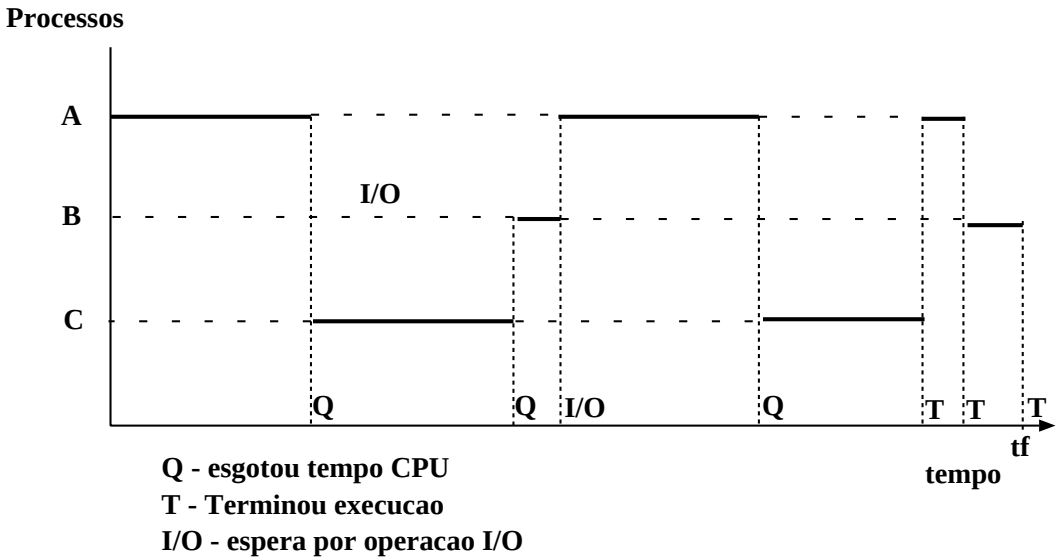
- Um processo não tem apenas associado a si, o código de um programa, inclui também:
  - program-counter – que indica a próxima instrução a executar.
  - pilha de execução (stack) – com valores temporários (parâmetros de funções, endereços de retorno, etc.)
  - região de dados – com os valores das variáveis globais.

Execução de processos

Consideremos 3 processos cuja traçagem de execução, se executados um de cada vez até terminarem, se ilustra na figura:



Suponha agora (2a. figura) que os processos são executados em time-sharing:

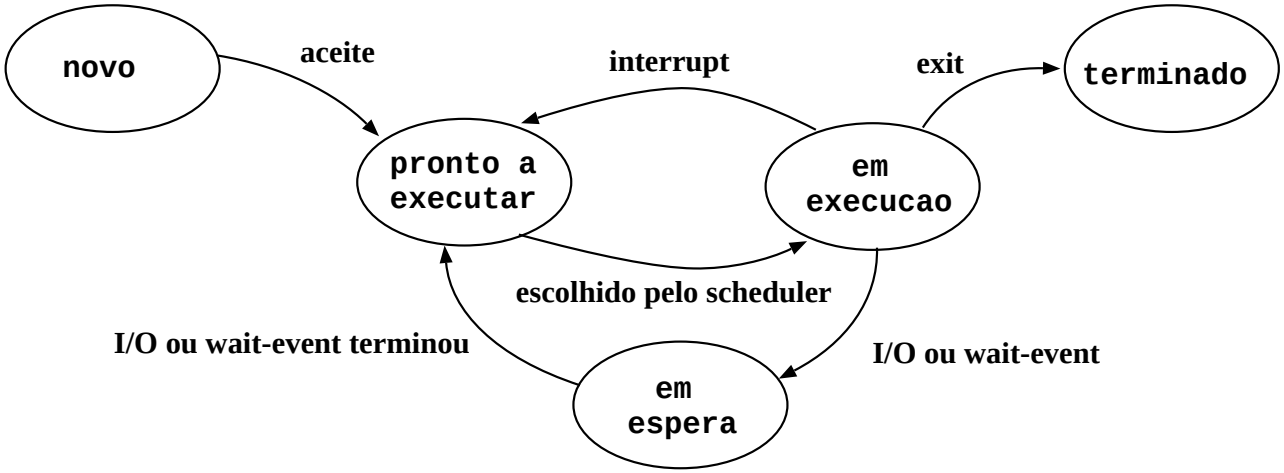


Observe que o tempo de execução, será menor quando ocorre time-sharing, i.e.:

$$tf < (t1 + t2 + t3)$$

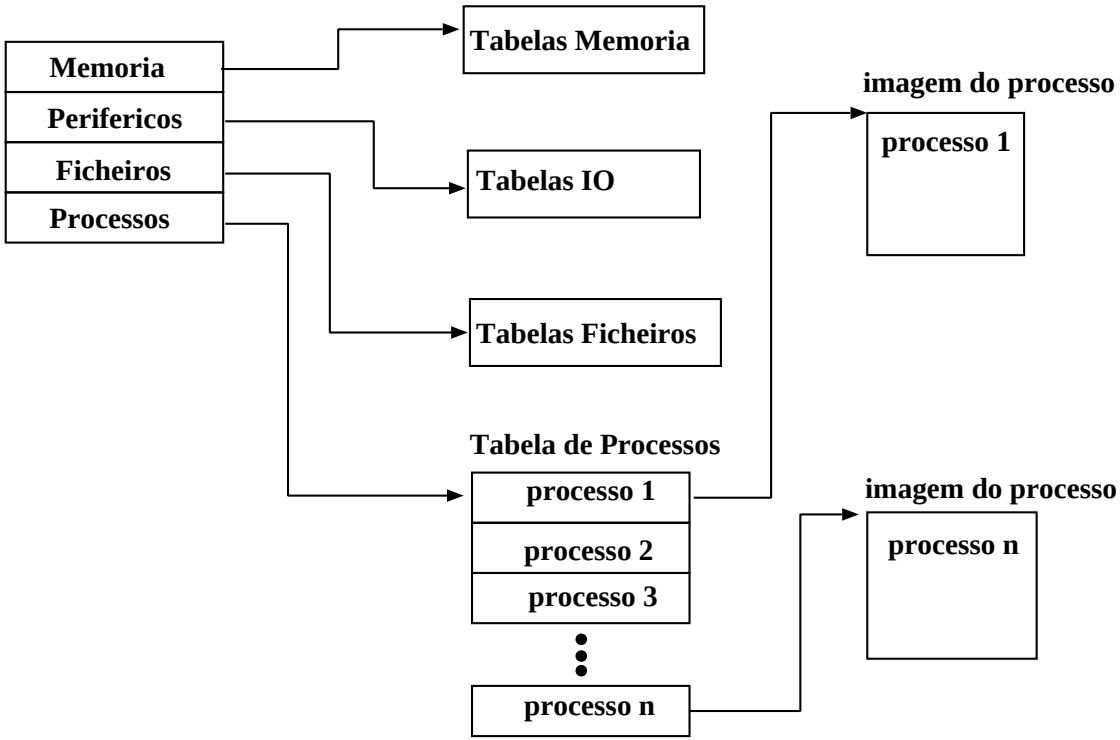
Estados de um Processo

- Um processo varia de estado durante a sua execução:
  - novo – o processo está a ser criado.
  - em execução – activo no CPU.
  - em espera – o processo está à espera de um evento externo.
  - pronto a executar – o processo está à espera de vez de CPU.
  - terminado – o processo terminou a execução.
- Transições possíveis entre os estados de um processo:


novo
aceite
interrupt
exit
terminado
pronto a executar
em execucao
escolhido pelo scheduler
I/O ou wait-event terminou
I/O ou wait-event
em espera

Estruturas de controlo do SO

O SO constroi e mantém tabelas com informação sobre cada entidade, processo e recurso do sistema, que gere.



Tabelas e atributos de processos

- $\mathbf{tabelas\ de\ memória}$ – guardam informação sobre a memória principal e secundária associada a processos, assim como atributos de protecção e informação para gerir memória virtual.
- $\mathbf{tabelas\ de\ I/O}$ – permitem que o SO saiba se um dado periférico está livre ou não, qual a operação em curso num dado periférico e qual a zona de memória associada.
- $\mathbf{tabelas\ de\ ficheiros}$ – têm informação sobre os ficheiros existentes, a sua localização na memória secundária, o seu estado corrente e outros atributos.
- $\mathbf{tabelas\ de\ processos}$ – permitem que o SO saiba localizar os processos correntes e quais os atributos que lhe estão associados.

Organização de um processo

- Associado a um processo (a sua imagem) temos:
  - um espaço de endereçamento que determina a execução do processo (memória onde estão partes do código executável e dados usados pelo processo).
  - um contexto do processo que define o ambiente de execução e determina o estado do processo. Estes atributos são normalmente referidos por PCB (Process Control Block).

Informação típica num PCB

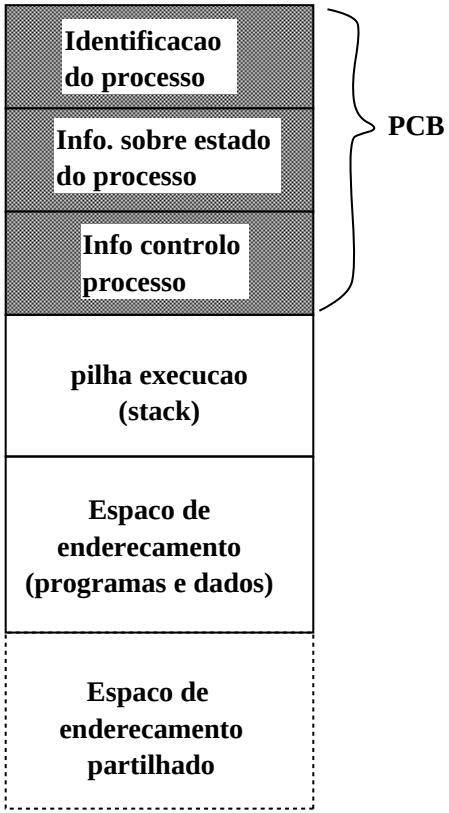
- identificação do processo:
  - *identificador do processo* (PID)
  - *identificador do processo que criou este* (processo-pai – PPID)
  - *identificador do utilizador dono do processo* (UID).
- estado do processo:
  - *program counter*: próxima instrução a executar.
  - *registos do cpu:* acessíveis ao utilizador.
  - *status info:* inclui flags sobre interrupts (activos/inactivos), modo de execução (kernel/utilizador).
  - *códigos de condição*: resultados de operações lógicas e aritméticas (overflow...).
  - *apontador para a stack*. A stack é usada para guardar parâmetros quando da chamada de funções.

PCB (continuação)

- **informação de controlo:**
  - *informação de scheduling*: estado do processo (a executar, pronto a executar, em espera, etc.), prioridade do processo.
  - *privilégios do processo*
  - *info. para gestão de memória*: páginas e segmentos de memória associados ao processo.
  - *recursos associados*: ficheiros abertos, etc.

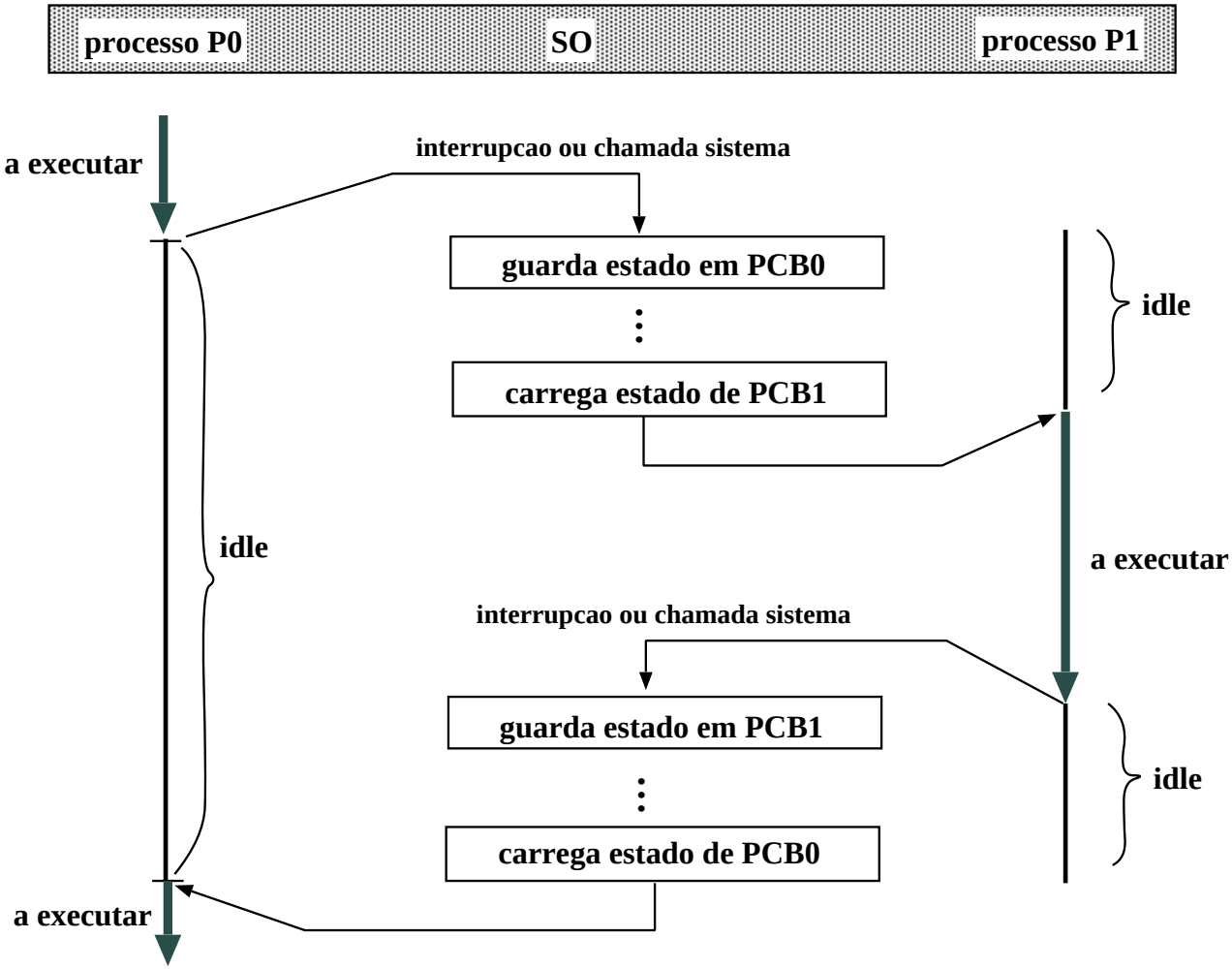
Imagem do Processo

A estrutura da imagem de um processo na memória (virtual) está ilustrada na figura:


Identificacao do processo
Info. sobre estado do processo
Info controlo processo
PCB
pilha execucao (stack)
Espaco de enderecamento (programas e dados)
Espaco de enderecamento partilhado

Troca de contexto (processos)

- Sempre que o CPU comuta de um processo para outro tem de guardar o estado do processo que sai (para poder ser retomado posteriormente) e carregar o estado do processo que entra.
- O estado do processo é guardado no PCB do processo.
- A troca de contexto é um *overhead*, i.e. o sistema não está a fazer nada de útil enquanto processa a troca.
- Ilustração do que ocorre numa troca de contexto:


processo P0
SO
processo P1
a executar
interrupcao ou chamada sistema
guarda estado em PCB0
carrega estado de PCB1
idle
idle
a executar
interrupcao ou chamada sistema
guarda estado em PCB1
carrega estado de PCB0
idle
a executar

Criação de processos

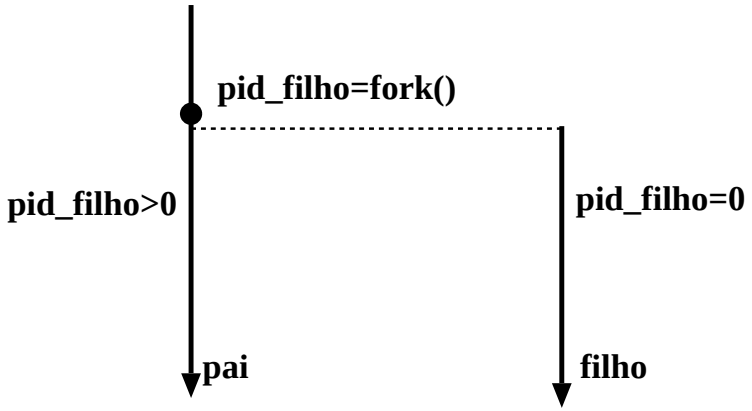
- Um processo pode criar novos processos, *processos-filho*, que por sua vez podem criar novos processos.
- recursos: o processo-filho é uma cópia da imagem do processo-pai e pode partilhar os recursos deste.
- execução:
  - Pai e filho executam concorrentemente, ou
  - Pai espera que os filhos terminem.
- espaço de endereçamento:
  - Filho é uma cópia do pai, executa o mesmo programa, ou
  - Filho pode executar um programa diferente.
- em UNIX:
  - `fork` – cria um novo processo.
  - `execve` – usado a seguir ao fork para substituir no espaço de memória do processo o programa a executar.

Criação de processos em UNIX

Faz-se através da função de sistema:

```
pid_t fork(void);
```

• fork retorna 0 ao processo-filho e retorna o PID do filho ao processo-pai. Caso não seja possível criar o novo processo, retorna -1.



Como distinguir a execução do processo-filho da do processo-pai?

```
if ( (pid_filho=fork())==0) {
  <código-processo-filho>
}
else {
  <código-processo-pai>
}
<código possivelmente comum>
```

Diferenças entre pai e filho:

- diferentes PID e PPID
- filho não herda locks de ficheiros que pertencem ao pai.

Exemplo do uso de `fork()`

```
#include <stdio.h>
main()
{
  int i;

  if (fork() == 0)   /* filho */
    for (i=0; i<5; i++) {
      printf("Filho: %d\n'', i);
      sleep(2);
    }
  else               /* pai */
    for (i=0; i<5; i++) {
      printf("Pai: %d\n'', i);
      sleep(3);
    }
}
```

```
 execução:
-----------
  Pai: 0
  Filho: 0
  Filho: 1
  Pai: 1
  Filho: 2
  Pai: 2
  Filho: 3
  Filho: 4
  Pai: 3
  Pai: 4
```

→ o processo-filho executa o mesmo programa que o processo-pai, mas partes diferentes!

→ pai e filho executam concorrentemente.

Novos processos para execução de programas

Processo-filho pode executar um programa diferente daquele que o criou.

As funções de sistema:

```
int execl(char *path, char *com, char *arg,...);
int execlp(char *file, char *com, char *arg,...);

int execv(char *path, char *argv[]);
int execvp(char *file, char *argv[]);
```

diferem na passagem dos argumentos para o programa a executar.

→ Estas funções permitem substituir o programa que faz a chamada por um outro programa executável.

→ Modificam o segmento de dados e texto do processo, mas não alteram o seu contexto. Isto é, mantêm os mesmos identificadores de recursos e ficheiros abertos que o processo possuía antes da execução do `exec()`.

Exemplo: fork() + exec()

```
main() // (Programa que cria um processo para executar ls -l)
{
  int filhoID, estado;
  char path[]=‘‘/bin/ls’’;
  char cmd[]=‘‘ls’’;
  char arg[]=‘‘-l’’;

  if ( (filhoID= fork()) == -1) {
    printf(‘‘Erro no fork’’);
    exit(1);
  }
  else if (filhoID==0) {
    if (execl(path, cmd, arg,NULL)<0) {
      printf(‘‘O exec falhou’’);
      exit(1);
    }
  }
  else if (filhoID != wait(&estado))
    printf(‘‘sinal antes do filho terminar’’);
  exit(0);
}
```

Identificação do processo

Os processos são identificados através de um inteiro único, atribuído na criação do processo.

- Identificador de um processo (PID), obtido por:

```
pid_t getpid(void);
```

- Identificador do processo-pai:

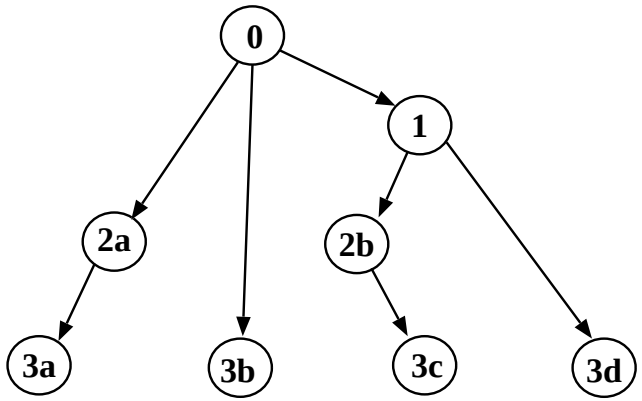
```
pid_t   getppid(void);
```

- Cada processo tem um “dono” (owner), que tem privilégios sobre o processo. Pode ser determinado por:

```
uid_t   getuid(void);
```

fornece o UID efectivo que permite determinar os privilégios do processo no acesso a recursos.

Como criar uma árvore de processos?




```
main()
{
  int i, n=4;
  ...

  for (i= 1; i<n; i++)
    if ((filhoID=fork()) == 0)
      break;
  fprintf(stderr,‘‘proc(\%ld) com pai(\%ld)\n’’,
          (long)getpid(), (long)getppid());
}
```

Funções: `exit()` e `wait()`

- **Terminar a execução de um processo:**

```
void exit(int estado)
```

retorna o controlo ao processo-pai, indicando-lhe através da variável `estado` como é que o processo-filho terminou:

`estado == 0` – terminou normal.
`estado` $\neq$ `0` – terminou com erro (valor é o código do erro).

- **Esperar que um processo-filho termine:**

```
pid_t wait(int *estado)
```

o processo que executa a função, suspende à espera que um dos processos-filho termine, ou até receber ele mesmo um sinal para terminar.

A função `wait()` retorna o identificador do processo que terminou e a forma como terminou (através da variável `estado`).

- **Esperar que um processo-filho específico termine:**

```
pid_t waitpid(pid_t pid, int *estado, int opções)
```

o processo suspende à espera que o processo `pid` termine. Se o valor de `pid` for `-1`, esta função equivale à função `wait()`.

Exemplo com `exit()` e `wait()`

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

main()
{
  pid_t filhoID;
  int estado;

  if ((filhoID= fork()) == -1) {
     printf(``O fork falhou\n'');
     exit(1);
  }
  if (filhoID==0)
    printf(``Sou o filho com pid=%ld\n'',(long)getpid());
  else if (wait(&estado) != filhoID)
    printf(``Um sinal interrompeu o wait()\n'');
  else
    printf(``Sou o pai com pid=%ld\n'',(long)getpid());
  exit(0);
}
```

Mais um exemplo sobre criação de processos: cadeia de N processos

Escrever um programa que crie uma cadeia de N processos tal que o 1º processo da cadeia cria o 2º, o 2º cria o 3º, etc. Cada processo deve escrever o seu PID e o PID do processo-pai.

Mais um exemplo sobre criação de processos: cadeia de N processos

```
#define is_child(P)  (P == 0)

void cadeia_procs()
{
  pid_t pai, new_proc;
  int i;

  for (i=1; i<=N; i++) {

    pai = getpid();
    new_proc = fork();

    if (is_child(new_proc))
      printf("FILHO %d: PID = %d\tPPID = %d\n", i, getpid(), pai);
    }
    else
      break;
  }
  return;
}
```

Mais um exemplo de sincronização de processos

Escrever um programa que crie um processo filho para calcular o número de fibonnacci de ordem 10, enviando o resultado ao processo pai por um sinal. O processo pai deve esperar que o processo filho termine, apanhar o sinal enviado e escrevê-lo.

Mais um exemplo de sincronização de processos

ATENÇÃO: este programa não funciona se o fibonacci devolver valores maiores do que 256, por causa do tipo de WEXITSTATUS!!

```
void sinc_fib10()
{
  pid_t new_proc;
  int status;

  new_proc = fork();
  if (is_child(new_proc))
    exit(fibonacci(10));
  else {
    wait(&status);
    if (WIFEXITED(status)) {
       printf("PAI: fib(10) = %d\n",WEXITSTATUS(status));
    }
    else {
       printf("Child did not terminate normally");
    }
  }
  return;
}
```

```
 long fibonacci(int n)
{
    if (n==0) return 1;
    if (n==1) return 1;
    return fibonacci(n-1) +
          fibonacci(n-2);
}
```