Instituto Superior de Engenharia do Porto

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Sistemas de Computadores

Processos

Apontamentos para Aulas Laboratoriais

Luís Lino Ferreira Luís Nogueira Luís Pinho
Orlando Sousa Paulo Ferreira Pedro Oliveira
Filipe Pacheco Alberto Sampaio

Fevereiro de 2013

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Índice

1 Processos

1.1 Processos

Definição

- Consistem num método de descrição das actividades de um sistema operativo;
- Todo o software incluído num sistema operativo (SO) é organizado num grupo de programas executáveis. Cada um destes programas quando *activo* forma um processo juntamente com o ambiente de execução associado (fazendo a gestão dos recursos, disponibilizando recursos aos processos à medida que são necessários);
- Cada processo pode "correr" num processador diferente, mas na prática é utilizada a multi-programação (cada processador "corre" vários processos). O sistema operativo deve fazer o escalonamento de processos de forma a ser efectuada a *comutação* de processos;
- Os sistemas operativos baseados neste modelo, fornecem primitivas para a criação e destruição de processos;
- Cada processo pode criar por sua vez *processos filho* independentes, podendo o primeiro continuar a sua execução *concorrentemente* com eles, ou então bloquear até à conclusão dos segundos.

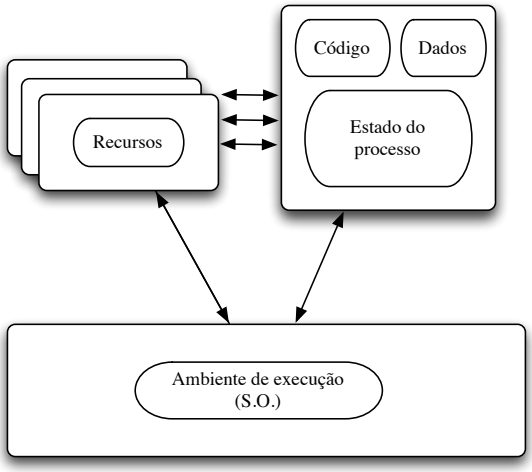

Figura 1.1: Estrutura de um processo

Estados de um processo

Desde o instante em que o processo é criado (com a função `fork`) até terminar o seu trabalho, sofre várias mudanças de *estado* (ex: quando o processo efectua uma *chamada ao sistema*, quando é a "vez" de outro processo estar activo ou quando o processo necessita de um recurso que está ocupado).

No estado **New** o processo está a ser criado, passando de seguida para o estado **Ready to Run**. Neste estado um processo espera até que o *escalonador* do SO liberte o(s) processador(es) e o coloque

em execução (estado de **Running**). Um processo é então executado por um determinado espaço de tempo, de acordo com as políticas de escalonamento do SO.

Pode deste estado passar para **Waiting**, na existência de qualquer situação de bloqueio, por exemplo quando um processo esteja à espera de um semáforo. Pode também regressar ao estado **Ready to run** por decisão do *scheduler* (o responsável pelo escalonamento dos processos). Finalmente quando termina a sua execução o processo passa para o estado de **Terminated**, caso tenha terminado correctamente. Se deixar alguma informação pendente (por exemplo, se a função *exit* tiver retornado algum valor) então o processo passa para o estado de **Zombie**. Um processo passa ao estado **Stopped** quando recebe uma indicação para suspender a sua execução, p.e. através de um sinal ou do comando *sleep*.

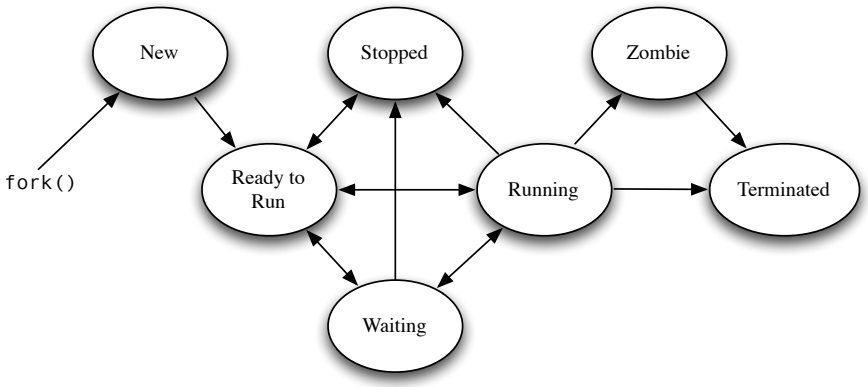

Figura 1.2: Estados de um processo

Note-se que, autores diferentes, descrevem os estados de um processo de forma diferente, e sistemas diferentes podem variar na implementação dos diferentes estados. Por isso, é necessário estar atento à forma específica como cada SO concretiza estas noções.

Contexto de um Processo

É toda a informação necessária à descrição completa do estado actual de um processo:

- PCB (Bloco de Controlo do Processo)
  - Identificação do processo, grupo, etc;
  - Informação sobre o escalonamento: estado, prioridade, etc;
  - Localização e tamanho dos contextos de memória, de dados, de pilha, ficheiros e código;
  - Registos de controlo;
  - Código;
  - Dados do programa;
  - Processos
  - Stack;
  - Descritores de ficheiros.

Através da utilização desta informação, o núcleo (*kernel*) do Sistema Operativo efectua a gestão da evolução de cada processo.

Nível de um Processo

Resulta do facto de um processo poder ou não ter privilégios para executar ou não certo tipo de operações que sejam críticas para a segurança do sistema:

- Nível 1: processos do *utilizador* – programas do utilizador que usam serviços do supervisor;
- Nível 2: processos do supervisor – executam funções como gestão de memória, ficheiros, escalonamento, etc;
- Nível 3: I/O – respondem tipicamente a interrupts;
- Nível 4: excepções – executam acções de manutenção do sistema (os erros e excepções são tratados aqui).


Note-se que nos processadores da família IA-32 (386 e sucessores) o mecanismo existente de anéis de protecção tem uma numeração inversa, sendo 0 o nível de maior privilégio, e 3 o nível com menores privilégios.

Gestão de Processos

Para efectuar a gestão de processos, são necessários processos "especiais" que executam funções como:

- Criação e eliminação de processos;
- Protecção de processos;
- Sincronização de processos;
- Comutação e escalonamento de processos.

Protecção de Processos

De modo a evitar que um processo do *utilizador* tenha acesso a um processo do supervisor, ou mesmo de outros utilizadores, é necessário garantir a integridade do sistema. Esta integridade deve basear-se em duas partes:

- *Estabilidade* – prevenção dos *estouros*;
- *Privacidade* – acesso controlado.

Sincronização de Processos

A sincronização de processos é necessária para controlar a ordem em que cada processo opera sobre os dados e recursos partilhados.

Comutação de Processos

Os processos do *utilizador* cooperam com processos de mais alto nível de prioridade, actuando concorrentemente sobre o mesmo CPU. É necessário escolher o melhor algoritmo de escalonamento para a atribuição de tempo de CPU a cada processo. A comutação deve ter em conta vários aspectos:

- Justiça: cada processo deve ter a sua "fatia" de tempo de CPU;
- Eficiência: 100% de utilização do CPU;
- Tempo de Resposta: minimizar o tempo de resposta de sistemas interactivos;
- *Throughput*: maximizar o número de processos por hora. Uma das técnicas mais usadas é a preempção (técnica de suspender processos durante um período de tempo).

1.2 Chamadas ao sistema para manuseamento de processos

Fork

Função fork – permite criar um processo em Unix.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Comportamento

- Verifica se há lugar na Tabela de Processos;
- Tenta alocar memória para o *filho*;
- Altera mapa de memória e copia para a tabela;
- Cria uma cópia exacta do processo original (código, dados, pilha, descritor de ficheiros, etc), criando uma relação "pai-filho" , mas seguindo ambos "percursos" diferentes;
- Copia para a tabela de processos a informação do processo *pai* (*pid*, prioridades, estado, etc);
- O *uid* (User Identifier) do *pai* é copiado para o *filho*;
- Ao *filho* é atribuído um novo *pid* (*Process IDentifier*);
- Informa o *Kernel* (Núcleo do Sistema Operativo) e o sistema de ficheiros que foi criado um novo processo.

Pontos a salientar

- A função *fork* é executada uma vez... mas tem **dois** retornos (um para o processo *pai* e outro para o processo *filho*);
- Geralmente o *pai* e o *filho* precisam de executar código diferente, logo o `fork` retorna 0 para o *filho* e retorna o *pid* do *filho* para o *pai*. Deste modo é possível saber-se em qualquer altura qual dos dois processos está a ser executado (isto nos casos em que se pretende executar código diferente no processo *filho* e no *pai*);
- A função retorna -1 em situações de erro;
- O processo *filho* **apenas** executa as linhas de código que se encontram depois da linha onde foi executada a função `fork`
- Todas as variáveis têm valores idênticos até à execução do `fork`, mas qualquer alteração feita em um, não se reflecte no outro, a partir dessa altura.

De uma forma mais pedagógica, podemos pensar na função `fork` como sendo de *clonagem* em vez de ser de *paternidade* (ou *maternidade*). O `fork` faz uma cópia do processo original, com o mesmo código, e os mesmos dados do processo original. De forma a distinguir-se qual é o *original* (*pai*) e qual é o *clone* (*filho*), a função devolve 0 no caso do *clone* e o Pid do *clone* para o processo *original*.

Como a execução de instruções continua a seguir à função `fork`, e o *clone* tem os dados replicados do processo *original*, para todos os fins práticos, para o *clone* ele "executou" o código anterior ao `fork`, porque os dados têm todas as alterações efectuadas neles, pelo código do processo *original*, porque a replicação dos dados apenas aconteceu na função `fork`.

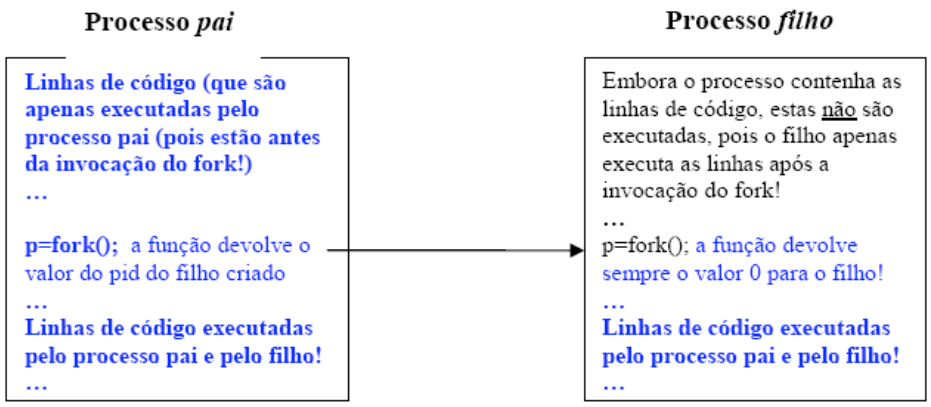

Figura 1.3: Código executado pelos processos pai e filho (a azul o que cada processo executa)


A replicação dos dados e das estruturas associadas ao novo processo faz que o uso da função `fork` seja mesmo muito pesado do ponto de vista de tempo de execução, e de recursos dos sistema. Devemos antes de optar pelo uso de um mais um processo, ponderar bem todas as possíveis alternativas, porque uma das formas mais fáceis de *aninhar* uma máquina é criar um grande número de processos.

Se quisermos impor um limite ao número máximo de processos que um utilizador pode criar, para prevenir eventuais acidentes, temos ao nosso dispor o comando `ulimit`.

Uma dúvida que poderá assaltar o leitor é a da utilidade da função `fork`, quando esta apenas faz *cópias* de processos, que são iguais ao *original*, quando o interessante seria criar processos realmente *diferentes*. Esta dúvida será dissipada quando mais à frente virmos a função `exec`, uma vez que na prática as duas se usam quase sempre em conjunto.

prdemo01.c – Exemplo de utilização da função `fork` e suas particularidades.

```
/* prdemo01.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{
    pid_t pid;
        int a;
        a = 1;
        pid = fork(); /* Cria um PROCESSO */
        if (pid < 0)
        {
            perror("Erro ao criar o processo:");
            exit(-1);
        }
        if (pid >0) /* Código do PAI */
        {
            printf("Pai: a=%d\n", a);
            a = a + 1;
            printf("Pai:a+1=%d\n", a);
        }
        else /* Código do FILHO */
        {
            /* O valor actual da variável a é 1, pois o
            filho herda esse valor do processo pai */
             printf("Filho:a=%d\n", a);
             a = a + 1000;
            /* O valor da variável a será 1001 no filho e 2
            no pai. Não esquecer que o que é alterado
            depois do fork só se reflecte no processo onde
            é executado! */
            printf("Filho:a+1000=%d\n", a);
        }
        printf("SistComp\n");
        exit(0);
}
```

Este exemplo permite verificar o funcionamento da função `fork`, isto é, o código que será executado por cada um dos processos assim como o valor das variáveis em cada processo. É necessário ter em atenção que a ordem pela qual serão apresentados os resultados é "aleatória", pois os dois processos estão a executar concorrentemente, podendo ocorrer a situação em que a ordem da apresentação do resultado seja diferente quando se executa o programa várias vezes.

A relação entre processo *pai* e *filhos* pode ser representada através de um árvore de processos. Neste tipo de árvores, os processos são representados por linhas verticais, caso necessário a execução de uma determinada linha de código também pode ser representada na mesma linha através de uma pequena linha horizontal. A execução de um *fork*, e a criação de um processo filho é representada por uma derivação à linha principal. A finalização de um processo é representada por uma circunferência a cheio. A comunicação entre os processos (troca de sinais, acessos à memória partilhada, etc) é representada através de setas. A figura 1.4, mostra a árvore (detalhada) de processos para o programa anterior.

Importa salientar que as linhas a azul são executadas no *pai* e no *filho*, pois como estão depois da invocação da função `fork` e não se encontram em nenhuma condição, são executadas em ambos os processos.

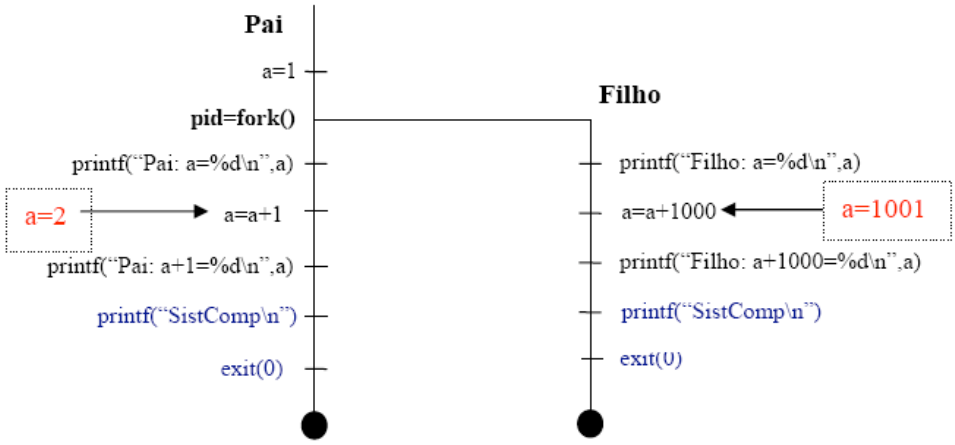

Figura 1.4: Árvore de processos do programa anterior

Exit

Função exit – Termina o processo actual.

Um processo pode ser terminado através da função `exit` ou através de uma chamada à função `return` (apenas se efectuado na função `main`). Seguidamente o *kernel* fecha todos os descritores abertos, liberta a memória usada pelo processo e guarda informação "mínima" sobre o estado de saída do processo filho: PID, estado de finalização e tempo de CPU gasto. Esta informação poderá ser obtida posteriormente pelo pai através das funções `wait` e `waitpid`[1].

Quando um processo termina, mas o seu pai ainda não foi buscar os seus dados de retorno, esse processo passa ao estado de **Zombie** até que o seu pai chame a função `wait` ou `waitpid`.

Um processo também pode terminar anormalmente, através de uma chamada à função de `abort` ou devido a ter recebido certo tipo de sinais (para mais detalhes quanto a este assunto consultar a bibliografia aconselhada).

A sintaxe da função `exit` é a seguinte:

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

A variável `status` é um inteiro e permite retornar **apenas** os seus 8 bits menos significativos para o pai. Na secção seguinte será descrito como é que o processo pai pode obter o valor de *status*, originando assim uma eliminação definitiva na tabela de processos.

Se no instante em que o *filho* invoca a função `exit` o *pai* está à espera, então a entrada na tabela é limpa, e o espaço de memória é libertado (o processo termina definitivamente). Se o *pai* não está à espera, o processo fica **Zombie**, activando um bit na entrada correspondente da tabela; o *scheduler* recebe uma mensagem de modo a evitar o processo. Se o processo *pai* morre antes do filho fazer `exit`, e de modo a evitar que o processo fique **Zombie** eternamente, este é adoptado pelo processo `init` (processo com o PID correspondente a 1) , já que este está sempre a fazer `wait`.

[1] Adicionalmente o kernel do SO envia o sinal SIGCHLD ao respectivo processo pai.

Quando um processo termina, o seu *pai* é informado desse facto através do sinal `SIGCHLD`. Este evento é assíncrono, por isso o pai pode receber este sinal em qualquer altura da sua execução. Juntamente com o sinal, o SO armazena o valor de retorno do processo *filho* juntamente com outra informação relativa ao estado de saída do *filho*. O pai pode optar por ignorar o sinal (situação por omissão) ou indicar uma função que irá ser chamada quando um *filho* terminar.

Funções wait e waitpid

Ao chamar as funções `wait` ou `waitpid`, o processo invocador poderá ter um dos seguintes comportamentos:

- ficar bloqueado até que o filho termine;
- retornar imediatamente, caso o filho já tenha terminado;
- retornar imediatamente, com um erro, caso já não existam mais filhos em execução.

A função `wait` espera até que *qualquer* filho termine. A função `waitpid` espera até que um processo filho *específico* termine, além disso, esta função também permite esperar por um filho sem bloquear o processo pai. Os protótipos destas funções são apresentados a seguir.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pit_t pid, int *status, int options);
```

`status`: é um apontador para um inteiro. Se este apontador não for passado como nulo, então poderá ser usado para armazenar o estado de finalização do processo filho (no endereço que foi passaado). Note-se que este apontador deve ser inicializado antes da invocação da função.

`pid`: processo pelo qual a função `waitpid` vai esperar. Se `pid` = -1, `waitpid` espera por qualquer processo filho. Se `pid` > 0 então `waitpid` espera por um processo cujo *pid* seja igual a `pid`.

`options`: 0 de modo a que o processo fique bloqueado à espera do processo filho; pode também ser igual ao OR (ou lógico) de duas constantes: `WNOHANG`, `WUNTRACED`. A primeira constante significa que a função retorna imediatamente se nenhum filho tiver terminado e a segunda retorna também o estado dos filhos que se encontrem no estado de **Stopped** (opção pouco utilizada).

As funções `wait` e `waitpid` retornam o PID do processo que terminou em caso de sucesso e -1 em caso de erro.

A função `waitpid` retorna 0 caso nenhum filho tenha terminado e se a opção `WNOHANG` tivesse sido usada na chamada da função. Mais uma vez, em caso de erro, a variável global `errno` é actualizada podendo o seu significado ser apresentado ao utilizador através da função `perror`.

No valor de `status` vem codificada bastante informação, nomeadamente: os 8 bits menos significativos que foram passados como parâmetro à função exit (chamada pelo filho), o sinal que provocou o fim (anormal) do processo filho, se o fim do filho deu origem a um ficheiro de core, etc.

Contudo, a forma mais fácil de extrair essa informação é através de macros específicas. A macro `WIFEXITED(status)` devolve verdadeiro se o filho retornou normalmente e nessa altura pode-se chamar outra macro, `WEXITSTATUS(status)`, que vai retornar os 8 bits menos significativos que foram passados como parâmetro à função exit. A macro `WIFSIGNALED(status)` permite verificar se o processo filho terminou devido a um sinal não interceptado, podendo posteriormente ser utilizada a macro `WTERMSIG(status)`, que obtém o número do sinal que causou a morte do processo filho. Também se pode utilizar a macro `WCOREDUMP(status)` para verificar se foi gerado um ficheiro core. A macro `WIFSTOPPED(status)` determina se o processo que originou o retorno se encontra "congelado"

(**Stopped**), sendo possível utilizar a macro `WSTOPSIG(status)` para obter o número do sinal que provocou o "congelamento" do processo filho. A macro `WIFCONTINUED(status)` determina se o processo filho foi "descongelado" (**resumed**)[2].
O exemplo seguinte cria um filho que após ficar suspenso de execução durante 5 segundos, termina retornando o número 5. O pai espera sem bloquear que o filho termine. A função `sleep` usada neste programa permite suspender a execução de um processo durante x segundos.

prdemo02.c – Exemplo de utilização do `fork`, `exit` e `waitpid`.

```
/* prdemo02.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
int main(void)
{
    pid_t pid;
    int aux;
    int status;
    pid=fork();
    if (pid<0)
       {
        perror("Erro ao cria o processo\n");
        exit(-1);
        }

    if (pid > 0) /* Código do Pai */
    {
        printf("Pai\n");
        do
        {
            aux = waitpid(pid, &status, WNOHANG);
            if (aux==-1)
                {
                perror("Erro em waitpid");
                exit(-1);
                }
            if (aux == 0)
                {
                printf(".\n");
                sleep(1);
                }
        } while (aux == 0);
        if (WIFEXITED(status))
           {
            printf("Pai: o filho retornou o valor:%d\n",
            WEXITSTATUS(status));
            }
    }
```

[2] `wait(&status)` é igual a executar `waitpid(-1, &status, 0)`

```
42      else /* Código do filho */
43          {
44          printf("Filho\n");
45          sleep(5);
46
47          printf("Filho a sair\n");
48          exit(5);
49          }
50
51      exit(0);
52     }
```

A figura 1.5 apresenta a árvore de processos para o exemplo anterior. Pode-se constatar que neste caso existe a "comunicação" entre processos através da obtenção do valor de saída do processo filho (usando a função `waitpid` no pai e a função `exit` no filho).

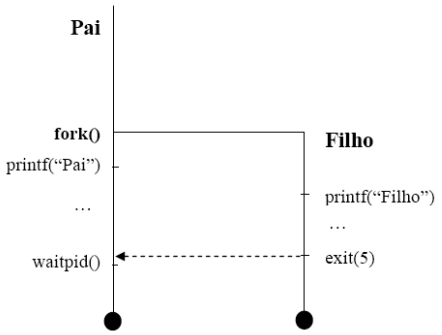

Figura 1.5: Árvore de processos do programa anterior

Funções getpid e getppid

A função `getpid` devolve o valor do pid do processo que invocou a função. Para obter o *pid* do processo pai do processo actual utiliza-se a função `getppid`.

Os protótipos das funções são:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

O exemplo seguinte ilustra a utilização destas funções assim como a possibilidade de constatar que quando o processo pai termina antes do filho, o processo filho é adoptado pelo processo `init`.

prdemo03.c – Exemplo de utilização das funções `getpid` e `getppid`.

```
1  /* prdemo03.c */
2  #include <stdio.h>
3  #include <sys/types.h>
4  #include <unistd.h>
```

```
5  int main ( void )
6  {
7      pid_t p = fork();
8      printf("p = %d pid = %d pid pai = %d\n " ,p , getpid(), getppid());
9      return 0;
10 }
```

Para uma melhor compreensão, de seguida é apresentado o que pode acontecer quando o processo pai termina depois do processo filho e quando o processo pai termina antes:

Situação 1: possível *output* da execução quando o pai termina depois do filho:

p=0 pid = 5234 pid pai = 5233

p=5234 pid = 5233 pid pai= 2359

Situação 2: possível *output* da execução quando o pai termina antes do filho invocar a função getppid():

p=5434 pid = 5433 pid pai= 2359

p=0 pid = 5434 pid pai = 1 (o filho foi "adoptado" pelo processo INIT)

1.3 Exercícios

1. Considere o seguinte programa:

```
1  /* prdemo04.c */
2  #include <stdio.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5  int main ( void )
6  {
7      pid_t p,a;
8      p = fork();
9      a = fork();
10     printf("Sistemas de Computadores\n");
11     return 0;
12 }
```

   a) Desenhe uma árvore que descreva os processos criados. Quantos processos são criados?

   b) Quantas vezes é apresentada "`Sistemas de Computadores`"? Justifique.

   c) Altere o programa de modo a que todas as possíveis situações de erro sejam tratadas.

2. Faça um programa que crie 2 processos e:

   - Escreve "Eu sou o pai" no processo pai, assim como deve apresentar o seu *pid*. O pai deve esperar que o primeiro filho termine e só depois espera pelo segundo. Também deve verificar se os filhos terminaram normalmente assim como apresentar o "valor de saída" de cada processo filho.
   - Escreve "Eu sou o 1º filho" no primeiro filho, assim como o seu *pid*. Este processo adormece durante 5 segundos e depois retorna como "valor de saída" o valor 1.
   - Escreve "Eu sou o 2º filho" no segundo filho, assim como o seu *pid*. Este processo retorna como "valor de saída" o valor 2.

Abra duas janelas: uma para executar o programa e outra para visualizar a tabela de processos. Execute o seu programa e na outra janela execute várias vezes o comando `ps` para consultar a tabela de processos (consulte `man ps`). Comente os valores obtidos, em especial as novas entradas da tabela assim como o estado dos processos criados ao longo da sua "vida".

3. Considere o seguinte programa:

```
/* prdemo05.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ( void )
{
    pid_t pid;
    int i;
    for (i=0; i<4; i++ )
    pid = fork();
    printf("SistComp\n");
    return 0;
}
```

   a) Desenhe uma árvore que descreva os processos criados. Quantos processos são criados?

   b) Quantas vezes é apresentado "`SistComp`"? Justifique.

4. Considere o seguinte programa:

```
/* prdemo06.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
int main(void)
{
    pid_t pid;
    int f;
    for (f = 0; f < 3; f++)
    {
        pid = fork(); /* Cria um PROCESSO */
        if (pid > 0) /* Código do PAI */
        {
            printf("Pai: Eu sou o PAI\n");
        }
        else /* Código do FILHO */
        {
            sleep(1);
        }
    }
    return 0;
} /* fim main */
```

   a) Analise o código deste programa e verifique quantos processos vão ser criados.

   b) Teste o programa de modo a verificar se o resultado da alínea anterior está correcto. Execute várias vezes o comando `ps` para visualizar as mudanças de estado ocorridas nos processos

criados.

c) Desenhe uma árvore que descreva os processos criados.

d) Altere o programa de modo a que apenas sejam gerados 3 filhos.

e) Altere o programa de modo a que todas as possíveis situações de erro sejam tratadas.

f) Altere o programa de modo que o pai espere pelo fim de cada um dos filhos.

g) Altere o programa de modo a que cada filho devolva o seu número de ordem ao pai. O pai deverá imprimir qual o número de ordem de cada um dos filhos que vai terminando, assim como o seu PID.

h) Altere o programa de modo a que o pai apenas espere pelo segundo filho, mas sem bloquear.

5. Considere o programa seguinte:

```
1  /* prdemo07.c */
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <sys/wait.h>
5  #include <sys/types.h>
6  int main(void)
7  {
8      fork();
9      printf("1\n");
10     fork();
11     printf("2\n");
12     fork();
13     printf("3\n");
14     return 0;
15 }
```

a) Desenhe uma árvore que descreva o conjunto dos processos criados.

b) Será possível que o número 1 apareça depois do 3? Porquê?

6. Suponha que vai programar um sistema baseado numa tecnologia que utiliza dois processadores. Tendo um vector de inteiros com 100 000 posições, crie um processo e faça o seguinte:

   - Os dois processos (pai e filho) fazem o seguinte cálculo: `resultado[i] = dados[i]*4 + 20`.
   - Cada processo é responsável pelo cálculo de 50000 posições do vector `dados`, colocando o resultado no vector `resultado`.
   - Os resultados devem ser apresentados segundo a ordem do vector (de 0 a 49999 e depois de 50000 a 99999).

   Nota: tenha em atenção que o cálculo de cada uma das partes do vector deve ser efectuado de uma forma paralela/concorrente (apenas a apresentação dos resultados deve ser sequencial).

   a) Apresente uma solução para o problema

   b) Execute o programa e noutra janela utilize o comando ps para acompanhar a evolução dos processos. Comente os resultados obtidos.

7. Assuma que vai programar um sistema baseado numa tecnologia que utiliza sete processadores. Faça um programa que crie 6 processos e:

   - Cada processo escreve 20000 números:

1º processo: 1 . . . 20000

2º processo: 20001 . . . 40000

3º processo: 40001 . . . 60000

4º processo: 60001 . . . 80000

5º processo: 80001 . . . 100000

6º processo: 100001 . . . 120000

- O processo pai tem de esperar que todos os processos filho terminem.

Nota: Todos os processos devem estar a executar em "simultâneo".

a) Apresente uma solução que cumpra os requisitos do problema.

b) O resultado aparece "ordenado"? Pode garantir que aparecerá sempre? Justifique.

c) Execute o seu programa e noutra janela execute várias vezes o comando ps. Comente os resultados.

8. Implemente um programa que obtenha a "funcionalidade" apresentada no seguinte diagrama:

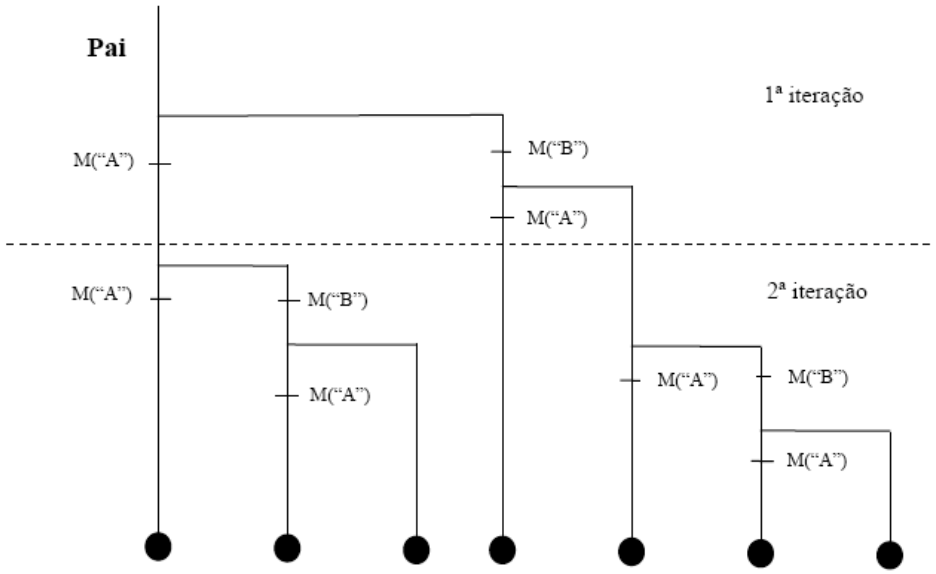

Figura 1.6: Diagrama do funcionamento do programa

Nota: A função `void M(char *)` apresenta no monitor uma string;

9. Considere que vai programar um sistema baseado numa tecnologia que utiliza seis processadores. Tendo um vector de 100000 posições, faça um programa que crie 5 processos e:
   - Dado um número, procurar esse número no array.
   - Cada processo filho, procura 20000 posições.
   - O processo que encontrar o número, deve imprimir a posição do array onde se encontra. Também deve devolver como valor de saída o número (não confundir com o pid!) do processo (1, 2, 3, 4, 5).

- Os processos que não encontrarem o número devem devolver como valor de saída o valor 0.
- O processo pai tem de esperar que todos os filhos terminem e imprimir o número do processo onde esse número foi encontrado (1, 2, 3, 4, 5).

Nota: O vector não tem números repetidos. O processo de pesquisa deve ser feito de uma forma paralela/concorrente.

a) Apresente uma solução para o problema

b) A solução encontrada é a mais eficiente na utilização de recursos? O que acontece quando um processo já encontrou o número e os outros ainda estão a executar?

10. Crie a seguinte função:

a) `char cria_gémeos(pid_t lista[2])` – esta função deve criar dois processos e retorna para o primeiro processo criado o caracter `'a'` e para o segundo processo o caracter `'b'`. Para o *pai*, a função retorna o caracter `'p'`. No processo pai, os identificadores dos dois processos criados devem ser armazenados no vector que foi passado como argumento à função.

b) Utilize a função da alínea anterior em substituição da função `fork` para resolver o exercício 7.

Bibliografia

Bryant, Randal E.; O'Hallaron, David – *Computer systems: a programmer's perspective*. Upper Saddle River, New Jersey: Pearson Education, Junho 2003 ⟨URL: `http://csapp.cs.cmu.edu/`⟩, p. 1008. ISBN 013178456

Butenhof, David R. – *Programming with POSIX(R) Threads*. 1.ª edição. Reading, Massachusetts: Addison-Wesley Professional, Maio 1997, p. 381. ISBN 0201633922

Curry, David A. – *UNIX Systems Programming for SVR4*. Sebastopol, CA: O'Reilly & Associates, 1996. ISBN 1–56592–163–1

Dowdney, Allen B. – *The little book of semaphores*. 2.ª edição. Needham, MA: Green Tea Press, 2005 ⟨URL: `http://greenteapress.com/semaphores/`⟩, p. 291

Ferreira, Luís L. *et al.* – *Programação Concorrente em Linux: gestão de processos*. Porto: ISEP, 2005

Gancarz, Mike – *Linux and the Unix Philosophy*. New York: Digital Press, 2003. ISBN 1–55558–263–X

Goodheart, Berny; Cox, James – *The Magic Garden Explained*. Sydney, Australia: Prentice Hall, 1994. ISBN 0–13–098138–9

Gough, Brian – *An Introduction to GCC*. Bristol, United Kingdom: Network Theory Limited, 2004 ⟨URL: `http://www.network-theory.co.uk/gcc/intro/`⟩. ISBN 0–9541617–9–3

Gray, John Shapley – *Interprocess communications in UNIX®: the nooks and crannies*. Upper Saddle River, New Jersey: Prentice Hall Ptr, 1997, p. 364. ISBN 0131868918

——— *Interprocess communications in LINUX®: the nooks and crannies*. Upper Saddle River, New Jersey: Prentice Hall Ptr, 2003, p. 624. ISBN 01304604727

Grötker, Thorsten *et al.* – *The Developer's Guide to Debugging*. Springer, 2008 ⟨URL: `http://www.debugging-guide.com`⟩. ISBN 978–1–4020–5540–9

Hagen, William von – *The Definitive guide to GCC*. 2.ª edição. Berkeley, CA: Apress, 2006 ⟨URL: `http://www.apress.com/book/downloadfile/2932`⟩. ISBN 1–59059–585–8

Horton, Ivor – *Beginning C: from novice to professional*. 4.ª edição. Berkeley, CA: Apress, 2006 ⟨URL: `http://www.apress.com`⟩, p. 640. ISBN 1590597354

Jackson, L. Blunt – *NPTL: The New Implementation of Threads for Linux*. Julho 2005 ⟨URL: `http://www.ddj.com/dept/linux/184406204`⟩

Koenig, Andrew – *C Traps and Pitfalls*. Reading, Massachusetts: Addison-Wesley, 1989. ISBN 0–201–17928–8

Lee, Edward A. – *The Problem with Threads*. Berkeley, January 10 2006 (UCB/EECS-2006-1). – Relatório Técnico ⟨URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html`⟩

Lewis, Bil; Berg, Daniel J. – *PThreads Primer: A Guide to Multithreaded Programming*. 1.ª edição. Upper Saddle River, New Jersey: Prentice Hall PTR, Outubro 1995 ⟨URL: `http://www.lambdacs.com/classes/programs_6_Oct_02.tar.gz`⟩, p. 352. ISBN 0134436989

Linden, Peter van der – *Expert C Programming: Deep C Secrets*. Englewood Cliffs, NJ: Prentice Hall, 1994. ISBN 0–13–177429–8

Love, Robert – *Linux System Programming*. Sebastopol, CA: O'Reilly Media, 2007. ISBN 0–596–00958–5

Matthew, Neil; Stones, Richard – *Beginning Linux programming*. 4.ª edição. Wrox, 2007 ⟨URL: `http://www.wrox.com/WileyCDA/`⟩. ISBN 978–0–470–14762–7

Mitchell, Mark L.; Samuel, Alex; Oldham, Jeffrey – *Advanced Linux programming*. Indianapolis: Sams, 2001 ⟨URL: `http://www.advancedlinuxprogramming.com/`⟩, p. 340. ISBN 0735710430

Molay, Bruce – *Understanding Unix™/Linux Programming*. Upper Saddle River, New Jersey: Prentice Hall, 2003. ISBN 0–13–008396–8

Rehman, Rafeeq Ur; Paul, Christopher – *The Linux Development Platform*. Upper Saddle River, New Jersey: Prentice Hall PTR, 2003 ⟨URL: `http://www.informit.com/store/product.aspx?isbn=0130091154`⟩. ISBN 0–13–009115–4

Robbins, Arnold – *Linux Programming by Example: The Fundamentals*. New York: Prentice Hall PTR, 2004 ⟨URL: `http://authors.phptr.com/robbins/`⟩. ISBN 0131429647

Robbins, Kay; Robbins, Steve – *UNIX™ systems programming: communication, concurrency and threads*. 2.ª edição. Upper Saddle River, New Jersey: Prentice Hall PTR, 2003 ⟨URL: `http://vip.cs.utsa.edu/usp`⟩, p. 912. ISBN 0130424110

Rochkind, Marc J. – *Advanced UNIX Programming*. 2.ª edição. Addison-Wesley Professional, 2004 ⟨URL: `http://www.basepath.com/aup/`⟩, p. 736. ISBN 0131411543

Shukla, Vikram – *Linux threading models compared: LinuxThreads and NPTL*. Julho 2006 ⟨URL: `http://www-128.ibm.com/developerworks/linux/library/l-threading.html`⟩

Silberschatz, Abraham; Gagne, Greg; Galvin, Peter Baer – *Operating System Concepts*. 6.ª edição. New York: Wiley, Março 2002, p. 976. ISBN 0471250600

Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg – *Operating System Concepts*. 7.ª edição. New York: Wiley, Dezembro 2004, p. 944. ISBN 0471694665

Sousa, Orlando – *Apontamentos de Sistemas Operativos II*. 2005 ⟨URL: `http://www.dei.isep.ipp.pt/~orlando/so2/`⟩

Stevens, W. Richard; Rago, Stephen A. – *Advanced programming in the UNIX® environment*. 2.ª edição. New York: Addison-Wesley Professional, 2005 ⟨URL: `http://www.apuebook.com`⟩, p. 960. ISBN 0201433079

The Open Group – *The single UNIX specification, version 3*. ⟨URL: `http://www.unix.org/version3/`⟩