Sistemas Operacionais
Laboratorio 1 - System Calls
Adaptação do Laboratório 1 - Prof. Eduardo Zambon & Prof. Roberta L. Gomes

1 *System Calls* no Linux

Vamos mencionar aqui alguns pontos já discutidos em aula e introduzir novos conceitos e informações úteis.

1.1 Aonde fica o *kernel* do SO?

Na maioria dos sistemas operacionais, o *kernel* é carregado no espaço de endereçamento virtual de todos os programas em execução. Por exemplo, o Linux em uma arquitetura x86 32-bits é mapeado no gigabyte (GB) mais "alto" do espaço de endereçamento, começando no endereço `0xf0000000`.

Note que o espaço de endereçamento virtual de um processador 32-bits é $2^{32} = 4$ GB, o que leva a um espaço de endereçamento virtual efetivo de 3 GB para a aplicação em si e 1 GB para o *kernel*.

Então como o *kernel* evita que uma aplicação reescreva as estruturas do *kernel* ou chame as funções do *kernel* diretamente? Isso é tarefa do mecanismo de mapeamento de memória, que permite ao SO especificar em qual *ring* a CPU deve estar executando para poder acessar uma dada região de memória.

1.2 *Protection rings*

A CPU x86 possui quatro *rings*, ou níveis de privilégio. Entretanto, a maioria dos OSes usa somente dois *rings*: *ring* 0 (*kernel mode*) e *ring* 3 (*user mode*). Os *rings* de numeração mais alta são mais restritos, indicando que eles não podem executar certas instruções privilegiadas, tais como instruções que vão interagir diretamente com o *hardware*. De forma similar, os mecanismos de proteção de endereços de memória (mais precisamente, páginas de memória, mas estudaremos isso um pouco mais adiante no curso) conseguem diferenciar permissões de acesso dependendo do *ring* atual em que a CPU está executando.

1.3 Trocando de *rings*

Como a CPU sai de um *ring* para outro?

Em geral, uma vez que a CPU entrou no *ring* 3 (*user mode*), o único jeito de retornar ao *kernel mode* é por meio de uma *interrupção*. Uma interrupção pode ser um evento de *hardware*, tal como um disco sinalizando a conclusão de uma operação de leitura/escrita; ou pode ser também uma *interrupção de software* (a famosa *chamada de sistema* ou *System Call*), em que o *software* intencionalmente levanta uma interrupção usando uma instrução especial. Por fim existe um terceiro tipo de interrupção (que pode ser entendida como de *hardware*... mas não há consenso sobre essa definição) chamada de *exceção*, como no caso de uma divisão por zero. O Termo *trap* também é muito usado na literatura ou pela comunidade, mas não há consenso sobre a terminologia (alguns definem *trap* como sendo as Syscalls, enquanto outros como sendo

qualquer interrupção que cause desvio para o Kernel). A figura 1 apresenta uma terminologia bem aceita pela comunidade acadèmica (Cornell Unitersity[1]):

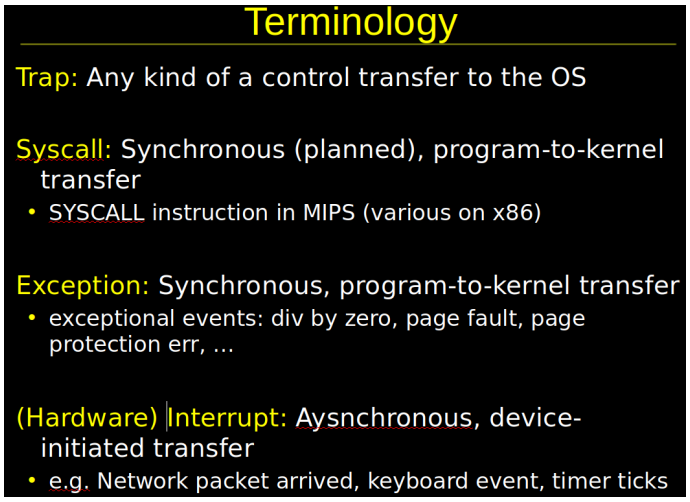

Figura 1: Uma terminologia bastante aceita.

Na arquitetura x86, interrupções são associadas com um valor 8-bits específico. Por exemplo, a exceção de divisão por zero recebe o número de interrupção 0. Este valor serve como um índice na *interrupt descriptor table (IDT)*, onde o *kernel* "instala" um *handler* (função) que é chamado quando uma interrupção dispara.

A IDT também especifica em qual *ring* o *handler* deve executar; em geral, o *ring* é zero. Assim, qualquer *software* que pode causar alguma interrupção vai levar a CPU a trocar para o *ring* zero e começar a executar o *handler* específico.

Alguns números de interrupção são designados pelo desenvolvedor do *hardware*. A Intel reserva as interrupções 0–31 para exceções, e por convenção, as 16 seguintes são tipicamente utilizadas para interrupções de dispositivos. Os outros 212 códigos de interrupção restantes ficam sob controle do *kernel*. O uso mais comum de um *handler* de interrupções é tratar as *System Calls* de uma aplicação. Por exemplo, o Linux utiliza 0x80, ou 128 em decimal, para a sua interrupção de *System Call*. O Windows, por outro lado, utiliza 0x2e, ou 46 em decimal. Essa escolha é totalmente arbitrária.

E como isso fica no código? Se você fizer um *disassemble* de um binário 32-bits antigo que faz uma chamada de sistema, você deve ver uma linha contendo `int $0x80`. A instrução `int` gera uma interrupção de *software* que leva a um salto (desvio) na execução para a função especificada como o *handler* da interrupção 0x80, que roda no *ring* 0. Antes que o salto seja realizado, alguns registradores são salvos na pilha. Durante a execução do *handler* da interrupção 0x80, o SO procura na *Syscall Table* o endereço da chamada de sistema em si que deve ser executada (usando o código que foi passado no EAX), gerando um novo desvio para a função que implementa a chamada de sistema. Após a execução da chamada de sistema, o *kernel* retorna o controle para a aplicação por meio da instrução `iret`, que restaura os registradores da aplicação e retorna para *ring* 3. A figura 2 resume os passos executados durante uma Syscall no Linux 32-bits.

Importante: `int $0x80` é um código legado e deve ser evitado, pois não está mais disponível em CPUs 64-bits (ele só foi utilizado como um exemplo!). O método atual de entrar em *kernel mode* em arquiteturas x86 64-bits é com a instrução `syscall`.

[1] http://www.cs.cornell.edu/courses/cs3410/2012sp/lecture/22-traps-i-g.pdf

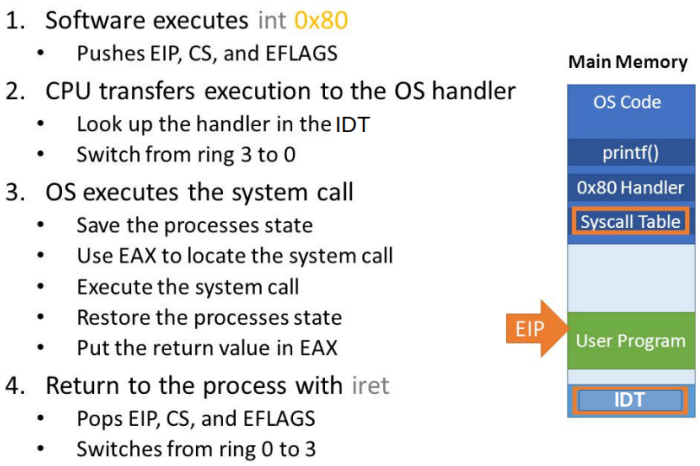

Figura 2: Passos executados durante uma Syscall no Linux 32-bits

2 Códigos de Exemplos de *System Calls*

O programa abaixo é o exemplo clássico de *Hello World* implementado em C.

```c
int main(void) {
   printf("Hello World!\n");
   return 0;
}
```

Esse programa faz uso da função `printf` que está definida em `stdio.h`. Esse arquivo define as funções de I/O que estão implementadas na biblioteca padrão do C (`libc`). Para um usuário normal, essa biblioteca provê a interface com as funcionalidades do SO.

Descendo um nível na API, é possível ver que as funções em `stdio.h` utilizam outras funções de mais baixo nível, as chamadas *system call wrappers*, que são funções que preparam a chamada da *system call* real. O programa abaixo utiliza os *wrappers* para reimplementar o programa de *Hello World*, empregando somente a função `write`, que faz parte do padrão POSIX, definido em `unistd.h`.

```c
#include <unistd.h>
int main(void) {
   const char *msg = "Hello World!\n";
   write(STDOUT_FILENO, msg, 13);
   return 0;
}
```

Baixando mais ainda o nível, é possível realizar diretamente as *system calls* do *kernel*, mas para tal é preciso programar diretamente no *assembly* da arquitetura, como ilustrado no programa a seguir.

```
#-----------------------------------------------------------------
# Writes "Hello World!" to the console using only system calls.
# Runs on 64-bit Linux only.
# To assemble and run:
# gcc -c hello2.s
# ld -o hello2 hello2.o
# ./hello2
#-----------------------------------------------------------------
```

```
9         .global _start
10
11        .text
12 _start:
13        # write(1, message, 13)
14        mov $1, %rax            # system call 1 is write
15        mov $1, %rdi            # file handle 1 is stdout
16        mov $message, %rsi      # address of string to output
17        mov $13, %rdx           # number of bytes
18        syscall                 # invoke operating system to do write
19
20        # exit(0)
21        mov $60, %rax           # system call 60 is exit
22        xor %rdi, %rdi          # we want return code 0
23        syscall                 # invoke operating system to exit
24 message:
25        .ascii "Hello World!\n"}
```

O programa acima está escrito em Assembly x86_64, no padrão AT&T, que é o utilizado pelo `as`, o montador do `gcc`. A *system call* que escreve no terminal é invocada pelo comando `syscall`. Esse comando não possui operandos pois cada *system call* tem um número variável de argumentos. Esses argumentos são passados em registradores, que precisam ser preenchidos corretamente antes da chamada. O registrador `rax` sempre deve conter o código da *system call* que deve ser executada. Os demais registradores variam conforme esse código. Uma tabela completa de todas as *system calls* do Linux 64-bits (com os respectivos registradores) pode ser vista em `http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/`.

3 Chamada *fork()*

Agora que você já tem uma visão geral de como as *Syscalls* funcionam, vamos estudar algumas *Syscalls* importantes para a criação e gerência de processos. A primeira delas é o `fork()`!

O `fork()` é usado para criar um novo processo em sistemas do tipo Unix. Quando criamos um processo por meio do `fork()`, dizemos que esse novo processo é o *filho*, e processo *pai* é aquele que chamou o `fork()`.

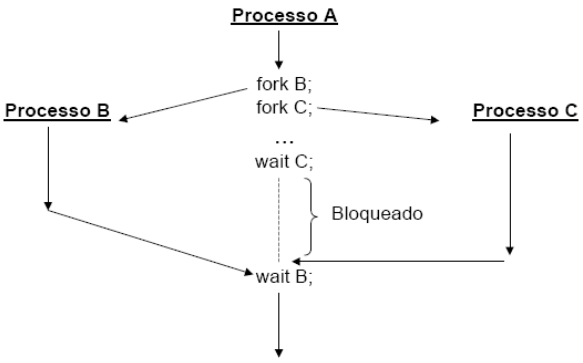

Figura 3: Processo A é o *pai* dos Processos B e C.

Quando o `fork()` for usado, será criado o processo filho, que será idêntico ao pai (**uma cópia do pai!**), inclusive tendo mesmo código, mesmas variáveis (e valores!), ponteiros para arquivos abertos, etc. Ou seja, o processo filho é uma cópia do pai, "exatamente" igual. As

aspas aqui devem-se ao seguinte: na verdade não será exatamente igual... algumas informações de controle (presentes no bloco de controle do proecesso filho) serão diferentes... como o caso do `PID` ou do `PPID` (*parent PID*).

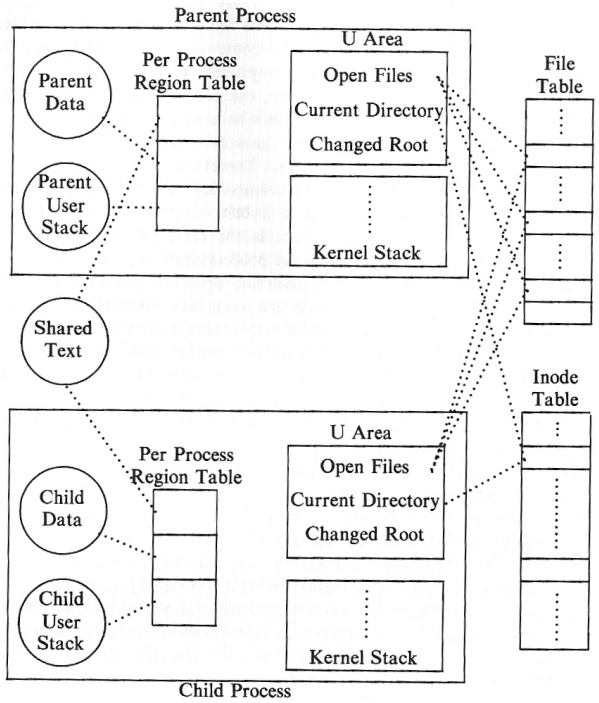

Figura 4: O processo filho é uma cópia do processo pai.

Como isso o processo filho tem **seu próprio espaço de endereçamento**, com cópia de todas as variáveis do processo pai (lembrem-se que elas ficam armazenadas no segmento de dados ou na pilha). Essas são independentes em relação às variáveis do processo pai. Além das variáveis (globais e locais), o processo filho "herda" do pai alguns atributos, tais como: variáveis de ambiente, privilégios e prioridade de escalonamento. O processo filho também herda alguns recursos, tais como arquivos abertos e devices. No entanto, alguns atributos e recursos, tais como PID, PPDI, sinais pendentes e estatísticas do processo, não são herdados pelo processo filho. A figura 4 ilustra os espaços de endereçamento do processo pai e filho. Observem na figura que o segmento de código (*Text*) não precisa ser copiado... o mesmo pode ser compartilhado entre os dois processos, uma vez que esse é *read-only*.

Copy-on-Write
Como alternativa à significante ineficiência do fork(), no Linux o fork() é implementado usando uma técnica chamada **copy-on-write** (COW). Essa técnica atrasa ou evita a cópia dos dados. Quando um processo filho é criado, ao invés de copiar o espaço de endereçamento do processo pai, ambos podem compartilhar uma única cópia somente de leitura. Se uma escrita é feita (seja pelo pai ou pelo filho), uma duplicação do seguimento é realizada imediatamente antes da escrita ser efetivada, e cada processo recebe sua própria cópia do seguimento. Consequentemente, a duplicação é feita apenas quando necessário, economizando tempo e espaço. Assim, o único overhead inicial do fork() é a duplicação das tabelas de gerência de memória (pai -> filho) e a criação de um novo bloco de controle de processo (BCP) para o filho.

Voltando à figura 3, vocês podem observar outra chamada de sistema: a chamada `wait()`. A sincronização entre processo pai e filho(s) é feita por meio da *Syscall* `wait()`, que bloqueia o processo pai até que um processo filho termine (mas veremos isso melhor mais tarde, em outro laboratório).

Agora vamos a algumas notas sobre a chamada `fork()`:

- A função fork() é invocada uma vez (no processo pai) mas retorna duas vezes quando há sucesso: uma no processo que a invocou e outra no novo processo criado, o processo filho.
- Em caso de sucesso da chamada, o valor de retorno da função `fork()` no processo pai é igual ao número do PID do processo filho recém criado (lembrando que todos os processos em Unix têm um identificador, geralmente designado por PID – *process identifier*). Em caso de erro, a chamada retorna o valor -1 e nenhum processo filho é criado. Isso pode ocorrer, por exemplo, porque o usuário estourou o limite de processos que ele pode criar.
- Para o processo filho, o valor de retorno da função `fork()` é igual a 0 (zero).

Tarefas

1. Faça o *download* dos arquivos exemplos para a aula de hoje: lab1.zip
2. Execute o arquivo `simple_fork.c`, analise o código e observe as diferenças nos valores exibidos pelos processos *pai* e *filho*. *Obs:* as chamadas `getpid()` e `getppid()` imprimem o próprio PID do processo e o PID do processo pai, respectivamente.

4 Diferenciando "Processo Pai" do "Processo Filho"

Uma vez que processos pai e filho rodam o mesmo código (sim! é o mesmo arquivo executável!!), para permitir que após o `fork()` pai e filho possam executar instruções distintas, normalmente o código do programa é estruturado conforme mostrado na listagem a seguir. No código mostrado, notem que quando o processo filho for executar pela primeira vez, ele começará a rodar exatamente a partir da linha 2. O processo filho NÃO executa o código que antecede `fork()` no qual ele foi criado

Para entender melhor quais são as primeiras instruções (de máquina!) executadas pelo processo filho, teríamos que enxergar o código em assembly. A grosso modo, dentre outras coisas, é colocado no registrador EAX o valor de retorno 0. Em seguida, no caso do código mostrado, é feita uma cópia desse valor para a região de memória correspondente à variavel pid (`pid=fork()`).

```
1  //...
2      pid=fork();
3      if(pid < 0) {
4              /* falha do fork */
5      }
6      else if (pid > 0) {
7              /* codigo do pai */
8      }
9      else { //pid == 0
10             /* codigo do filho */
11     }
```

A figura 5 ilustra um exemplo de código usando `fork()`. As setas azuis indicam o instante logo após a execução do `fork()` em que os valores retornados pela função são atribuídos às variáveis `pid` (no pai e no filho). Observem que se a máquina possuir duas CPUs isso pode até ser executado simultaneamente. Mas se não for o caso, o escalonador é quem vai definir quem roda primeiro (pai ou filho).

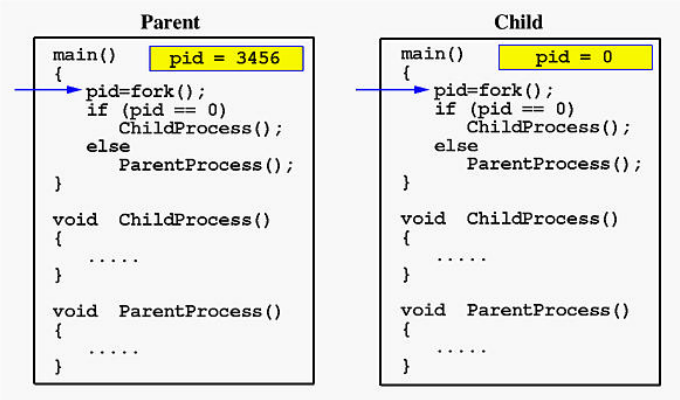

Figura 5: O processo filho começa a rodar a partir da linha de código em que é feita a chamada fork() que o criou.

Tarefas

3. Agora vamos diferenciar *Pai* e *Filho*... Execute o arquivo `two_procs.c` e analise o código. Por que são exibidos valores distintos para a variável `glob` (pelo pai e pelo filho) se a variável é global?!? [Transcreva sua resposta no formulário online]

4. Cuidado ao dar nomes às variáveis do programa! Execute arquivo `myPID.c`, analise o código. A variável `mypid` está sendo exibida com o mesmo valor no pai e no filho... você não achou isso estranho?!? Por que isso ocorre?

5 *User ID*, *Group ID* e *Process Group*

No Unix, cada processo tem um proprietário, um usuário que seja considerado seu dono. Por meio das permissões fornecidas pelo dono, o sistema sabe quem pode e não pode executar o processo em questão. Para lidar com os donos, o Unix usa os números **UID** (*User Identifier*) e **GID** (*Group Identifier*). Os nomes dos usuários e dos grupos (de usuários) servem apenas para facilitar o uso humano do computador.

Cada usuário precisa pertencer a um ou mais grupos. Como cada processo (e cada arquivo) pertence a um usuário, logo esse processo pertence ao grupo de seu proprietário. Assim sendo, cada processo está associado a um `UID` e a um `GID`.

Os números `UID` e `GID` variam de 0 a 65536. Dependendo do sistema, o valor limite pode ser maior. No caso do usuário root, esses valores são sempre 0 (zero). Assim, para fazer com que um usuário tenha os mesmos privilégios que o root, é necessário que seu `GID` seja 0.

Outro conceito também definido pelo UNIX é o de *Grupo de Processos*. No UNIX, **por default**, um processo e todos os seus descendentes formam um grupo de processos (identificado pelo **PGID** - *Process Group ID*). Isso facilita a gerência dos processos (por exemplo, é possível

terminar um grupo inteiro de processos com apenas uma chamada de sistema). Além disso também facilita o compartilhamento de recursos. Quando um processo é criado, ele é colocado no mesmo *process group* de seu pai. Mas vale ressaltar que um processo, por meio de *syscalls*, pode se excluir de um grupo, migrando para um novo grupo ou para um grupo já existente (dentro da mesma *session*... mais sobre isso em breve!).

A seguir são listadas algumas chamadas de sistema para verificar/setar UID[2], GID e PGID:

```
//Chamadas para consultar o user:
    uid_t getuid(void) //UID
    uid_t geteuid(void)  //effective user id

//Chamadas para consultar o user group:
    gid_t getgid(void)  //GID
    gid_t getegid(void) //effective group id

//Chamada para consultar o process group
    pid_t getpgrp(void); //PGID
    //...ou
    pid_t getpgid(pid_t pid); //'0' como parametro retorna o PGID
                              //do processo que executou a chamada

//Chamada para alterar o process group
    int setpgid(pid_t pid, pid_t pgid);
                 // seta o valor do ID do grupo do processo
                // (especificado por pid) para pgid */
```

Tarefas

5. Altere o arquivo `two_procs.c` de forma que tanto o pai quanto o filho imprimam os valores do *User ID* e do *Process Group ID*. O que você observou sobre o grupo de processos? Pai e filho estão no mesmo grupo? [Responda no formulário online]

6. Agora altere o código novamente de forma que o filho altere seu grupo de processo, criando um novo grupo cujo PGID será igual ao PID deste processo. *Dica:* `man setpgid`[3]

6 Relembrando: Comando PS

(Retirado de `man ps`) *By default, ps selects all processes with the same effective user ID (euid=EUID) as the current user and associated with the same terminal as the invoker. It displays process ID (pid=PID), terminal associated with the process (tname=TTY), cumulated CPU time in [dd-]hh:mm:ss format (time=TIME), and the executable name (ucmd=CMD). Output is unsorted by default.*

Opções interessantes:

[2]Na realidade, todo processo tem mais de um ID de usuário: além do *uid* (também é denominado *real uid*) há também o *effective uid*, ou *euid*. O *euid* é quem de fato define os direitos de acesso para um processo. Quando você faz o login, o shell de login define o *real uid* e *effective uid* com o mesmo valor (seu id de usuário real). Mas em algumas situações, dependendo do programa que um determinado processo executa, o *euid* pode ser alterado para outro usuário, geralmente o *root*. Com isso o processo passa a ter direto de acesso de root. Um exemplo famoso dessa funcionalidade é o comando *passwd*: quando o processo executa esse programa, ele passa a ter o *euid* setado para *root*, podendo com isso alterar o arquivo de senhas de usuários.

[3]O que foi pedido no exercício é simples... mas as regras para mudança de grupo de processo são bem complicadinhas. Caso tenha interesse, neste livro (sessão 10.6.3) você encontra mais informações.

- `$ ps` Lista os processos do usuário associados ao terminal
- `$ ps l` Idem, com informações mais completas
- `$ ps a` Lista também os processos não associados ao terminal
- `$ ps u` Lista processos do usuário
- `$ ps U <user>` ou `$ps -u <user>` Lista processos do usuário <user>
- `$ ps p <PID>` Lista dados do processo PID
- `$ ps r` Lista apenas os processos no estado running
- `$ ps f` Mostra a árvore de execução dos processos
- `$ ps t` Mostra todos os processos do terminal (incluindo a coluna STAT)
- `$ ps x` Mostra todos processos do usuário
- `$ ps al`, `$ ps au`, `$ ps aux`

Tarefas

7. Escreva um programa C que receba como parâmetro de entrada um inteiro N. Esse programa deve criar uma sequência de N filhos. Você deve usar a estrutura `for(...)`. Em um outro terminal (`Ctrl-Alt-t`), use o comando `$ ps` (e suas variantes) para exibir os processos que foram criados.

   > *Dica:* A chamada fork() deve aparecer dentro do loop... mas visto que após cada execução do fork() passamos a ter um processo pai e um filho, apenas o pai deve permanecer dentro do loop... o filho deve sair do loop assim que for criado.

8. Dado o código a seguir, quantos processos são criados (incluindo o processo principal) quando `n=4`? [Responda no formulário online]

```c
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;
    /* check for valid number of command-line arguments */
    n = atoi(argv[1]);
    for (i = 1; i < n; i++)
        if ((childpid = fork()) == -1)
            break;
    return 0;
}
```

9. Monte a árvore de processos gerada com a execução do código a seguir (a figura 6 ilustra um exemplo de árvore de processos). [Faça o desenho em arquivo ou papel, e use o formulário online para fazer upload do seu arquivo ou scan do seu papel (PDF ou IMAGEM)]

```
    c2 = 0;
    c1 = fork();                    /* fork number 1 */
    if (c1 == 0)
       c2 = fork();                 /* fork number 2 */
    fork();                         /* fork number 3 */
    if (c2 > 0)
       fork();                      /* fork number 4 */
    exit();
```

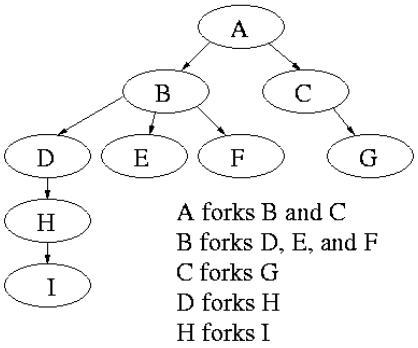

Figura 6: Exemplo de árvore de processos

========== Exercícios de Consolidação ==========

O objetivo desses exercícios é ajudar no estudo individual dos alunos. Soluções de questões específicas poderão ser discutidas em sala de aula, conforme interesse dos alunos.

1. Descreva o funcionamento da função `fork()`. Após o *fork*, como os processos pai e filho podem se comunicar/sincronizar (considere apenas as chamadas `fork()`, `exec()`, `exit()` e `wait()`)?

2. Explique o que o código a seguir faz:

```
int main (int argc, char *argv[]) {
   pid_t childpid;
   int i, n;
   pid_t waitreturn;
   if (argc != 2){   /* check for number of command-line arguments
      */
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if (childpid = fork()) break;
   while (childpid != (waitreturn = wait(NULL)))
      if ((waitreturn == -1) && (errno != EINTR))
         break;
   fprintf(stderr, "I am process %ld, my parent is %ld\n",
            (long)getpid(),(long)getppid());
   return 0;
}
```

3. Implemente um programa C que possui uma variável do tipo array contendo 10 números desordenados. Esse processo `MAIN` deve criar um filho. Em seguida o `MAIN` deve ordenar o array usando "ordenação simples" enquanto o filho deve fazer "quick sort". Ao final da ordenação, cada processo deve exibir o tempo gasto para realizar a mesma. O processo que acabar primeiro deve finalizar (`kill()`) o seu "parente"e imprimir uma msg avisando sobre o "assassinato"(ex. "Sou o pai, matei meu filho!"). Observem que não deve ser possível que os dois processos mostrem as mensagens de assassinato.

   Dicas:

```c
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
/*
- If pid is positive, then signal sig is sent to the process with the ID specified by pid.
- SIGKILL and SIGINT are examples of signals that can cause the process to be terminated
- Return Value: On success (at least one signal was sent), zero is returned. On error, -1 is returned, and errno is set appropriately.
*/
```

```c
#include <time.h>
...
clock_t c1, c2; /* variaveis que contam ciclos de processador */
float tmp;
c1 = clock();
//... codigo a ser executado
c2 = clock();
tmp = (c2-c1)*1000/CLOCKS_PER_SEC; //tempo de execucao em milisec.
```

```c
void quickSort(int valor[], int esquerda, int direita)
{
    int i, j, x, y;
    i = esquerda;
    j = direita;
    x = valor[(esquerda + direita) / 2];
    while(i <= j){
        while(valor[i] < x && i < direita){
            i++;
        }
        while(valor[j] > x && j > esquerda){
            j--;
        }
        if(i <= j){
            y = valor[i];
            valor[i] = valor[j];
            valor[j] = y;
            i++;
            j--;
        }
    }
    if(j > esquerda){
        quickSort(valor, esquerda, j);
```

```
24      }
25      if(i < direita){
26          quickSort(valor,  i, direita);
27      }
28  }
```