Sistemas Operacionais
Laboratorio - 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 páginas de memória, que serão estudados 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 a seguir apresenta uma das definições:

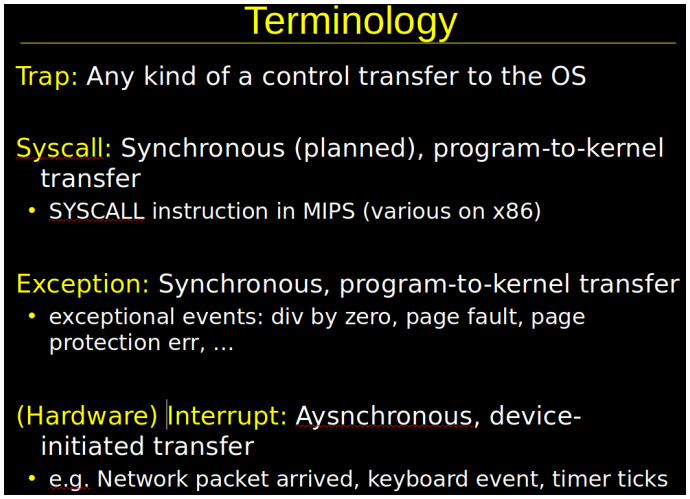

Figura 1: Uma terminologia bastante aceita.

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. O *kernel* retorna o controle para a aplicação por meio da instrução `iret`, que restaura os registradores da aplicação e retorna para *ring* 3.

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`.

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;
}
```

Por fim, 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
#-----------------------------------------------------------------
        .global _start

        .text
_start:
        # write(1, message, 13)
        mov $1, %rax            # system call 1 is write
        mov $1, %rdi            # file handle 1 is stdout
        mov $message, %rsi      # address of string to output
        mov $13, %rdx           # number of bytes
        syscall                 # invoke operating system to do write

        # exit(0)
        mov $60, %rax           # system call 60 is exit
        xor %rdi, %rdi          # we want return code 0
        syscall                 # invoke operating system to exit
message:
        .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 (com os respectivos registradores) pode ser vista em `http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/`.

3 Chamada *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()`.

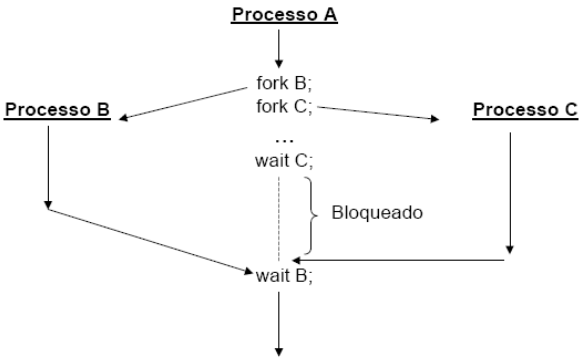

Figura 2: Processo A é o *pai* dos Processos B e C.

Quando usamos o `fork()` , será criado o processo filho, que será idêntico ao pai, inclusive tendo as mesmas variáveis, registros, descritores de arquivos etc. Ou seja, o processo filho é uma cópia do pai, "exatamente" igual. As aspas aqui deve-se ao seguinte: na verdade não será exatamente igual, já que algumas informações de controle (presentes no bloco de controle do proecesso filho) serão diferentes... como o caso do `PID` ou do `PPID` (*parent PID*).

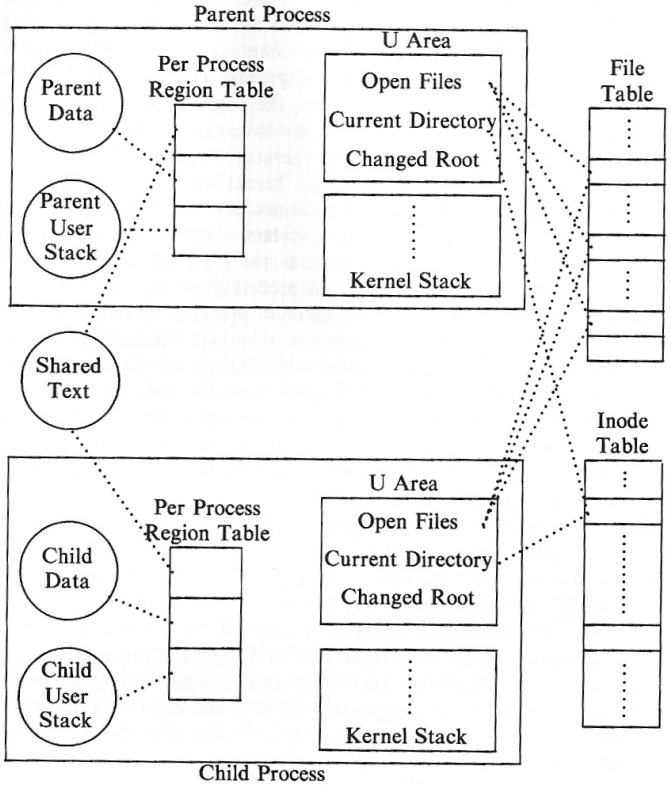

Figura 3: O processo filho é uma cópia do processo pai.

Observem na figura 3 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*.

Voltando à figura 2 vocês podem observar outra chamada de sistema: a chamada `wait()`. A sincronização entre processo pai e filho(s) é feita por meio da SVC wait(), que bloqueia o processo pai até que um processo filho termine (mas veremos isso melhor mais a diante). Agora vamos a algumas notas sobre a chamada `fork()`:

- A função fork() é invocada uma vez (no processo-pai) mas retorna duas vezes: uma no processo que a invocou e outra no novo processo criado, o processo-filho.
- O retorno da função fork(), no processo pai, é igual ao número do PID do processo filho recém criado (todos os processos em Unix têm um identificador, geralmente designado por PID – *process identifier*).
- O retorno da função fork()é igual a 0 (zero) no processo filho.

Com isso, normalmente o código é estruturado conforme mostrado a seguir:

```
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     }
```

4 Tarefas

1. Faça o *download* dos arquivos exemplos para a aula de hoje: lab1.zip
2. Execute 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.
3. Agora vamos diferenciar *Pai* e *Filho*... Execute arquivo `two_procs.c`, analise o código. Por que são exibidos valores distintos para a variável `glob` se a variável é global?!?
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?!?

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 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 "matar"um grupo inteiro de processos com apenas uma chamada de sistema). Além disso também facilita o compartilhamento de recursos. Mas vale ressaltar que um processo, eventualmente, pode se excluir de um grupo grupo e criar um novo grupo, também fazendo uso de chamadas de sistema.

A seguir você visualiza as chamadas de sistema para verificar os UID, GID e PGID:

```
1  //Chamadas para consultar o user:
2      uid_t getuid(void)
3      uid_t geteuid(void)  //effective user id
4
5  //Chamadas para consultar o user group:
6      gid_t getgid(void)
7      gid_t getegid(void) //effective group id
8      }
9
10 //Chamada para consultar o process group
11     pid_t getpgid(pid_t pid);
12
13
14 //Chamada para alterar o process group
15 int setpgid(pid_t pid, pid_t pgid);
16                     /* seta o valor do ID do grupo do */
17                     /* especificado por pid para pgid */
```

6 Tarefas

1. Altere o arquivo `two_procs.c` de forma que tanto o pai quanto o filho imprimam os falores do UID e do Processo Group. O que você observou sobre o grupo de processos? Agora altere o código de forma que o filho altere seu grupo de processo.

7 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.*

Alguns tributos: a Lista todos os processos e Mostra as variáveis associadas aos processos f Mostra a árvore de execução dos processsos l Mostra mais campos u Mostra o nome do usuário e a hora de inicio x Mostra os processos que não estão associados a terminais t Mostra todos os processos do terminal

Opções interessantes:

- `$ 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 al`, `$ ps au`, `$ ps aux`

8 Tarefas

1. Escreva um programa C que receba como parâmetro de entrada um inteiro N. Este 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.

2. Dado o código a seguir, calcule quantos processos são criados (além do processo principal) quando `n=3`?

```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;
    fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld
       \n", i, (long)getpid(), (long)getppid(), (long)childpid);
    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 matar (`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);
    }
    if(i < direita){
        quickSort(valor,  i, direita);
    }
}
```

4. Montar a árvore de processos gerada com a execução o código a seguir.

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