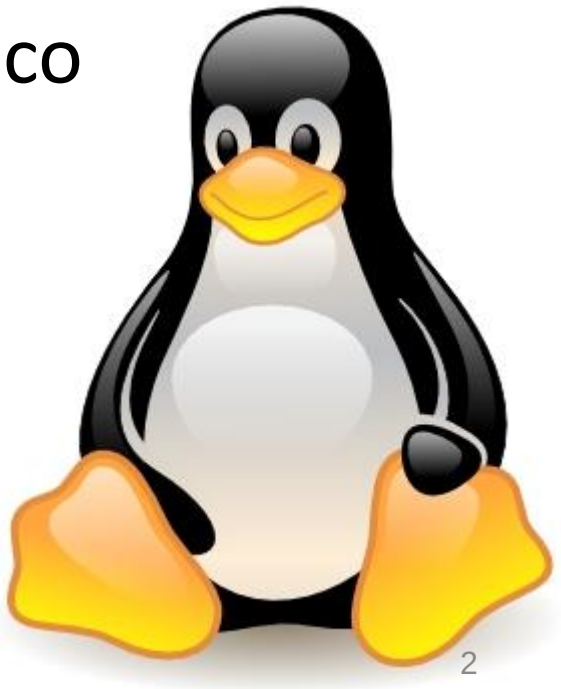
Sistemas Operacionais II

Chamadas de Sistema do Linux

Fabricio Breve
fabricio.breve@unesp.br
https://www.fabriciobreve.com

Sumário

- **`strace`** – depurar chamadas de sistema
- **`access`** – teste de permissão de arquivos
- **`fcntl`** – travas e outras operações com arquivos
- **`fsync`** e **`fdatasync`** – descarga de buffers de disco
- **`getrlimit`** e **`setrlimit`** – limites de recursos
- **`getrusage`** – estatísticas de processos
- **`gettimeofday`** – hora do relógio



Chamadas de Sistema do Linux

- Neste curso, utilizamos uma variedade de funções para diversas tarefas, como analisar opções de linha de comando, manipular processos, etc.
- Tais funções podem ser divididas em dois grupos, dependendo de como são implementadas:
  - Funções de bibliotecas
  - Chamadas de sistema

Chamadas de Sistema do Linux

- Funções de bibliotecas:
  - Uma função comum que reside em uma biblioteca externa ao programa.
  - A maioria das funções que vimos, estão na biblioteca padrão do C.
  - A chamada a uma função de biblioteca é igual a qualquer outra chamada.
- Chamadas de sistema:
  - Implementadas no núcleo do Linux.
  - Quando uma chamada é feita, o núcleo assume a execução do programa até que a chamada se complete.
  - A chamada é um procedimento especial que requer a transferência de controle para o núcleo.
    - Porém a biblioteca GNU C padrão empacota as chamadas de sistema do Linux em funções, para que você possa chamá-las facilmente.

Chamadas de Sistema do Linux

- São a interface mais básica entre programas e o núcleo do Linux.
- Algumas são bastante poderosas e só podem ser invocadas pelo superusuário.
- O Linux tem aproximadamente 200 chamadas de sistema diferentes.
  - Algumas são de uso interno do sistema.
- Vamos apresentar algumas das mais úteis para programadores de aplicativos e sistemas.
  - A maioria delas está declarada em **`<unistd.h>`**

Usando **`strace`**

- O comando **`strace`** rastreia a execução de outro programa, listando qualquer chamada de sistema que o programa faça e sinais que ele recebe.
  - Na saída de erro padrão.
- Para usar, basta chamar **`strace`** passando como argumento o programa a ser rastreado e sua lista de argumentos.
  - Exemplo:
    - **`strace hostname`**

```
execve("/bin/hostname", ["hostname"], 0x7ffde1ca9150 /* 68 vars */) = 0
brk(NULL)                               = 0x561fbd578000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=100986, ...}) = 0
mmap(NULL, 100986, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f8af554b000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2030544, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8af5549000
mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8af4f4c000
mprotect(0x7f8af5133000, 2097152, PROT_NONE) = 0
mmap(0x7f8af5333000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7f8af5333000
mmap(0x7f8af5339000, 15072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f8af5339000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f8af554a500) = 0
mprotect(0x7f8af5333000, 16384, PROT_READ) = 0
mprotect(0x561fbcc18000, 4096, PROT_READ) = 0
mprotect(0x7f8af5564000, 4096, PROT_READ) = 0
munmap(0x7f8af554b000, 100986)          = 0
brk(NULL)                               = 0x561fbd578000
brk(0x561fbd599000)                     = 0x561fbd599000
uname({sysname="Linux", nodename="ubuntu-donald", ...}) = 0
fstat(1, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
write(1, "ubuntu-donald\n", 14)         = 14
exit_group(0)                           = ?
+++ exited with 0 +++
```

Usando **strace**

- **strace hostname**
  - A chamada de sistema **execve** chama o programa **hostname**:
    - **execve("/bin/hostname", ["hostname"], 0x7ffe8e8f4c40 /* 68 vars */) = 0**
      - Nome do programa, lista de argumentos, lista do ambiente.
  - As próximas aproximadamente 25 linhas fazem parte do mecanismo de carregar a biblioteca C padrão.
  - Perto do final, temos a chamada **uname**, que ajuda o programa a cumprir sua tarefa:
    - **uname({sysname="Linux", nodename="ubuntu-donald", ...}) = 0**
  - Finalmente, a chamada **write** é usada para produzir a saída na saída padrão:
    - **write(1, "ubuntu-donald\n", 14) = 14**
      - Descritor, *string* com nome do hospedeiro, quantidade de caracteres

Veja em execução:
https://youtu.be/eip3b6vsNoU

Testando permissões de arquivos com **`access`**

- A chamada de sistema **`access`** determina se o processo chamador tem permissão de acesso em um arquivo.
  - Ele pode checar qualquer combinação de permissões de leitura, escrita e execução, e pode checar pela existência do arquivo.

Testando permissões de arquivos com **`access`**

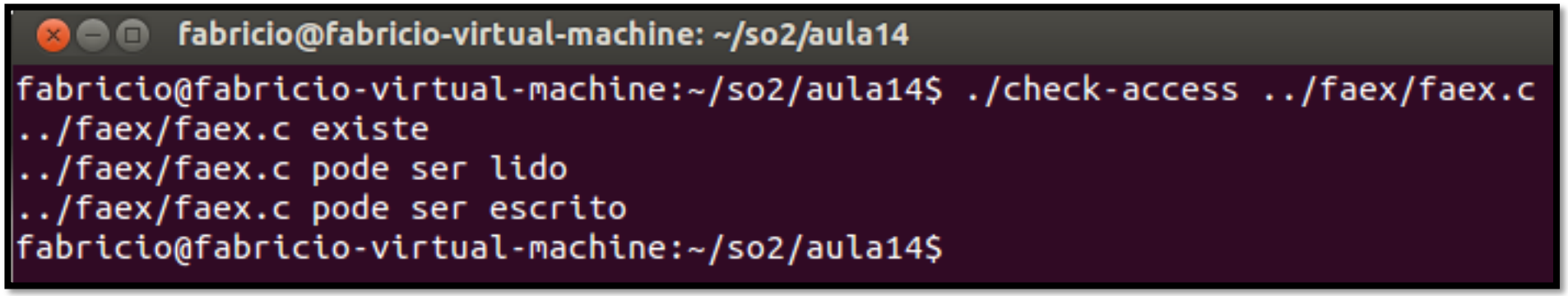
- Checando permissões:
  - Argumentos:
    - Caminho para o arquivo a ser checado.
    - **`R_OK`**, **`W_OK`**, e **`X_OK`** para checar por leitura, escrita, e execução, respectivamente. Podem ser conectados pelo OU binário **|**
  - O valor de retorno é:
    - **`0`** se o processo tem todas as permissões especificadas.
    - **`-1`** e ajusta **`errno`** para **`EACCES`** (ou **`EROFS`**, se a permissão de escrita foi pedida em um sistema somente de leitura).

Testando permissões de arquivos com **`access`**

- Checando existência:
  - Argumentos:
    - Caminho para o arquivo a ser checado.
    - **`F_OK`**
  - O valor de retorno é:
    - **`0`** se o arquivo existir.
    - **`-1`** e ajusta **`errno`** para **`ENOENT`** se o arquivo não existir. **`errno`** também pode ser ajustado para **`EACCES`** se um diretório no caminho do arquivo estiver inacessível.

Testando permissões de arquivos com **access**

- O programa a seguir usa **access** para checar se um arquivo existe e determina permissões de leitura e escrita.
  - Especifique o nome do arquivo na linha de comando.



```c
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  char* path = argv[1];
  int rval;

  /* Checa a existência do arquivo.  */
  rval = access (path, F_OK);
  if (rval == 0)
    printf ("%s existe\n", path);
  else {
    if (errno == ENOENT)
      printf ("%s não existe\n", path);
    else if (errno == EACCES)
      printf ("%s não está acessível\n", path);
    return 0;
  }

  /* Checa o acesso de leitura.  */
  rval = access (path, R_OK);
  if (rval == 0)
    printf ("%s pode ser lido\n", path);
  else
    printf ("%s não pode ser lido (acesso negado)\n", path);

  /* Checa o acesso de escrita.  */
  rval = access (path, W_OK);
  if (rval == 0)
    printf ("%s pode ser escrito\n", path);
  else if (errno == EACCES)
    printf ("%s não pode ser escrito (acesso negado)\n", path);
  else if (errno == EROFS)
    printf ("%s não pode ser escrito (sistema de arquivos somente de leitura)\n", path);

  return 0;
}
```

check-access.c

Checa as Permissões de Acesso de um Arquivo

Veja em execução:
https://youtu.be/tzeowsfHeZM

`fcntl`: Travas e outras operações com arquivos

- A chamada de sistema **`fcntl`** é o ponto de acesso para várias operações avançadas em descritores de arquivos.
- Argumentos:
  - Descritor de arquivos aberto.
  - Operação a ser realizada.

`fcntl`: Travas e outras operações com arquivos

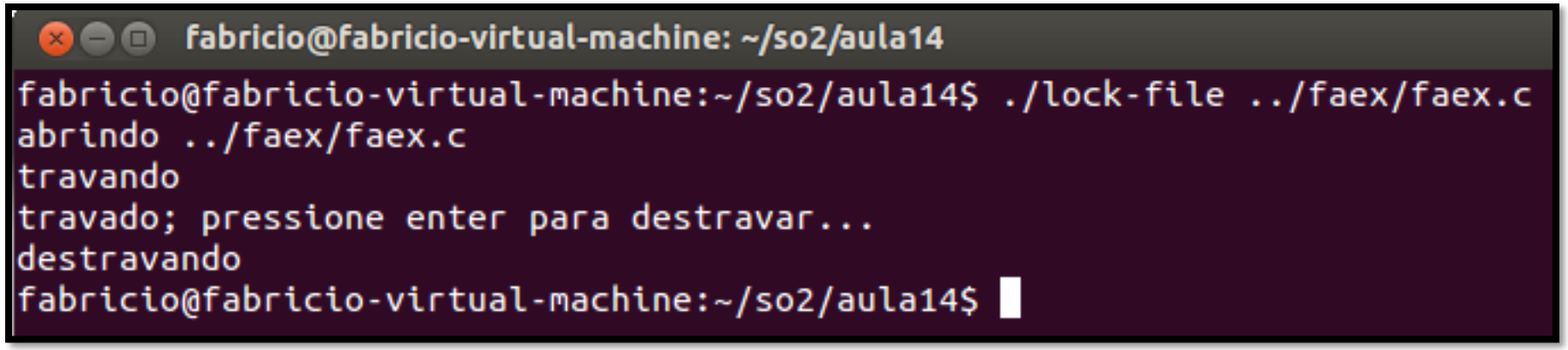
- **`fcntl`** permite colocar travas de leitura ou de escrita em arquivos.
  - Análogo ao uso de **`mutex`** para trechos críticos.
  - Mais de um processo pode colocar trava de leitura em um arquivo ao mesmo tempo.
  - Apenas um processo pode colocar trava de escrita de cada vez.
  - O mesmo arquivo não pode ser travado para leitura e escrita ao mesmo tempo.
- As travas não impedem que outros processos abram, leiam ou escrevam em arquivos, a menos que eles também usem **`fcntl`** para adquirir travas.

`fcntl`: Travas e outras operações com arquivos

- Para colocar uma trava em um arquivo:
  - Crie uma variável **`struct flock`** e preencha-a com zeros.
  - Ajuste o campo **`l_type`** da estrutura para **`F_RDLCK`** para trava de leitura ou **`F_WRLCK`** para trava de escrita.
  - Chame **`fcntl`** passando um descritor de arquivo para o arquivo, o código de operação **`F_SETLCKW`** e um ponteiro para a variável **`struct flock`**.
  - Se outro processo tem uma trava que não permite a aquisição de uma nova trava, **`fcntl`** bloqueia até que a trava seja liberada.

`fcntl`: Travas e outras operações com arquivos

- O programa a seguir abre um arquivo para escrita, cujo nome é fornecido na linha de comando, e então coloca uma trava de escrita no mesmo. O programa espera o usuário pressionar Enter e então destrava e fecha o arquivo.



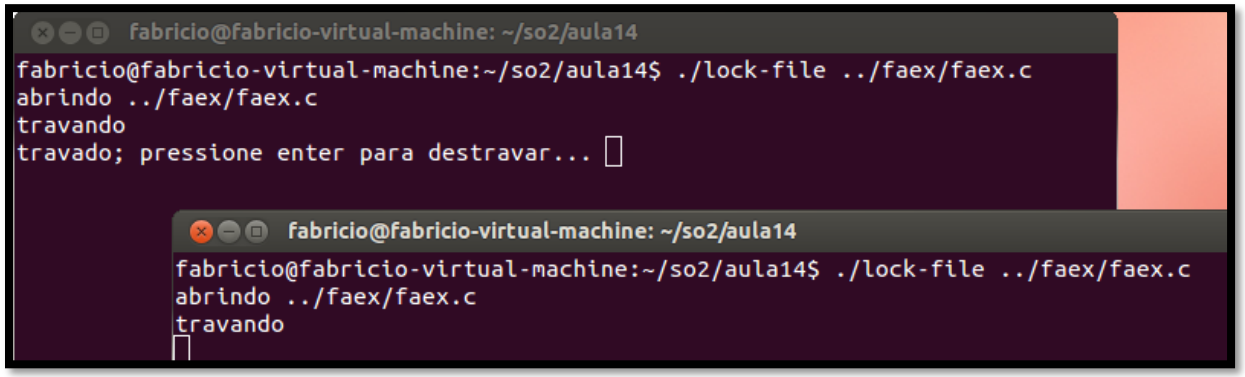
```c
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  char* file = argv[1];
  int fd;
  struct flock lock;

  printf ("abrindo %s\n", file);
  /* Abre um descritor de arquivo para o arquivo.  */
  fd = open (file, O_WRONLY);
  printf ("travando\n");
  /* Inicializa a estrutura flock.  */
  memset (&lock, 0, sizeof(lock));
  lock.l_type = F_WRLCK;
  /* Coloca uma trava de escrita no arquivo.  */
  fcntl (fd, F_SETLKW, &lock);

  printf ("travado; pressione enter para destravar... ");
  /* Espera o usuário pressionar enter.  */
  getchar ();

  printf ("destravando\n");
  /* Libera a trava.  */
  lock.l_type = F_UNLCK;
  fcntl (fd, F_SETLKW, &lock);

  close (fd);
  return 0;
}
```

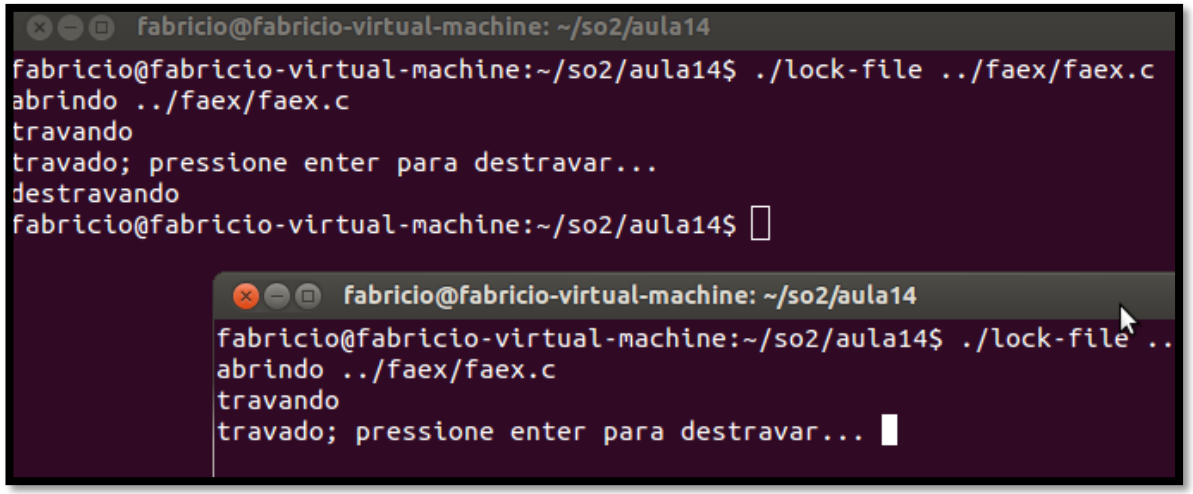
lock-file.c

Cria uma Trava de Escrita com **fcntl**

`fcntl`: Travas e outras operações com arquivos

- Abra dois terminais.
- Trave um arquivo no primeiro terminal.
- Tente travar o mesmo arquivo no segundo terminal.
- Tecle Enter no primeiro terminal.





Veja em execução:
https://youtu.be/L4xB8gfAIhg

`fcntl`: Travas e outras operações com arquivos

- Se não quiser que **`fcntl`** bloqueie se não conseguir obter a trava, utilize **`F_SETLK`** em vez de **`F_SETLKW`**.
  - Se não conseguir a trava, **`fcntl`** retorna **`-1`** imediatamente
- O Linux também oferece a chamada **`flock`** para travar arquivos. A vantagem do **`fcntl`** é que ela funciona em sistemas de arquivos NFS (*Network File System*).
  - Funciona inclusive com máquinas diferentes que estejam acessando o mesmo arquivo.

`fsync` e `fdatasync`: descarga de *buffers* de disco

- A maioria dos sistemas operacionais, incluindo o Linux, fazem cache dos dados escritos em disco em um *buffer* de memória para reduzir o número de requisições de escrita em disco, aumentando o desempenho do sistema.
  - Quando o *buffer* enche, o *timer* expira, etc. os dados são escritos no disco.
  - Isto é um problema em programas que dependem da integridade dos registros escritos em disco.
    - Se o sistema falha de repente, dados que ainda estavam no cache são perdidos.
    - Para contornar este problema, é preciso escrever em disco imediatamente quando necessário.

`fsync` e `fdatasync`: descarga de *buffers* de disco

- A chamada de sistema **`fsync`** recebe como argumento um descritor de arquivos com permissão de escrita, e descarrega para o disco quaisquer dados escritos para este descritor.
  - A chamada não retorna até que os dados estejam fisicamente escritos no disco.
- **`fdatasync`** é semelhante a **`fsync`**, porém ele não atualiza os metadados do arquivo (como a data de última modificação) a menos que uma operação subsequente precise deles para ser corretamente executada.
  - O objetivo é reduzir a atividade de disco em aplicações que não requerem que os metadados sejam sincronizados.

fsync e **fdatasync**: descarga de *buffers* de disco

- A função abaixo ilustra o uso de **fsync**:

```c
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

const char* journal_filename = "journal.log";

void write_journal_entry (char* entry)
{
  int fd = open (journal_filename, O_WRONLY | O_CREAT | O_APPEND, 0660);
  write (fd, entry, strlen (entry));
  write (fd, "\n", 1);
  fsync (fd);
  close (fd);
}
```

write_journal_entry.c

`fsync` e **`fdatasync`**: descarga de *buffers* de disco

- Também é possível abrir um arquivo para escrita síncrona com a flag **`O_SYNC`** na chamada **`open`**.
  - Toda escrita será efetivada em disco imediatamente, dispensando o uso de **`fsync`**.

`getrlimit` e **`setrlimit`**: Limites de Recursos

- As chamadas de sistema **`getrlimit`** e **`setrlimit`** permitem que um processo leia e defina limites de recursos de sistema que pode consumir.
  - Faz programaticamente o mesmo que o comando **`ulimit`** faz na *shell*.
- Para cada recurso há um *hard limit* e um *soft limit*.
  - O *soft limit* nunca pode exceder o *hard limit*.
  - O *hard limit* só pode ser alterado pelo superusuário.
  - Tipicamente, aplicações baixam o *soft limit* para limitar os recursos que podem utilizar.

`getrlimit` e `setrlimit`: Limites de Recursos

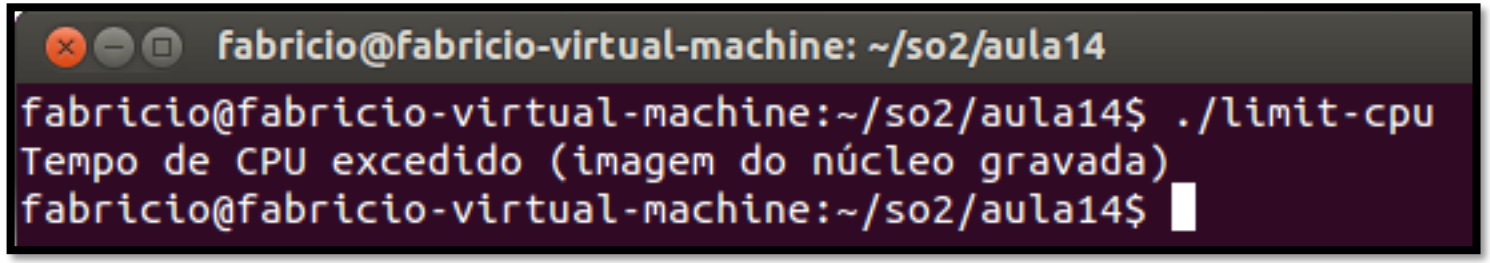
- Argumentos:
  - Código especificando tipo de limite de recurso.
  - Ponteiro para uma variável **`struct rlimit`**.
- A chamada **`getrlimit`** preenche os valores da estrutura.
- A chamada **`setrlimit`** muda os limites baseado nas informações da estrutura.
- A estrutura **`rlimit`** tem dois campos:
  - **`rlim_cur`** – *soft limit*
  - **`rlim_max`** – *hard limit*

`getrlimit` e **`setrlimit`**: Limites de Recursos

- Alguns dos tipos de limites mais úteis:
  - **`RLIMIT_CPU`** – Tempo máximo de CPU, em segundos, usado pelo programa. Esta é a quantidade de tempo total que o programa passa realmente executando na CPU. Se o programa exceder este limite, será terminado com o sinal **`SIGXCPU`**.
  - **`RLIMIT_DATA`** – Quantidade máxima de memória que um programa pode alocar para seus dados. Alocação adicional além do limite irá falhar.
  - **`RLIMIT_NPROC`** – Quantidade máxima de processos filhos que podem estar executando para este usuário. Se o processo chamar **`fork`** e o limite já tiver sido atingido, o **`fork`** falha.
  - **`RLIMIT_NOFILE`** – Quantidade máxima de descritores de arquivos que o processo pode ter aberto de cada vez.
- Veja a lista completa na página de manual de **`setrlimit`**.

`getrlimit` e **`setrlimit`**: Limites de Recursos

- O programa a seguir ilustra o ajuste de tempo de CPU consumido por um programa.
  - Ele ajusta o limite de 1 segundo e então entra em um *loop* infinito.
  - O Linux logo mata o processo, quando ele excede 1 segundo de tempo de CPU.
- Quando o programa é terminado por **`SIGXCPU`**, a *shell* imprime uma mensagem interpretando o sinal.



```c
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>

int main ()
{
  struct rlimit rl;

  /* Obtém os limites atuais.  */
  getrlimit (RLIMIT_CPU, &rl);
  /* Ajusta o limite de CPU em um segundo.  */
  rl.rlim_cur = 1;
  setrlimit (RLIMIT_CPU, &rl);
  /* Faz o trabalho.  */
  while (1);

  return 0;
}
```

limit-cpu.c

Demonstração de limite de tempo de CPU

Veja em execução:
https://youtu.be/FeUNzCEOgws

`getrusage`: Estatísticas de Processos

- A chamada de sistema **`getrusage`** recupera estatísticas de processo do núcleo.
  - Argumentos:
    - **`RUSAGE_SELF`** para obter estatísticas para o processo atual ou **`RUSAGE_CHILDREN`** para obter estatísticas de todos os processos filhos finalizados criados por este processo e seus filhos.
    - Ponteiro para uma variável **`struct rusage`**, que será preenchida pelas estatísticas.

`getrusage`: Estatísticas de Processos

- Alguns dos campos mais interessantes na estrutura **`rusage`**:
  - **`ru_utime`** – um campo com uma **`struct timeval`** contendo a quantidade de *tempo de usuário*, em segundos, que o processo usou.
    - Tempo de usuário é o tempo que a CPU gasta executando o programa do usuário, não incluindo chamadas de sistemas do núcleo.
  - **`ru_stime`** – um campo com uma **`struct timeval`** contendo a quantidade de *tempo de sistema*, em segundos, que o processo usou.
    - Tempo de sistema é o tempo que a CPU gasta executando chamadas de sistema em favor do processo.
  - **`ru_maxrss`** – a maior quantidade de memória física ocupada pelos dados do processo no curso de sua execução.
- A página de manual de **`getrusage`** lista todos os campos disponíveis.

`getrusage`: Estatísticas de Processos

- A função abaixo imprime o tempo de usuário e tempo de sistema do processo atual:

print-cpu-times.c

```
#include <stdio.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>

void print_cpu_time()
{
  struct rusage usage;
  getrusage (RUSAGE_SELF, &usage);
  printf ("Tempo de CPU: %ld.%06ld segundos de tempo de usuário, %ld.%06ld segundos
de tempo de sistema\n",
          usage.ru_utime.tv_sec, usage.ru_utime.tv_usec,
          usage.ru_stime.tv_sec, usage.ru_stime.tv_usec);
}
```

gettimeofday: Hora do Relógio

- A chamada de sistema **gettimeofday** pega a hora do relógio do sistema.
  - O primeiro argumento é um ponteiro para uma variável **struct timeval**.
    - Esta estrutura representa a hora, em segundos, dividida em dois campos.
      - O campo **tv_sec** contém o número inteiro de segundos.
      - O campo **tv_usec** contém um número adicional de microssegundos.
      - Representa o número de segundos passados desde o início da *época* UNIX, à meia noite de 1 de Janeiro de 1970 (UTC).
  - O segundo argumento é **NULL**.
  - Inclua **<sys/time.h>** para usar esta chamada de sistema.

gettimeofday: Hora do Relógio

- As funções de biblioteca **localtime** e **strftime** ajudam a manipular o valor de retorno de **gettimeofday**.
- A função **localtime** recebe um ponteiro para o número de segundos (campo **tv_sec** da estrutura **timeval**) e retorna um ponteiro para um objeto **struct tm**.
  - Esta estrutura contém alguns campos úteis:
    - **tm_hour**, **tm_min**, **tm_sec** – Tempo do dia, em horas, minutos e segundos.
    - **tm_year**, **tm_mon**, **tm_day** – O ano, mês e dia.
    - **tm_wday** – O dia da semana. Zero representa o domingo.
    - **tm_yday** – O dia do ano.
    - **tm_isdst** – Uma flag que indica se o horário de verão está em vigor.

`gettimeofday`: Hora do Relógio

- A função **`strftime`** pode produzir, a partir da **`struct tm`**, uma *string* formatada para mostrar a data e a hora. O formato é especificado de uma maneira similar ao **`printf`**, como uma *string* com códigos embutidos indicando onde os campos devem ser incluídos.
  - Exemplo:
    - **`“%d/%m/%Y %H:%M:%S”`** especifica a data e hora desta forma: **`08/06/2012 15:46:50`**

gettimeofday: Hora do Relógio

- **strftime**
  - Argumentos:
    - Buffer de caracteres para receber a *string*.
    - Tamanho do *buffer* de caracteres.
    - Formato da *string*.
    - Ponteiro para a variável **struct tm**.
  - Consulte a página de manual de **strftime** para uma lista completa de códigos que podem ser usados na *string*.
  - Observações:
    - **localtime** e **strftime** tem precisão de 1 segundo (não usam **tv_usec**).
    - Inclua **<time.h>** se usar **localtime** ou **strftime**.

`gettimeofday`: Hora do Relógio

- A função a seguir imprime data e hora atual, incluindo milissegundos:

print-time.c

```c
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

void print_time ()
{
  struct timeval tv;
  struct tm* ptm;
  char time_string[40];
  long milliseconds;

  /* Obtém data e hora atual, e converte-as em uma estrutura tm.  */
  gettimeofday (&tv, NULL);
  ptm = localtime (&tv.tv_sec);
  /* Formata data e hora, até os segundos.  */
  strftime (time_string, sizeof (time_string), "%d/%m/%Y %H:%M:%S", ptm);
  /* Computa milisegundos de microsegundos.  */
  milliseconds = tv.tv_usec / 1000;
  /* Imprime a hora formatada, até os segundos, seguido de um ponto decimal
     e os milissegundos.  */
  printf ("%s.%03ld\n", time_string, milliseconds);
}
```

Referências Bibliográficas

1. NEMETH, Evi.; SNYDER, Garth; HEIN, Trent R.;. *Manual Completo do Linux: Guia do Administrador*. São Paulo: Pearson Prentice Hall, 2007.
2. DEITEL, H. M.; DEITEL, P. J.; CHOFFNES, D. R.; *Sistemas Operacionais: terceira edição*. São Paulo: Pearson Prentice Hall, 2005. Cap. 20.
3. MITCHELL, Mark; OLDHAM, Jeffrey; SAMUEL, Alex; *Advanced Linux Programming*. New Riders Publishing: 2001. Cap. 8.
4. TANENBAUM, Andrew S.; BOS, Herbert. *Sistemas Operacionais Modernos.* 4ed. São Paulo: Pearson Education do Brasil, 2016. Cap. 10.