Processos e Sinais - Linux

MCTA026-13 - Sistemas Operacionais

Emilio Francesquini e Fernando Teubl Ferreira
e.francesquini@ufabc.edu.br / fernando.teubl@ufabc.edu.br

2019.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

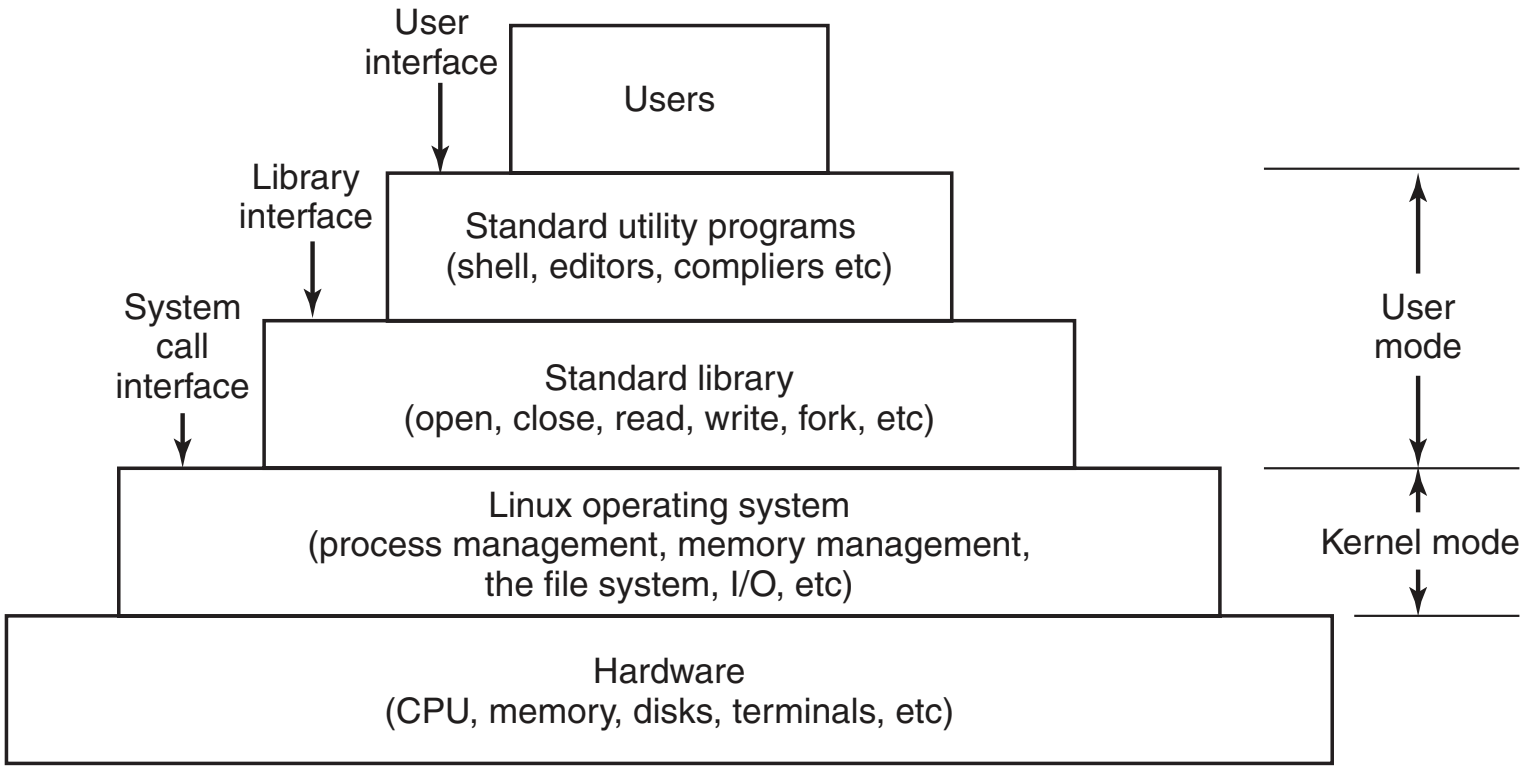
UFABC

- Estes slides foram preparados para o curso de **Sistemas Operacionais na UFABC**.
- Este material foi preparado com base naquele elaborado e gentilmente cedido pelo Prof. Gustavo Souza Pavani (UFABC)
- As figuras e tabelas foram retiradas do livro [TB]: *Andrew S. Tanenbaum, Herbert Bos.* **Modern Operating Systems**

Processos no Linux

- Chamadas de sistema (*system calls*, *syscalls*)
  - Para cada chamada de sistema há uma função correspondente disponível em uma biblioteca
  - No linux, grande parte da biblioteca segue o padrão POSIX (*Portable Operating System Interface*)
  - *Man pages* seção 2 (*system calls*)
    - `$ man 2 <nome_da_syscall>`

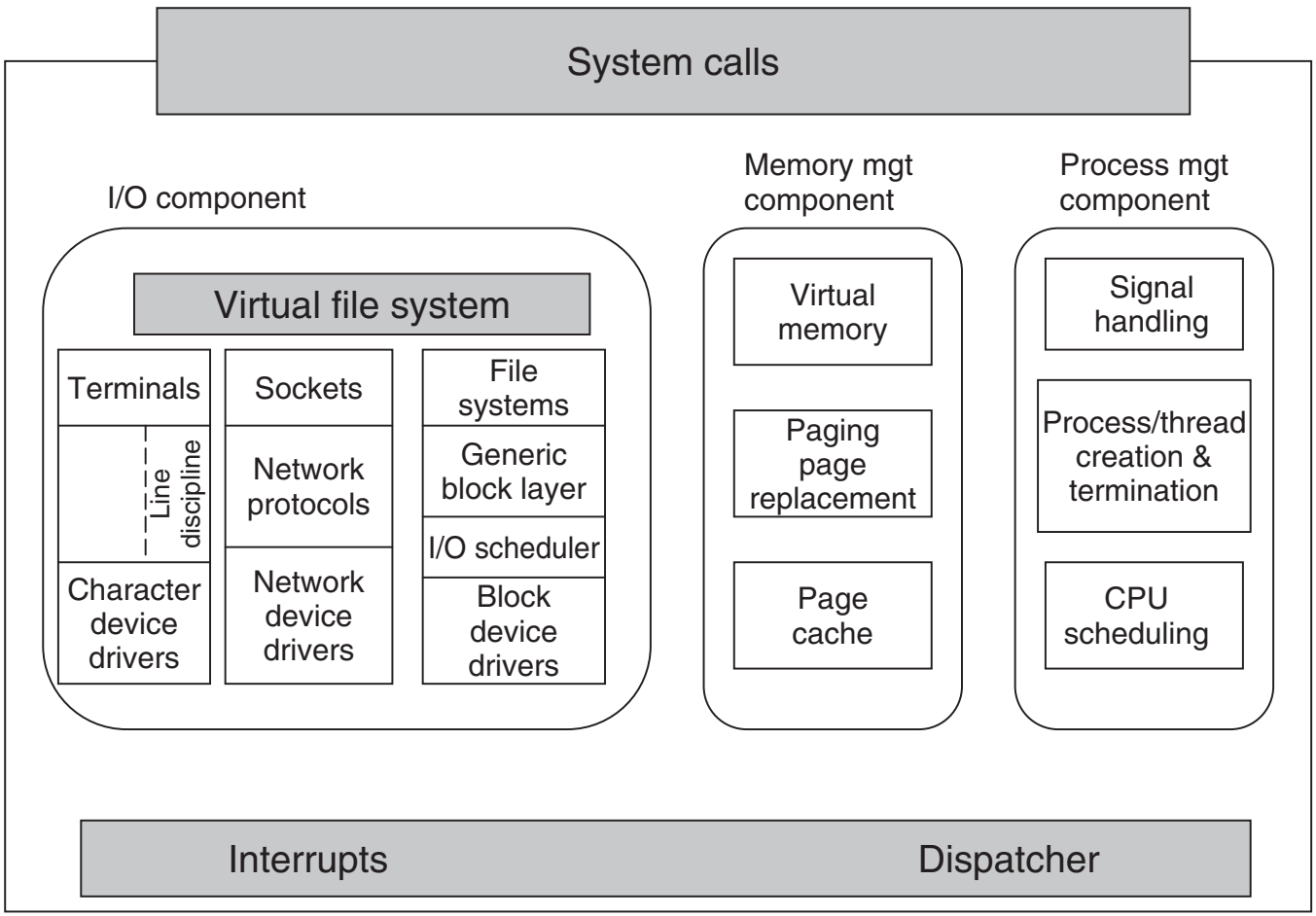
Linux - Camadas do sistema



Estrutura do núcleo do Linux

UFABC

- Três componentes principais
  1. Entrada e saída (E/S)
  2. Gerenciamento de memória
  3. Gerenciamento de processo



Sinais em Linux

UFABC

- Sinais (*signals*) permitem a comunicação inter-processo através de interrupções de software
  - Um processo envia um sinal a outro processo, o qual pode:
    1. Ignorá-lo
    2. Capturá-lo
    3. Forçá-lo a morrer
  - Um processo pode apenas mandar um sinal para o seu **grupo de processos**, que consiste do seu pai (e ancestrais), irmãos e filhos (e descendentes).
    - Todos os membros de um grupo de processos podem receber um sinal com uma única chamada de sistema.
  - Veja a lista de sinais em `man 7 signal`
  - O terminal pode enviar sinais aos processos. Ver as combinações em `man 3 termios`

Sinais em Linux - Tabela parcial

UFABC

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Syscalls em Linux para gerenciamento de processos

UFABC

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

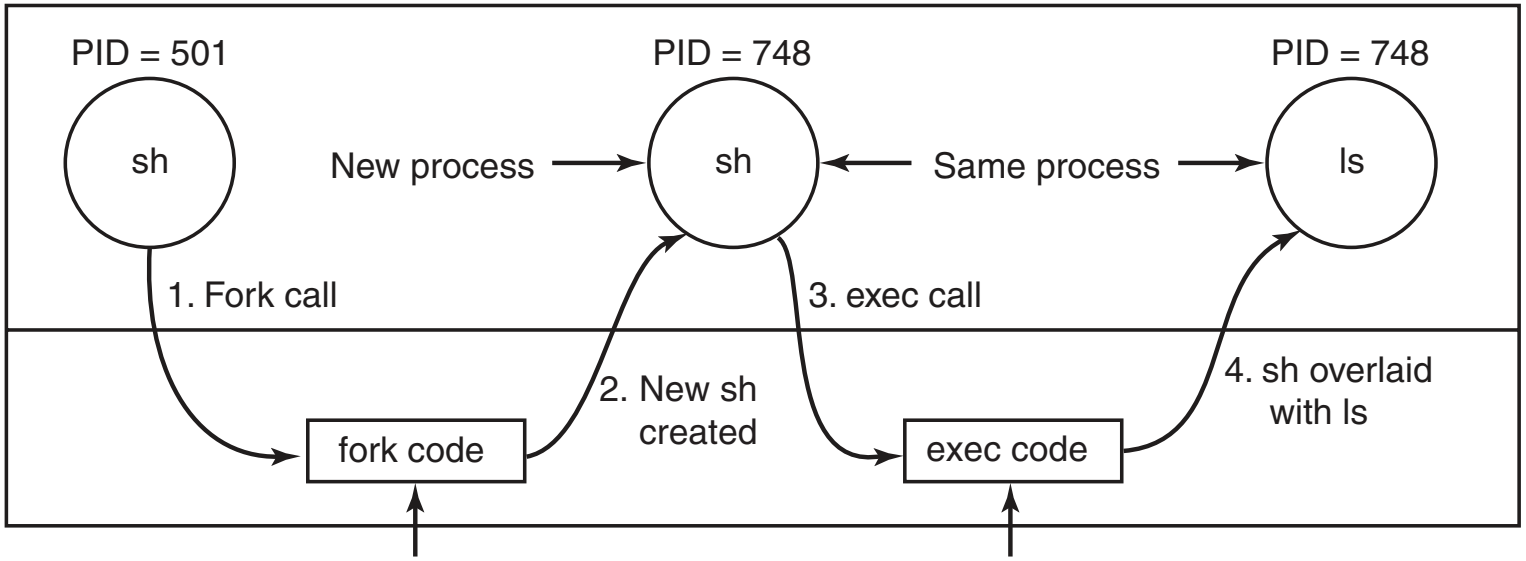
Implementação de processos no Linux

UFABC

- O núcleo do Linux representa os processos internamente como tarefas (*tasks*)
  - Representado pela estrutura `task_struct`, que é definida em `<linux/sched.h>`
  - O núcleo organiza todos os processos em uma lista duplamente encadeada de estrutura de tarefas, chamada de lista de tarefas
    - Processo identificado pelo seu identificador de processos (*PID – Process IDentifier*).
- Estrutura de tarefas possui diversos campos
  - Ocupa cerca de 1,7 KB de memória em uma máquina 32 bits
  - Contém toda informação que o núcleo tem sobre o processo

UFABC

- Exemplo de criação de um processo no Linux: usuário digita o comando `ls` no shell



Roteiro

UFABC

- Exercício 1 Uso da chamada de sistema fork.
  1. Compile e execute o programa `prog1.c`. Explique o seu funcionamento, detalhando o uso das chamadas de sistema usadas.
  2. Envie o sinal `SIGINT` ao processo filho: `$ kill -SIGINT <PID>`, em que PID é o PID do processo filho mostrado na tela. O que aconteceu nesse caso? Justifique.

UFABC

- Exercício 2 Sinais como condições de erro.
  1. Compile e execute o programa `prog2.c`. Por que o processo filho é terminado?
  2. Compile e execute o programa `prog3.c`. Por que o processo filho é terminado?
- Exercício 3 Como tratar um sinal?
  1. Compile e execute o programa `prog4.c`. Em um segunda execução, envie o sinal `SIGINT` ao processo filho. Compare as duas execuções e comente. O que é necessário fazer para tratar um sinal?
  2. Modifique o programa `prog4.c` para tratar a condição de erro do programa `prog3.c`. Foi possível tratar o sinal e continuar a execução? Justifique.

UFABC

- Exercício 4 Como mascarar um sinal?
  1. Compile e execute o programa `prog5.c`. Durante sua execução, envie o sinal `SIGQUIT` ao processo filho. Em um segunda execução, envie o sinal `SIGTERM` ao processo filho. Compare as duas execuções e comente. O que é necessário fazer para mascarar um sinal?
  2. É possível tratar ou mascarar o sinal `SIGKILL` ou `SIGSTOP`? Justifique.

UFABC

- Exercício 5 Produtor-consumidor com troca de mensagens.
  1. Compile com a opção `-o cons` o programa `prog6c.c`. Compile com a opção `-o prod` o programa `prog6p.c`. Execute os programas `cons` e `prod` em diversas combinações. Explique o funcionamento dos programas.

UFABC

- Exercício 6 Modifique os programas `prog6c.c` e `prog6p.c` para implementar o sistema produtor-consumidor com N mensagens da figura do slide a seguir, mas fazendo uso de caixa postal.

UFABC

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
        int item;
        message m;                              /* message buffer */

        while (TRUE) {
                item = produce_item( );         /* generate something to put in buffer */
                receive(consumer, &m);          /* wait for an empty to arrive */
                build_message(&m, item);        /* construct a message to send */
                send(consumer, &m);             /* send item to consumer */
        }
}


void consumer(void)
{
        int item, i;
        message m;

        for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
        while (TRUE) {
                receive(producer, &m);          /* get message containing item */
                item = extract_item(&m);        /* extract item from message */
                send(producer, &m);             /* send back empty reply */
                consume_item(item);             /* do something with the item */
        }
}
```