FATEC ZONA LESTE

CURSO DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

**Augusto Rocha Nascimento**

**Brendo Maciel**

**João Henrique Maciel**

**Kelvin Vicente Costa**

UNIX

Comportamento do Sistema Operacional

São Paulo – SP

2018

**Augusto Rocha Nascimento**

**Brendo Maciel**

**João Henrique Maciel**

**Kelvin Vicente Costa**

UNIX

Comportamento do Sistema Operacional

|  |  |
| --- | --- |
|  | O presente trabalho tem como objetivo apresentar o comportamento do sistema operacional UNIX®  Orientador: Prof. M. Sc. Leandro Colevati dos Santos |

São Paulo – SP

2018

Sumário

[Breve Histórico 1](#_Toc515987604)

[Kernel 1](#_Toc515987605)

[Processos 2](#_Toc515987606)

[O que são 2](#_Toc515987607)

[Propriedades 2](#_Toc515987608)

[Criação 3](#_Toc515987609)

[Finalização 3](#_Toc515987610)

[Zumbie e órfãos 3](#_Toc515987611)

[Daemons 3](#_Toc515987612)

[Sinais 4](#_Toc515987613)

[Deadlock 4](#_Toc515987614)

[Condições 4](#_Toc515987615)

[Estratégias para lidar 4](#_Toc515987616)

[Prevenção 4](#_Toc515987617)

[Sistema de Arquivos 5](#_Toc515987618)

[Gerenciamento de memória 5](#_Toc515987619)

[Bloqueios 5](#_Toc515987620)

[Arquivos mapeados 6](#_Toc515987621)

[Proteção 6](#_Toc515987622)

[I/O 6](#_Toc515987623)

[Classes de dispositivos 6](#_Toc515987624)

[Visão Geral 6](#_Toc515987625)

[Drivers de dispositivos de carácter 7](#_Toc515987626)

[Interface de dispositivos de bloco 7](#_Toc515987627)

[Drivers de dispositivos de bloco 7](#_Toc515987628)

# Breve Histórico

O UNIX é um sistema operacional desenvolvido para ser usado em qualquer tipo de dispositivo. As versões atuais do sistema rodam em diversos hardwares: de supercomputadores a smartphones, passando por servidores, laptops e smartwatches.

Em 1964, os Laboratórios Bell, a General Eletrics e o MIT juntaram-se para criar um novo sistema operacional chamado Multics. O time de desenvolvedores era formado por nomes como  Ken Thompson, Dennis Ritchie, Douglas McIlroy e Peter Weiner.

Mais tarde, em 1969, Ken Thompson iniciou uma versão simplificada do Multics escrita em Assembly. O sistema foi então batizado de UNIX.

Em 1973, Ken Thompson e Dennis Ritchie re-escreveram o sistema utilizando a linguagem C.

# Kernel

Tecnicamente falando, o Linux é um kernel Unix verdadeiro, mesmo que não seja um sistema Unix completo uma vez que não inclui todas as aplicações; entretanto, como a maior parte destas aplicações se encontram sob licença GPL, elas podem ser instaladas em qualquer sistema baseado em Linux  
  
Todas as versões comerciais são derivadas ou da SVR4 ou da 4.4BSD, e todas tendem a seguir alguma padronização como POSIX e CAE  
  
A maioria dos kernels comerciais são monolíticos, com exceção do Mac OS X e do GNU Hurd, ambos derivados do Carnegie-Mellon’s Mach, que utilizam micro-kernel  
  
As versões SVR4.2 e Solaris podem carregar e descarregar porções do código do kernel dinamicamente (tipicamente, drivers de dispositivos) usualmente chamados módulos  
  
Estas versões também são organizadas como um conjunto de threads que podem ser associados a um programa ou simplesmente executar alguma função do kernel  
  
Solaris e Mach 3.0 possuem kernels preemptivos; SVR4.2/MP introduz alguns pontos fixos de preempção como forma de limitar a capacidade preemptiva  
  
Diversas versões do Unix tomam vantagem de sistemas com multiprocessadores  
Um exemplo disto, são sistemas NUMA, onde o sistema pode utilizar múltiplos processadores e cada pode controlar quaisquer tarefas

Esconde todos os detalhes de baixo nível sobre a organização física do computador de aplicações que sejam executadas pelo usuário, se um programa deseja utilizar um recurso de hardware, é necessário solicitar ao kernel que avalia a requisição e, se escolher por permitir a utilização do recurso, interage com os componentes de hardware em nome do programa; para reforçar este mecanismo, o kernel proíbe que programas do usuário interajam de forma direta com componentes de hardware em baixo-nível ou acessar endereços arbitrários de memória, introduzindo dois modos de execução: user mode e kernel mode

O MINIX foi um dos primeiros sistemas do tipo UNIX baseado em um projeto de micronúcleo, tendo o gerenciamento de memória e o sistema de arquivos empurrados para os processos de usuário   
O núcleo tratava da troca de mensagens entre processos e outras poucas coisas

Por conta da arquitetura 8088, os drivers de I/O estavam no núcleo; o sistema de arquivos e o gerenciador de memória eram executados como dois processos do usuário em separado

# Processos

## O que são

Processos são como containers que possuem todas as informações do programa em execução e que pode receber ou enviar mensagens entre eles

## Propriedades

* PID: Todo processo possui um ID associado a ele, que funciona como um identificador único e permite que o processo seja referenciado
* PPID: ID do processo pai, quase todos os processos possuem um processo responsável por sua criação
* TTY: Identificador do terminal que iniciou o processo, chamado de terminal de controle; com exceção de daemons, todos os processos possuem um TTY associado
* UID: User ID, identificada o usuário dono do processo e define as permissões que o processo terá
* ARGS: o comando e argumentos que executam o processo

## Criação

A criação de processos possui 2 passos: *fork* e *exec*

Todo processo é criado a partir de uma chamada de sistema *fork*

O fork cria uma cópia do processo que o chamou; o novo processo (filho) herda tudo que que o processo pai possui sendo quase uma cópia idêntica, mudando o PID e PPID

O próximo passo é o *exec*, que substitui o processo atual por um novo, fazendo com que o processo que realiza a chama termine e o novo tome seu lugar

## Finalização

Todo processo finaliza com um código de saída que varia de 0 a 255

É possível controlar estes códigos de saída ao manipular processos, mas é ideal seguir a padronização

## Zumbie e órfãos

Um processo se torna zumbi quando finaliza sem que o processo pai realize o wait

Para que seja possível ler o código de saída, o kernel não descarta imediatamente tão logo realize a finalização esperando que o processo pai realize a chamada do wait  
Todo processo passa por esse estado, mesmo que por um curto período, entre o momento de finalização e o momento em que o processo lê o código de saída

Um processo se torna órfão quando, ainda em execução, o processo pai finaliza

O processo filho é então “adotado” pelo processo inicial, normalmente chamado init

O PPID de um processo orfão é 1

Isso pode acontecer devido uma falha no processo pai ou então quando o usuário decide desassociar um processo da sessão de usuário, como é o caso de daemons

## Daemons

Processo que é executado nos “bastidores” e que não está associado a um TTY

Banco de dados e servidores web são exemplos de daemons

O processo init é um daemon especialmente importante para o sistema, sendo o primeiro criado e é considerado “avô” de todos os outros processos

Novos daemons podem ser criados tanto a partir do init quando “abandonando” os processos filhos intencionalmente   
Por convenção, daemons possuem a letra ‘d’ no final de seus nomes   
Exemplos: syslogd, sshd, httpd

## Sinais

Os sinais são as mensagens que ocorrem entre os processos

O comando KILL e CTRL+C (INTERRUPT) nada mais são do que sinais que solicitam que o processo finalize

Caso o processo possua uma exceção de SIGKILLou SIGSTOP, ele pode “prender” essa requisição para efetuar alguma ação ou mesmo ignorar

Processos zumbis, processos bloqueados e o processo init ignoram sinais de finalização ou de interrupção

# Deadlock

## Condições

1. Exclusão mútua: Cada recurso está atualmente associado a exatamente um processo ou está disponível
2. Posse e espera: Processos atualmente de posse de recursos que foram concedidos antes podem solicitar novos recursos
3. Não preempção: Recursos concedidos antes não podem ser tomados à força de um processo, sendo necessário que o processo libere o recurso explicitamente
4. Espera circular: Deve haver uma lista circular de dois ou mais processos, cada um deles esperando por um processo de posse do membro seguinte da cadeia

Todas estas condições devem estar presentes para que o deadlock ocorra

## Estratégias para lidar

1. Ignorar: Quem sabe ele ignore também? (Algoritmo do avestruz)
2. Detecção e recuperação: Tome medidas cabíveis quando ocorrer
3. Alocação cuidadosa de recursos
4. Prevenção: Negar estruturalmente uma das quatro condições

## Prevenção

Evitar deadlock é essencialmente impossível sendo necessárias informações de solicitações futuras, ainda não conhecidas

Porem se conseguirmos que ao menos uma das condições de deadlock jamais seja satisfeita, o deadlock se torna estruturalmente impossível  
O POSIX determina uma série de regras para a criação e manipulação de threads a fim de evitar a ocorrência de deadlocks

# Sistema de Arquivos

O Filesystem Hierarchy Standard define os principais diretórios e seus conteúdos em um sistema operacional UNIX tendo a versão mais recente 3.0 sido publicada em 19 de Março de 2015

Com essa padronização torna-se possível que tanto software quanto usuário saibam de antemão onde estão os arquivos e diretórios

Temos então duas distinções independentes entre os arquivos:

* compartilhável x não-compartilhável
* variável x estático

|  |  |  |
| --- | --- | --- |
| Exemplos: | compartilhável | não-compartilhável |
| variável | /usr  /opt | /etc  /boot |
| estático | /var/mail  /var/spool/news | /var/run  /var/lock |

O FHS determina que alguns diretórios são obrigatórios (e.g. /bin e /boot) e outros são opcionais (e.g. /home e /root incluindo o conteúdo destes diretórios  
Um exemplo disto é que o diretório /bin não deve possuir sub-diretórios e determina quais comandos devem existir sob este diretório

O FHS possui um escopo limitado não interferindo no local de armazenagem de arquivos locais e nem em problemas de endereçamento onde o armazenamento do arquivo necessita ser coordenado por múltiplas partes

# Gerenciamento de memória

## Bloqueios

Garante a permanência de porções de espaços de endereçamento e tradução fixa entre endereço virtual e endereço fixo e não sendo herdado no *fork*.  
Toda memória bloqueada por um processo é liberada no *exec* ou na finalização do processo.

## Arquivos mapeados

Provê mecanismo que permite ao processo acessar arquivos incorporando seus dados diretamente em seu espaço de memória fazendo com que, desta forma, estes dados possam ser manipulados como memória.

Se um arquivos for mapeado por mais de um processo, seu conteúdo será compartilhado pelo processos.  
Caso este mapeamento permita escrita, os dados escritos no objeto de memória através do endereçamento de um processo aparecerá nos endereçamentos de todos os processos que utilizem este objeto.

## Proteção

Quando um objeto é mapeado, os acessos de diversas aplicações podem resultar em sinais

Dentro disto, o SIGBUS é utilizado para indicar um erro na utilização do objeto mapeado e o SIGSEGV é usado para indicar ou que uma proteção foi violada ou mal-uso de um endereço

# I/O

## Classes de dispositivos

Existem 2 classes:

* bloco: adequado para discos, fitas e fita DEC que trabalham, ou podem trabalhar, com blocos de endereçamentos de 512-bytes
* carácter: possuem interface mais direta, já que a maior parte do trabalho é realizado pelo próprio driver; exemplos: teclados e portas seriais

## Visão Geral

As chamadas de sistema *open* e *creat* têm como propósito configurar as entradas em 3 tabelas de sistemas separadas

A primeira destas tabelas (u\_ofile) é armazenada na área de dados *u* por processo do sistema, sendo indexada pelo descritor de arquivo que é retornado pelo *open* ou *create* e é acessado durante a operação que abra o arquivo (e.g.: *read* e *write*)

Cada entrada possui apenas um ponteiro à entrada coystema, já que a mesma instância de um arquivo aberto deve ser compartilhada entre diversos processos que podem resultar de *forks* após a abertura do arquivo

Entretanto, há, no máximo, uma entrada por arquivo na tabela de inodes

Um arquivo pode entrar na tabela de inodes não apenas por sua abertura, mas também por estar no diretório atualmente acessado por algum processo ou por se tratar de um arquivo especial contendo a montagem atual do file system

## Drivers de dispositivos de carácter

A tabela cdevsw especifica as rotinas presentes a estes dispositivos

Cada um deles provê 5 rotinas: *open*, *close*, *read*, *write*, e *special-function*, sendo que todas estas funções devem estar presentes

Caso uma destas rotinas seja ignorada, a entrada na tabela pode ser dada como nulldev ou nodev (caso deva ser considerado como erro)

## Interface de dispositivos de bloco

A utilização deste tipo de dispositivo é intermediada por uma coleção de rotinas que gere um set de buffers contendo as imagens dos blocos de dados de vários dispositivos, tendo como principal propósito assegurar que diversos processos que acessem um mesmo bloco mantenham a consistência de dados no bloco além de, como propósito secundário, aumentar a eficiência do sistema mantendo cópias dos blocos mais acessos

## Drivers de dispositivos de bloco

A tabela bdevsw contém os nomes das interfaces e dos dispositivos e, assim como com os dispositivos de carácter, os drivers podem suprir rotinas *open* e *close*

As rotinas *read* e *write* não são separadas e possuem uma rotina *strategy* que é chamado com um ponteiro para um buffer header como argumento   
O *buffer header* contém uma *read/write* *flag*, o endereçamento, o número do bloco, um contador de palavras negativo e o número do dispositivo

O papel da rotina *strategy* é realizar a operação de acordo com a requisição informada no *buffer header*