



FACULDADE DE TECNOLOGIA DA ZONA LESTE



SISTEMAS OPERACIONAIS I

LINUX

Robson Henrique Ferreira	1110481823026
Luiz Fernando Geraldo dos Santos	1110481823051
Isis Paloma Viegas Cellini dos Santos	1110481923049
Fernanda Pinheiro Reis	1110481823022
Bruno Bega Harnik	1110481823052

Prof.: Leandro Colevati

São Paulo
2019

Introdução

O Linux não é um sistema operacional, mas sim um kernel (núcleo) monolítico capaz de executar diferentes sistemas operacionais, tanto para servidores como para computadores pessoais. A partir desse núcleo, diversos sistemas operacionais foram desenvolvidas, como o Linux Mint, Mandriva, Kalango e Kurumin (mais recentes), Debian, Fedora, Ubuntu entre outros. A combinação de um projeto GNU de sistema operacional com o kernel Linux recebe o nome de GNU/Linux.



Criado em 1991 por Linus Torvalds, foi inspirado no Minix (um sistema operacional, similar ao Unix, desenvolvido em C e Assembly) que segue a filosofia Unix:



- Everything is a process; if it's not a process, it's a file - os periféricos são entendidos como "files" — Virtual Filesystem Switch (VFS)
- One tool to do one task
- Three standard I/O channel
- Combine tools seamlessly
- Plain text preferred CLI, not GUI
- Modular, designed to be repurposed by others
- Provide the mechanism, not the policy

Controvérsia do nome GNU/Linux

O nome "GNU/Linux" é uma disputa entre a comunidade do software livre com a comunidade do software open-source. Basicamente, a comunidade *Open-Source* propõe que a combinação de kernel e sistema operacional seja chamada apenas de Linux, acreditando que isso seja mais reconhecível por parte do público. Já os que defendem a utilização do termo *GNU/Linux* (*Free Software Foundation*), acreditam que essa seja uma forma mais adequada para atribuir devidamente os créditos ao projeto GNU, criado em 1984 com o intuito de desenvolver um sistema operacional livre – a criação do núcleo Linux possibilitou essa relação, porém ambos possuem grande importância.

Introdução	2
Processos	4
Tipos de Processos	5
Comunicação de processos	7
Escalonamento de processos	8
Threads	10
Threads do Kernel	12
Deadlocks	12
Sistemas de sincronismo oferecidos pelo kernel do Linux	13
Outras ferramentas de análise	14
Kernel	15
I/O	19
Sistemas de Arquivos	20
Tipos de Sistemas de arquivos	23
Gerenciamento de Memória	24
Curiosidades	26
Referências Bibliográficas	27

Processos

Os sistemas baseados em Unix necessitam de que haja um processo existente para que seja feita uma cópia exata do mesmo, a qual será atribuída a nova tarefa a ser executada – esse procedimento é chamado de *forking* (bifurcação). Nesse ponto, o espaço de endereço do processo filho é sobrescrito com os dados do novo processo e recebe uma nova identificação.

O processo pai concede metade de seu tempo restante de execução (quantum) ao processo filho. Esta é uma técnica de proteção do sistema que faz com que um usuário crie um processo filho que executa o mesmo código do processo pai, isso previne que cada processo filho receba um novo quantum integral e que um usuário crie vários processos em background ou execute diversas sessões *shell*. Quando um processo é encerrado normalmente, o programa retorna um *exit status* (informação de saída) ao processo pai.

Os processos executados no *kernel* Linux possuem um conjunto de características, como:

- proprietário do processo;
- estado (em execução, aguardando);
- prioridade e
- recursos alocados.

Essas informações são gerenciadas de forma organizada por meio de chaves identificadoras.

Através do proprietário do processo, o sistema pode, através das permissões fornecidas pelo mesmo, saber quem tem acesso ao processo em questão, para isso o sistema utiliza os identificadores UID e GID. Estes são identificadores pelos quais o sistema linux gerencia usuários e grupos deles, onde cada usuário precisa pertencer a pelo menos um grupo. Desta forma, como um processo pertence a um usuário, automaticamente este pertence a um grupo (grupo deste usuário). os valores de UID e GID podem variar entre 0 e 65536, podendo ser um valor maior, dependendo da distribuição. Usuários ROOT tem o valor 0, neste caso para que um usuário ou processo possam ter acessos a nível root, o seu GID deve ser 0.

Os processos possuem um PID (*Process Identifier*), um número de identificação único atribuído pelo sistema para cada novo processo gerado. Já aos identificadores dos "processos pai", dá-se o nome de PPID (*Parent Process Identifier*).

```

top - 00:44:44 up 1:02, 1 user, load average: 0,34, 0,12, 0,11
Tarefas: 233 total, 1 em exec., 188 dormindo, 0 parado, 0 zunbt
%CPU(s): 1,2 us, 0,7 sls, 0,0 nl, 98,2 oc, 0,0 ag, 0,0 lh, 0,0 ls, 0,0 tr
KB mem : 2017144 total, 132508 livre, 1023728 usados, 860808 buff/cache
KB swap: 2097152 total, 2050876 livre, 46276 usados, 664892 mem dispon.

```

PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TEMPO	COMANDO
2501	cachaca	20	0	494964	40572	20972	S	1,0	2,0	2:28.34	xorg
5649	cachaca	20	0	51464	4164	3456	R	0,7	0,2	0:00.31	top
34	root	20	0	0	0	0	I	0,3	0,0	0:05.43	kworker/1:1-eve
2713	cachaca	20	0	3357052	183268	49612	S	0,3	9,1	1:08.90	gnome-shell
2737	cachaca	20	0	378080	9784	8036	S	0,3	0,5	0:05.24	dbus-daemon
5630	cachaca	20	0	803072	37356	27784	S	0,3	1,9	0:00.67	gnome-terminal-
1	root	20	0	225808	8700	6344	S	0,0	0,4	0:07.69	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.00	kthreadd
3	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	rcu_par_gp
6	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/0:0H-kb
8	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	mm_percpu_wq
9	root	20	0	0	0	0	S	0,0	0,0	0:00.49	ksoftirqd/0
10	root	20	0	0	0	0	S	0,0	0,0	0:02.05	rcu_sched
11	root	rt	0	0	0	0	S	0,0	0,0	0:00.05	migration/0
12	root	-S1	0	0	0	0	S	0,0	0,0	0:00.00	idle_inject/0
14	root	20	0	0	0	0	S	0,0	0,0	0:00.00	cpuhp/0
15	root	20	0	0	0	0	S	0,0	0,0	0:00.00	cpuhp/1
16	root	-S1	0	0	0	0	S	0,0	0,0	0:00.00	idle_inject/1
17	root	rt	0	0	0	0	S	0,0	0,0	0:00.05	migration/1
18	root	20	0	0	0	0	S	0,0	0,0	0:00.98	ksoftirqd/1
20	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/1:0H-kb
21	root	20	0	0	0	0	S	0,0	0,0	0:00.00	kdevtmpfs
22	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	netns
23	root	20	0	0	0	0	S	0,0	0,0	0:00.00	rcu_tasks_kthre
24	root	20	0	0	0	0	S	0,0	0,0	0:00.00	auditd
25	root	20	0	0	0	0	S	0,0	0,0	0:00.00	khungtaskd
26	root	20	0	0	0	0	S	0,0	0,0	0:00.00	oom_reaper
27	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	writeback
28	root	20	0	0	0	0	S	0,0	0,0	0:00.21	kcompactd0
29	root	25	5	0	0	0	S	0,0	0,0	0:00.00	ksmd
30	root	39	19	0	0	0	S	0,0	0,0	0:00.00	khugepaged

Figura 3: Comando *top* exibe uma série de informações sobre os processos.

No Linux, os processos podem ser executados de duas maneiras: em *background* (plano de fundo) ou em *foreground* (primeiro plano). Enquanto um processo é executado em primeiro plano, outro processos podem continuar suas execuções em segundo plano.

Tipos de Processos

Processos Interativos

São processos que dependem de alguém para que sejam executados, normalmente o usuário – não são processos relacionados às funções do sistema operacional. Um processo interativo rodar em primeiro plano, ocupando o terminal que o iniciou e impedindo que outras aplicações sejam iniciadas enquanto o processo estiver em execução, como também pode rodar em plano de fundo, permitindo a entrada de novos comandos durante sua execução.

Processos Automáticos

Também chamados de processos *batch* (lote), não são conectados a um terminal. Em contrapartida, são enfileirados de forma FIFO (*first in, first out*) para suas execuções. O critério para execução de um processo automático pode ser um determinado momento, por meio do comando *at*, ou quando a carga total do sistema é baixa o suficiente para aceitar novas tarefas, por meio do comando *batch* – por padrão as tarefas são colocadas em fila e são executadas quando a carga do sistema é inferior a 0.8.

Processos Daemons

São processos que rodam continuamente e normalmente estão relacionados a alguma funcionalidade do sistema. Na maior parte das vezes, são iniciados junto com o sistema operacional e aguardam em *background* até que sejam requisitados. São exemplos de *Daemons* processos que realizam logs, que controlam recursos de entrada, conexões de rede, entre outros.

Processo de inicialização

Quando um computador é iniciado, a BIOS fornece uma interface mínima com detecção de periféricos para controlar o processo de inicialização.

A BIOS segue procurando por dispositivos, unidades de CD, USB e unidades de disco rígido (de acordo com a ordem pré configurada). Quando há um Linux instalado em um disco rígido do sistema, a BIOS busca pelo *Master Boot Record* (MBR) – gravação mestra de inicialização - que é iniciado no primeiro setor do disco rígido, carregando seus conteúdos na memória e passando o controle para ele.

O primeiro passo da inicialização do sistema é a inicialização do processo pai de todos os processos – ou "processo avô". Este processo é encarregado de inicializar tudo que o sistema precisa, definir caminhos, iniciar o *swapping*, além de definir o clock do sistema.



Figura 4: O comando *ps tree* permite visualizar a árvore de processos. É possível notar que o *systemd* é o "pai" de todos os processos.

GRUB

O GRUB (ou GNU GRUB) é um método de *"direct loading"* (carregamento direto), que pode ser instalado e, então, carregado pelo MBR na inicialização. Ele permite que o usuário escolha inicializar as diferentes opções disponíveis para cada sistema operacional instalado em uma unidade ou partição.

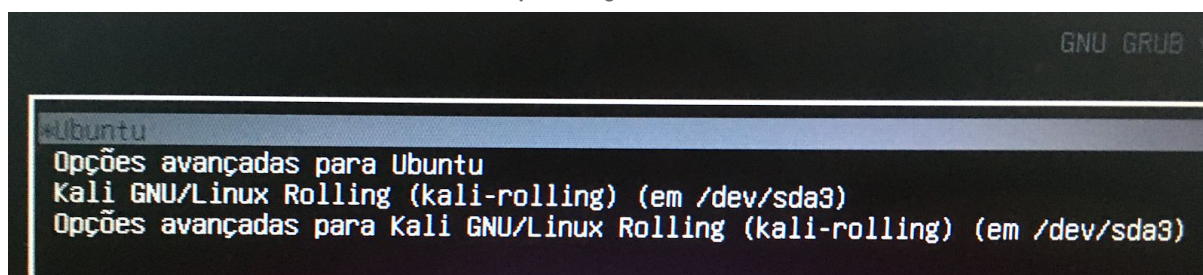


Figura 5: Tela do **GNU GRUB** de escolha dos sistemas operacionais instalados.

Comunicação de processos

Através de sinais, é possível realizar a comunicação entre diferentes processos e também manipular a suas execuções. Ao receber um sinal com as instruções sobre como responder a este sinal, o sistema pode executar ações de acordo com suas rotinas. Como por exemplo:

STOP - Interromper processo até que se receba o sinal CONT.

CONT - Informa que o processo interrompido pode ser continuado.

SEGV - Informa erros de endereços de memória.

TERM - Termina completamente o processo, deixa de existir.

KILL - Função de eliminar o processo em situações críticas (travamentos).

O comando Kill, quando usado sem parâmetros, assume a execução do comando TERM, o comando kill é usado da seguinte forma: KILL -SINAL PID. Por exemplo:

KILL - STOP 4220 → Interrompe temporariamente a execução do processo de ID 4220.

KILL - CONT 4220 → Continua a execução do processo de ID 4220 interrompido anteriormente.

Caso seja necessário enviar o sinal a todos os processos, pode ser usado o valor -1 na posição do PID.

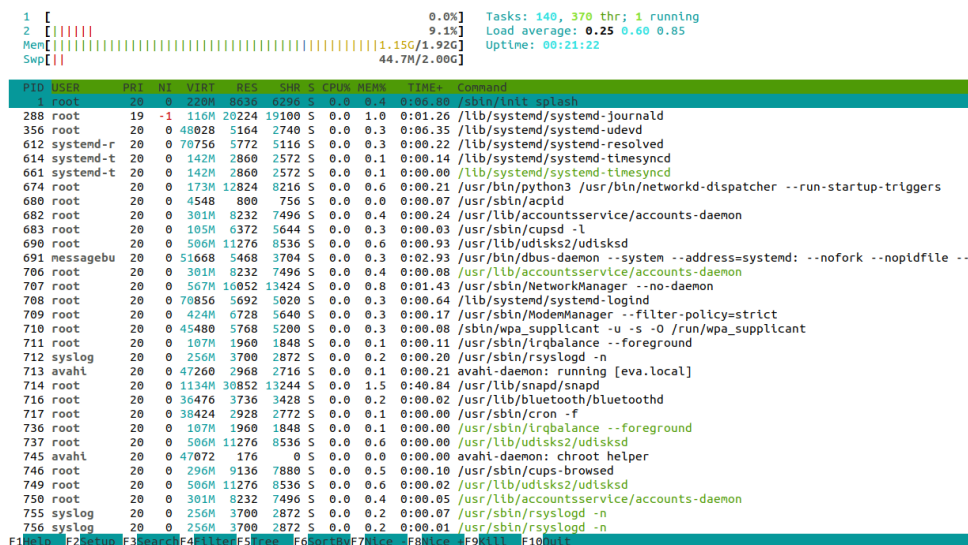


Figura 6: O programa *htop*, além de exibir os processos em uma interface mais amigável, possibilita a execução de alguns sinais aos processos de maneira simplificada.

Escalonamento de processos

O kernel Linux não distingue processos interativos de processos batch, só os difere de processos em tempo real. Seguindo o padrão dos escalonadores UNIX, o linux concede prioridade maior aos processos I/O bound se comparados aos CPU bound, fornecendo assim melhor tempo de resposta a aplicações interativas (ex: terminais de comando).

Este escalonador é baseado em compartilhamento de tempo (quantum), preempção por tempo. Este tempo é dividido em (epochs), quando um processo é criado, este recebe o quantum, e diferentes processos podem receber diferentes valores de quantum (que é a duração da epoch), e é um múltiplo de 10 milissegundos e inferior a 100 milissegundos.

Ao estourar o quantum, o estado do processo é salvo e os recursos são direcionados a outro processo (troca de contexto), este procedimento é transparente ao processo principal e promove escalonamento preemptivo.

O kernel Linux também trabalha com escalonamento por prioridade estática e dinâmica. Sendo a priorização estática utilizada exclusivamente em processos em tempo real podendo variar de valor entre 1 a 99, onde a prioridade destes processos é definida pelo usuário e não pode ser modificada pelo sistema. Somente usuários de

nível maior de privilégio podem criar e definir processos em tempo real.

Com a prioridade dinâmica, sendo aplicada a processo batch e interativos recebendo valor de 100 a 139. Na prioridade dinâmica, o sistema monitora o comportamento dos processos e ajusta a sua prioridade, a fim de equilibrar o uso do processador pelos processos pendentes. Quando um processo tem ocupado o processador por um tempo considerado “longo”, a sua prioridade é reduzida. E de forma inversa, aqueles processos que tem seu acesso ao processador considerado “curto”, recebe um aumento de prioridade para as próximas atividades de escalonamento.

Os processos de prioridade dinâmica são executados somente quando não há mais processos de prioridade estática (em tempo real) na fila de execução.

O escalonador linux utiliza 2 políticas para o escalonamento de processos:

- First in First out, aplicada a processo em tempo real. quando este tipo de processo é criado, é inserido ao final da fila que corresponde a sua prioridade (estática). Quando este processo entra em execução, permanece em execução até que: outro processo de prioridade estática maior que a sua entre na fila, o processo libere voluntariamente o processador de prioridade igual a sua, ou o processo é terminado ou bloqueado em operações de entrada e saída ou sincronismo.
- Round Robin, aplicado a processos de prioridade dinâmica, o qual é inserido ao final da fila que corresponde a sua prioridade (dinâmica) quando é criado. Ao entrar em execução, permanece até que: o seu quantum seja esgotado, sendo então o processo interrompido, seu estado salvo e o mesmo é inserido no final de sua fila. Um processo de prioridade superior chega a sua fila de execução. O processo libera voluntariamente o processador para outro processo de prioridade igual ou quando processo termina ou bloqueia para operação de entrada e saída ou sincronismo.

Threads

Nas versões de kernel após 2.4 e antes da 2.6 houve uma implementação de “threads” inicialmente conhecida como LinuxThreads porém essa versão falhou em muitos aspectos e não atingiu o padrão POSIX (Interface Portável entre Sistemas Operativos). A POSIX define as interfaces de programação de aplicações (APIs), juntamente com as shells de comando e interfaces utilitárias, para compatibilidade de softwares variantes de Unix e outros SOs. O LinuxThreads tentou inicialmente utilizar threads no espaço de usuário, mas por inúmeros problemas na implementação com a chamada de sistema clone acabou não tendo sucesso. Exemplos de utilização ANTES da implementação da versão 2.6:

- `fork()`: Criava um processo filho que obtém um PID diferente e absorve o PID do processo pai como seu PPID(parental process). Os dois processos executam os mesmo código (o objetivo do `fork` é possuir o máximo de informações incomum apesar de não copiar tudo, por exemplo, os limites de recursos) porém poderia ocasionar variáveis indesejadas no pai já que o filho pode alocar o endereço do pai. Além do novo PID e PPID os retornos eram diferentes, na qual o filho retorna 0 e o pai retorna o PID do filho.

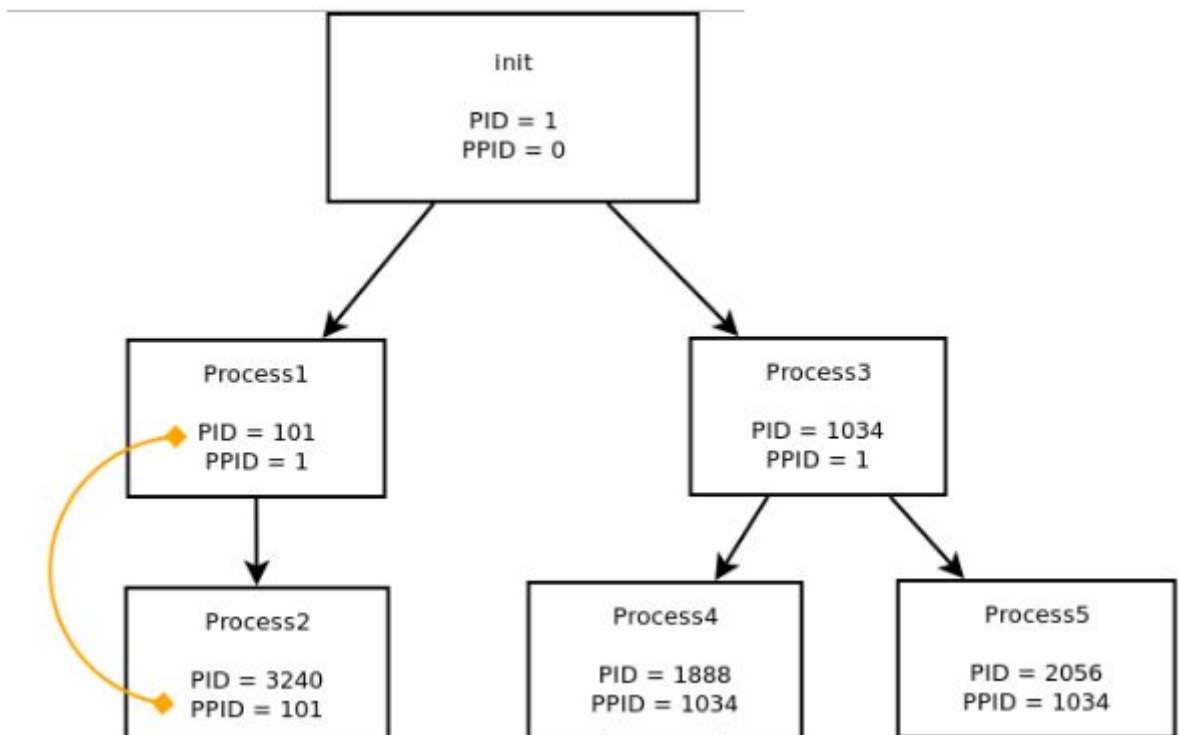


Figura 7: Hierarquia de processos

- clone(): como fork, cria um novo processo porém ao contrário do fork é possível usar flags para indicar quais partes serão compartilhadas entre o processo pai e o filho como endereços de memórias ou arquivos.

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Figura 8: Flags "clone"

Já na versão 2.6 do kernel conhecida como NPTL (Native POSIX Thread Library) foi implementada no kernel uma biblioteca para o suporte de threads, agora todas threads de um processo compartilham do mesmo PID e é utilizado o comando pthread_create para criação de novas threads. Alguns exemplos dos novos comando de manipulação e criação de threads na nova versão: (em C)

- Criação de uma thread:

```
1 int pthread_join (pthread_t thread, void **thread_return);
```

- Obtenção do identificador da thread:

```
1 pthread_t pthread_self (void);
```

- Indicação de bloqueio da thread para execução da próxima:

```
1 int sched_yield (void);
```

```
top - 00:46:03 up 1:04, 1 user, load average: 0,36, 0,20, 0,14
Threads: 579 total, 1 em exec, 535 dormindo, 0 parado, 0 zumbi
%CPU(s): 0,8 us, 1,3 sis, 0,0 ni, 97,9 oc, 0,0 ag, 0,0 lb, 0,0 is 0,0 tr
KB mem : 2017144 total, 286528 livre, 1015832 usados, 714784 buff/cache
KB swap: 2097152 total, 2039828 livre, 57324 usados, 685188 mem dispon.

PID USUARIO PR NI VIRT RES SHR S %CPU %MEM TEMPO COMANDO
6066 cachaca 20 0 51828 4696 3572 R 2,6 0,2 0:01.11 top
691 message+ 20 0 51668 5368 3632 S 0,3 0,3 0:03.64 dbus-daemon
2561 cachaca 20 0 494720 39776 20176 S 0,3 2,0 2:25.54 Xorg
2713 cachaca 20 0 3357316 181908 48176 S 0,3 9,0 1:15.71 gnome-shell
5506 root 20 0 0 0 0 I 0,3 0,0 0:01.79 kworker/0:2-eve
6049 cachaca 20 0 803892 37316 27940 S 0,3 1,0 0:00.71 gnome-terminal-
1 root 20 0 225808 8668 6324 S 0,0 0,4 0:07.74 systemd
2 root 20 0 0 0 0 S 0,0 0,0 0:00.00 kthreadd
3 root 0 -20 0 0 0 I 0,0 0,0 0:00.00 rcu_gp
4 root 0 -20 0 0 0 I 0,0 0,0 0:00.00 rcu_par_gp
6 root 0 -20 0 0 0 I 0,0 0,0 0:00.00 kworker/0:0H-kb
8 root 0 -20 0 0 0 I 0,0 0,0 0:00.00 m_percpu_wq
9 root 20 0 0 0 0 S 0,0 0,0 0:00.50 ksoftirqd/0
10 root 20 0 0 0 0 I 0,0 0,0 0:02.17 rcu_sched
11 root rt 0 0 0 S 0,0 0,0 0:00.05 migration/0
12 root -51 0 0 0 S 0,0 0,0 0:00.00 idle_inject/0
14 root 20 0 0 0 S 0,0 0,0 0:00.00 cpuphp/0
15 root 20 0 0 0 S 0,0 0,0 0:00.00 cpuphp/1
16 root -51 0 0 0 S 0,0 0,0 0:00.00 idle_inject/1
17 root rt 0 0 0 S 0,0 0,0 0:00.06 migration/1
18 root 20 0 0 0 S 0,0 0,0 0:01.00 ksoftirqd/1
20 root 0 -20 0 0 I 0,0 0,0 0:00.00 kworker/1:0H-kb
21 root 20 0 0 0 S 0,0 0,0 0:00.00 kdevtmpfs
22 root 0 -20 0 0 I 0,0 0,0 0:00.00 netns
23 root 20 0 0 0 S 0,0 0,0 0:00.00 rcu_tasks_kthre
24 root 20 0 0 0 S 0,0 0,0 0:00.00 kauditd
25 root 20 0 0 0 S 0,0 0,0 0:00.00 khungtaskd
26 root 20 0 0 0 S 0,0 0,0 0:00.00 oom_reaper
27 root 0 -20 0 0 I 0,0 0,0 0:00.00 writeback
28 root 20 0 0 0 S 0,0 0,0 0:00.22 kcompactd0
29 root 25 5 0 0 S 0,0 0,0 0:00.00 ksm
30 root 39 19 0 0 S 0,0 0,0 0:00.00 khugepaged
```

*Figura 9: Comando **top -H** exibe, além das informações sobre os processos e consumo de memória, detalhes sobre threads em seu cabeçalho.*

Threads do Kernel

O kernel executa processos em segundo plano criando threads de kernel (são processos que só existem no kernel). O que diferencia as threads de kernel dos demais processos é que elas não possuem um endereço (o ponteiro é NULL) elas operam apenas no kernel e não interagem com o usuário mesmo sendo tarefas agendadas e preemptivas. Essas threads são geradas quando o sistema inicia e perduram até uma reinicialização, criando um loop na qual a thread é chamada, executa sua função e retorna para um estado de aguardo (dormente).

Deadlocks

O kernel do linux disponibiliza vários mecanismos de sincronização para evitar alguns tipos de erros, tais como:

- **Erro de região crítica:** Um código que deverá ser executada sobre exclusão mútua. Por exemplo: threads que estão usando a mesma variável no espaço de endereço que o processo pai, então ambos os códigos estão atualizando e compartilhando o mesmo recurso. É extremamente necessário proteger essa área crítica para evitar enganos no código e/ou sistema.
- **Condição de "corrida":** É uma falha que ocorre quando a região crítica não é protegida de maneira adequada ocasionando um erro no tempo ou na ordem dos eventos afetando o programa e podendo ocasionar em um resultado errôneo.
- **Deadlock:** Quando não é usado o mecanismo de sincronização de maneira correta ocasionando uma situação em que o processo fica à espera de recursos que jamais serão liberados .

Sistemas de sincronismo oferecidos pelo kernel do Linux

- **Operações atômicas:** São instruções que executam sem interrupções. O kernel oferece duas interfaces: Operações sobre inteiros e Operações em bit individuais. é usado para assegurar que o compilador não atualize de maneira errada a variável. Pode ser utilizado para evitar deadlocks e condições de corrida, pois só uma operação atômica pode ser executada por vez e de maneira ininterrupta. Usada quando dois ou mais processos compartilham uma variável que precisa ser alterada.

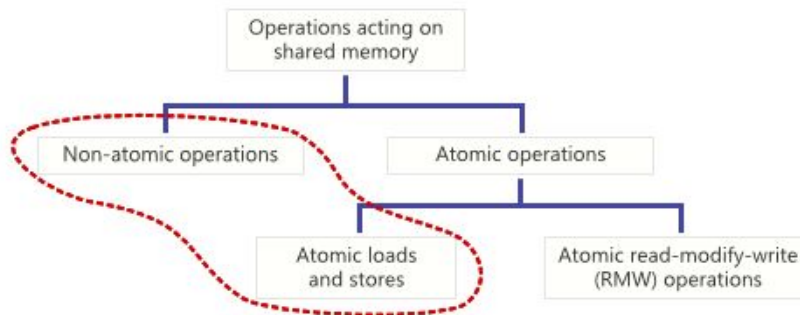


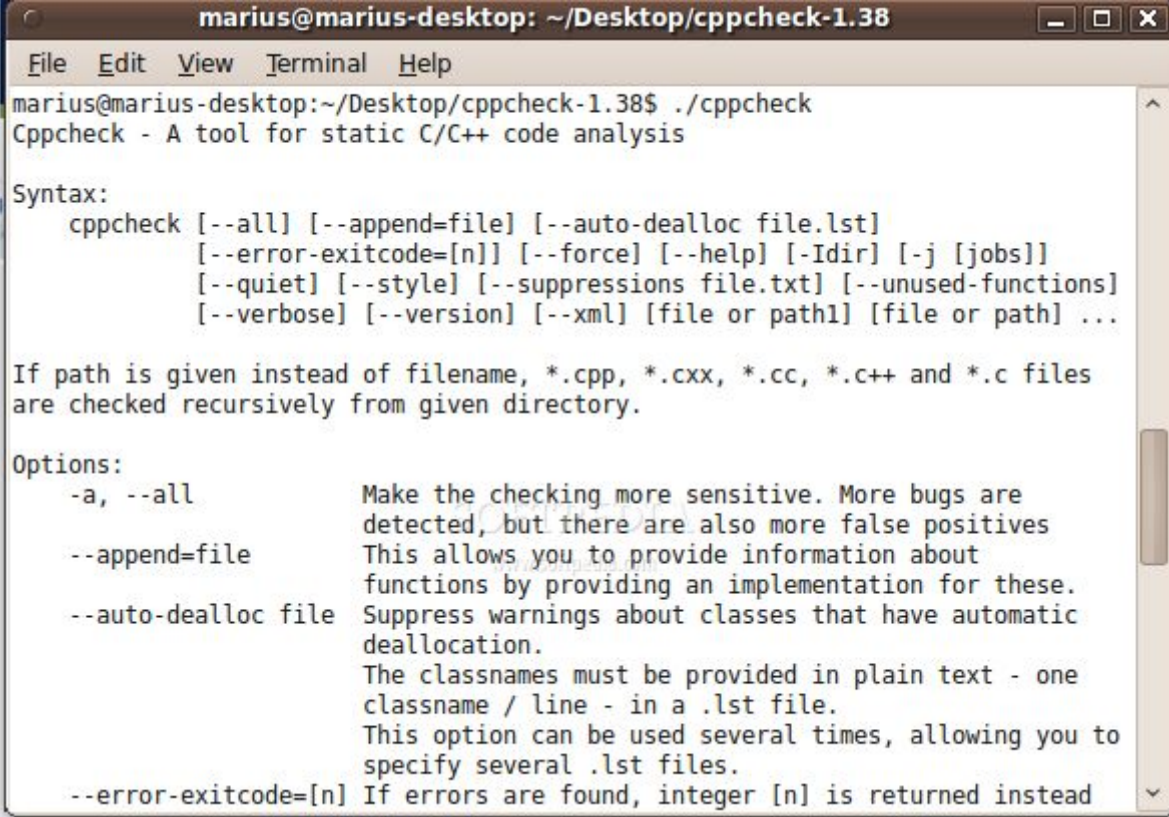
Figura 10: Operações atômicas

- **Semáforos:** Usado para trocar recursos entre threads de maneira segura. Quando um processo está tentando acessar o semáforo que não está disponível o semáforo coloca o processo na fila de espera (FIFO) e entra em modo de suspensão, quando o semáforo é liberado uma das tarefas da fila será chamada.
- **Spinlock:** É usada principalmente em casos em que o código não pode entrar no modo de suspensão. Spinlock usa o mesmo conceito de semáforo mas usando o estado de "disponível" e "fechado". Se uma thread tenta obter o spinlock enquanto este está fechado, a thread entra em um loop esperando o spinlock se tornar disponível, quando disponível thread pode imediatamente continuar sua execução. Assim prevenindo que mais de uma thread entre na região crítica por vez. Apesar de tentar prevenir erros o próprio spinlock pode provocar um deadlock, pois em linux o spinlock não é recursivo e se um processo tentar pedir um recurso na qual ele já obtém abrindo outra chamada de spinlock ele entrara em um loop esperando que ele mesmo devolva o recurso, gerando um deadlock.
- **Bloqueios em sequência(seqlock):** Este é um mecanismo de sincronização muito útil para fornecer um bloqueio leve e escalável para o cenário em que há muitos leitores e alguns escritores, é um bloqueio sequencial que permite acesso de

recurso para leitores, porém cada leitor deve verificar a existência de conflitos com um escritor. Cada escritor que adquire um bloqueio sequencial, aumenta o contador e adquire um spinlock, quando o escritor terminar ele libera o spinlock para dar acesso aos outros escritores para incrementar o contador de bloqueio sequencial novamente.

Outras ferramentas de análise

O “tratamento” de deadlocks se baseia em prevenção e além dos sistemas de sincronismo, existem outras ferramentas à parte que tentam detectar erros que os compiladores não detectam, analisando o código e procurando por comportamentos incomuns ou estruturas inseguras, um exemplo de ferramenta é o CPPCHECK que analisa códigos em C/C++ buscando por comportamentos inesperados ou bloco de códigos potencialmente nocivos.

A screenshot of a terminal window titled "marius@marius-desktop: ~/Desktop/cppcheck-1.38". The window shows the command prompt where the user has entered ". /cppcheck". Below the command, the help text for cppcheck is displayed. It includes the syntax for the tool, a note about recursive checking of directories, and a list of options with their descriptions. The options listed are: -a, --all; --append=file; --auto-dealloc file; and --error-exitcode=[n]. The descriptions for these options explain their functions, such as making checking more sensitive, providing information about functions, suppressing warnings about classes, and returning an integer instead of a non-zero exit code.

```
marius@marius-desktop: ~/Desktop/cppcheck-1.38
File Edit View Terminal Help
marius@marius-desktop:~/Desktop/cppcheck-1.38$ ./cppcheck
Cppcheck - A tool for static C/C++ code analysis

Syntax:
  cppcheck [--all] [--append=file] [--auto-dealloc file.lst]
           [--error-exitcode=[n]] [--force] [--help] [-Idir] [-j [jobs]]
           [--quiet] [--style] [--suppressions file.txt] [--unused-functions]
           [--verbose] [--version] [--xml] [file or path1] [file or path] ...

If path is given instead of filename, *.cpp, *.cxx, *.cc, *.c++ and *.c files
are checked recursively from given directory.

Options:
  -a, --all           Make the checking more sensitive. More bugs are
                      detected, but there are also more false positives
  --append=file       This allows you to provide information about
                      functions by providing an implementation for these.
  --auto-dealloc file Suppress warnings about classes that have automatic
                      deallocation.
                      The classnames must be provided in plain text - one
                      classname / line - in a .lst file.
                      This option can be used several times, allowing you to
                      specify several .lst files.
  --error-exitcode=[n] If errors are found, integer [n] is returned instead
```

Figura 11: CPPCHECK para identificar comportamentos inesperados

Os deadlocks devem ser prevenidos já que quando ocorrem não são tratados pelo kernel pois o tratamento seria muito custoso. Então em caso de deadlocks o algoritmo do avestruz é aplicado.

Kernel

O kernel gerencia os recursos de um sistema, sejam processos, memória ou dispositivos de hardware. Esse gerenciamento serve para intermediar o acesso a esses recursos entre vários usuários (IBM).

O kernel Linux foi desenvolvido no início dos anos 90 sob a GPL (General Public License), o que permitiu que tivesse a colaboração de milhares de desenvolvedores pelo mundo e que não fosse explorado comercialmente, se tornando muito popular em um período curto de tempo. Ele tem várias versões e ao passar do tempo são lançadas ainda outras novas, para melhoria do sistema e correção de falhas. As versões são representadas por 3 números separados por pontos, indicando a série (os dois primeiros) e a versão do kernel dentro dessa série (o último). Quando o segundo número for ímpar, a série ainda está em desenvolvimento e pode ser considerada instável. Quando esse número for par, a série é estável e segura (ESCOLALINUX)

Em sua estrutura, o kernel Linux é dividido em camadas, agrupadas dentro do próprio kernel, com subsistemas distintos detentoras de funções diferentes. O Linux, portanto, pode ser considerado um kernel “monolítico”, por agrupar todos os serviços básicos, como comunicação, E/S e gerenciamento de memória e processo.

Uma das características mais marcantes do Linux é a portabilidade. Ele tem a capacidade de ser compilado e executado em uma ampla variedade de processadores e plataformas, independentemente da arquitetura empregada. Por exemplo, ele roda tanto em processos com MMU (Memory Management Unit) quanto em processos que não oferecem essa funcionalidade (IBM)

Para complementar essa portabilidade, o Linux também oferece suporte à inclusão e remoção de métodos de software durante a execução do núcleo, chamados “módulos de kernel dinamicamente carregáveis”. Esses módulos são bibliotecas que contenham funções específicas que poderão ser usadas por um software específico que execute no sistema geral.

Na figura 12, pode-se observar as subdivisões contidas dentro do Kernel Linux, onde há a interface de chamada de sistema, o gerenciamento de processos, o gerenciamento de memórias, o sistema

de arquivo virtual, a pilha de redes, os drivers do dispositivo e a camada de código dependente da arquitetura.

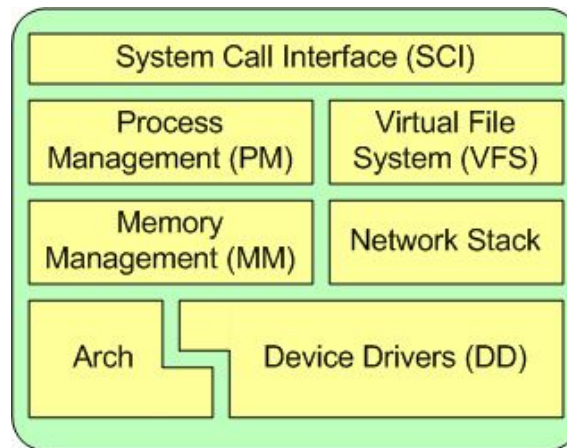


Figura 12: Subdivisões do Kernel Linux

A “Interface de chamada do sistema” é responsável pela multiplexação e demultiplexação de chamada de funções e por meio dele o usuário pode solicitar serviços de processos e memórias.

O “Gerenciamento de processos” do Linux é focado na execução desses processos e usa para o gerenciamento o API através da Interface de chamada do sistema para criar, parar, executar ou sincronizar processos. Esse gerenciamento, no Linux, é realizado utilizando o planejador $O(1)$, que opera em tempo constante independentemente do número de processos competindo pelos recursos da CPU. Ou seja, o tempo não é alterado em função do número de encadeamento de funções.

O “Gerenciamento de memória” no Kernel Linux faz seu trabalho utilizando o que chama de “páginas”, que são conjuntos de endereços de memória de 4KB (buffers). Ainda, há um esquema de alocação chamado “Slab”, que usa as páginas de 4KB como base para verificar e/ou modificar as estruturas de memória de acordo com páginas completas, parcialmente usadas ou vazias, permitindo que o uso da memória seja otimizado dinamicamente de acordo com a necessidade do sistema geral. Para completar, o gerenciamento de memória pode utilizar a tática de “troca” (swap) em ocasião de falta de memória por uso de vários usuários. O swap faz com que parte das páginas em buffers sejam movimentadas direto para o disco rígido e utilizadas por demanda. Novamente, o fato de o Linux poder ser executado em sistemas com ou sem MMU permite que ele tenha uma portabilidade acima da média em nível de Kernel, sendo possível seu

uso em plataformas integradas pequenas, como o processador DragonBall da Motorola.

O “Sistema de arquivo virtual” é uma subdivisão do kernel que possibilita a abstração de interface aos sistemas de arquivos, fornecendo uma camada de troca entre a Interface de chamada de sistema e os sistemas de arquivo, fato que pode ser verificado na figura 2.

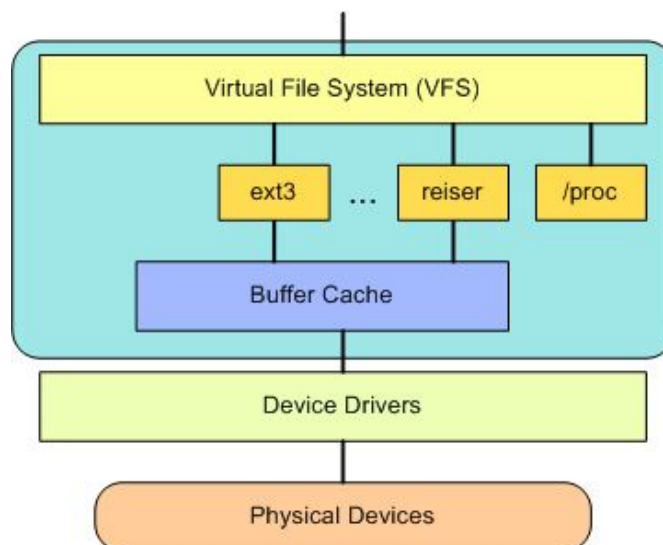


Figura 13: Sistema de arquivo virtual.

De acordo com a imagem, no topo do sistema está justamente as funções comuns da API (close, open, read, write) que serão chamadas pelo usuário. Na parte inferior estão as abstrações de sistema que serão utilizadas em um nível mais baixo de interpretação do chamado. Então há uma intermediação através de um buffer para otimização do acesso dos dispositivos físicos, sendo a última camada de intermediação os próprios drivers de dispositivos.

A subdivisão chamada “Pilha de Redes” tratará de fornecer a interface necessária para intermediar a relação usuário-protocolos de rede, sendo estes protocolos o IP, TCP ou UDP. É utilizado uma camada denominada “camada de soquetes” para gerenciar essas conexões e mover dados entre os terminais.

Nos “Drivers do Dispositivo” são encontrados códigos-fonte que poderão estar no kernel Linux para que possam ser utilizáveis. De acordo com subdiretórios os drivers são divididos pelos dispositivos suportados (como Bluetooth, I2C e serial).

Por fim, a última subdivisão do kernel Linux é a camada onde é armazenado o código dependente da arquitetura. Apesar de esse kernel ter alta portabilidade, há funções específicas que devem ser

guardadas dentro do próprio kernel para que possa ser suportado e desempenhado de modo eficaz.

I/O

O gerenciamento de entrada e saída do Linux ocorre de maneira totalmente integrada ao sistema de arquivos, já que cada dispositivo E/S é associado a um ou mais arquivos chamados “arquivos especiais”. Portanto, o acesso a esses dispositivos é realizado exclusivamente por meio desses arquivos e o usuário consegue localizá-los em seus diretórios como se fossem arquivos comuns e utilizá-los através de system calls pela Interface de chamada de sistema.

As operações realizadas no Linux em dispositivos de Entrada e Saída ocorrem utilizando sequências de bytes, possibilitando o envio do mesmo dado para diferentes dispositivos de saída e, assim, permitir a portabilidade do Linux para diferentes arquiteturas computacionais de maneira descomplicada.

Os dispositivos de Entrada e Saída devem ser sempre associadas aos seus respectivos drivers (que são os arquivos especiais) e no Linux eles podem ser integrados e desintegrados com o sistema funcionando, não havendo a necessidade de uma reinicialização do kernel. Esses drivers seguem dois padrões: o orientado a bloco e o orientado a caractere. O primeiro padrão visa permitir maior agilidade na troca de dados entre o dispositivo e o processador, por garantir que grandes blocos de dados sejam transferidos a cada contato, e é mais utilizado para dispositivos como HD, CD-ROMs e outros discos. Já o segundo padrão é mais preciso na troca de dados e por isso é mais lento, enviando/recebendo caractere a caractere. Esse é utilizado para impressoras, por exemplo.

O buffer cache pode ser associado às comunicações realizadas pelo padrão orientado a bloco para que os dados possam ser enviados de maneira mais ágil ao processador e visando a melhoria de desempenho do sistema.

```

cachaca@eva:~$ dmesg | grep Linux
[ 0.000000] Linux version 5.0.0-36-generic (build@lgw01-amd64-060) (gcc vers
ion 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1)) #39~18.04.1-Ubuntu SMP Tue Nov 12 11:
09:50 UTC 2019 (Ubuntu 5.0.0-36.39~18.04.1-generic 5.0.21)
[ 0.359763] ACPI: Added _OSI(Linux-Dell-Video)
[ 0.359763] ACPI: Added _OSI(Linux-Lenovo-NV-HDMI-Audio)
[ 0.359763] ACPI: Added _OSI(Linux-HPI-Hybrid-Graphics)
[ 0.411764] pps_core: LinuxPPS API ver. 1 registered
[ 2.254579] Linux agpgart interface v0.103
[ 2.283444] usb usb1: Manufacturer: Linux 5.0.0-36-generic ehci_hcd
[ 2.303454] usb usb2: Manufacturer: Linux 5.0.0-36-generic ehci_hcd
[ 2.304787] usb usb3: Manufacturer: Linux 5.0.0-36-generic uhci_hcd
[ 2.305820] usb usb4: Manufacturer: Linux 5.0.0-36-generic uhci_hcd
[ 2.306842] usb usb5: Manufacturer: Linux 5.0.0-36-generic uhci_hcd
[ 2.307918] usb usb6: Manufacturer: Linux 5.0.0-36-generic uhci_hcd
[ 2.308925] usb usb7: Manufacturer: Linux 5.0.0-36-generic uhci_hcd

```

Figura 14: Comando `dmesg | grep` exibe os dispositivos reconhecidos

Sistemas de Arquivos

Uma das grandes vantagens da utilização de sistemas Linux se dá pelo seu amplo suporte a diversos tipos de sistemas de arquivos diferentes. O que promove a compatibilidade com uma ampla gama de dispositivos além de cada tipo de sistema de arquivos prover melhor desempenho para tipo de aplicação específica (ex: Clusters, bancos de dados, Storage, etc).

Definição:

Um sistema de arquivo consiste na organização dos dados e metadados em uma unidade de armazenamento. Podemos definir os sistemas de arquivos como protocolos de comunicação (Como TCP/IP, por exemplo), que realizam a ordenação e interpretação de um determinado fluxo de dados e os decodificam.

Em sistemas Linux, quando realizamos a “montagem” de um dispositivo de armazenamento, estamos associando este dispositivo a um determinado sistema de arquivos.

Em sua grande maioria, o código dos sistemas de arquivos se localiza no espaço do kernel, (com exceção de alguns componentes de espaço de usuário.)

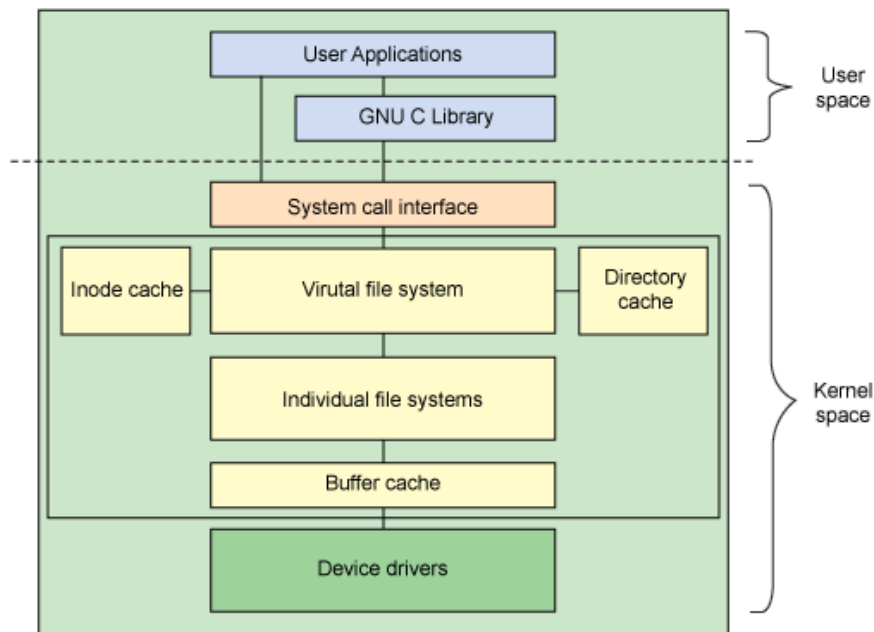


Figura 15: componentes do sistema de arquivo

A ilustração acima representa o relacionamento entre os componentes do sistema de arquivos, o espaço de usuário contém algumas aplicações, que são as responsáveis de fornecer a interface de acesso a manipulação de arquivos (rotinas de leitura e escrita, por exemplo).

O Virtual File system é responsável por fornecer um conjunto de interfaces, desta forma cada tipo de sistemas de arquivos irá importar um conjunto de interfaces, que são esperadas pelo VFS. também possui informações de todos os sistemas de arquivos suportados no momento (fornecida pelo Kernel) e dados de montagem de sistemas de arquivos.

O cache de buffer contém o buffer com as requisições entre os sistemas de arquivos e os dispositivos que estes controlam (dispositivos em bloco, como os setores de um disco, onde os dados que chegam até ele formam o bloco). Como exemplo, as requisições de leitura e escrita que são enviadas ao driver, migram pelo buffer, sendo armazenadas em cache, o que faz com que estas sejam acessadas mais rapidamente, ao invés de se realizar o acesso diretamente no hardware. Isto torna o sistema mais eficiente, fornecendo os dados destas operações para cada sistema de arquivos individual.

```

cachaca@eva:~$ sudo dumpe2fs -h /dev/sda2
[sudo] senha para cachaca:
dumpe2fs 1.44.1 (24-Mar-2018)
Filesystem volume name: <none>
Last mounted on: /
Filesystem UUID: de47fb13-9028-4f62-b63b-1ae4307de27b
Filesystem magic number: 0xEF53
Filesystem revision #: 1 (dynamic)
Filesystem features: has_journal ext_attr resize_inode dir_
ile huge_file dir_nlink extra_isize metadata_csum
Filesystem flags: signed_directory_hash
Default mount options: user_xattr acl
Filesystem state: clean
Errors behavior: Continue
Filesystem OS type: Linux
Inode count: 7331840
Block count: 29296875
Reserved block count: 1464843
Free blocks: 23764409
Free inodes: 7054112
First block: 0
Block size: 4096
Fragment size: 4096
Group descriptor size: 64
Reserved GDT blocks: 1024
Blocks per group: 32768
Fragments per group: 32768
Inodes per group: 8192
Inode blocks per group: 512
Flex block group size: 16
Filesystem created: Thu Aug 22 19:55:34 2019
Last mount time: Mon Nov 18 23:42:03 2019
Last write time: Mon Nov 18 23:41:57 2019
Mount count: 113
Maximum mount count: -1
Last checked: Fri Aug 23 03:38:44 2019
Check interval: 0 (<none>)
Lifetime writes: 211 GB

```

*Figura 16: **dumpe2fs** fornece informações sobre o sistema de arquivos*

O Sistema Linux define todos os sistemas de arquivos como objetos comuns, sendo estes (superblock, inode, dentry e file).

Superblock: Presente na raiz de cada sistema de arquivos, descreve o estado do sistema de arquivos (nome, tamanho, referência para o dispositivo de bloco). Normalmente é armazenado no dispositivo de armazenamento, mas também pode ser criada em tempo real. As operações da superblock definem o conjunto de funções para a manipulação de inodes.

Inode: Cada objeto manipulado por um sistema de arquivos (diretório ou pasta) contém metadados para gerenciamento de objetos, operações possíveis em cada um. Cada sistema de arquivos possui seus métodos inodes, que irão fornecer a abstração comum ao VFS.

Dentry: realiza a conversão entre nome de inode, possui cache que mantém os objetos usados mais recentemente.

File: representa um arquivo aberto (seu estado, operações de deslocamento de dados).

Tipos de Sistemas de arquivos

Atualmente o sistema de arquivos padrão utilizado em sistemas operacionais Linux é o EXT4, mas há a possibilidade de incluir módulos de suporte a outros sistemas de arquivos no kernel. É possível verificar quais os sistemas de arquivos suportados no arquivo `/proc/filesystems`. Dentre os sistemas de arquivos mais utilizados em distribuições Linux destacam-se:

EXT3 (third extended FileSystem): Adotado como padrão a partir do ano de 2001. Utilizando o registro (journal) o que ampliou a confiabilidade e a possibilidade de recuperação do sistema em caso de desligamentos repentinos. Suporta 16 TB de tamanho máximo e 2TB de tamanho por arquivo, além de suportar 32 mil subdiretórios por diretório.

EXT4 (fourth extended FileSystem): Adotado como padrão a partir do ano de 2008, suportando 1 Exabyte de tamanho máximo e 16TB de tamanho por arquivo, além de um número ilimitado de subdiretórios.

XFS (Extended FileSystem): Usado como padrão em algumas distribuições a partir de 20014, é um sistema 64, mas compatível com sistemas de 32 bits, suporta 16 Exabytes de tamanho total e 8 Exabytes por arquivo, é um sistema de alto desempenho.

JFS (Journaled FileSystem): Introduzido pela IBM em 1990 é um sistema 64 bits que suporta 32 PetaBytes de tamanho total e 4 PetaBytes por arquivo.

ReiserFS: Introduzido em 2001, utiliza o recurso de journaling, permite tamanho máximo de 16 Tebibytes, sendo 1 Exbibyte por arquivo em sistemas 64 bits e 8 Tebibytes em sistemas de 32 bits. Sua versão atual é o Reiser 4 inserida em 2007.

VFAT (Virtual File Allocation table, extensão do FAT32 do Windows): Sem suporte a journaling, nos dias atuais seu uso é feito para a troca de dados entre sistemas Microsoft Windows e Linux que estejam armazenados no mesmo disco, permitindo a leitura e escrita de arquivo por parte dos dois sistemas operacionais.

Journaling: Sistemas que implementam este recurso, mantém um log das mudanças no sistema de arquivos antes que os dados sejam escritos no disco. Desta forma, diminui-se a probabilidade dos dados serem corrompidos em caso de travamento ou desligamento repentino, além de promover uma recuperação mais rápida, já que é necessário verificar no disco apenas os arquivos mencionados na log. o Sistemas EX3 e 4, XFS, JFS e Reiser utilizam este recurso.

Gerenciamento de Memória

Os sistemas de alocação de memória do Linux, segundo Tanenbaum, são 3: memória virtual, alocador `kmalloc` arbitrário e caches de dados persistentes no kernel.

A memória física do Linux trabalha com um sistema de paginação onde o espaço de endereçamento lógico dos processos é linear e unidirecional, e cada endereço representa uma *página*. O espaço físico de alocação é chamado de quadro (frame) e cada página pode estar, assim, associada a qualquer quadro.

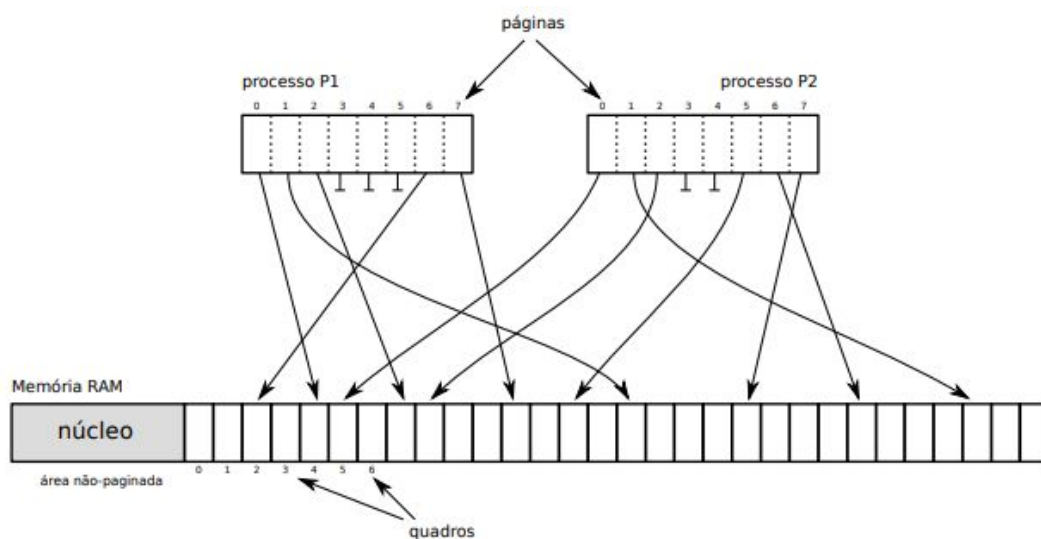


Figura 5.11: Alocação de memória por páginas.

Figura 17: Alocação de memória por páginas. Disponível em: <http://www.lcvdata.com/ads/so-livro.pdf>. Acesso em: 20/11/2019.

Um alocador divide a memória, por meio de um algoritmo, em um conjunto de páginas e permite que os recursos sejam alocados sem que haja perda de eficiência. A essa maneira de alocação, onde direciona-se uma quantidade variável de memória a um recurso dá-se o nome de alocação dinâmica.

O alocador `kmalloc` arbitrário executa o algoritmo de buddy heap, ou paginação por demanda, onde a memória é sempre reduzida ou unida em pares: se um processo é pequeno e necessita de pouca memória alocada, ele quebra a parte maior em duas páginas e assim, pode ser alocado. Quando o processo é liberado em 2 páginas “vizinhas”, estas se unem e formam uma página maior.

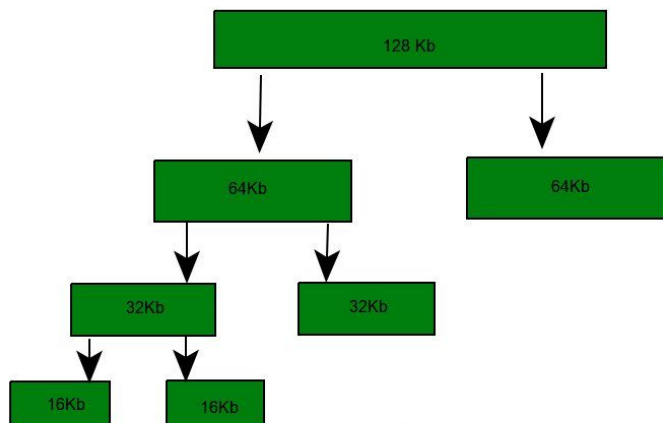


Figura 18: buddy heap. Disponível em: <https://www.geeksforgeeks.org/operating-system-allocating-kernel-memory-buddy-system-slab-system/>. Acesso em: 19/11/2019.

Quando a memória física não é suficiente para rodar todos os processos em execução, os processos fazem uso da **memória virtual**, que é relação de mutualismo entre a memória RAM e a área de disco que performa o SWAP. A memória virtual é utilizada e, depois, os processos retornam à execução de memória (RAM) de forma dinâmica, visto que os processos precisam estar na mesma para serem executados.

A técnica de *swapping*, ou escalonamento, traz o conceito de execução de tarefas por prioridades: se um processo A de prioridade mais alta necessita de recursos, o gerenciador de memória descarrega o processo B de menor prioridade na área de swapping, que pertence ao disco, para que os recursos sejam realocados às necessidades do processo prioritário A.

Quando o prioritário A ou outro processo libera recursos (swap out), o processo B pode, finalmente, retornar à memória RAM (swap in).

Apesar de funcionar como uma extensão da memória RAM, o espaço de swapping é uma partição à parte, no Linux,

```

cachaca@eva:~$ cat /proc/meminfo
MemTotal:      2017144 kB
MemFree:       243104 kB
MemAvailable:  686408 kB
Buffers:       108740 kB
Cached:        572884 kB
SwapCached:    5796 kB
Active:        755920 kB
Inactive:      638216 kB
Active(anon):  377404 kB
Inactive(anon): 479596 kB
Active(file):  378516 kB
Inactive(file): 158620 kB
Unevictable:   128296 kB
Mlocked:       32 kB
SwapTotal:     2097152 kB
SwapFree:      2040084 kB
Dirty:         8 kB
Writeback:     0 kB
AnonPages:     837016 kB
Mapped:        158188 kB
Shmem:         144904 kB
KReclaimable:  75360 kB
Slab:          152516 kB
SReclaimable:  75360 kB
SUnreclaim:    77156 kB
KernelStack:   9264 kB
PageTables:    41448 kB
NFS_Unstable:  0 kB
Bounce:        0 kB
WritebackTmp:  0 kB
CommitLimit:   3105724 kB
Committed_AS:  5308536 kB
VmallocTotal:  34359738367 kB
VmallocUsed:   0 kB
VmallocChunk:  0 kB
Percpu:        1248 kB
HardwareCorrupted: 0 kB
AnonHugePages: 0 kB
ShmemHugePages: 0 kB
  
```


Por fim, o cache de dados persistente é um recurso não só do Linux que permite que a alocação temporária de recursos recém utilizados seja feita, o que possibilita que seus endereços de memória sejam armazenados e, quando novamente requisitados, executem o processo com mais rapidez. Por armazenar informações de recursos recém requisitados, é comum que encontremos vídeos na internet de usuários questionando o alto consumo de cache pelo Linux.

Os sistemas operacionais GNU/Linux têm parte de seus blocos de código compostos por interfaces criadas para o ambiente desktop, também de código aberto. As principais interfaces gráficas utilizadas em sistemas operacionais Linux são Unity (depreciado, substituído agora pelo GNOME), GNOME, KDE, XFCE, LXDE, CINNAMON e MATE.

O GNOME (GNU Network Object Model Environment) é um projeto de software de código aberto de interface de usuário. Após o decaimento do Unity, em 2017, passou a ser a interface gráfica do Ubuntu 18.04LS. Sua principal função é fornecer padrões mundialmente reconhecidos de usabilidade e acessibilidade para compor a interface com o usuário. O Linux Mint, famosa distribuição GNU/Linux, por sua vez, utiliza o Cinnamon, uma interface derivada do GNOME.



Referências Bibliográficas

<http://escalonamentoprocessos.blogspot.com/2010/10/escalonador-linux.html>

<https://trabalhosprontos.net/processos-e-threads-em-windows-e-linux-8548/>

<https://www.programacaoprogessiva.net/2014/09/A-Chamada-de-Sistema-fork-Como-Criar-e-Gerenciar-Processos.html>

<https://tudodeasunto.wordpress.com/threads-definicao/windows-vs-linux-threads/>

<https://www.it-swarm.net/pt/linux/diferenca-entre-fork-vfork-exec-e-clone/972067631/>

<https://kerneltweaks.wordpress.com/tag/deadlock/>

<https://0xax.gitbooks.io/linux-insides/content/SyncPrim/linux-sync-6.html>

<https://github.com/selbyk/operating-systems/blob/master/4.threads.md>

<https://delightfullylinux.files.wordpress.com/2012/06/pid.png>

<http://www.lcvdata.com/ads/so-livro.pdf>

<https://www.ibm.com/developerworks/br/library/l-linux-process-management/index.html>

<https://www.certificacaolinux.com.br/trabalhando-com-processos-no-linux/>

<http://www.inf.ufrgs.br/~asc/livro/secao94.pdf>

<https://www.infowester.com/linprocessos.php>

<http://www.makelinux.net/ldd3/chp-5-sect-5.shtml>

<http://linux.die.net/man/1/cppcheck>

https://www.researchgate.net/publication/221033382_Static_Deadlock_Detection_in_the_Linux_Kernel

<http://www.informit.com/articles/article.aspx?p=370047&seqNum=3>

<https://www.ibm.com/developerworks/br/library/l-linux-kernel/index.html>

<https://www.escolalinux.com.br/blog/kernel-do-linux-o-que-e-e-para-que-e-serve>

<https://www.diolinux.com.br/2015/07/sistemas-de-arquivos-populares-e-suas-caracteristicas.html>

<https://www.ibm.com/developerworks/br/library/l-linux-filesystem/index.html>

<http://guialinux.uniriotec.br/sistemas-de-arquivos/>

<https://www.infowester.com/reiserfs.php>

Billimoria, K. M. Hands-On System Programming With Linux. 2018. Packt Publishing. Birmingham, UK.

Garcia, D., Menna da Silva, Eduardo. Gerência de Memória: Conceitos e Aplicação no Sistema Linux. UNESC. Criciúma, SC.