Sistema de ficheiros

ficheiro -> coleção dados persistentes identificado pelo nome, organizado hierarquia de pastas.

- nome para utilizadores, fd para SO
- offset -> cursor
- metadata

long **ftell**(FILE *stream) -> saber o cursor do ficheiro aberto (stream)

int **fseek**(FILE *stream, long offset, int whence) -> troca posição do cursor para whence + offset recebido SEEK_END(fim), SEEK_SET(início), SEEK_CUR(valor original),

int **fflush**(FILE *stream) -> grava buffer temp write no disco

Hard link: conceito de cópia de um ficheiro (sem cópia real dos dados), ficheiro só é removido quando o último hardlink for apagado, entradas em diversos diretórios apontam p/ o mesmo ficheiro (mesmo inode) porque do ponto de vista do sistema, são ind. arquivos
unlink() elimina um link e não apaga forçosamente um ficheiro

Symbolic Link: ficheiro que contém o caminho de acesso para o ficheiro original, se o original for apagado o link fica quebrado

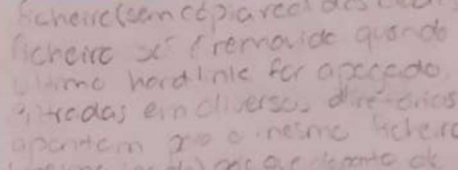
stat -> vê informação do ficheiro

strace -> vê chamadas sistemas que os programas estão a executar

chmod -> permite mudar permissões de acesso ao ficheiro

mount -> liga a raiz de novo sistema de ficheiros a um diretório do sistema de ficheiros raiz

Sistema Ficheiros CP/M -> tipo notas



cada bloco possui por omissão 1KB
mapa de blocos c/ 16 entradas
dimensão máxima de um ficheiro = 16KB

Sistema Ficheiros MS-DOS

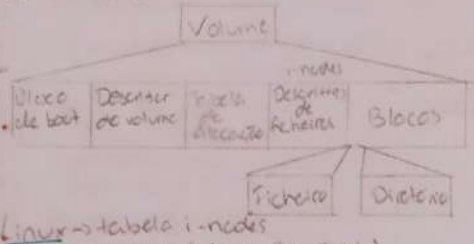
evolução de CP/M, em vez de mapa de blocos por ficheiro, tem tabela blocos global partilhada por todos os ficheiros deu origem ao nome do sistema de ficheiros FAT

três secções:
tabela de alocação, diretoria com os nomes dos ficheiros no sistema de ficheiros, restante espaço dividido em blocos para conter os dados do ficheiro

FAT -> vetor composto por 2^n inteiros de n bits
entradas FAT: 0 -> bloco livre
≠ diferente 0 -> bloco faz parte de um ficheiro
tabelas destas dimensões não são possíveis manter em RAM permanentemente

Organização c/ i-nodes

manter descritor ficheiro num descritor próprio de cada ficheiro, chamado i-node, pode guardar vários atributos (tipo de ficheiro, dono, datas de últimos acessos, permissão, dimensão ...)
é a estrutura que está entre as entradas dos diretórios que referenciam o ficheiro e os seus blocos de dados
vantagem: podem existir várias entradas de diretório a apontar para o mesmo ficheiro
i-nodes são guardados numa estrutura especial de tamanho fixo antes dos blocos de dados



Linux -> tabela i-nodes
Windows -> MFT (Master File Table)
nº max ficheiros numa partição = nº max i-nodes nessa tabela

Descritor do Volume -> possui informação geral de descrição do sistema de ficheiros é geralmente replicado noutros blocos (a informação nele guardada é de importância fundamental), se se corromper pode ser impossível recuperar a informação do sistema de ficheiros

Unix -> bloco especial denominado superbloco

NTFS -> ficheiro especial

FAT -> a informação em causa é descrita diretamente no setor de boot

Tabela de Alocação (Tabela Blocos Livres) -> mantém um conjunto de estruturas necessárias à localização de blocos livres, pode ser um simples bitmap (um bit por bloco de operação, indica se o respetivo bloco está livre ou ocupado), esta tabela desacoplada dos i-nodes tem vantagens: é possível ter estruturas muito mais densas, pode-se organizar a tabela de blocos livres em várias tabelas de menor dimensão para blocos adjacentes

Todos os sistemas de ficheiros definem um conjunto de estruturas em memória volátil para gerir a informação persistente mantida em disco

Tabela de ficheiros abertos do processo: descritor p/ cada ficheiro aberto que referencia a tabela global de ficheiros abertos, índice nessa tabela é fd

Tabela de ficheiros abertos global/sistema: informação relativa a ficheiro aberto: cursor com posição atual, modo como ficheiro foi aberto

pq duas tabelas?
garantir isolamento processos, permitir partilha ficheiros sempre que necessário

Objeto ficheiro -> criado quando se chama open(), adicionado na tabela de descritores do processo, pode existir mais do que um para o mesmo ficheiro

Ler Ficheiro
localizar bloco de disco onde está informação a aceder, calcular ordem bloco aceder p/ determinar onde ir buscar índice respetivo, obter inode e eventuais blocos de índice p/ chegar nº bloco, ler bloco p/ memória principal, calcular endereço memória dentro bloco mapeado em memória q contém a informação a aceder

Estruturas em m...
objetivo: criação ... virtual, a suporta... ativas, em simult...
neste modelo: cad... por conjunto de op... dependendo do s... que estão arma...
cada entrada ... tem informações ... ficheiros, ter vetor ... obter a informaç...

Journaling
objetivo da cam... Block Device (JBD... ficheiro fica em... (atualmente apenas ... evita que uma op... parcialmente real... forma atómica
regista os blocos m... de registar no sis...
o EXT3 verifica j... pendentes q preci... o sistema mais...

Tarefas e proc...

Entre exit e wai... depois de wait ap...

exec -> substitui ... do processo onde ... contido num ficheiro ... programa substitui... dados substituídos ... ficheiro, pilha é es... apenas para 1ª ins... programa
quando tem suces...

UID -> user ID desi...
GID -> cada user ... grupo ID

Mudar UID/GID
setuid(int) set... ou usando exec

pthread_create(&...
pthread_t tid -> pont...
attr -> atributos da...
function -> função ...
arg -> ponteiro p/ ...
devolve 0 se corre...

pthread_exit(void ...
pthread_join(pthre...

Exclusão Mútu...

limitações trincos:
só servem para pro... suficientemente exp... problemas sincroniza...
ativa -> processo fica p... à espera de uma e...

int **pthread_mutex_**...
pthread_mutexattr_t ...
int **pthread_mutex_l**...
int **pthread_mutex_u**...

Trincos leitura e escrita
usados quando estru... com leitores e escr... que mutex e por ...

Estruturas em memória VFS

objetivo: criação sistema ficheiro comum, virtual, q suporta vários sistemas ficheiros ativos, em simultâneo

neste modelo: cada ficheiro manipulado por conjunto de operações diferente, dependendo do sistema ficheiros nativo que estão armazenados

cada entrada designada VFS superblock, tem informações sobre cada sistema ficheiros, tem vetor ponteiros p/ funções q sabem obter a informação do sistema de ficheiros

Journaling

objetivo da camada do núcleo Journaling Block Device (JBD) é impedir q sistema ficheiro fica em estado inconsistente (atualmente apenas usado por EXT3)

evita que uma operação escrita seja parcialmente realizada → efetuada de forma atómica

regista os blocos modificados no journal antes de registar no sistema de ficheiros, ao iniciar o EXT3 verifica journal se há operações pendentes q precisam ser aplicadas, torna o sistema mais ineficiente

Tarefas e processo

Entre exit e wait processo diz-se zombie depois de wait o processo é totalmente esquecido

exec → substitui espaço de endereçamento do processo onde é invocado por aquele contido num ficheiro executável:
programa substituído pelo programa no ficheiro, dados substituídos pelos dados iniciais do ficheiro, pilha é esvaziada, instruction pointer aponta para 1ª instrução do main do novo programa
quando tem sucesso n tem retorno

UID → user ID do sistema (0 para root)

GID → cada user pertence a um grupo com um grupo ID

Mudar UID/GID
setuid(int) setgid(int)
ou usa nas exec

pthread_create(&tid, attr, function, arg)
pthread_t tid → ponteiro p/ identificador da tarefa
attr → atributos da tarefa
function → função a ser executada
arg → ponteiro p/ parâmetros dados à função
devolve 0 se correr bem

pthread_exit(void *value_ptr)

pthread_join(pthread_t thread, void ** value_ptr)

Exclusão Mútua

limitações trincos:
só servem para proteger secções críticas e n são suficientemente expressivos p/ resolver outros problemas sincronização, podem provocar espera ativa → processo ficar preso num ciclo repetidamente à espera de uma condição

int pthread_mutex_init(pthread_mutex_t *A, pthread_mutexattr_t *attr)

int pthread_mutex_lock(pthread_mutex_t *mutex)

int pthread_mutex_unlock(pthread_mutex_t *mutex)

Trincos leitura e escrita
usados quando estrutura de dados partilhada com leitores e escritores, são mais complexos que mutex e por isso mais lentos

int pthread_rwlock_rdlock(pthread_rwlock_t *A)

int pthread_rwlock_wrlock(pthread_rwlock_t *A)

int pthread_rwlock_unlock(pthread_rwlock_t *A)

try_lock em vez de lock evita problema míngua

Semáforos

objeto do SO q mantém um contador
contador inicializado c/ nº threads q podem entrar numa secção crítica ao mesmo tempo sem se bloquearem
permite gerir recursos compartilhados entre processos

int sem_init(sem_t *sem, int pshared, unsigned value)
pshared=0 significa q semáforo só pode ser usado entre threads mesmo processo, caso contrário pode ser usado entre processos independentes
value → contador

int sem_wait(sem_t *sem)
é chamado quando uma tarefa entra numa secção crítica decrementando contador, bloqueia semáforo até contador >0

int sem_post(sem_t *sem)
é chamado quando uma tarefa sai de uma secção crítica incrementando novamente o contador

(1) sem_wait(...) antes de pthread_mutex_lock( ) pode gerar interblocagem

(2) pthread_mutex_unlock( ) antes de sem_post(...) o programa é menos eficiente mas funciona corretamente

Variáveis de condição

permite uma tarefa esperar por uma condição (booleana) q depende da ação de outra tarefa, variável associada a um trinco (uma ou mais), conjunto trinco + variáveis condição chamado monitor

pthread_cond_t

int pthread_cond_init(condition, attr)

int pthread_cond_destroy(condition)

int pthread_cond_wait(condition, mutex)
desbloqueia mutex e coloca tarefa na fila de espera associada à variável condition

int pthread_cond_signal(condition)
acorda uma das tarefas bloqueadas

int pthread_cond_broadcast(condition)
acorda todas as tarefas bloqueadas

Signals

interrupções causadas por sinais são eventos assíncronos

quando processo passa modo núcleo para utilizador, o núcleo verifica se existe signal pendente se sim a rotina de tratamento do signal é posta a correr em modo utilizador

void (*signal(int sig, void (*func)(int)))(int)
↳ redefinir tratamento de um signal

int kill(pid_t pid, int sig) → permite enviar um signal sig a processo com id pid

unsigned alarm(unsigned int segundos)
signal SIGALRM é enviado p/ processo depois de decorrerem nº segundos espec. Acesso, se o nº segundos é zero é cancelado

unsigned sleep(unsigned int segundos) → chama alarm e bloqueia-se à espera do signal

int raise(int sig) → signal especificado em input é enviado p/ próprio processo

int pause(void) → aguarda a chegada de um signal

Diferentes semânticas em Unix

System V → Ao receber o signal, o tratamento associado ao mesmo tem efeito para uma única ativação, depois disso o tratamento volta ao padrão. Usada em sistemas mais antigos

BSD → A associação signal-rotina não é desfeita após a ativação. A receção de um novo signal é inibida durante a execução de tratamento. Utilizada em sistemas mais recentes.

Funções async-signal-safe → podem ser chamadas a partir de um signal sem causar problemas de segurança ou de estabilidade no sistema, funções reentrantes, funções cuja execução não pode ser interrompida por signals

Pipes e comunicação

2 formas comunicar entre processos
memória partilhada, troca de mensagens (pipes)
Unix System V não usa pipes, usa: shared memory, message queues, semaphores

Pipes → mecanismo do Unix p/ comunicar entre processos, tem um descritor equivalente aos fd's dos ficheiros
o processo fica bloqueado quando escreve num pipe cheio e quando lê de um pipe vazio

int pipe(int fds[2])
- fds[0] fd para ler
- fds[1] fd para escrever

int dup(int fd) duplica fd de forma a que aponta para o mesmo ficheiro que fd, tem mesmo modo de acesso que fd, tem o mesmo offset, o descritor de ficheiro retornado é o mais baixo disponível

Named Pipes ou FIFO

canal unidirecional identificado por um nome do ficheiro, permitem que processos (sem ser pai e filho) comuniquem, existe apenas enquanto está aberto, comporta-se externamente como um ficheiro, existindo uma entrada na diretoria correspondente e tem um dono e permissões de acesso, mas não é um ficheiro uma vez que o seu conteúdo não é persistente
- fazer unlink do pathname
- criar pipe com int mkfifo(const char *pathname, mode_t mode)
- read bloqueia até que alguém escreva no pipe
- write bloqueia até que alguém leia a mensagem escrita

Sofia Nunes 106609

Despacho e escalonamento

separação entre
micro-núcleo → reduzidas dimensões q se continha essencial de SO: gestão fluxos execução-threads; gestão espaço endereçamento; comunicação entre processos; gestão interrupções
servidores sistema → executam em processos independentes a restante funcionalidade gestão processos; memória virtual; device drivers; sistema ficheiro

Boot núcleo de SO
computadores pessoais → BIOS → copia bloco código do disco p/ RAM, salta p/ 1ª instrução deste programa chamado bootloader
Linux → bootloader é GRUB

Gestor de processos
entidade do núcleo responsável por suportar a execução processos: trata interrupções; faz escalonamento; implementação chamadas sistema (syscalls); implementação funções sistema relacionadas processos sincronização: criação, sincronização, informação, eliminação; multiplexagem processador. Despacho → efetua transferência contexto entre 2 processos; Escalonamento → otimiza gestão processador; sincronização no núcleo

Despacho
comuta processador sempre q lhe seja indicado
copia o contexto hardware processo q estava em execução quando interrupção foi registada p/ respetivo descritor (entrada da lista processos); escolhe processo p/ executar (c/ maior prioridade); carrega contexto hardware do processo no processador, transfere controlo p/ novo processo, colocando PC na pilha, assim RTI é enganado a "voltar" para o processo "errado".
chamado por syscalls desencadeadas por software interrupts
Interrupção → modo núcleo. contexto processo execução guardado na pilha
se alterar estado processos invoca Despacho p/ eventualmente escolher outro processo p/ execução
caso processo ñ largue CPU existe interrupções de temporização → time-slice

Pilha utilizador e pilha núcleo
o uso de pilhas distintas é medida de segurança q impede q processos tenham acesso informação privilegiada núcleo

Escalonamento
definir q processo tem acesso CPU em cada momento
Escalonamento c/ prioridades → multi-lista
cada lista tem processos c/ dada prioridade; processos mais prioritários recebem CPU primeiro; prioridade pode ser: fixa → processos c/ menor prioridade podem nunca receber CPU; dinâmica → vai tornando processos q não recebem CPU a mais tempo mais prioritários

Preempção
retirar processador a processo em execução assim que existe outro processo mais prioritário
pode causar desperdício de tempo entre trocas processos se estiverem sempre a surgir processos mais prioritários

pseudo-preempção → só perde CPU para o mais prioritário se já tiver usado CPU durante tempo mínimo.

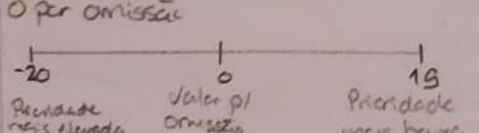
Escalonadores atuais
multi-filas c/ prioridades dinâmicas e fixas, pseudo-preemptivos c/ time-slice variável; atuam sobre tarefas e ñ processos; privilegia processos q consomem pouco CPU

Escalonamento em Unix
dividido em 2 estruturas
proc → informação tem de estar disponível p/ poder efetuar escalonamento e funcionamento dos signals; informação ñ pode deixar de estar em RAM, mesmo q processo tenha sido guardado em disco (swap)
u (user) → resto do contexto núcleo q só é necessário quando processo em execução; podia estar em disco quando ñ está em execução p/ diminuir necessidade memória núcleo
dois tipos prioridade → modo núcleo e modo utilizador
modo núcleo → fixas, consoante acontecimento q processo está a tratar; têm valores negativos (mais negativo mais prioritário), sempre mais prioritárias q os em modo utilizadores
modo user → escalonamento feito de acordo c/ prioridades (preempção); 0 → maior prioridade N → menor prioridade; prioridades calculadas em função tempo CPU usado (+ tempo recentemente → menos prioridade); processos c/ maior I/O mais prioritários

Algoritmo escalonamento
- escalonador escolhe processo + prio. durante um time-slice (100ms)
- round-robin entre os c/ prio. + elevada
- depois de 1s escalonador recalcula prioridades

TempoProcesso = TempoProcesso/2
Prioridade = TempoProcesso/2 + PrioridadeBase + Nice

Nice
0 por omissão



Tradicionalmente reservada a processos privilegiados

Escalonamento no Linux
Completely Fair Schedule (CFS)
distribuição equitativa recursos do CPU todos processos, evita monopolize CPU; time-slice sem duração fixa; não tem prioridades atribuídas aos processos; cada processo tem atributo vruntime → tempo acumulado execução modo user do processo
- quando processo criado vruntime = menor dos tempos execução atual, quando perde CPU vruntime incrementado c/ tempo q esteve executado nesse time-slice
menor vruntime => maior prioridade
nice → serve para pesar a distribuição tempo
Complexidade inserir tarefa de O(n) a O(log(n))
n → nº tarefas
otimizações → O(1)

Chamadas de Sistema
int fork()
reserva entrada tabela pro...
se user e/ou SO a exce...
subprocessos; atribui val...
vruntime igual mínimo...
sendo filho corrente contr...
contexto processo pai, com...
é partilhado, só é increm...
contador de nº users...
copia regiões de Dados...
copy-on-write p/ evita...
filho estado Executável...
processo
void exit(int status)
recebe inteiro correspo...
q acaba e termina...
apaga os seus time...
todos os recursos (fi...
concorrente, regiões...
atualiza ficheiro e re...
processador, memória...
ficheiros abertos e a...
nenhum hard link...
ficheiro aberto; estado...
chamada a exit ou...
envia signal death o...
a pai; guarda PID fi...
pai bem como status...
wait, mantendo filho...
até q pai o encontr...
task-struct para se...
pelo pai (zumbie)
int wait(int *status...
bloqueia pai p/ se sinc...
filho
recebe um ponteiro a...
filho onde guarda c...
exit do filho; procura...
ainda ñ há filho zom...
se há filhos zombie n...
pid filho e estado ex...
filho é libertado; de...
exec()
executa um novo ficheir...
verifica se ficheiro ex...
args chamada exec...
liberta regiões dados...
processo; reserva nova...
tabela ficheiros aber...
código executável; c...
p/ pilha user; ñ retor...
signals
- processo recebe sinal, se...
rotina tratamento ass...
flag no task descript...
pendente
- quando retorna de n...
verifica se existênc...
núcleo modifica pilh...
modo user executar...

Gestão de Mem...
memória primária → R...
memória secundária...
cada processo tem a...
ilusão q tem acesso a...
SO tem fornecer um...
memória forma se...
Dimensão espaço en...
processador n-bits →...

Chamadas de Sistema Linux

int fork()

reserva entrada tabela processos; verifica se user e/ou SO n excederam nº max subprocessos; atribui novo pid; atribui vruntime igual mínimo existente, em teoria filho corre antes pai; copia contexto processo pai, como região código é partilhada, só é incrementado contador de nº users q acedem a região; copia regiões de Dados e pilha, usa copy-on-write p/ evitar cópia inicial; filho estado Executável; devolve PID processo

void exit(int status)

recebe inteiro correspondente estado q acaba e termina-o

apaga os seus timers, liberta todos os recursos (ficheiro, diretoria corrente, regiões memória); atualiza ficheiro e registo utilização processador, memória e I/O; fecha os ficheiros abertos e apaga os q n tiverem nenhum hard link ou processos c/ ficheiro aberto; estado terminação, definido chamada a exit ou por razões internas; envia signal death of child (SIGCHILD) a pai; guarda PID filho terminado e do pai bem como status p/ quando pai chamar wait, mantendo filho estado zombie até q pai o encontre, mantém a task-struct para ser coletada na wait() pelo pai (zombie)

int wait(int *status)

bloqueia pai p/ se sincronizar c/ terminação filho

recebe um ponteiro a variável status do filho onde guarda o valor recebido no exit do filho; procura filhos zombie, se ainda n há filho zombie pai fica bloqueado; se há filhos zombie retorna imediatamente, pid filho e estado exit retornados e estrutura filho é libertada; devolve pid filho terminado

exec()

executa um novo ficheiro

verifica se ficheiro existe e é executável; copia args chamada exec da pilha user p/ núcleo; liberta regiões dados e pilha ocupadas pelo processo; reserva novas regiões memória; mantém tabela ficheiros abertos; carrega ficheiro código executável; copia args pilha núcleo p/ pilha user; n retorna nada em caso sucesso

signals

processo recebe sinal, se n o ignora e existe rotina tratamento associada, coloca-se uma flag na task descriptor indicando qual o sinal pendente

quando retorna de modo núcleo p/ modo user verifica se existência sinais pendentes núcleo modifica pilhas p/ quando retorne a modo user executar handler do signal

Gestão de Memória

memória primária → RAM + cache

memória secundária → disco

Cada processo tem espaço endereçamento p/ dar ilusão q tem acesso a uma memória só dele

SO tem fornecer utilizador interface p/ acender memória forma segura → Gestão de Memória

Dimensão espaço endereçamento

processador n-bits → 2^n bytes podem ser endereçados

endereçamento virtual → endereços referenciados por um processo "não existem" nem correspondem diretamente a endereços físicos pelo q têm de ser traduzidos

Memory Management Unit (MMU) → converte endereços virtuais em físicos ↳ UGM em português

Endereço virtual

número bloco	deslocamento dentro bloco

- 2 tipos blocos: segmentos e páginas

Segmentação

- c/ base programas segmentos lógicos (tamanho variável)

dimensão seg. → nº bits deslocamento (bits mais significativos), n excede memória principal

Desvantagens: o programador pode ter de se preocupar c/ gestão memória quando escreve programa; tamanho variável blocos faz com que possa sobrar espaço entre blocos após substituição por blocos menores → fragmentação externa

Tabela segmentos

P	Prot	Tamanho	Base

P → presente na memória

Prot → Leitura/Escrita/execução

Tamanho → Dimensão segmento (inferior a $2^{\text{bits deslocamento}}$)

Base → Endereço base na memória primária

endereço físico → Base + deslocamento

Fragmentação

percentagem espaço memória primária n usado devido gestão blocos

fragmentação interna → bloco > informação e fica desocupada uma parte

fragmentação externa → blocos colocados na primária e quando substituídos são por blocos menores, ao fim de algum tempo a memória tem vários blocos pequenos desocupados

Paginação

Página	Deslocamento

divide em blocos tamanho fixo → páginas

Tabela de páginas

P	R	M	Prot	Base

P → presente na memória

R → referenciada

M → modificada

Prot → leitura/escrita/execução

Base → Endereço base na memória primária

Dimensão página influencia

fragmentação interna, nº falta de páginas, tempo transferência (cresce c/ dimensão pág.), dimensão tabelas de pág. e listas de pág. mantidas pelo SO

Valor típico → 4KB

Translation Lookaside Buffer (TLB)

tabela tradução endereços

memória associativa c/ N entradas cada uma c/ end. pág virtual, base pág física e bits proteção; associado a cada end. pág. existe comparador hardware, q ao detetar igualdade dá como "resultado da leitura" base de pág. e bits proteção

TLB é limpa a cada comutação de processos

Tabela de páginas multi-nível

tabela páginas nível 1, endereça páginas que, consistem em tabelas de páginas.

Partilha memória entre processos

feita através de uma tabela páginas c/ entrada c/ mesma base

fork() duplica contexto pai mas n faz cópia física das páginas

Copy-on-write

permite partilha memória, onde págs só são copiadas quando necessário

- nas entradas tabela páginas dos processos (pai e filho) troca-se as permissões de escrita por CoW
- se pai ou filho tentam escrever em página c/ CoW ocorre exceção e núcleo acorda para alocar nova página p/ qual copia conteúdo original e atualiza a tabela de página
- se pág. original só é referenciada por 1 processo, permissão escrita ativada, CoW desativada

Sistema de Ficheiros

EXT3

dimensão máxima ficheiro

$B \times (12 + B/R + (B/R)^2 + (B/R)^3)$

B → dimensão em bytes bloco de dados

R → dimensão em bytes referência para bloco

Índice blocos ficheiro → vetor c/ 15 posições

12 primeiras entradas são diretas → bloco com dados

restantes entradas referências indiretas p/ outros blocos, com nível de indireção um (entrada 13), dois (entrada 14) e três (entrada 15).

$2^0 \to 1B$ $2^{10} \to 1KB (1024)$

$2^{20} \to 1MB$ $2^{30} \to 1GB$ $2^{40} \to 1TB$ $2^{50} \to 1PB$