

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

ЛАБОРАТОРНА РОБОТА №4

з дисципліни «ОПЕРАЦІЙНІ СИСТЕМИ»

Тема: «Драйвер файлової системи»

Виконав:

студент 2-го курсу

групи ІП-42

Дзюба Влад

Київ – 2016

**Завдання:**

Розробити драйвер файлової системи для пристроїв збереження інформації блочного типу. В якості файлової системи взяти звичайний файл, розмір якого означає об’єм пристрою збереження інформації. Самостійно обрати розмір блока, кількість посилань на блоки у дескрипторі файла, максимальну довжину імені файла.

**Лістинг програми:**

**Я обрав такі параметри:**

module Config where

blockSize, maxRefCount, maxFilenameLength, maxDescriptorCount :: Int

blockSize = 48

maxRefCount = 4

maxFilenameLength = 14

maxDescriptorCount = 8

fileFieldLength = refLength + maxFilenameLength

refsInBlock = div blockSize refLength

fileFieldsInBlock = div blockSize fileFieldLength

blocksForDescriptors = 1 + (div (descriptorLength\*maxDescriptorCount-1) blockSize)

fileTypeLength, sizeLength, refLength, linkCountLength:: Int

fileTypeLength = 1

sizeLength = 1

linkCountLength = 2

refLength = 2

descriptorLength = fileTypeLength

+ sizeLength

+ linkCountLength

+ refLength\*maxRefCount

**Структури даних:**

type Filename = String

type Reference = Int

type DescIndex = Int

newtype Memory = Memory (M.Map Reference MemoryBlock) deriving (Show)

data MemoryBlock = MemoryBlock {

getContent :: B.ByteString,

isFree :: Bool

} deriving (Show)

data DriverInfo = DriverInfo {

getBlockSize :: Int,

getFileRefCount :: Int,

getMaxFilenameLength :: Int,

getMaxDescriptorCount :: Int

}

data MemorySession = MemorySession {

getFDs :: M.Map ReadFileDescriptor DescIndex,

getMax :: ReadFileDescriptor

}

emptyMemSes = MemorySession M.empty (-1)

type ReadFileDescriptor = Int

type DescriptorIndex = Int

data FileType = File | Folder | None deriving (Eq, Show)

data Descriptor = Descriptor {

getType :: FileType,

getSize :: Int,

getLinkCount :: Int,

getRefs :: [Reference]

}

**Функції для роботи**

instance Show Descriptor where

show (Descriptor fileType size linkCount refs) =

(show fileType) ++ ":" ++

(show size) ++ ":" ++

(intercalate ":" $ map show refs)

class Noneable a where

isNone :: a -> Bool

notNone :: a -> Bool

isFile :: a -> Bool

notNone = not . isNone

isNone = not . notNone

instance Noneable FileType where

isNone None = True

isNone \_ = False

isFile File = True

isFile \_ = False

instance Noneable Descriptor where

isNone = isNone . getType

isFile = isFile . getType

class HasDescriptors a where

readDescriptors :: a -> [Descriptor]

writeDescriptors :: [Descriptor] -> a -> a

setDescriptor :: DescriptorIndex -> Descriptor -> a -> a

takeDescriptor :: Int -> a -> Descriptor

takeDescriptor i = (!! i) . readDescriptors

instance HasDescriptors B.ByteString where

readDescriptors = map readDescriptor

. splitAtChunks C.descriptorLength

writeDescriptors descs = mappend descsStr . B.drop (B.length descsStr)

where descsStr = mconcat

. map showDescriptor

$ descs

setDescriptor i desc mem = mconcat [before, showDescriptor desc, after]

where (before, after) = B.splitAt (i\*C.descriptorLength) mem

(\_, after') = B.splitAt C.descriptorLength after

instance HasDescriptors MemoryBlock where

readDescriptors = readDescriptors . getContent

writeDescriptors descs = modifyContent $ writeDescriptors descs

setDescriptor i desc = modifyContent $ setDescriptor i desc

instance HasDescriptors Memory where

readDescriptors (Memory mem) = take C.maxDescriptorCount

. concat . M.map readDescriptors

$ mem

writeDescriptors descs (Memory mem) = Memory

. snd

. (\xs -> M.mapAccum saveMemory xs isFreeMap)

. splitAtChunks C.blockSize

. writeDescriptors descs

. M.foldr mappend mempty

. M.map getContent

$ mem

where isFreeMap = M.map isFree $ mem

saveMemory (x:xs) fr = (xs, MemoryBlock x fr)

setDescriptor i desc mem = flip writeDescriptors mem

. insertAt i desc

. readDescriptors

$ mem

splitAtChunks :: Int -> B.ByteString -> [B.ByteString]

splitAtChunks n = map (B.take n)

. takeWhile ((>0).(B.length))

. iterate (B.drop n)

modifyContent :: (B.ByteString -> B.ByteString) -> MemoryBlock -> MemoryBlock

modifyContent f (MemoryBlock cont free) = MemoryBlock (f cont) free

modifyMemory f (Memory mem) = Memory (f mem)

readFileType 0 = None

readFileType 1 = File

readFileType 2 = Folder

writeFileType None = B.singleton 0

writeFileType File = B.singleton 1

writeFileType Folder = B.singleton 2

setFileType fileType desc = desc { getType = fileType}

insertBetween :: ([a] -> ([a], [a])) -> a -> [a] -> [a]

insertBetween f el xs = before ++ (el:after)

where (before, \_:after) = f xs

insertAt :: Int -> a -> [a] -> [a]

insertAt i = insertBetween (splitAt i)

insertWhen :: (Descriptor -> Bool) -> Descriptor -> [Descriptor] -> [Descriptor]

insertWhen f = insertBetween (break f)

insertWhenNone :: Descriptor -> [Descriptor] -> [Descriptor]

insertWhenNone = insertWhen isNone

readDescriptor :: B.ByteString -> Descriptor

readDescriptor str = Descriptor {

getType = readFileType fileType,

getSize = size,

getLinkCount = linkCount,

getRefs = refs

} where mapFst f (a, b) = (f a, b)

(fileType, str') = mapFst readNumber $ B.splitAt C.fileTypeLength str

(size, str'') = mapFst readNumber $ B.splitAt C.sizeLength str'

(linkCount, str''') = mapFst readNumber $ B.splitAt C.linkCountLength str''

refs = take C.maxRefCount . map (readNumber.fst) . tail

. iterate (B.splitAt C.refLength . snd)

$ (B.empty, str''')

showDescriptor :: Descriptor -> B.ByteString

showDescriptor (Descriptor fileType size linkCount refs) = mconcat $

[writeFileType fileType

,writeNumberN C.sizeLength size

,writeNumberN C.linkCountLength linkCount] ++

map (writeNumberN C.refLength) refs

showDescriptors :: [Descriptor] -> B.ByteString

showDescriptors = mconcat . map showDescriptor

writeNumberN :: Int -> Int -> B.ByteString

writeNumberN n = B.pack

. reverse

. map (fromIntegral . (`mod` 256))

. take n

. iterate (`div` 256)

writeNumbersN :: Int -> [Int] -> B.ByteString

writeNumbersN n = mconcat . map (writeNumberN n)

readNumberN :: Int -> B.ByteString -> Int

readNumberN n = readNumber . B.take n

readNumbersN :: Int -> B.ByteString -> [Int]

readNumbersN n = map (readNumberN n)

. takeWhile ((>=n) . B.length)

. iterate (B.drop n)

readNumber :: B.ByteString -> Int

readNumber = foldl1 (\acc b -> acc\*256+b)

. map fromIntegral

. B.unpack

toByteString :: String -> B.ByteString

toByteString = B.pack . map BS.c2w

fromByteString :: B.ByteString -> String

fromByteString = map BS.w2c . B.unpack

readFiles :: B.ByteString -> [(DescIndex, Filename)]

readFiles =

let readPair = transformPair . B.splitAt C.refLength

transformPair (s1, s2) = (readNumberN C.refLength s1, fromByteString s2)

in map readPair

. splitAtChunks C.fileFieldLength

readDriverInfo :: String -> DriverInfo

readDriverInfo str =

case map read . lines $ str of

bs:frc:mfl:mdc:\_ -> DriverInfo bs frc mfl mdc

\_ -> error $ "Not DriverInfo format "

takeBlock :: Int -> Memory -> MemoryBlock

takeBlock i (Memory mem) = mem M.! i

readMemory :: B.ByteString -> Memory

readMemory str = calcMemoryFreedom

. map (B.take C.blockSize)

. takeWhile ((>0) . B.length)

. iterate (B.drop C.blockSize)

. B.append (B.pack (replicate addByteCount 0))

$ str

where len = B.length str

addByteCount = mod (-(mod len C.blockSize)) C.blockSize

calcMemoryFreedom :: [B.ByteString] -> Memory

calcMemoryFreedom blocks = Memory . snd

. M.mapAccum (\(x:xs) el -> (xs, (MemoryBlock x el))) blocks

. foldr (`M.insert` False) initMap

. concat

. map descriptorAllRefs

. filter notNone

$ descriptors

where descriptors = take C.maxDescriptorCount

. readDescriptors . mconcat $ blocks

blocksLen = length blocks

initMap = M.fromList initList

initList = zip [0..blocksLen-1] $ (replicate C.blocksForDescriptors False)

++ (repeat True)

descriptorAllRefs d = neededRefs ++ (if isFile d then take size moreRefs else [])

where size = getSize d

neededRefsCount = calcRefsCount size

neededRefs = take neededRefsCount refs

refs = getRefs d

moreRefs = concat

. map (readNumbersN C.refLength

. (blocks!!))

$ refs

getDirFiles :: DescriptorIndex -> Memory -> [(DescIndex, Filename)]

getDirFiles folderId mem = files

where folder = (!! folderId) . readDescriptors $ mem

size = getSize folder

refs = getRefs folder

files = take size

. concat

. map (readFiles

. getContent

. (`takeBlock` mem))

$ refs

showFileField :: (Reference, String) -> B.ByteString

showFileField (descindex, filename) = mappend

(writeNumberN 2 descindex) (toByteString filename)

putFileInDir :: (DescIndex, Filename) -> DescIndex -> Memory -> Memory

putFileInDir (descindex, filename) folderIndex mem@(Memory blocks) = memory'

where fileField = showFileField (descindex, filename)

folderDesc = takeDescriptor folderIndex mem

size = getSize folderDesc

refs = getRefs folderDesc

size' = size + 1

refsCount = calcFileFieldsCount size

refsCount' = calcFileFieldsCount size'

isNewBlock = refsCount' > refsCount

(freeBlockIndex,\_) = head

. M.toList

. M.filter isFree

$ blocks

folderDesc' = folderDesc {

getSize = size',

getRefs = if isNewBlock

then insertAt refsCount freeBlockIndex refs

else refs

}

fieldIndexToWrite = mod size C.fileFieldsInBlock

addFileField = modifyContent $

mconcat

. insertAt fieldIndexToWrite fileField

. splitAtChunks C.fileFieldLength

changedBlock = (getRefs folderDesc') !! (refsCount'-1)

memory' = setDescriptor folderIndex folderDesc'

. Memory

. ( if isNewBlock

then M.adjust (setFreedom False) freeBlockIndex

else id

)

. M.adjust addFileField changedBlock

$ blocks

setFreedom :: Bool -> MemoryBlock -> MemoryBlock

setFreedom isFree block = block {isFree = isFree}

createFile :: Filename -> DescriptorIndex -> Memory -> Memory

createFile filename folderId mem = putFileInDir (desc'Index, filename)

folderId

$ writeDescriptors descriptors' mem

where descriptors = readDescriptors mem

descriptors' = insertWhenNone desc' descriptors

desc'Index = fst . head . dropWhile (notNone . snd)

$ zip [0,1..] descriptors

desc' = Descriptor {

getType = File,

getSize = 0,

getLinkCount = 1,

getRefs = replicate C.maxRefCount 0

}

filenameEquals :: Filename -> Filename -> Bool

filenameEquals f1 f2 = foldr strEq True $ zip (f1++(repeat '\NUL')) f2

where strEq ('0', '0') \_ = True

strEq (x, '0') \_ = False

strEq ('0', x) \_ = False

strEq (x, y) b = x == y && b

findFileIndex :: Filename -> Memory -> DescIndex

findFileIndex filename mem = fst . head

. filter (filenameEquals filename . snd)

. getDirFiles 0

$ mem

openFile :: Filename -> Memory -> MemorySession

-> (ReadFileDescriptor, MemorySession)

openFile filename mem memSes =

(curId, memSes {

getFDs = M.insert curId fileDescIndex $ getFDs memSes,

getMax = curId

})

where fileDescIndex = findFileIndex filename mem

curId = 1 + (getMax memSes)

closeFile :: ReadFileDescriptor -> MemorySession -> MemorySession

closeFile fd memSes = memSes {

getFDs = M.delete fd $ getFDs memSes

}

clearBlock :: Reference -> Memory -> Memory

clearBlock index = modifyMemory

$ M.adjust (modifyContent $ const zeroString)

index

where zeroString = B.pack $ replicate C.blockSize 0

truncateF :: Filename -> Int -> Memory -> Memory

truncateF filename newSizeB mem = mem'

where fileDescIndex = findFileIndex filename mem

descriptor = takeDescriptor fileDescIndex mem

descriptor' = descriptor { getSize = newSize }

curSize = getSize descriptor

curRefCount = calcRefsCount curSize

newSize = div newSizeB C.blockSize

newRefCount = calcRefsCount newSize

(Memory blocks) = mem

refs = take curSize . concat

. map (readNumbersN C.refLength

. getContent

. (`takeBlock` mem))

$ getRefs descriptor

freeBlockRefs = map fst . M.toList $ M.filter isFree blocks

(freeBlockRefs1, rest) = splitAt (newSize - curSize) freeBlockRefs

freeBlockRefs2 = take (newRefCount-curRefCount) rest

refs' = chunk C.refsInBlock (refs ++ freeBlockRefs1)

refToRefs' = zip ((getRefs descriptor) ++ freeBlockRefs2) refs'

changedBlocks = drop (div curSize C.refsInBlock) refToRefs'

insertChangedBlock (i, refs) =

let str = writeNumbersN C.refLength refs

in M.adjust ( setFreedom False

. modifyContent (mappend str

. B.drop (B.length str))) i

freeMemBlock = MemoryBlock {

getContent = B.pack $ replicate C.blockSize 0,

isFree = False

}

mem' = if newSize <= curSize

then setDescriptor fileDescIndex descriptor'

. Memory . foldr (M.adjust $ setFreedom True) blocks

$ ((drop newSize refs)

++ (drop newRefCount $ getRefs descriptor))

else Memory

. flip (foldr insertChangedBlock) changedBlocks

. foldr (`M.insert` freeMemBlock) blocks

$ freeBlockRefs1

makeLink :: Filename -> Filename -> Memory -> Memory

makeLink source target mem = setDescriptor fileDescIndex descriptor'

. putFileInDir (fileDescIndex, target) 0

$ mem

where fileDescIndex = findFileIndex source mem

descriptor = takeDescriptor fileDescIndex mem

descriptor' = descriptor {

getLinkCount = 1 + (getLinkCount descriptor)

}

deleteLink :: Filename -> Memory -> Memory

deleteLink filename mem = freeBlocks

. changeFileDesriptor

. changeFolderDescriptor

. swapLinks

$ mem

where folder = takeDescriptor 0 mem

folder' = folder {

getSize = size - 1

}

changeFolderDescriptor = setDescriptor 0 folder'

size = getSize folder

refs = getRefs folder

refsCount = calcFileFieldsCount size

fields = take size . concat

. map (readFiles

. getContent

. (`takeBlock` mem))

$ refs

(before, (ind, field):after) = break (filenameEquals filename . snd . snd) $ zip [0..] fields

refIndex = div ind C.fileFieldsInBlock

inBlockIndex = mod ind C.fileFieldsInBlock

field' = snd $ last after

swapLinks = if null after then id

else modifyMemory $

M.adjust (modifyContent $

mconcat

. insertAt inBlockIndex (showFileField field')

. splitAtChunks C.fileFieldLength

) (refs !! refIndex)

freeBlocks = if mod size C.fileFieldsInBlock /= 1

then id

else modifyMemory $ M.adjust (setFreedom True) (refs !! (refsCount-1))

fileDescInd = fst field

fileDesc = takeDescriptor fileDescInd mem

newLinkCount' = getLinkCount fileDesc - 1

fileDesc' = fileDesc {

getLinkCount = newLinkCount'

}

fileSize = getSize fileDesc

fileRefsCount = calcRefsCount fileSize

fileRefs = take fileRefsCount (getRefs fileDesc)

fileMoreRefs = take fileSize . concat

. map (readNumbersN C.refLength

. getContent

. (`takeBlock` mem))

$ fileRefs

changeFileDesriptor =

if newLinkCount' > 0

then setDescriptor fileDescInd fileDesc'

else setDescriptor fileDescInd (setFileType None fileDesc')

. (modifyMemory $

flip (foldr (M.adjust $ setFreedom True))

(fileRefs ++ fileMoreRefs)

)

type Offset = Int

type Size = Int

write :: ReadFileDescriptor -> Offset -> String

-> MemorySession -> Memory -> Memory

write fd offset str memSes mem = (modifyMemory $

flip (foldr insertBlock) blockData)

$ mem

where insertBlock (i, block) =

M.adjust (modifyContent $ const block) i

fileDescIndex = (getFDs memSes) M.! fd

fileDesc = takeDescriptor fileDescIndex mem

fileSize = getSize fileDesc

refsCount = calcRefsCount fileSize

refs = take refsCount (getRefs fileDesc)

moreRefs = take fileSize . concat

. map ( readNumbersN C.refLength

. getContent

. (`takeBlock` mem))

$ refs

fileData = mconcat

. map ( getContent

. (`takeBlock` mem))

$ moreRefs

(before, rest) = B.splitAt offset fileData

rest' = snd $ B.splitAt (length str) rest

fileData' = B.take (fileSize\*C.blockSize)

$ mconcat [before, toByteString str, rest']

blockData = drop (div offset C.blockSize)

. take (calcRefsCount (offset + (length str)))

. zip moreRefs

$ splitAtChunks C.blockSize fileData'

readFromFile :: ReadFileDescriptor -> Offset -> Size

-> MemorySession -> Memory -> String

readFromFile fd offset size memSes mem = fromByteString

. B.drop offset

. B.take (offset+size)

$ fileData

where fileDescIndex = (getFDs memSes) M.! fd

fileDesc = takeDescriptor fileDescIndex mem

fileSize = getSize fileDesc

refsCount = calcRefsCount fileSize

refs = take refsCount (getRefs fileDesc)

moreRefs = take fileSize . concat

. map ( readNumbersN C.refLength

. getContent

. (`takeBlock` mem))

$ refs

fileData = mconcat

. map ( getContent

. (`takeBlock` mem))

$ moreRefs

calcCount objInBlock size = 1 + (div (size-1) objInBlock)

calcRefsCount = calcCount C.refsInBlock

calcFileFieldsCount = calcCount C.fileFieldsInBlock

**Висновок:**

Використання блоків пам’яті та системи дискрипторів файлі з посиланнями на блоки з посиланнями дозволяє зменшити час знаходження вільного місця, зберігати файл не у послідовних блоках пам’яті та виділяти пам’ять для файла за необхідністю.