Pseudocode Design

Main()

-----------------------------------------------------------------

Test each function for use

init()

create()

open()

close()

read()

write()

status()

FUNCTIONS

------------------------------------------------------------------

Init

initialize frames to 0 (# of frames to use)

initialize framesize to 0 (the size in bytes of each page frame)

if return = 0,

return 0

execute function

else

return 1

exit program

CREATE

initialize character fname (name of file)

initial size to 0 (size for file)

print(“Name of File: ”)

set character “fname” to input of user

create file with (fname) as its name

assign file size up to next whole page (5 frames)

assign size to file(size) to be used for making the page table

write file to disk with # of bytes for file.

if return = 0,

return 0

else

return 1

exit program

OPEN

initialize character fname

prompt user for a file name (print: enter a filename to open)

if file exists,

open file (fname)

if file opens,

create page table with initially 0 page size

page size(length of the file) = file size

read page frames as needed when page faults occur for it

else,

print “error, file does not exist”

if return = 0,

return 0

else

return 1

exit program

CLOSE

initialize character fname

prompt user for file name (print: enter filename)

if file exists,

write modified pages to disk

find all existing unused frames

all existing frames with pages are made available to be reused.

delete page table for existing file

close the file

if return = 0,

return 0

else

return 1

exit program

READ

initialize char fname (name of file)

initialize int address (virtual address of the first byte to read)

initialize int length(# of bytes to read > 0)

initialize int err return error code = 0

reads a string of bytes from an open file

if pages of file are not in memory

then page fault

fetch necessary pages into page frames

if return = 0,

return 0

else

return 1

exit program

if err value = 0

success

else

error message

if error message AND err value != 0

return a NULL string for string of bytes for file

WRITE

initialize char fname

initialize int address

initialize char value

prompts user for filename (print: enter file name)

if file exists

write pages of file to disk

if pages of files are not in memory

page fault occurs

get pages from main memory to fill necessary space.

if page replacements are written OR file is closed

write modified pages to disk

length of string = bytes written to file

if return = 0,

return 0

else

return 1

exit program

if err = 0

success

else

error message (tried to write to file, or to write past the end of file)

STATUS

Do

if file exists

print fname

print page table for files with fname

print entries for table (use a virtual address) for fname

label each entry and all info (all associated bits and modified bits) for fname

While

files exist

if return = 0,

return 0

else

return 1

exit program

Da rules

* Your virtual file manager will use a fixed number of page frames that is specified when the virtual file manager is initialized.
* The size of a page frame is also specified at this time.
* You must also create the page tables for the virtual files managed by the virtual file manager.
* Use the LRU page replacement algorithm for handling page faults in the virtual file manager.
* The virtual file manager, once initialized, can handle any number of files. Each file must have its own page table.
* The files share the available page frames using a global LRU replacement policy. In other words, when a page fault occurs, the least recently used page in any of the open files is replaced.
* You should not restrict the candidates to only those page frames containing pages from the same file.
* Your functions should detect and handle any reasonable errors that could arise.
* The page table for a file must contain a modified bit so that you know when you need to write a page to disk (either when it is replaced or the file is closed).
* If a page has not been modified, then it should not be written to disk when replaced or closed.
* You must also include a referenced bit to implement the LRU algorithm.
* Both these bits do not need to be physical bits; you can use an entire int to store/test your bit.
* You will need to make sure that the physical file on the disk is the correct size.
* Round the given file size up to the next full page.
* You will need to write “filler” pages with all ‘.’ Characters into the file.
* You will also want to record the size of the file in its first few bytes so that you know how big it is when you open it with vfmopen() in the future.
* You need to know how big it is so that you can create the page table for it.
* Feel free to experiment with variations of the LRU algorithm to try to minimize the number of page faults in your virtual file manager.
* Since you do not know ahead of time how many page frames you will have and how many page tables you will need, you will need to allocate space for them dynamically when you create them. To do this, use the malloc() function or a similar function in C or C++ ( new() is the C++ equivalent).

interp of jfile\_chunker.Main

create filename with no values

initialize scanner

print intro page

try

do

create new object with filename

assign filename to file and close it

print error

while true

catch IOexeption

print create file name

try

initialize new file with filename in read/write mode

catch

IO exeption

print could not create name

return

do

print how many kbs? (1-0)

set filekbytes = input

while (file bytes < 1) | | (filebytes > 10)

filesize = filebtyes \* 1024

do

print select size

print A) 128

print B) 256

print C) 512

while (choice is not a,b & c)

switch

a = 128

b = 256

c = 512

default

print error message

chunkcount initialized = filesize/chunksize

print create file with name and size with chunksize and count

chunk = bytebuffer and allocated to (chunksize)

for (i = 0 i< chunksize, i++)

temp = ‘d’

chunk.put(temp)

for (i = 0 i< chunksize, i++)

temp = ‘d’

chunk.rewind()

for (j = 0; j < chunksize; j++)

fp.writeByte(chunk.get(j))

print success

do

print modify chunk int?

initialize mod chunk at next input

while (modchunk < 0 ) | | (modchunk >= chunkcount)

print increment chunk

file seek (modchunk \* chunksize)

rewind chunk

for

put chunk value in for file readByte

rewind chunk

initialize temp with chunk value i

increment temp

set chunk with first value of i to temp

seek file with modchunk \* chunksize

rewind chunk

for

file is written to byte with value chunk at i position

close file

print success