# Topic 3: (Part E)

I/O and Peripheral Devices Management

# Secondary storage management in the Nintendo DS

Systèmes Embarqués Microprogrammés

Prof. David Atienza Alonso, SEL-STI

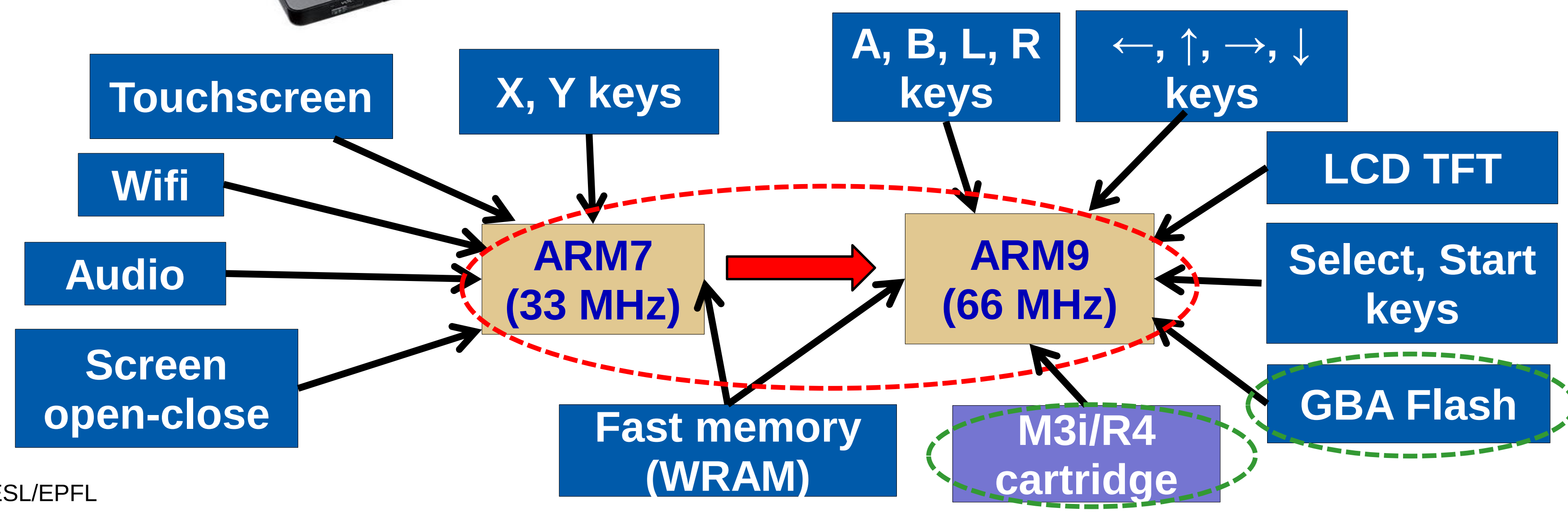
# Content of Session

- Use of secondary storage devices in the NDS
  - Comparisons between memories and secondary storage units
  - Components of a file system: files and directories
  - Implementations of a file system: contiguous and indirect allocation
  - libFAT library methods for files and directories in the NDS

- Enabling File Allocation Table (FAT) systems in combination to other I/O devices of NDS
  - Listing Root directory of the NDS into a file
  - Listing all files of a NDS unit into a file
  - Adding score counters and managing scores record (store and retrieval) in Tetris Game
  - Adding storage and playing saved melodies in the Piano Player

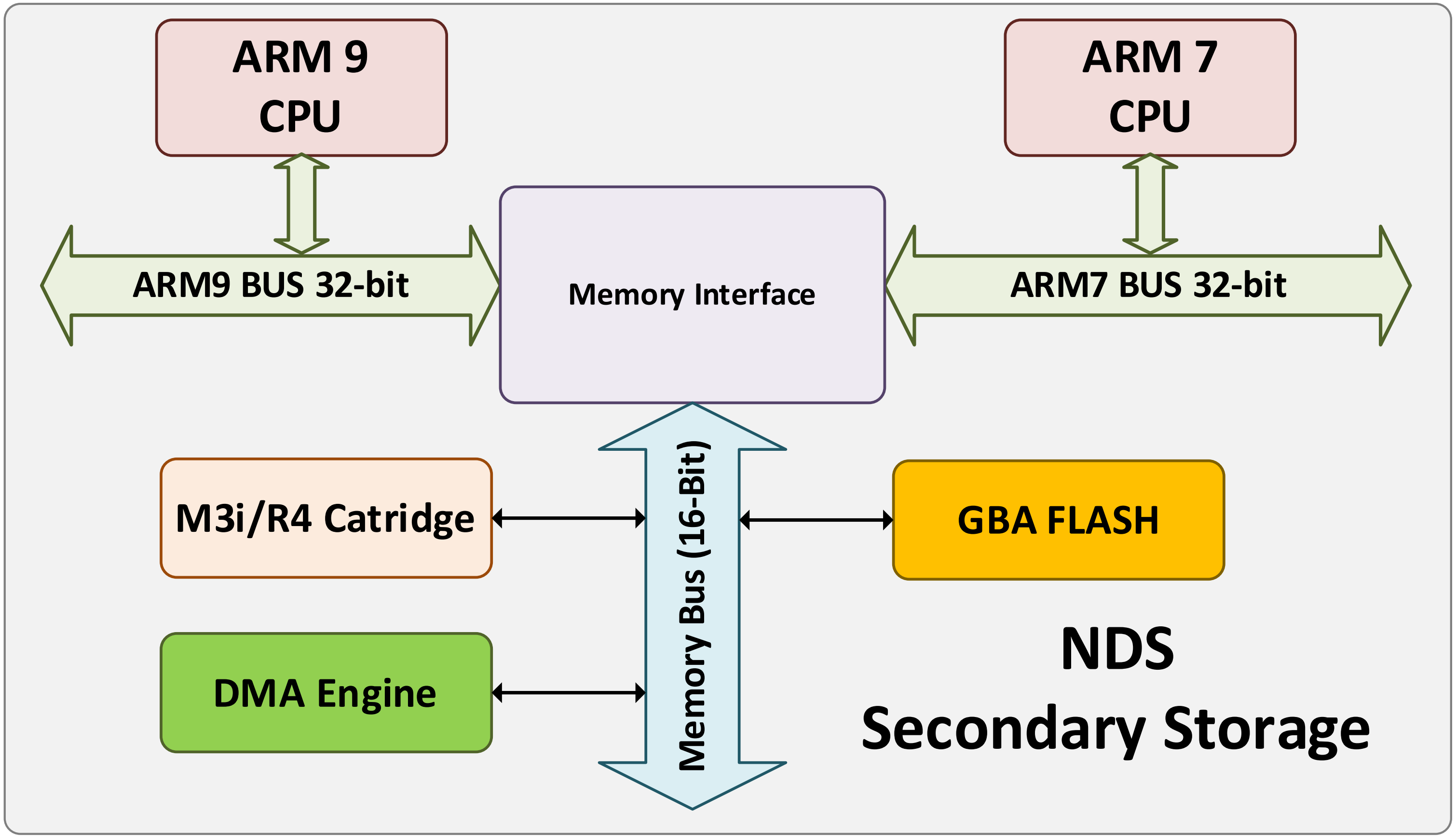
# I/O subsystem management on the NDS: Secondary Storage

- Both processors have shared access to secondary storage
  - ARM 7TDMI-S: standard NDS back-up for user configuration in GBA Flash
  - ARM 946E-S: able to perform backups and add a filesystem in M3i/R4 cartridge
    - Use of MicroSD card external to basic NDS configuration

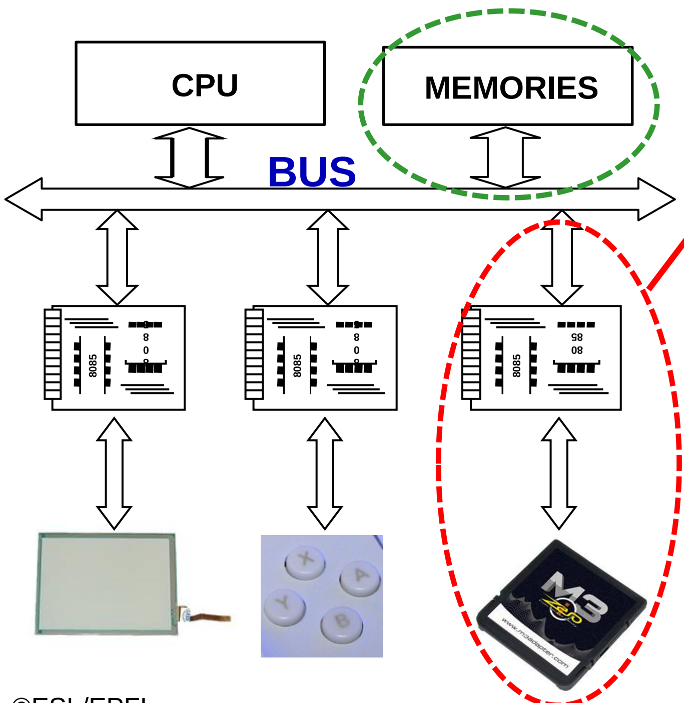
# Secondary Storage on the NDS

- The ARM9 and ARM7 processors have access to secondary storage via the memory interface



# Different types of storage: main memories and secondary memories storage

| Main memories (SRAM/DRAM) | Secondary storage (FLASH, HDD, …) |
|---|---|
| Volatile memory (only valid if NDS on) | Permanent storage (retains state even if NDS is off) |
| Fast memories (but limited space) | Slow memories (but large space) |
| Minimum access size : **WORD** (depends on processor instruct. size) | Minimum access size : **BLOCK** (depends on the I/O device) |

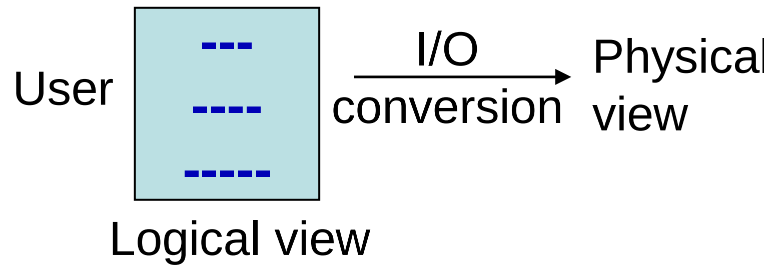



**I/O subsystem interface translates requests from words to block size**

- Secondary storage organization: ***File System***
  - Method to store data into an easy-to-manipulate database and human-readable names
  - Hierarchical data organization in 2 types:
    - Directories and Files
  - Mem. buffers as cache to avoid bus transactions

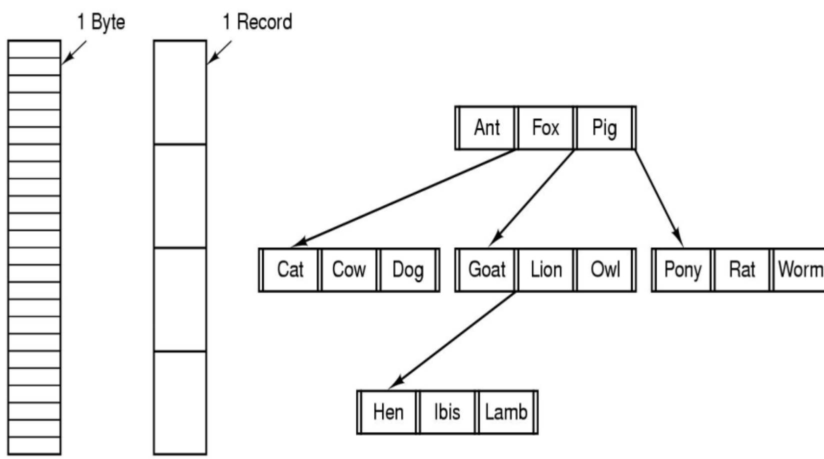
# Components of a file system: Files

- Fundamental logical storage unit defined by I/O subsystem interface to provide a mechanism to group data on physical storage device




- File structure determined by each system designer, three options:
  A. Original physical view
  B. Sequence of fixed length records
  C. Index-like structure (e.g., tree)

  - Structure stored in first fields of file (***header***), typically:
    1. **Id:** file identifier in the system
    2. **Offset:** displacement to data start
    3. **Size:** size of file
    4. **Type:** type of file
    5. **Name:** human-readable name

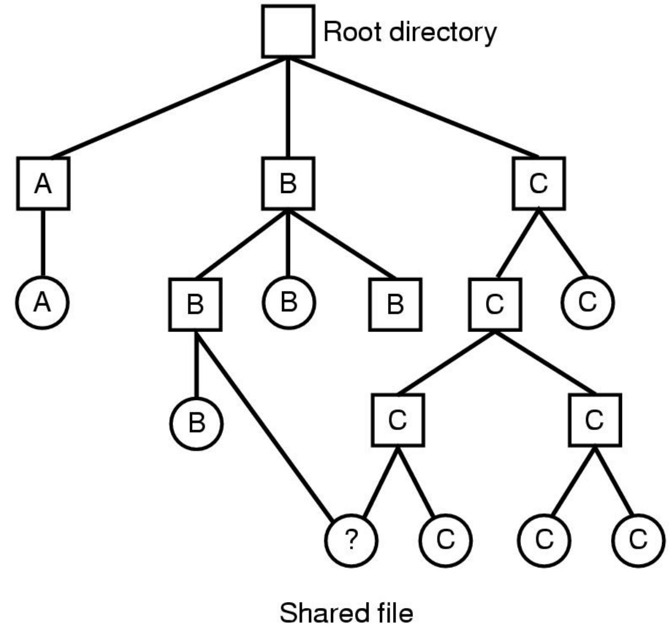


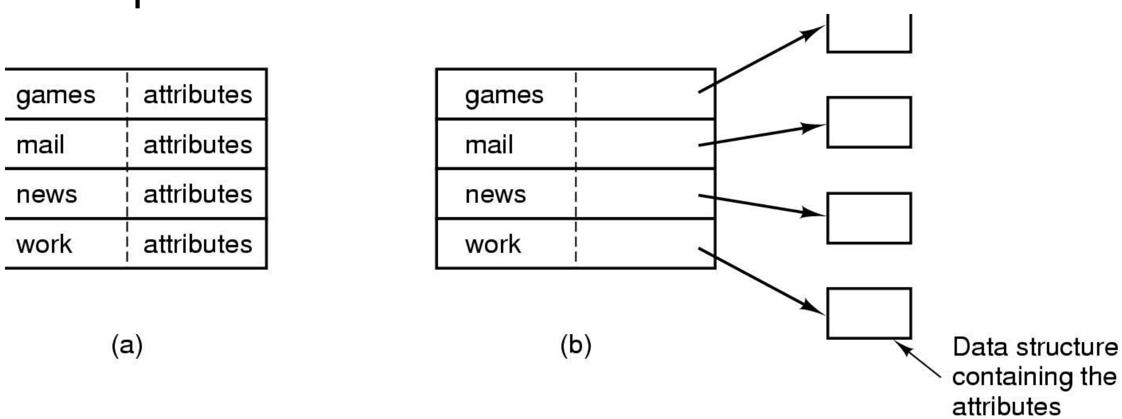

(a) (b) (c)

- Generally two types of access to a file are provided
  - Sequential access: start accessing from the beginning and read sequentially
  - Random access: access to any byte in the file directly

# Components of a file system: Directories

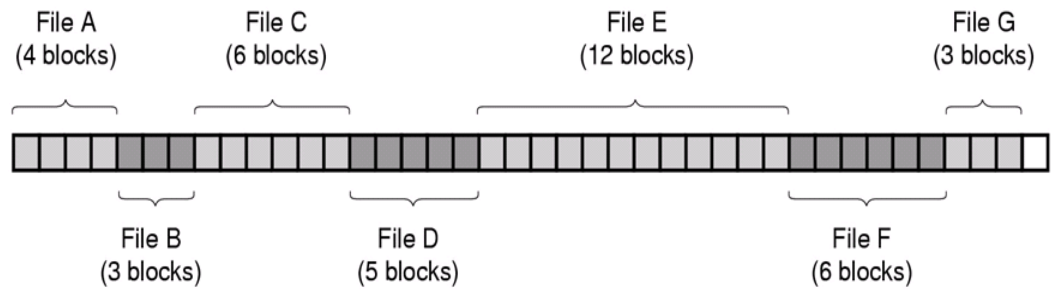
- Container of files to provide a mechanism to keep track of files
  - A directory is a file that stores one record/pointer for each file in that directory
  - Recursive/hierarchical structures possible
- A directory records info about the files in its particular partition
  - Typically contains per file:
    - A. Name and Attributes
    - B. Name and pointer to Attribute information



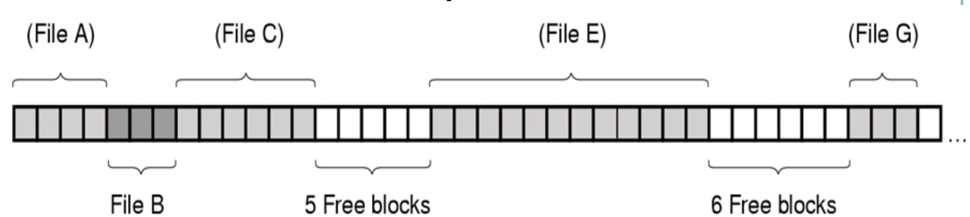



# Implementations of a file system in physical storage devices: Contiguous allocation

- Each file/directory stored on consecutive disk blocks
  - E.g., Disk with 4K block size, a 20K file is stored on 5 blocks



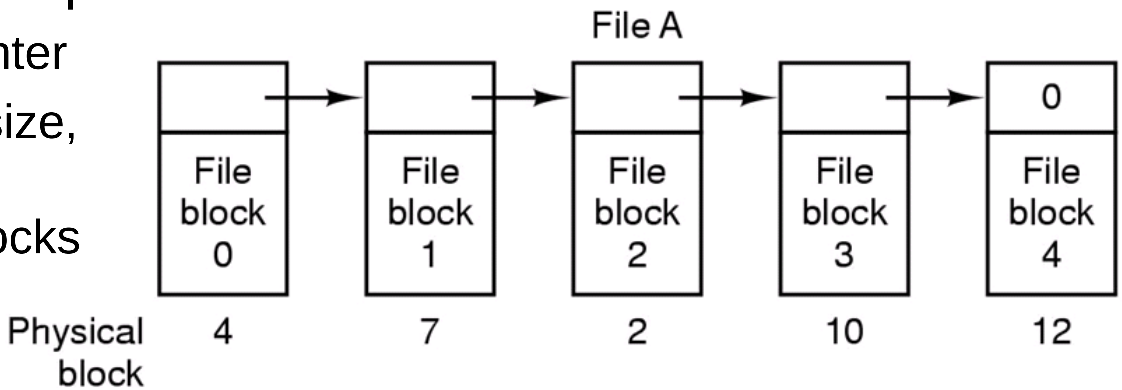

- Advantages
  - Simple to implement: only needed disk address of 1st block and nr. of blocks
  - Excellent read performance because only one disk operation reads entire file
- Disadvantages:
  - Disk fragmentation: occurs when files are removed. Keep track of used blocks?
  - Large files: must know final size of new file to be able to choose the correct hole to place it




- Consecutive allocation is ideal for write-once devices: CD-ROMs, DVDs, etc.

# Implementations of a file system in physical storage devices: Indirect allocation

- A linked list of disk blocks is kept in this method
  - First block: header and pointer
  - E.g., In disk with 4K block size, a 20K file is stored on **<u>any</u>** 5 available physical blocks



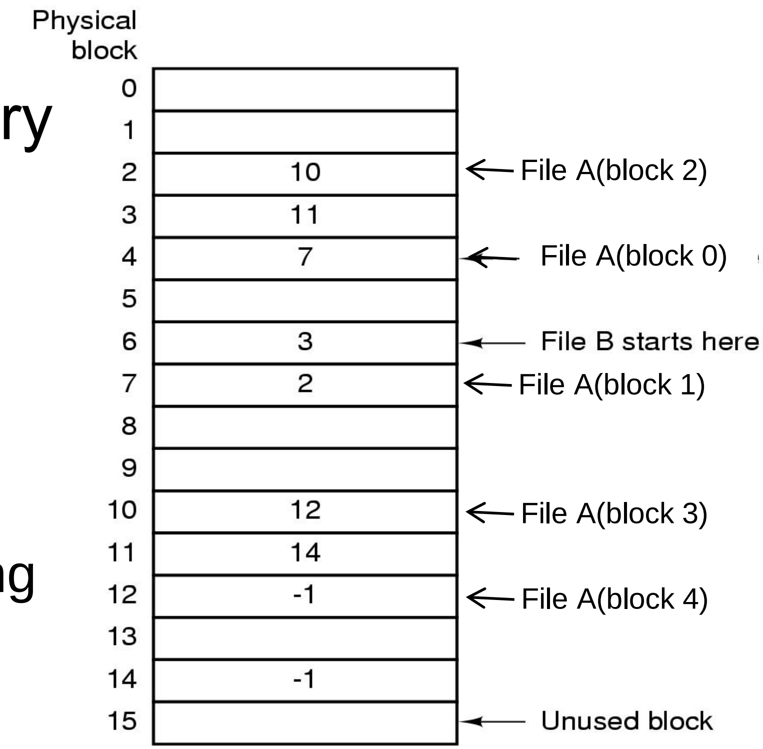

- Advantages
  - Every disk blocks can be used (except for internal fragmentation)
  - Still easy to perform sequential reads
- Disadvantages:
  - Random access to each block is very costly in time because we have to read all the previous blocks of a file before that block
  - Because of pointer the amount of data stored in each block is not a power of two

**Can we overcome these disadvantages for better performance?**

- Indirect allocation is the general method used in I/O devices with R/W support: Flash memories, hard disk drives, etc.

# Enhancing indirect allocation: File Allocation Table (FAT) systems

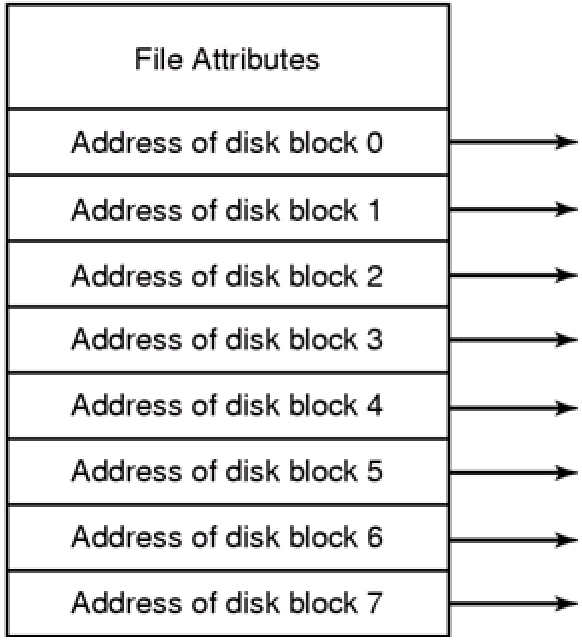
- FAT: separate table of pointers to keep the files' blocks starts in memory
  - E.g., In disk with 4K block size, a 20K file (file A) is stored on **<u>any</u>** 5 available physical blocks




- Advantage
  - Random access only requires the starting block number because there is no disk reference involved
- Disadvantage
  - Large number of blocks to index in the FAT
  - E.g., in a 20 GB disk, 1 KB block size, 4 bytes per entry, how much space for FAT?
    - Approx. **80 MB used only in indexing!** (20 M entries x 4 bytes)

# Enhancing indirect allocation: i-node systems

- i-node: separate table per file contains attributes and disk addresses of blocks of that file
  - Typically the size of files is small




- Advantage:
  - Use of index table depends only on the number of open files instead of disk size
    - If an i-node uses ***n*** bytes for ***k*** files open: only ***kn*** bytes of memory are used
- Disadvantage
  - If each i-node has room for a fixed number of disk addresses what happens when a file grows beyond this limit?
    - **Limited maximum possible size per file!**

# Libfat:
# A library for FAT systems in the NDS

- ***Libfat*** provides functions and methods to manage a FAT-based file system (typically for the ARM9)
  - Different possible units where to configure it
    - MicroSD card included in the M3i/R4 cartridge
    - External units connected to the Slot 2
  - Close interface to usual C programming functionality for file systems
    - Standard C management interface to files: fopen, fclose, etc.
    - Enhanced standard C management interface for directories

- It is compatible with devKitPro
  - Included in toolchain, but not included by default in project template
  - Necessary to add ***–lfat*** in the ***Makefile*** (first of the libraries)
    - **LIBS = -lfat  -lnds9  -lmm9**

# Libfat Application Programmer Interface (API): Two fundamental initialization methods

- Default configuration settings
  - bool fatInitDefault( void );
    - This function initializes the file system in the MicroSD card

- User-made configuration
  - bool fatInit (uint32_t cacheSize, bool setAsDefaultDevice);
    - ***cacheSize*** = Number of intermediate buffers (cache sectors) to avoid multiple accesses to external secondary storage devices
      - By default 8 in the NDS
      - Useful to reduce it in large programs where memory consumption is critical
    - ***SetAsDefaultDevice*** = Device where the file system will be mounted
      - By default this parameter is set to *true* (MicroSD card)

- More information/methods in: **/opt/devkitPro/libnds/include/fat.h**

# Libfat API: Consulting directories content

- Iterator system to consult directories, three operations:
  - DIR* *opendir*( char* dirPath );
    - Returns a pointer to a ***DIR*** structure (hidden structure)
  - dirent* *dirNext*( DIR* dt, char* fileName);
    - Returns a pointer to a Directory Entry (**dirent**) structure (defined in ***dirent.h***)
    - Five main fields:
      1. ***d_ino***: I-node number
      2. ***Offset***: displacement to file start
      3. ***d_reclen***: size of file
      4. ***d_type***: type of file
      5. ***d_name***: name of file

```
struct dirent
{
  __ino_t d_ino;
  __off_t d_off;
  unsigned short int d_reclen;
  unsigned char d_type;
  char d_name[256];
};
```

  - int *stat*( char* entryName, stat* st);
    - Two steps to use this functionality
      1. Status information of stored item saved in structure type **stat** (defined in ***fat.h***)
      2. Check the kind of entry: subdirectory or not: macro ***S_ISDIR(st->st_mode)*** will return ***true*** if the item is a subdirectory, or ***false*** if it is a regular file.

# Libfat API: Directories management – Create / Delete

- Create a directory
  - bool *mkdir*( char* dirPath, mode_t mode );
    - The new directory must be included in an existing one
    - The second parameter establishes the permit bits.  An OR mask can be used with the following macros:
      - S_IREAD, S_IWRITE, S_IEXEC

- Remove a directory
  - bool *remove*( char* dirPath );
    - If the path is correct, the function deletes the directory and returns true
    - If the directory does not exist, it only returns false

# Libfat API:
# Files management – Open / close

- Open a file
  - FILE* ***fopen***( char* filePath, char* mode );
    - It returns a file descriptor if it succeeds, otherwise it returns ***NULL***
    - Six modes can be specified to open a file:
      1. "r" : open for reading
      2. "r+" : open for reading and writing
      3. "w" : open for writing
      4. "w+" : open for reading and writing. Creates the file if it does not exist
      5. "a" : open for appending (at the end of the existing file)
      6. "a+" : open for reading (at the beginning) and appending (at the end)
- Close an opened file
  - bool ***fclose***( FILE* f_id );
    - All the opened files **must be closed** before the application finishes

# Standard C: Files management – Read / Write

- Read data from a file
  - int ***fscanf***( FILE* file_id, char* format, &param1, &param2, …. );
    - Read the parameters specified in the given format string (second parameter) from the file specified by ***file_id***
    - Similar padding process to ***scanf***
      - The parameters are pointers to specific data types
    - Example: Reading name, surname and age from a file called "**identity.txt**"

```
FILE* f = fopen("/identity.txt", "r");
char name[256], surname[256];
int age;
fscanf(f, "%s %s %i", name, surname, &age);
```

  - int ***fread***( void* buff, size_t sizeElem, size_t nElem, FILE* file_id);
    - Read of ***nElem*** elements with a fixed size per elements (***sizeElem***)
    - Input: pointer to file specified by ***file_id***
    - Output: data is saved in the ***buff*** buffer

# Standard C: Files management – Read / Write

- Write data in a file
  - int ***fprintf***( FILE* file_id, char* format, param1, param2, …. );
    - Same functionality as ***printf***, but output written in file instead of on terminal
    - Parameters are associated (padded) with the input formatted string
    - Example: Printing an address in a file called "**myfile.txt**"

```
FILE* f = fopen("/myfile.txt", "w+");
int number = 4, zip = 1015;
char* str = "New Orleans Street";
fprintf(f, "Street: %s, Number: %i, ZIP Code: %i", str, number, zip);
```

  - int ***fwrite***( void* buff, size_t sizeElem, size_t nElem, FILE* file_id);
    - Write of ***nElem*** elements with a fixed size per elements (***sizeElem***)
    - Input: buffer (***buff***)
    - Ouput: file specified by ***file_id***

# Example: Listing Root directory of NDS into a file

- How can we create an NDS program that lists all the element of the root directory (“***/***”) and write the list in a file (“***List.txt***”) in the same directory

- The main steps to follow are the following:
  1. Include necessary header files for the library
  2. Initialize the ***libfat*** library
  3. Declare the necessary structures (DIR*, dirent*****, stat).
  4. Open the necessary files and directories
  5. Read / Write files or directories
     1. Get directory entry
     2. Get the status information
     3. Write the corresponding entry in the file
  6. Close the files and directories

# Example: Listing Root directory of NDS into a file

1. Include the necessary files
   - "***fat.h***" includes the initialization of the library
   - "***sys/dir.h***" includes DIR* manipulation (opendir, closedir)
   - "***dirent.h***" Includes the dirent declaration.
2. Initialize the library
   - Default configuration
3. Declare the necessary structures and variables
   - Files and directories
4. Open files and directories

```c
#include <nds.h>
#include <stdio.h>

#include <sys/dir.h>
#include <fat.h>
#include <dirent.h>


int main(void) {
    //Initialization
    fatInitDefault();

    //Structures
    DIR* di;
    struct dirent *pent;
    FILE* file;
    struct stat st;

    //Open directory
    di = opendir("/");
    //Open file
    file = fopen("/List.txt", "w+");
```

# Example: Listing Root directory of NDS into a file

5. Read / Write files and directories
   A. Get Entry
   B. Get stat
   C. Write in file

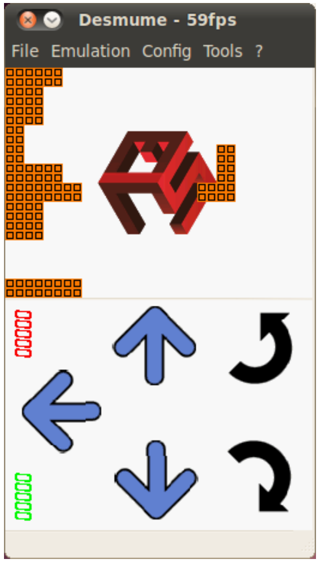
6. Close files or directories

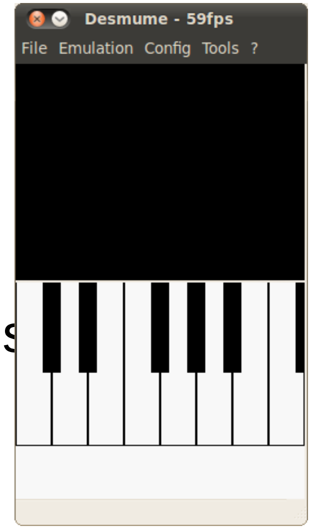
```
    //Open directory
    di = opendir("/");
    //Open file
    file = fopen("/List.txt", "w+");

    //List the items of the directory into the file
    while((pent=readdir(di))!=NULL)
    {
        stat(pent->d_name,&st);
        fprintf(file, "%s: %s\n",
                (S_ISDIR(st.st_mode ) ? "DIR" : "FILE"),
                pent->d_name);
    }

    //Close opened structures
    closedir(di);
    fclose(file);

    while(1) {
        swiWaitForVBlank();
    }
}
```

# Practical Work 11:
# File system use in the NDS

- Exercises
  - Exercise 1 – Listing root directory of the NDS into a file
  - Exercise 2 – Listing all files of a NDS unit into a file
  - Exercise 3 – Tetris Game: Inserting the score counters in the bottom screen
  - Exercise 4 – Tetris Game: Managing score in the Tetris Game
  - Exercise 5 – Tetris Game: Storing and retrieving highest score in the Tetris Game

  - *Exercise 6 – Piano Player: Listing melody files into the music directory
  - *Exercise 7 – Piano Player: Playing stored melodies
  - *Exercise 8 – Piano Player: Playing complex melodies with note length




* Additional exercises

# Questions?

# Let’s use the file system in the NDS!