Operating Systems (CS 3430/5430) Lecture 28 Name:_____

1. POSIX File API - interface between OS kernel and software running at the user-level. You can and should largely avoid by using a cleaner higher-level API built on top of it (e.g., Java and C++ file standards).

file descriptor (integer) ← open("path/filename", access rights requested)

All processes inherit at least three file descriptors form their parent:

Common name:	standard input	standard output	standard error output
File descriptor:	0	1	2
unistd.h name:	STDIN_FILENO	STDOUT_FILENO	STDERR_FILENO
Default meaning when run for shell:	keyboard	shell window	shell window

a) What happens when you redirect standard output to a file (e.g., ps l > my-processes.txt)? (See Figure 8.2)

b) How does the shell pipe ('|') the output of one command as input to another command (ls | grep .pdf)?

2. File *metadata* ("data about data") attributes can be retrieved using `fstat` call.

```
>g++ -o fstater fstater.cpp
>fstater < /etc/passwd
Standard input is owned by user number 0
and was last modified Thu Jun 25 14:52:07 2009
It is a 1082-byte file.
>ls -ln /etc/passwd
-rw-r--r-- 1 0 0 1082 Jun 25  2009 /etc/passwd
>
```

In the "data about data":

a) What does the first "data" apply to?

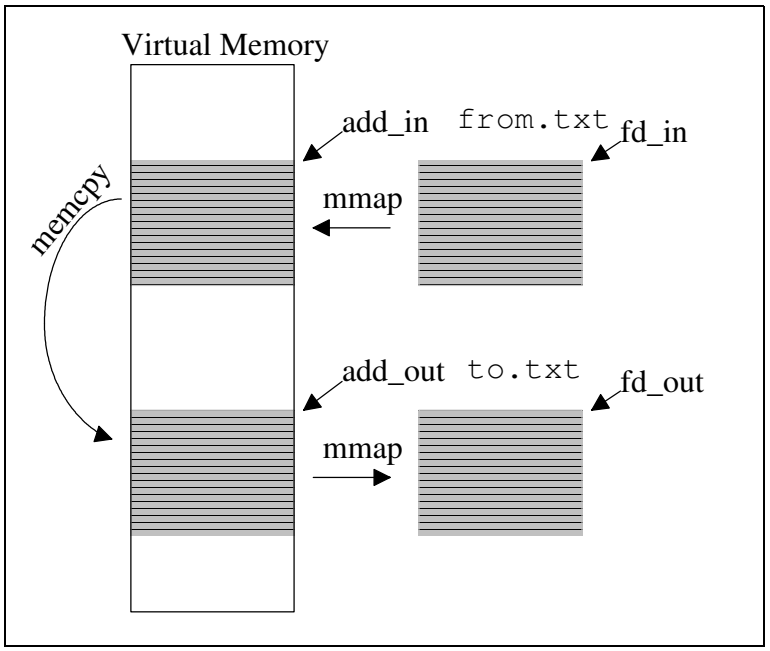
b) What does the second "data" apply to?

```
#include <unistd.h>
#include <time.h>
#include <sys/stat.h>
#include <stdio.h>
#include <iostream>
using namespace std;

int main(){
    struct stat info;
    if(fstat(STDIN_FILENO, &info) < 0){
        perror("Error getting info about standard input");
        return -1;
    }
    cout << "Standard input is owned by user number "
         << info.st_uid << endl;
    cout << "and was last modified " << ctime(&info.st_mtime);
    if(S_ISREG(info.st_mode)){
        cout << "It is a " << info.st_size << "-byte file." << endl;
    } else {
        cout << "It is not a regular file." << endl;
    }
    return 0;
}
```

3. POSIX API provides three ways to access (read/write) a file:

	Explicit Positions	Sequential
Memory Mapped	mmap	—
External	pread/pwrite	read/write

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <stdio.h>
#include <string.h>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]){
  if(argc != 3){
    cerr << "Usage: " << argv[0] << " infile outfile" << endl;
    return -1;
  }
  int fd_in = open(argv[1], O_RDONLY);
  if(fd_in < 0){
    perror(argv[1]);
    return -1;
  }
  struct stat info;
  if(fstat(fd_in, &info) < 0){
    perror("Error stating input file");
    return -1;
  }
  void *addr_in =
    mmap(0, info.st_size, PROT_READ, MAP_SHARED, fd_in, 0);
  if(addr_in == MAP_FAILED){
    perror("Error mapping input file");
    return -1;
  }
  int fd_out =
    open(argv[2], O_RDWR | O_CREAT | O_TRUNC, S_IRUSR |
                 S_IWUSR);
  if(fd_out < 0){
    perror(argv[2]);
    return -1;
  }
  if(ftruncate(fd_out, info.st_size) < 0){
    perror("Error setting output file size");
    return -1;
  }
  void *addr_out =
    mmap(0, info.st_size, PROT_WRITE, MAP_SHARED, fd_out, 0);
  if(addr_out == MAP_FAILED){
    perror("Error mapping output file");
    return -1;
  }
  memcpy(addr_out, addr_in, info.st_size);
  return 0;
}
```



Consider the cpmm.cpp code from Figures 8.4 and 8.5
Consider the call to executable program:

```
> cpmm from.txt to.txt
```

Relate the disadvantages of using mmap to this example:

a) process has no easy way to control time its updates are made persistent

b) process can write file only if it has read and write permissions

c) mapping a file into a range of addresses presumes you know the size of the file. Typically, hard to know in advance how much data will be written to the file.

d) What is the difference between the "external" access methods `pread/pwrite` vs. `read/write`?

4. OS views the disk as a linear sequence of block (0, 1, 2, ...):

Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7	Block 8	Block 9	...

a) Why is the Block size a multiple of the disk's sector size?

One definition of *fragmentation* (not the authors) - files scattered on disk into multiple *extents* (group of blocks). An OS disk defragmentation program

- move files into contiguous disk block, so each file has one extent, and
- moves all files so free blocks cluster together (e.g., one end of the disk)

b) How does disk defragmentation improve system performance?

c) Author's meaning of *fragmentation* is split into two parts:

- *internal fragmentation* - unused space in the last file block -- On average, how much internal frag. per file?

- *external fragmentation* - small blocks of free space between used blocks on the disk

Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7	Block 8	Block 9	...
File 1			Free	Free	File 2			Free	File 3	

What impact does increasing the file system block size have on:

- *internal fragmentation*?

- *external fragmentation*?

5. Locality - When allocating space for a file to improve access time. The general OS guidelines are:

a) Each file should be broken into as few extents as possible. Why?

b) If files need to be allocated to multiple extents, extents should be close together on the disk. Why?

c) Files commonly used together should be close together on the disk. How could the OS guess which files might be used together?

6. How big of files can be handled if 10 entries in the inode, 1024 entries per indirect block, and 4 KB blocks?

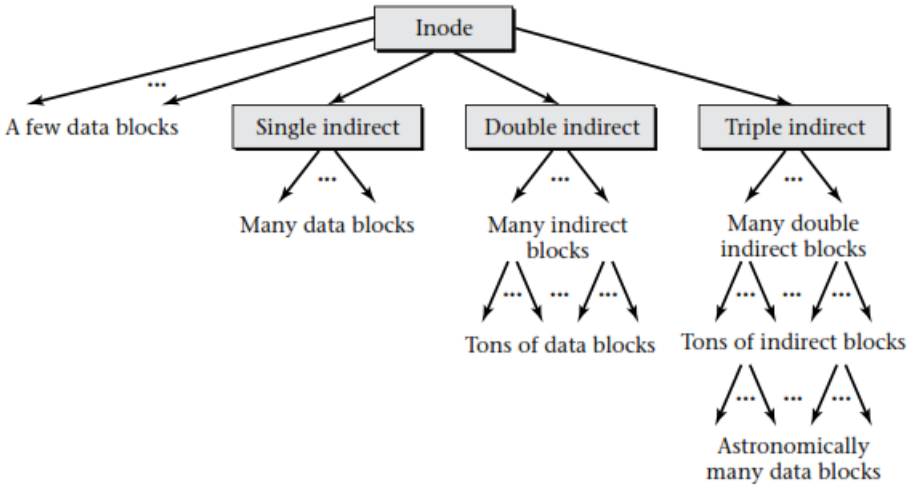


Figure 8.10: The full structure of a file starts with an inode and continues through a tree of single, double, and triple indirect blocks, eventually reaching each of the data blocks.

7. In the B+ extent map, how do we local block 95?

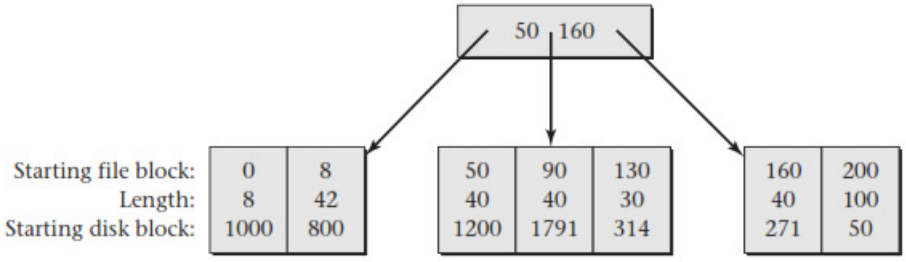


Figure 8.13: This small B^+-tree extent map contains information that can be used to find each extent's range of file block numbers and range of disk block numbers. Because the tree is a B^+-tree rather than a B-tree, all the extents are described in the leaves, with the nonleaf node containing just navigational information.