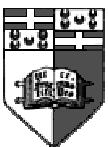




File I/O

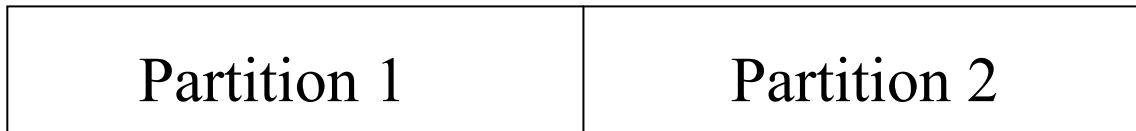
- In this section we will cover unbuffered I/O which is at the lowest level in UNIX.
- Knowledge of `fopen()`, `getc()`, `putc()`, `fread()` and `fwrite()` is assumed.
- We will be using file descriptors instead of `FILE*` objects.
- All this allows for lower level access and direct manipulation of files, symbolic links and directories



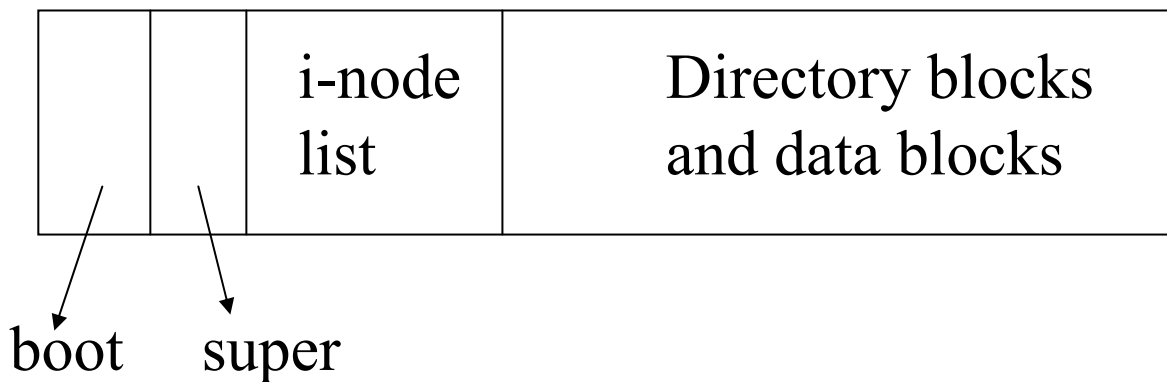


Filesystem Structure

Disk Drive Details



Partition Details

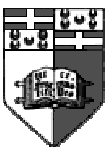


Typical I-node

type of file
of links
U-ID
G-ID
size
time
list of addresses

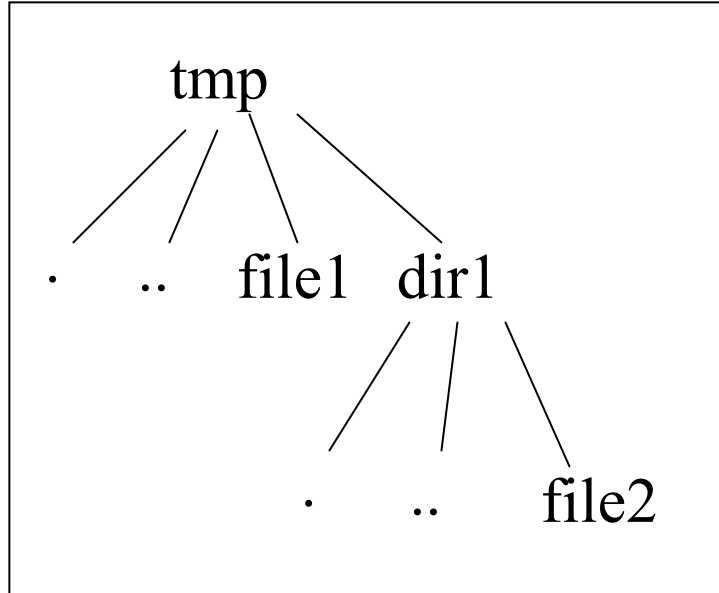
Typical Directory block

i-node no	.
i-node no	..
i-node no	file1
i-node no	file2



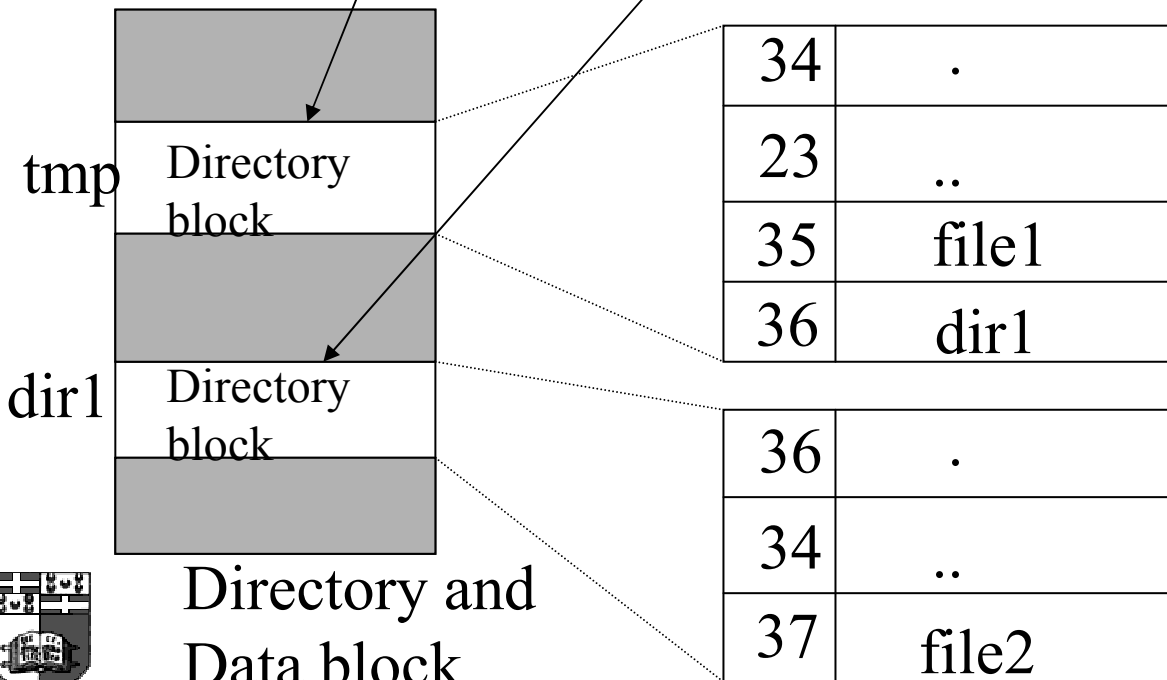


Filesystem Example

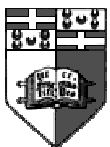


i-node list

	Node	Node	Node	Node	
	34	35	36	37	
	#3(+)	#1	#2	#1	



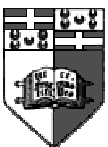
Directory and Data block





Filesystem (cont)

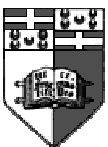
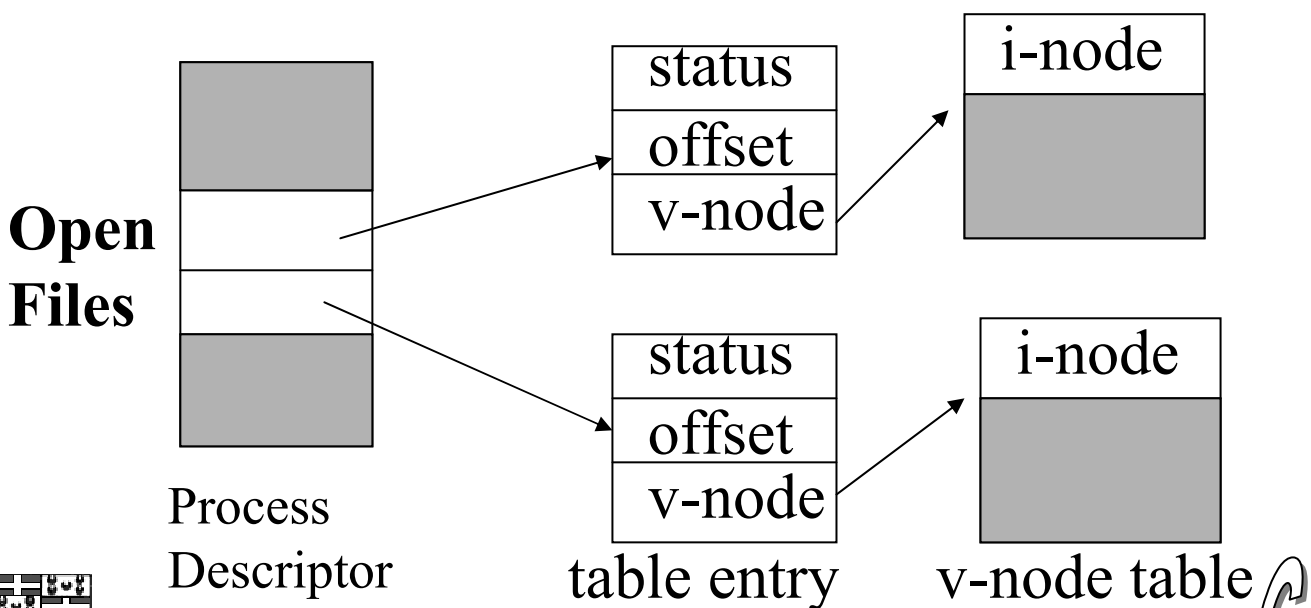
- Each i-node contains the number of links to that file (*ls -li*).
- Only when the number of links becomes 0 is that data block freed for reuse.
- A symbolic link contains the name of the file it points to inside its data block.
- Note: a leaf directory always has a link count of 2.





Process \Downarrow \diamond File

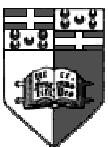
- Each process contains a process descriptor entry.
- This process descriptor contains a list of open files and for each a pointer to the file table entry.
- The file table entry contains the current file offset, the file status and a pointer to the v-node table entry.
- The v-node structure contains the file i-node together with a pointer to all the functions that operate on this file.



Unix

Process ↓ ◇ File (cont)

- This hierarchical structure allows more than one process to have the same file open (sharing v-node entries)
- In this case, each process would have a different file offset.
- When a file is opened for writing and the offset exceeds the i-node file size, the i-node is updated accordingly.
- Periodically and when closing the file, the I-node is rewritten to the file system.
- There is always one unique v-node table per file and it is stored in the kernel.
- Child processes share the same file table entry !!





File Descriptors

- In the kernel all files are given a non-negative integer as a reference up to `OPEN_MAX` (defined in `<limits.h>`).
- Three files are always open:
 - `STDIN_FILENO : 0`
 - `STDOUT_FILENO : 1`
 - `STDERR_FILENO : 2`
- All primitive file access goes through these file descriptors

```
#include <sys/types.h>
```

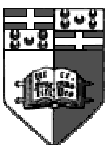
```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
int open(const char *pathname,int oflag)
```

```
int open(const char * pathname,int oflag,  
         mode_t mode)
```

```
int close(int filedes);
```





Basic File System Calls

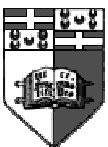
open oflag constants (OR'd together)

O_RDONLY	exclusive
O_WRONLY	
O_RDWR	
O_APPEND	
O_CREAT	
O_EXCL	error if O_CREAT on existing file
O_TRUNC	set length to 0
O_SYNC	reflect to physical I/O immediately

open mode constants (used in O_CREAT)

S_IRWXU	S_IRWXG	S_IRWXO
S_IRUSR	S_IRGRP	S_IROTH
S_IWUSR	S_IWGRP	S_IWOTH
S_IXUSR	S_IXGRP	S_IXOTH

- *open* returns the lowest numbered descriptor or -1 on error





Other File System Calls

- *lseek* sets the file offset in the table entry.
- *read* reads *nbytes* from the file into *buff* returning 0 on EOF.
- *write* writes *nbytes* to a file.

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
off_t lseek(int fd, off_t offset, int whence);
           return new file offset, -1 on error
```

```
ssize_t read(int fd, void *buff, size_t nbytes);
```

```
ssize_t write(int fd, const void *buff,
              size_t nbytes);
```

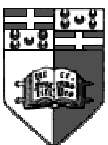
```
           return number of bytes written, -1 on error
```

whence constants:

SEEK_SET

SEEK_CUR

SEEK_END





dup and *dup2*

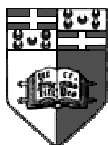
- These duplicate the current file descriptor yet retaining the same file table entry.
- *dup* returns the lowest numbered available file descriptor.
- Writing to a duplicated file descriptor changes the offset of the original file and vice versa.
- *dup2* closes `filedes2` if open and returns `filedes2`. If `filedes2 == filedes`, nothing occurs.

```
#include <unistd.h>
```

```
int dup(int filedes);
```

```
int dup2(int filedes, int filedes2);
```

```
return -1 on error
```





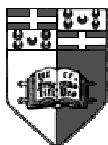
fcntl and *ioctl*

- *fcntl* is used to set or request properties of opened files.
- *ioctl* is a catch all function which gives raw access to all file attributes.
- Use these functions with caution since they are quite powerful

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <ioctl.h>
```

```
int fcntl(int filedes, int cmd, ...);
int ioctl(int filedes, int request,...);
```

- When opening a file in UNIX, a special file is created in `/dev/fd/n` where `n` is the file descriptor. This can be used for sharing file descriptors using the same table entry.





Exercises

- Try drawing up a typical directory hierarchy and then construct a typical i-node representation of it.
- Construct two programs that open two files and display the file descriptor number of each. Does the order of execution vary the file descriptor number.
- Try using `dup` to duplicate standard output and try writing to this file descriptor. Try *lseek*'ing on this new file.
- Familiarize yourself with `fcntl` and `ioctl`, learning their utility and power.

