Outline for Today

- Objective
  - Review of basic file system material
- Administrative
  - ??

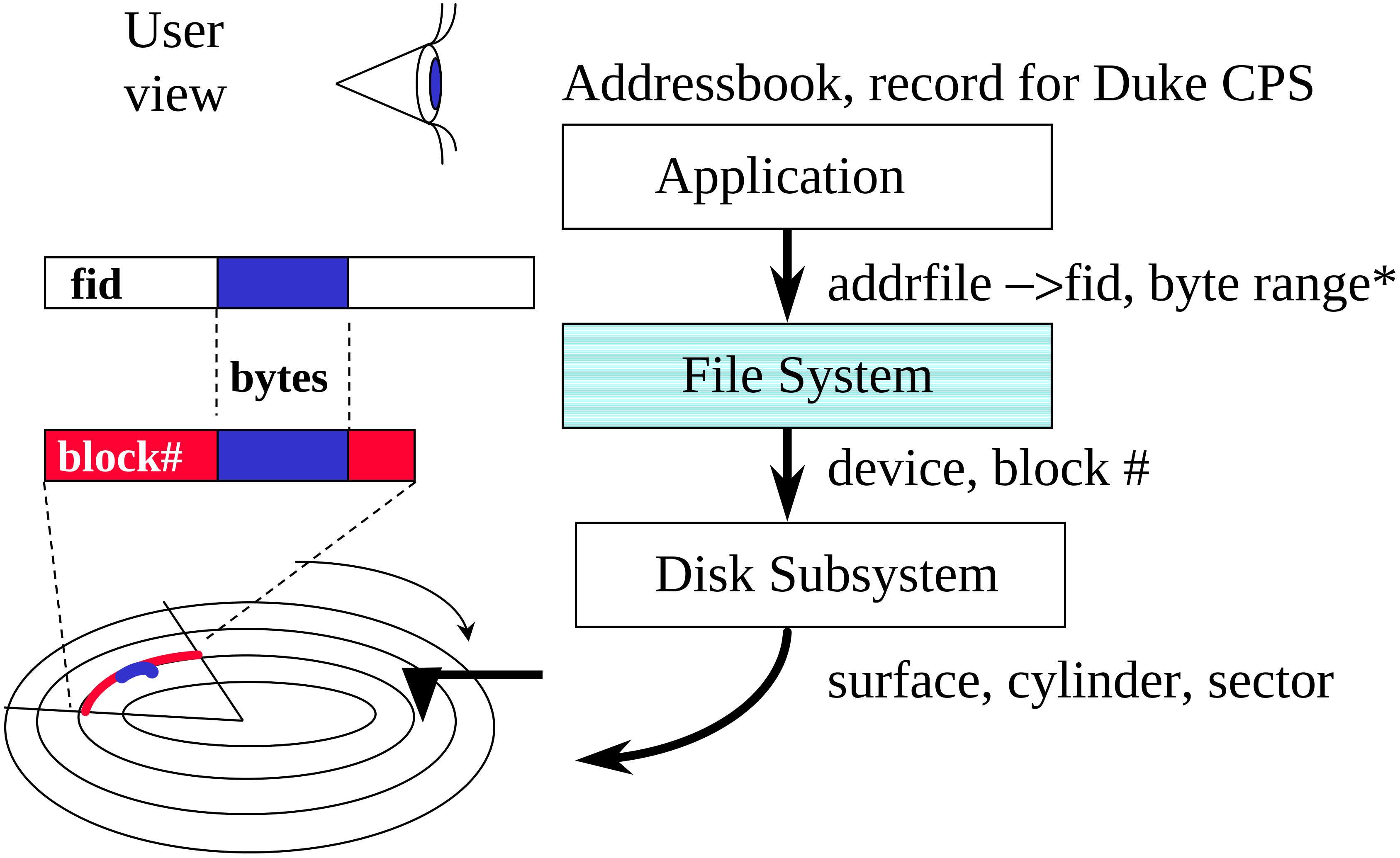
Review of File System Issues

- What is the *role* of files?
  What is the file abstraction?
- File naming. How to find the file we want?
  Sharing files. Controlling access to files.
- Performance issues - how to deal with the bottleneck of disks?
  What is the "right" way to optimize file access?

Role of Files

- Persistance – long-lived – data for posterity
    - → non-volitile storage media
    - → semantically meaningful (memorable) names

Abstractions



User
view

Addressbook, record for Duke CPS

Application

addrfile –>fid, byte range*

File System

device, block #

Disk Subsystem

surface, cylinder, sector

Functions of File System

- (Directory subsystem) Map filenames to fileids-open (create) syscall. Create kernel data structures. Maintain naming structure (unlink, mkdir, rmdir)
- Determine layout of files and metadata on disk in terms of blocks. Disk block allocation. Bad blocks.
- Handle read and write system calls
- Initiate I/O operations for movement of blocks to/from disk.
- Maintain buffer cache

Functions of Device Subsystem

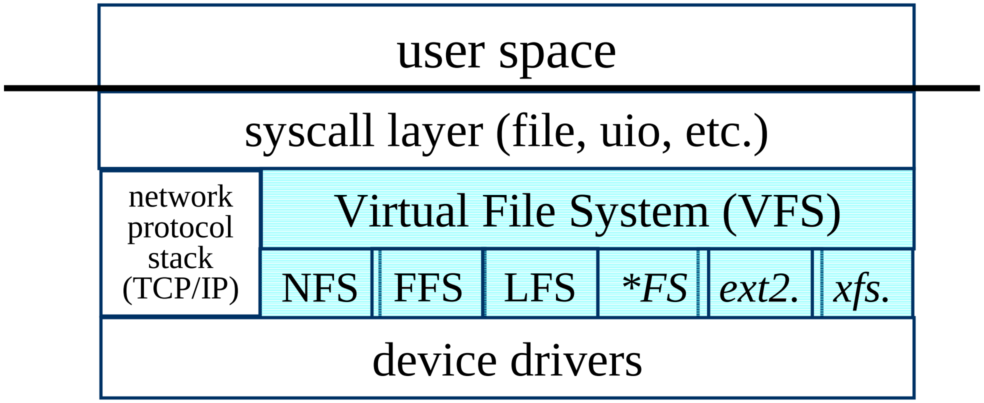
In general, deal with device characteristics

- Translate block numbers (the abstraction of device shown to file system) to physical disk addresses.
  Device specific (subject to change with upgrades in technology) intelligent placement of blocks.
- Schedule (reorder?) disk operations

VFS: the Filesystem Switch

Sun Microsystems introduced the *virtual file system* framework in 1985 to accommodate the Network File System cleanly.

- VFS allows diverse *specific file systems* to coexist in a file tree, isolating all FS-dependencies in pluggable filesystem modules.



Other abstract interfaces in the kernel: device drivers, file objects, executable files, memory objects.

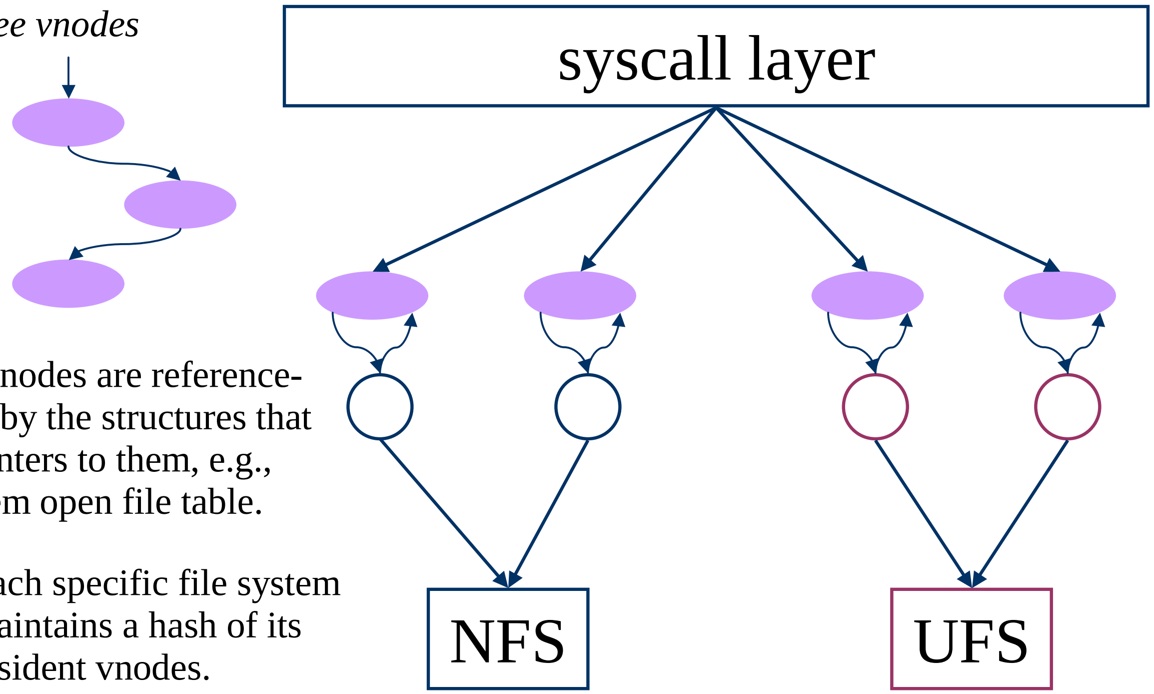
VFS was an internal kernel restructuring with no effect on the syscall interface.

Incorporates object-oriented concepts: a generic procedural interface with multiple implementations.

Based on abstract objects with dynamic method binding by type...in C.

Vnodes*

In the VFS framework, every file or directory in active use is represented by a *vnode* object in kernel memory.



Active vnodes are reference-counted by the structures that hold pointers to them, e.g., the system open file table.

Each specific file system maintains a hash of its resident vnodes.

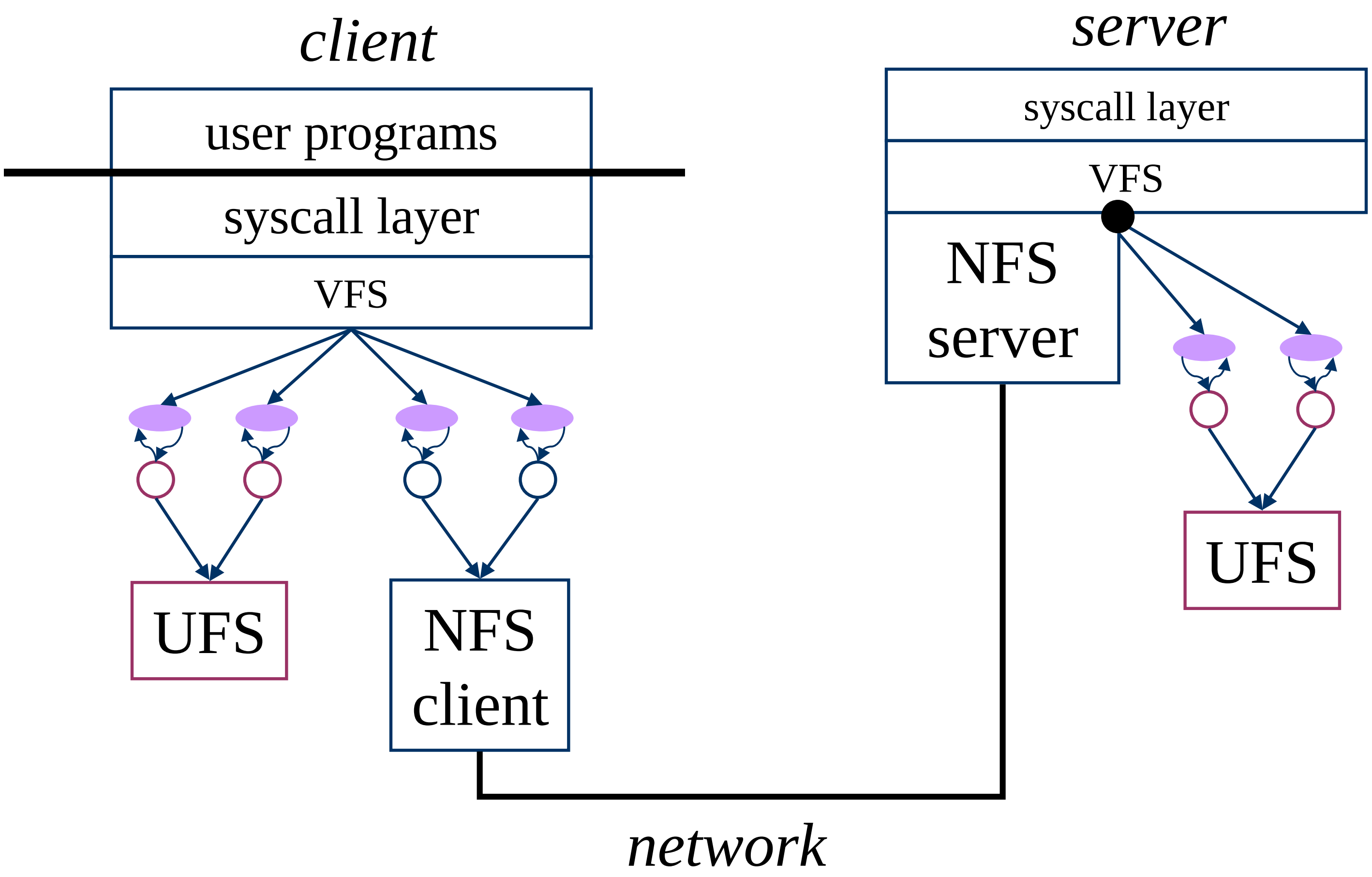
Each vnode has a standard *file attributes* struct.

Generic vnode points at filesystem-specific struct (e.g., *inode*, *rnode*), seen only by the filesystem.

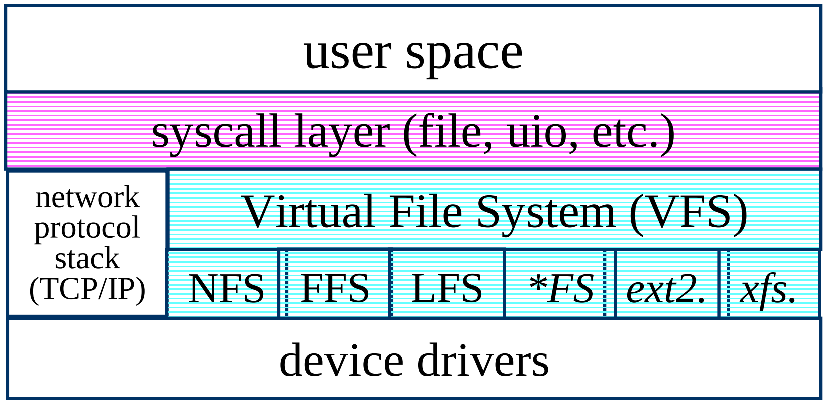
Vnode operations are macros that vector to filesystem-specific procedures.

*inode object in Linux VFS

Network File System (NFS)



File Abstractions

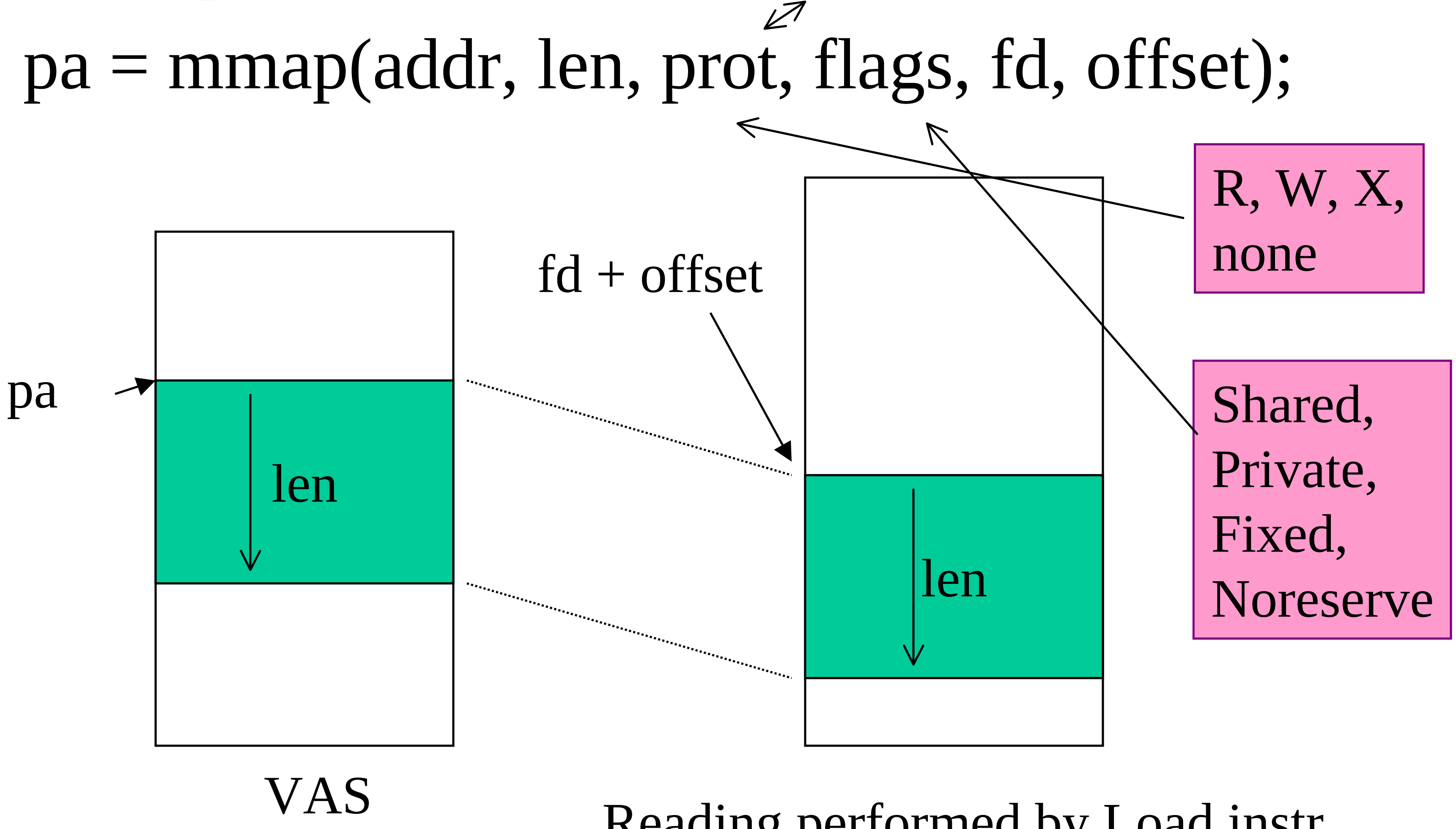


- UNIX-like files
  - Sequence of bytes
  - Operations: open (create), close, read, write, seek
- Memory mapped files
  - Sequence of bytes
  - Mapped into address space
  - Page fault mechanism does data transfer
- Named, Possibly typed

Memory Mapped Files

fd = open (somefile, *consistent_mode*);

pa = mmap(addr, len, prot, flags, fd, offset);



R, W, X, none

Shared, Private, Fixed, Noreserve

Reading performed by Load instr.

UNIX File System Calls

*Open files are named to by an integer **file descriptor**.*

*Pathnames may be relative to process **current directory**.*

```
char buf[BUFSIZE];
int fd;

if ((fd = open(“../zot”, O_TRUNC | O_RDWR) == -1) {
        perror(“open failed”);
        exit(1);
}
while(read(0, buf, BUFSIZE)) {
        if (write(fd, buf, BUFSIZE) != BUFSIZE) {
                perror(“write failed”);
                exit(1);
        }
}
```

*Process passes status back to parent on **exit**, to report success/failure.*

*Process does not specify current file offset: the **system** remembers it.*

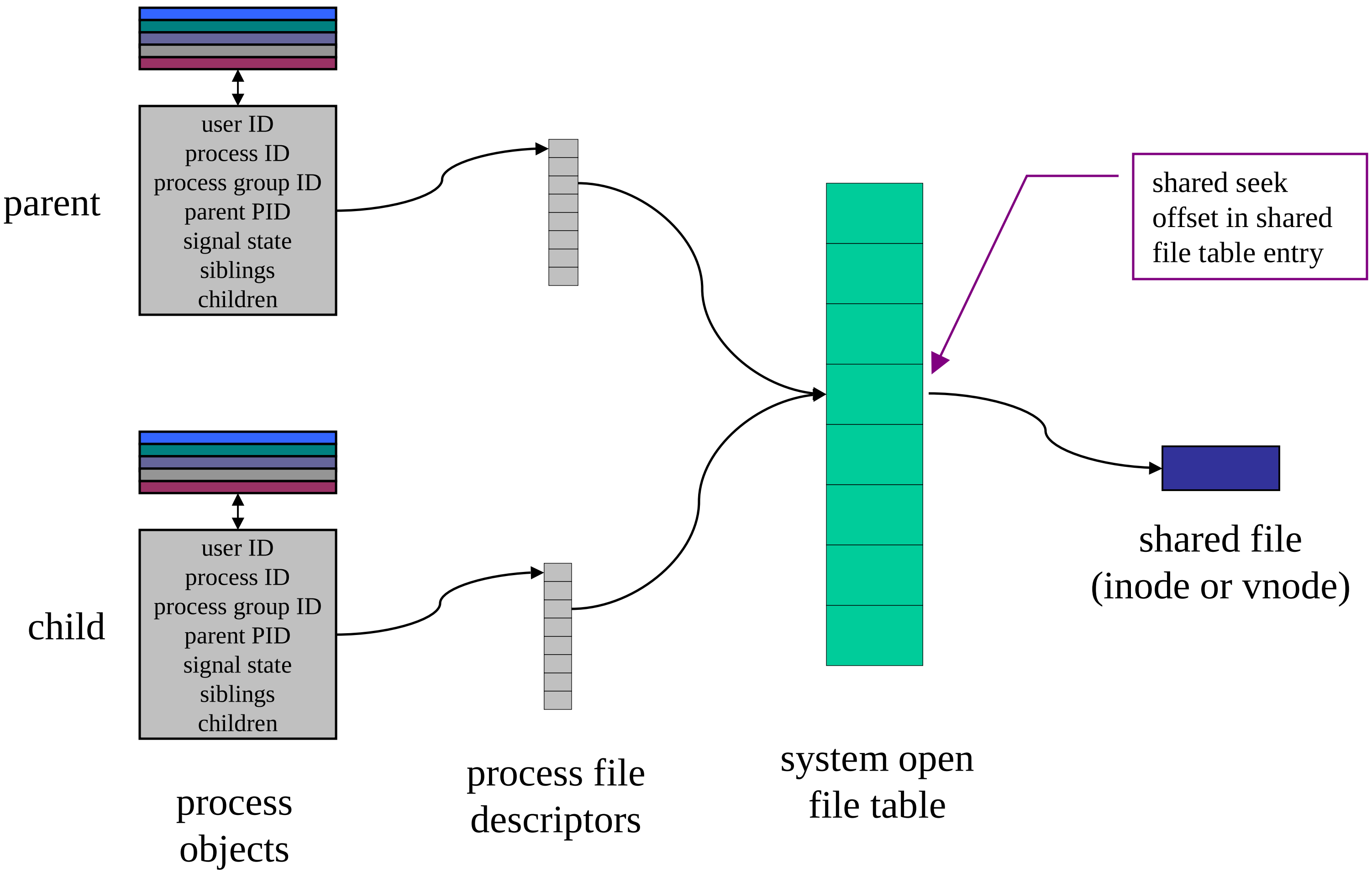
*Standard descriptors (0, 1, 2) for input, output, error messages (**stdin**, **stdout**, **stderr**).*

File Sharing Between Parent/Child (UNIX)

```
main(int argc, char *argv[]) {
        char c;
        int fdrd, fdwt;

        if ((fdrd = open(argv[1], O_RDONLY)) == -1)
                exit(1);
        if ((fdwt = creat([argv[2], 0666)) == -1)
                exit(1);

        fork();

        for (;;) {
                if (read(fdrd, &c, 1) != 1)
                        exit(0);
                write(fdwt, &c, 1);
        }
}
```

[Bach]

Sharing Open File Instances



parent

user ID
process ID
process group ID
parent PID
signal state
siblings
children

child

user ID
process ID
process group ID
parent PID
signal state
siblings
children

process objects

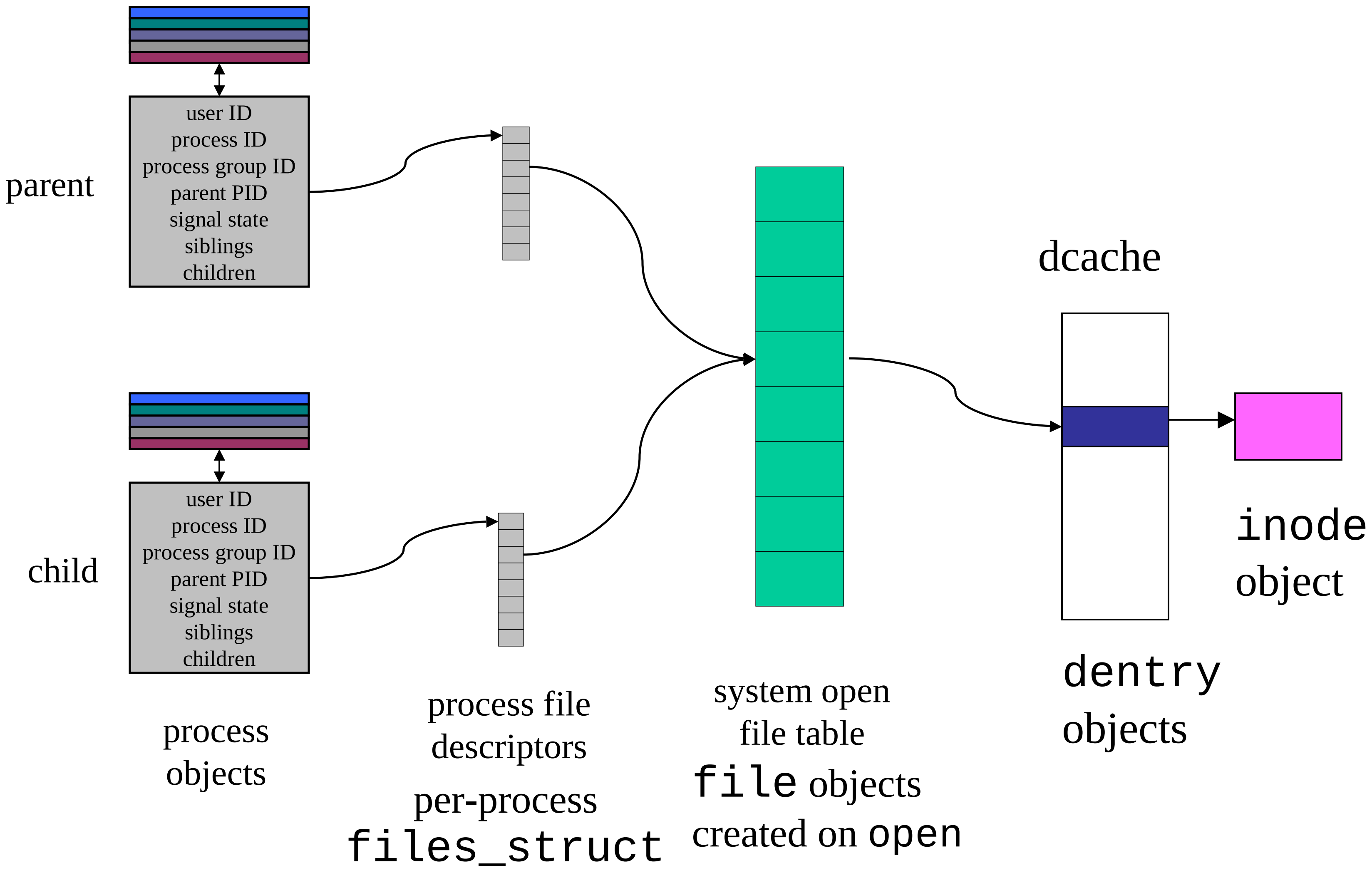
process file descriptors

system open file table

shared seek offset in shared file table entry

shared file (inode or vnode)

Corresponding Linux File Objects



parent

user ID
process ID
process group ID
parent PID
signal state
siblings
children

child

user ID
process ID
process group ID
parent PID
signal state
siblings
children

process
objects

process file
descriptors

per-process
`files_struct`

system open
file table
`file` objects
created on `open`

dcache

`dentry`
objects

`inode`
object

Goals of File Naming

- Foremost function - to find files,
  Map file name to file object.
- To store ***meta-data*** about files.
- To allow users to choose their own file names without undue ***name conflict*** problems.
- To allow ***sharing***.
- Convenience: ***short*** names, groupings.
- To avoid implementation complications

Meta-Data

- File size
- File type
- Protection - access control information
- History:
  creation time,
  last modification,
  last access.
- Location of file - which device
- Location of individual blocks of the file on disk.
- Owner of file
- Group(s) of users associated with file

Operations on Directories (UNIX)

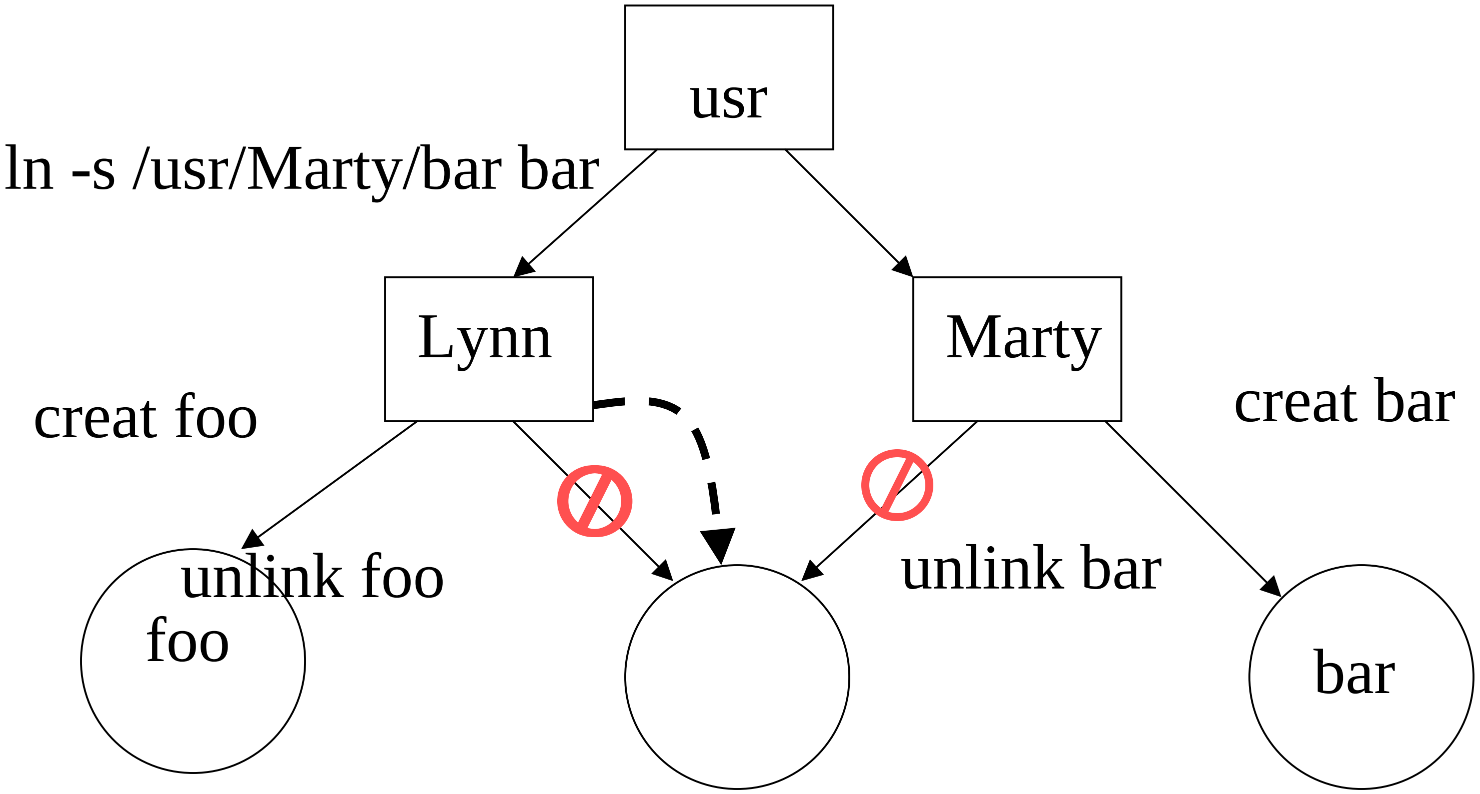
- Link - make entry pointing to file
- Unlink - remove entry pointing to file
- Rename
- Mkdir - create a directory
- Rmdir - remove a directory

Naming Structures

Naming Hierarchy

- Component names - ***pathnames***
  - ***Absolute*** pathnames - from a designated *root*
  - ***Relative*** pathnames - from a ***working directory***
  - Each name carries how to resolve it.
  - No cycles – allows reference counting to reclaim deleted nodes.
- Links
  - Short names to files anywhere for convenience in naming things – ***symbolic links*** – map to pathname

Links



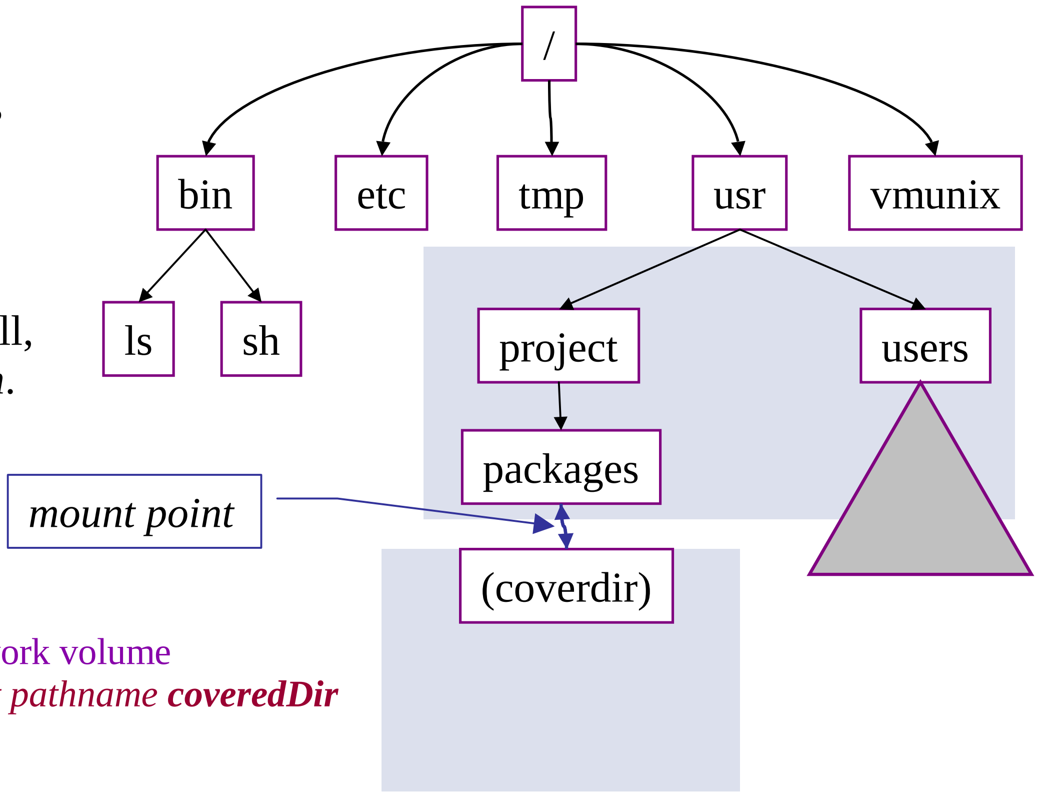
usr
ln -s /usr/Marty/bar bar
Lynn
Marty
creat foo
creat bar
unlink foo
foo
unlink bar
bar

A Typical Unix File Tree

Each volume is a set of directories and files; a host's *file tree* is the set of directories and files visible to processes on a given host.

File trees are built by *grafting* volumes from different devices or from network servers.

In Unix, the graft operation is the privileged *mount* system call, and each volume is a *filesystem*.


/
bin
etc
tmp
usr
vmunix
ls
sh
project
users
packages
mount point
(coverdir)

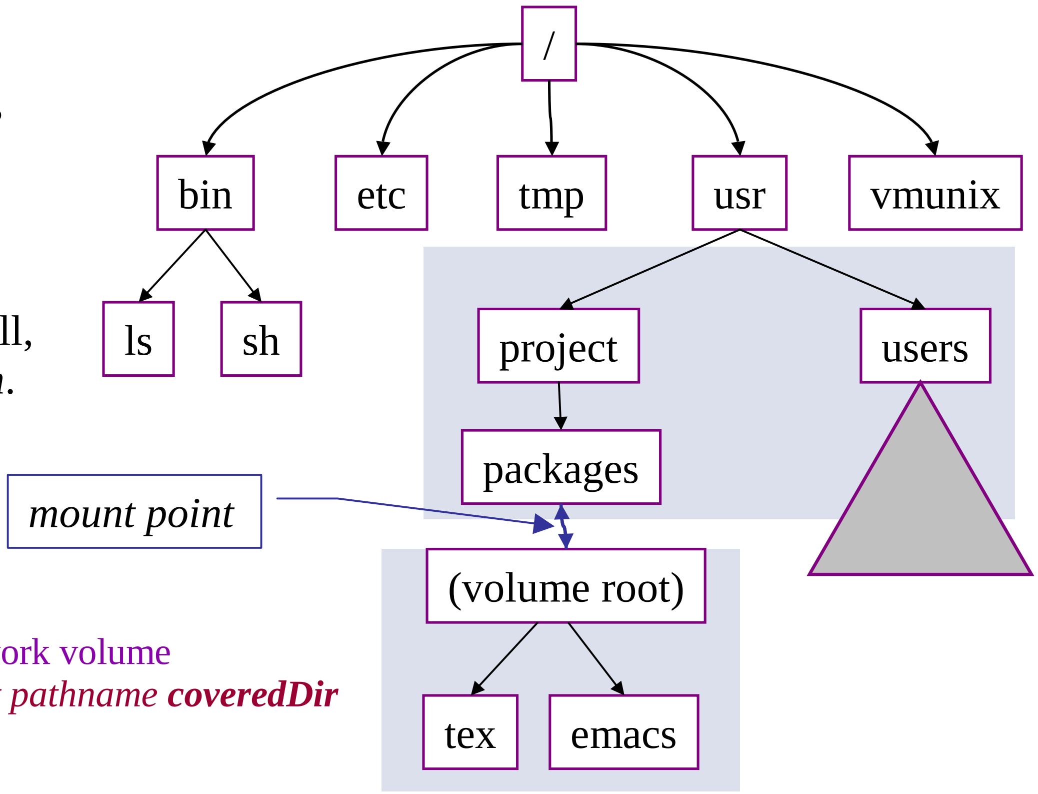
mount (coveredDir, volume)
coveredDir: directory pathname
volume: device specifier or network volume
volume root contents become visible at pathname ***coveredDir***

A Typical Unix File Tree

Each volume is a set of directories and files; a host's *file tree* is the set of directories and files visible to processes on a given host.

File trees are built by *grafting* volumes from different devices or from network servers.

In Unix, the graft operation is the privileged *mount* system call, and each volume is a *filesystem*.



mount (coveredDir, volume)
coveredDir: directory pathname
volume: device specifier or network volume

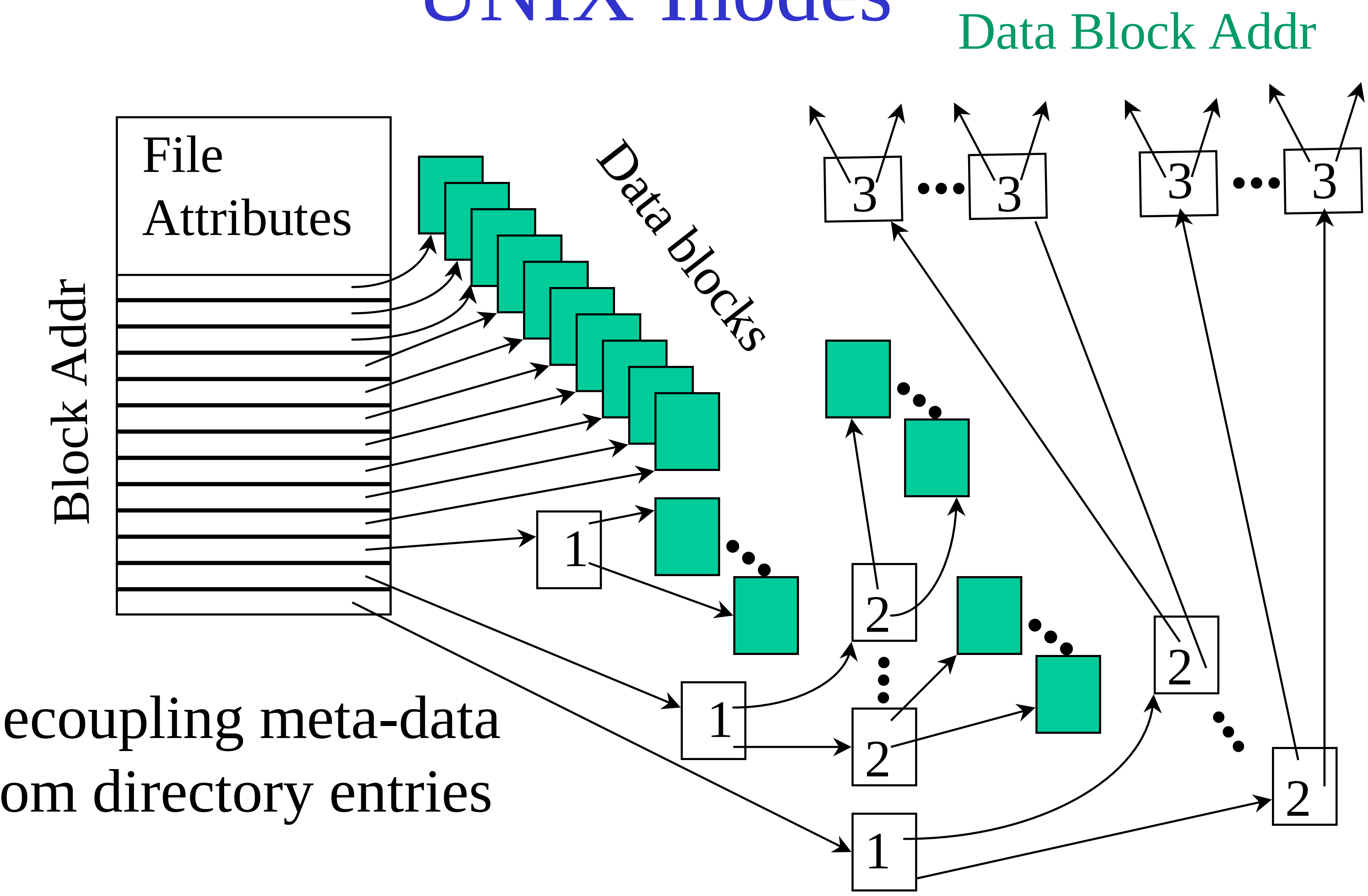
*volume root contents become visible at pathname **coveredDir***

/usr/project/packages/coverdir/tex

Access Control for Files

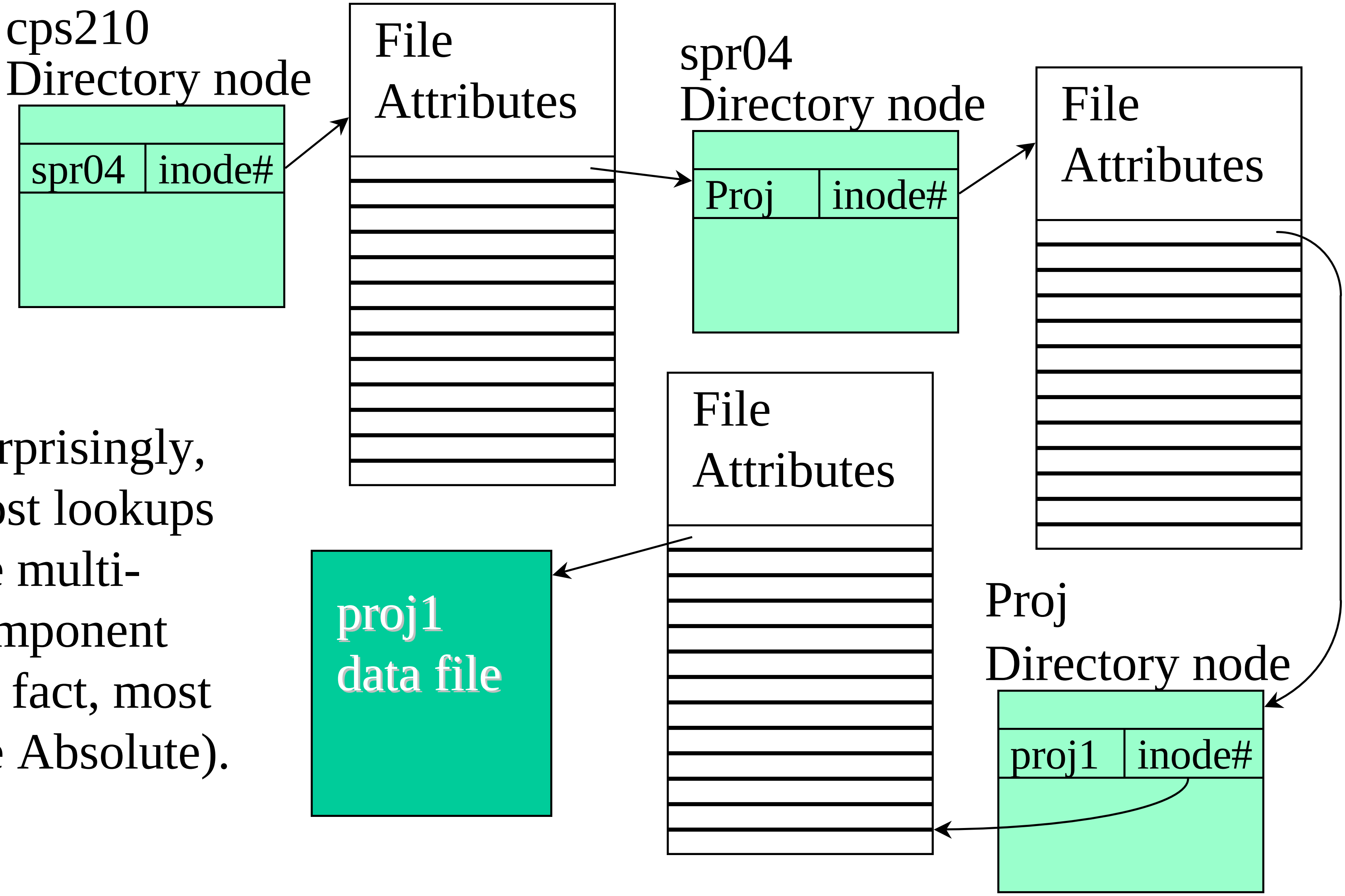
- Access control lists - detailed list attached to file of users allowed (denied) access, including kind of access allowed/denied.
- UNIX RWX - owner, group, everyone

Implementation Issues: UNIX Inodes



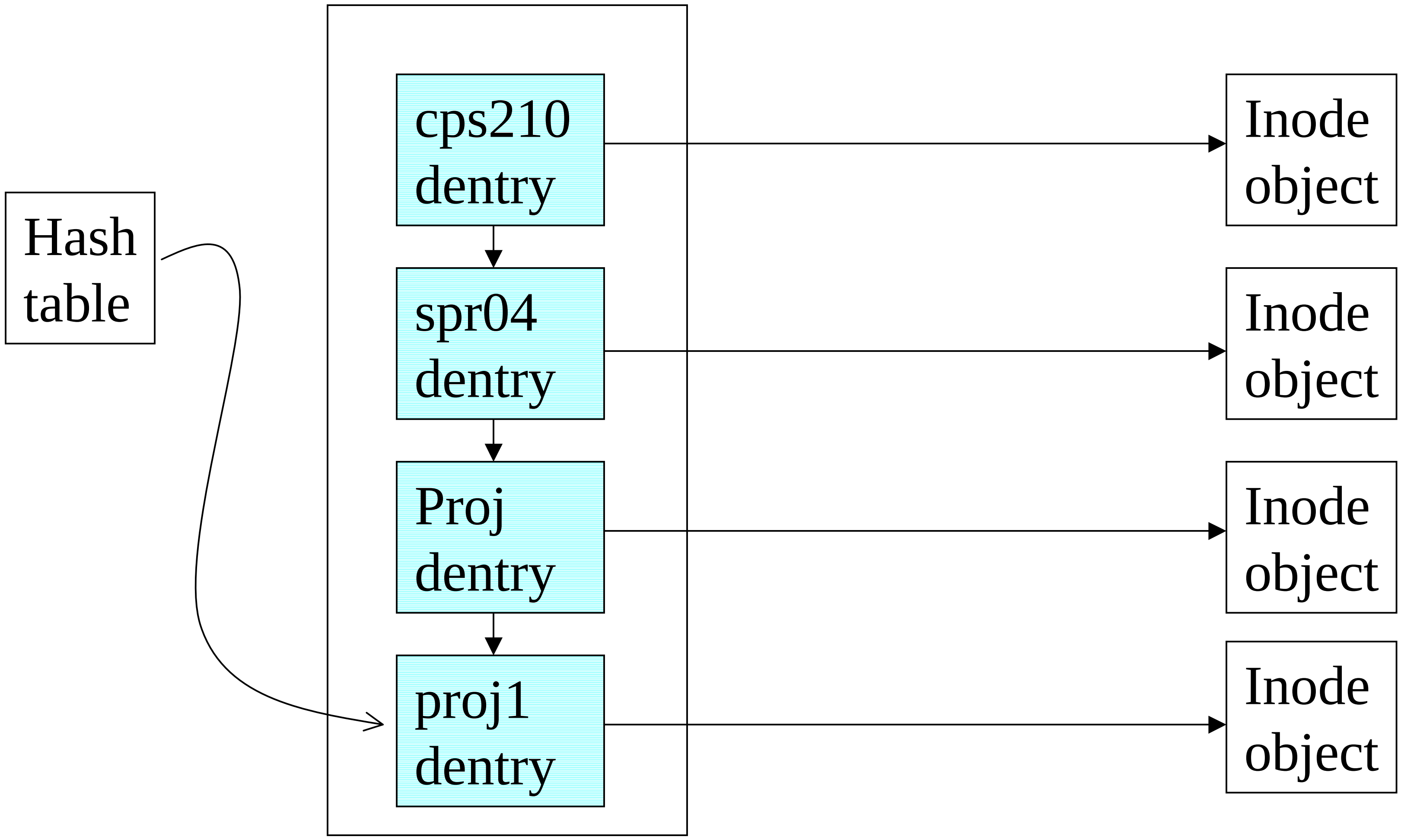
Decoupling meta-data from directory entries

Pathname Resolution



Surprisingly, most lookups are multi-component (in fact, most are Absolute).

Linux dcache



Hash table

cps210 dentry

spr04 dentry

Proj dentry

proj1 dentry

Inode object

Inode object

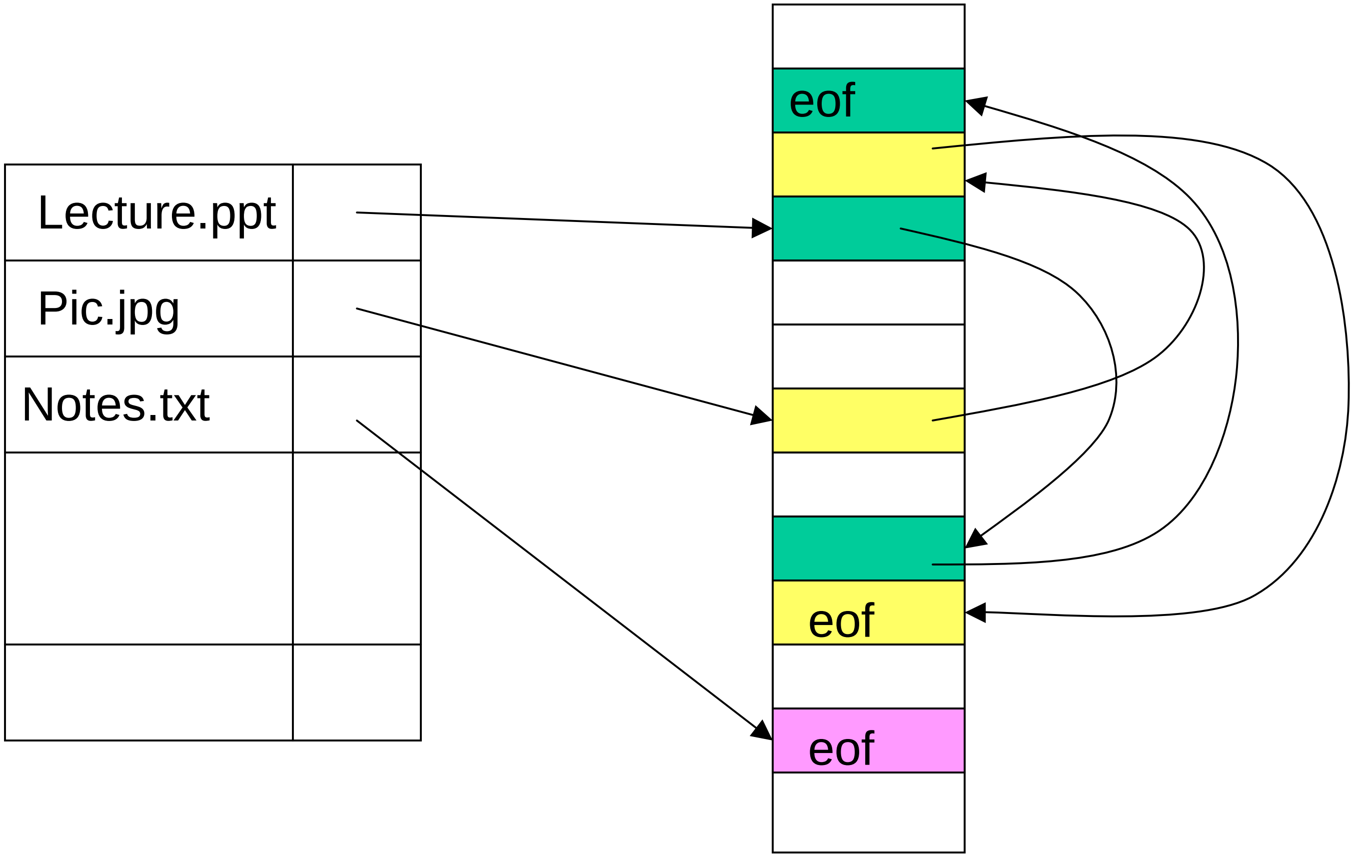
Inode object

Inode object

File Structure Alternatives

- Contiguous
  - 1 block pointer, causes fragmentation, growth is a problem.
- Linked
  - each block points to next block, directory points to first, OK for sequential access
- Indexed
  - index structure required, better for random access into file.

File Allocation Table (FAT)

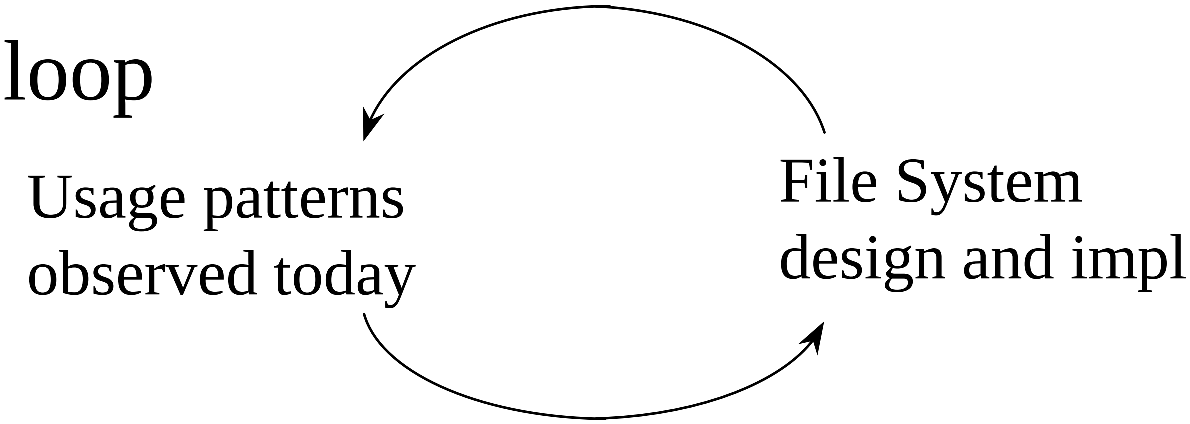


Finally Arrive at File

- What do users *seem* to want from the file abstraction?
- What do these usage patterns mean for file structure and implementation decisions?
  - What operations should be optimized 1st?
  - How should files be structured?
  - Is there temporal locality in file usage?
  - How long do files really live?

Know your Workload!

- File usage patterns should influence design decisions. Do things differently depending:
  - How large are most files? How long-lived? Read vs. write activity. Shared often?
  - Different levels “see” a different workload.
- Feedback loop



Usage patterns observed today

File System design and impl

Generalizations from UNIX Workloads

- Standard Disclaimers that you can’t generalize…but anyway…
- Most files are small (fit into one disk block) although most bytes are transferred from longer files.
- Most opens are for read mode, most bytes transferred are by read operations
- Accesses tend to be sequential and 100%

More on Access Patterns

- There is significant reuse (re-opens) – most opens go to files repeatedly opened & quickly. Directory nodes and executables also exhibit good temporal locality.
  - Looks good for caching!
- Use of temp files is significant part of file system activity in UNIX – very limited reuse, short lifetimes (less than a minute).

What to do about long paths?

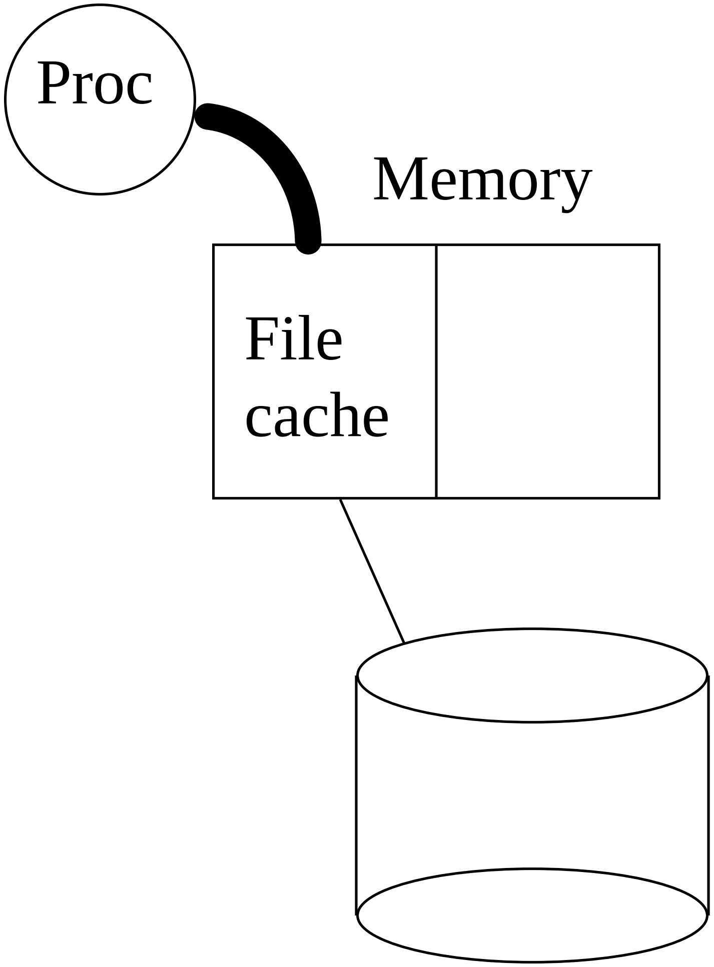
- Make long lookups cheaper – cluster inodes and data on disk to make each component resolution step somewhat cheaper
  - Immediate files – meta-data and first block of data co-located
- Collapse prefixes of paths – hash table
  - Prefix table
- “Cache it” – in this case, directory info

What to do about Disks?

- Disk scheduling
  - Idea is to reorder outstanding requests to minimize seeks.
- Layout on disk
  - Placement to minimize disk overhead
- Build a <u>better</u> disk (or substitute)
  - Example: RAID

File Buffer Cache

- ***Avoid*** the disk for as many file operations as possible.
- Cache acts as a filter for the requests seen by the disk – reads served best.
- Delayed writeback will avoid going to disk at all for temp files.



Handling Updates in the File Cache

1. Blocks may be modified in memory once they have been brought into the cache.
      - Modified blocks are *dirty* and must (eventually) be written back.
2. Once a block is modified in memory, the write back to disk may not be immediate (*synchronous*).
   - *Delayed writes* absorb many small updates with one disk write.
      - How long should the system hold dirty data in memory?
   - *Asynchronous writes* allow overlapping of computation and disk update activity (*write-behind*).
      - Do the **write** call for block ***n+1*** while transfer of block ***n*** is in progress.

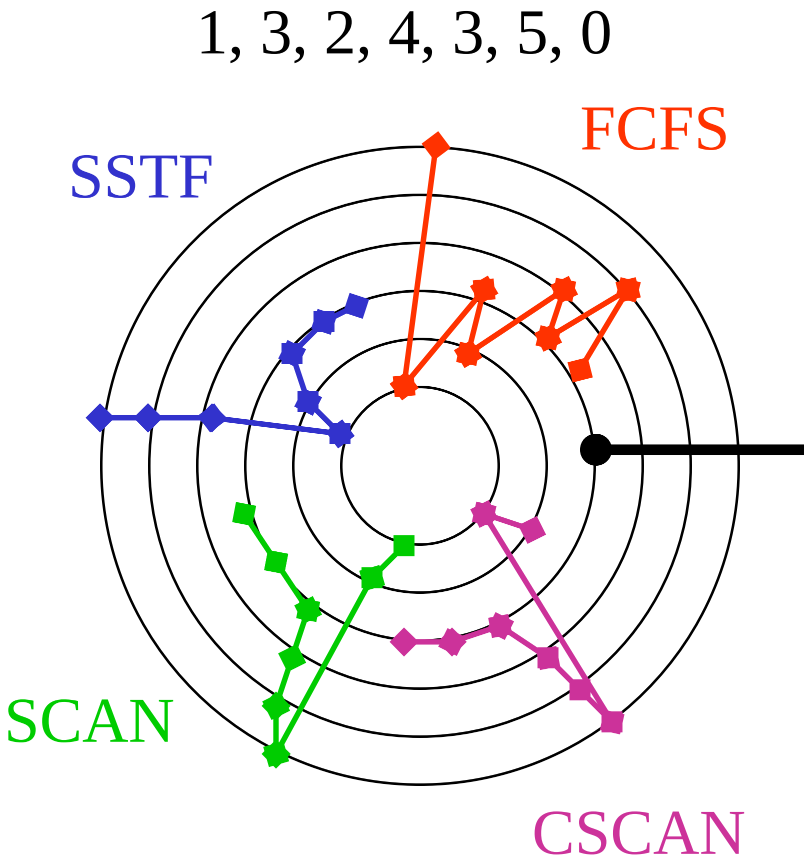
Disk Scheduling

- Assuming there are sufficient outstanding requests in request queue
- Focus is on seek time - minimizing physical movement of head.
- Simple model of seek performance

  Seek Time = startup time (e.g. 3.0 ms) +
    N (number of cylinders) *
    per-cylinder move (e.g. .04 ms/cyl)

Policies

- Generally use FCFS as baseline for comparison
- Shortest Seek First (SSTF) -closest
  - danger of starvation
- Elevator (SCAN) - sweep in one direction, turn around when no requests beyond
  - handle case of constant arrivals at same position
- C-SCAN - sweep in only one direction, return to 0
  - less variation in response



Layout on Disk

- Can address both seek and rotational latency
- Cluster related things together
  (e.g. an inode and its data, inodes in same directory (ls command), data blocks of multi-block file, files in same directory)
- Sub-block allocation to reduce fragmentation for small files
- Log-Structure File Systems

The Problem of Disk Layout

- The level of indirection in the file block maps allows flexibility in file layout.
    - “File system design is 99% block allocation.” [McVoy]
- Competing goals for block allocation:
    - *allocation cost*
    - *bandwidth* for high-volume transfers
    - *stamina*
    - *efficient directory operations*
- **Goal:** reduce disk arm movement and seek overhead.
    - metric of merit: *bandwidth utilization*

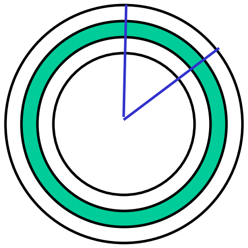
FFS and LFS

Two different approaches to block allocation:

- Cylinder groups in the Fast File System (FFS) [McKusick81]
  - clustering enhancements [McVoy91], and improved cluster allocation [McKusick: Smith/Seltzer96]
  - FFS can also be extended with metadata logging [e.g., Episode]
- Log-Structured File System (LFS)
  - proposed in [Douglis/Ousterhout90]
  - implemented/studied in [Rosenblum91]
  - BSD port, sort of maybe: [Seltzer93]
  - extended with self-tuning methods [Neefe/Anderson97]
- Other approach: *extent-based* file systems

FFS Cylinder Groups

- FFS defines *cylinder groups* as the unit of disk locality, and it factors locality into allocation choices.
  - typical: thousands of cylinders, dozens of groups
  - Strategy: place "related" data blocks in the same cylinder group whenever possible.
    - seek latency is proportional to seek distance
  - Smear large files across groups:
    - Place a run of contiguous blocks in each group.
  - Reserve inode blocks in each cylinder group.
    - This allows inodes to be allocated close to their directory entries and close to their data blocks (for small files).



FFS Allocation Policies

1. Allocate file inodes close to their containing directories.
    - For *mkdir*, select a cylinder group with a more-than-average number of free inodes.
    - For *creat*, place inode in the same group as the parent.
2. Concentrate related file data blocks in cylinder groups.
        - Most files are read and written sequentially.
    - Place initial blocks of a file in the same group as its inode.
        - How should we handle directory blocks?
    - Place adjacent logical blocks in the same cylinder group.
        - Logical block ***n+1*** goes in the same group as block ***n.***
        - Switch to a different group for each indirect block.

Allocating a Block

1. Try to allocate the rotationally optimal physical block after the previous logical block in the file.
   - Skip *rotdelay* physical blocks between each logical block.
   - (rotdelay is 0 on track-caching disk controllers.)
2. If not available, find another block a nearby rotational position in the same cylinder group
   - We’ll need a short seek, but we won’t wait for the rotation.
   - If not available, pick any other block in the cylinder group.
3. If the cylinder group is full, or we’re crossing to a new indirect block, go find a new cylinder group.
   - Pick a block at the beginning of a run of free blocks.

Clustering in FFS

- *Clustering* improves bandwidth utilization for large files read and written sequentially.
    - Allocate clumps/clusters/runs of blocks contiguously; read/write the entire clump in one operation with at most one seek.
  - – Typical cluster sizes: 32KB to 128KB.
- FFS can allocate contiguous runs of blocks "most of the time" on disks with sufficient free space.
  - – This (usually) occurs as a side effect of setting *rotdelay* = 0.
    - Newer versions may relocate to clusters of contiguous storage if the initial allocation did not succeed in placing them well.
  - – Must modify buffer cache to group buffers together and read/write in contiguous clusters.

Log-Structured File System (LFS)

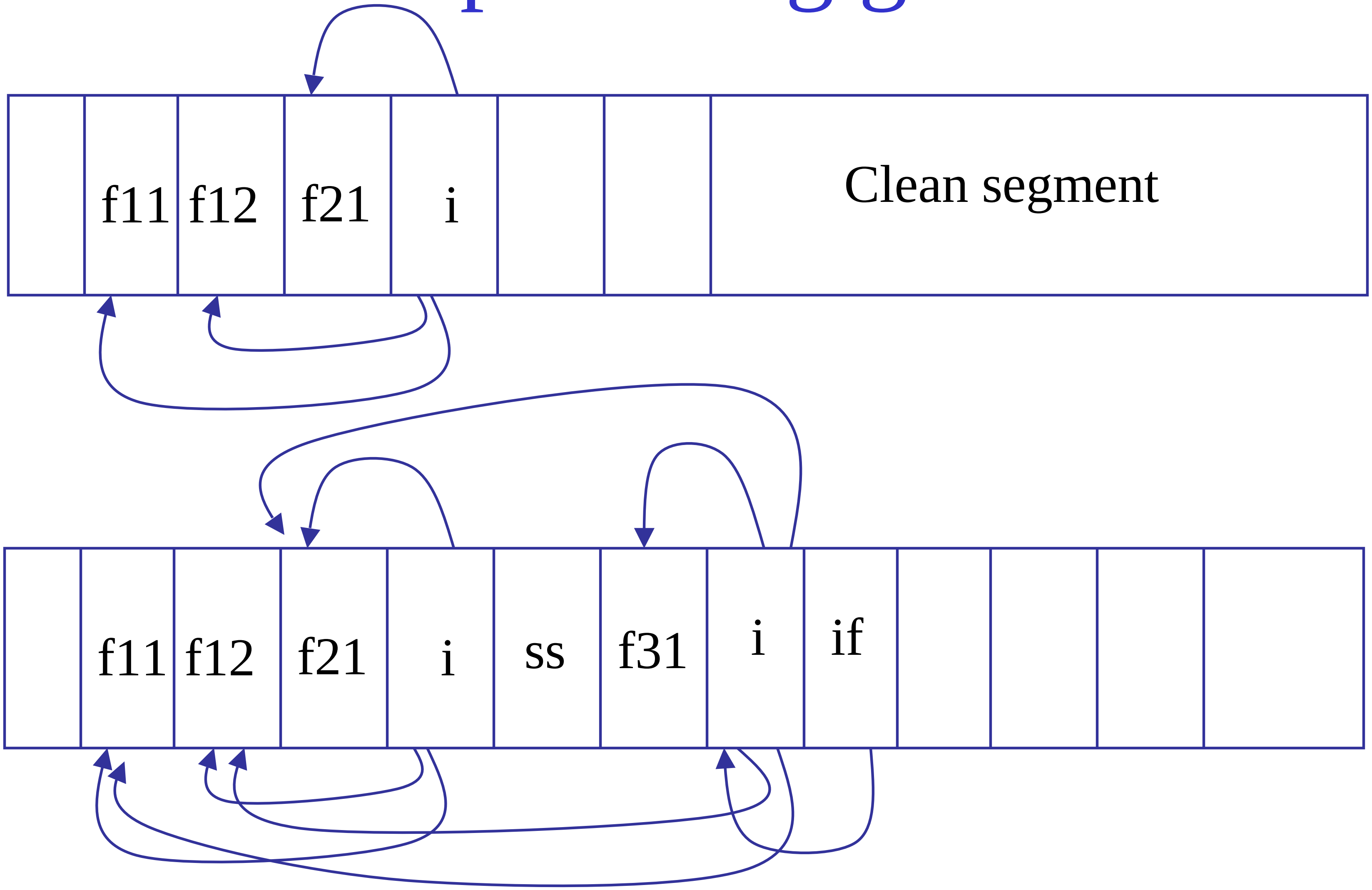
In LFS, *all* block and metadata allocation is log-based.

- LFS views the disk as "one big log" (logically).
- *All* writes are clustered and sequential/contiguous.
    - Intermingles metadata and blocks from different files.
- Data is laid out on disk in the order it is written.
- No-overwrite allocation policy: if an old block or inode is modified, write it to a new location at the *tail* of the log.
- LFS uses (mostly) the same metadata structures as FFS; only the allocation scheme is different.
    - Cylinder group structures and free block maps are eliminated.
    - Inodes are found by indirecting through a new map (the *ifile)*.

Writing the Log in LFS

1. LFS “saves up” dirty blocks and dirty inodes until it has a full *segment* (e.g., 1 MB).
   - Dirty inodes are grouped into block-sized clumps.
   - Dirty blocks are sorted by *(file, logical block number)*.
   - Each log segment includes summary info and a checksum.
2. LFS writes each log segment in a single burst, with at most one seek.
   - Find a free segment “slot” on the disk, and write it.
   - Store a back pointer to the previous segment.
     - Logically the log is sequential, but physically it consists of a chain of segments, each large enough to amortize seek overhead.

Example of log growth



Writing the Log: the Rest of the Story

1. LFS cannot always delay writes long enough to accumulate a full segment; sometimes it must push a *partial segment*.
    - fsync, update daemon, NFS server, etc.
    - Directory operations are synchronous in FFS, and some must be in LFS as well to preserve failure semantics and ordering.
2. LFS allocation and write policies affect the buffer cache, which is supposed to be filesystem-independent.
    - Pin (*lock*) dirty blocks until the segment is written; dirty blocks cannot be recycled off the free chain as before.
    - Endow *indirect blocks with permanent logical block numbers suitable for hashing in the buffer cache.

Cleaning in LFS

What does LFS do when the disk fills up?

1. As the log is written, blocks and inodes written earlier in time are superseded ("killed") by versions written later.
   - files are overwritten or modified; inodes are updated
   - when files are removed, blocks and inodes are deallocated
2. A cleaner daemon compacts remaining live data to free up large hunks of free space suitable for writing segments.
   - look for segments with little remaining live data
     - benefit/cost analysis to choose segments
   - write remaining live data to the log tail
   - can consume a significant share of bandwidth, and there are lots of cost/benefit heuristics involved.