EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

EC 440 – Introduction to Operating Systems
Project 4 – Discussion

(credit to: Manuel Egele)

Goals – Thread Local Storage

1. Provide protected memory regions for threads
2. Understand the basic concepts of memory management

Why Protected Memory for Threads?

- By default, all threads share the same address space
- Easy sharing of information
- But no protection from misbehaving threads
  - Thus, let's implement this protection
- Similar to Threads themselves, can be implemented in user space or kernel space
  - We'll implement it in user space

Thread Local Storage (TLS) Library

Threads

- Were the topic of projects 2 & 3 … i.e., should not be a matter of debate anymore

Storage

- An area of memory where data can be written to & read from

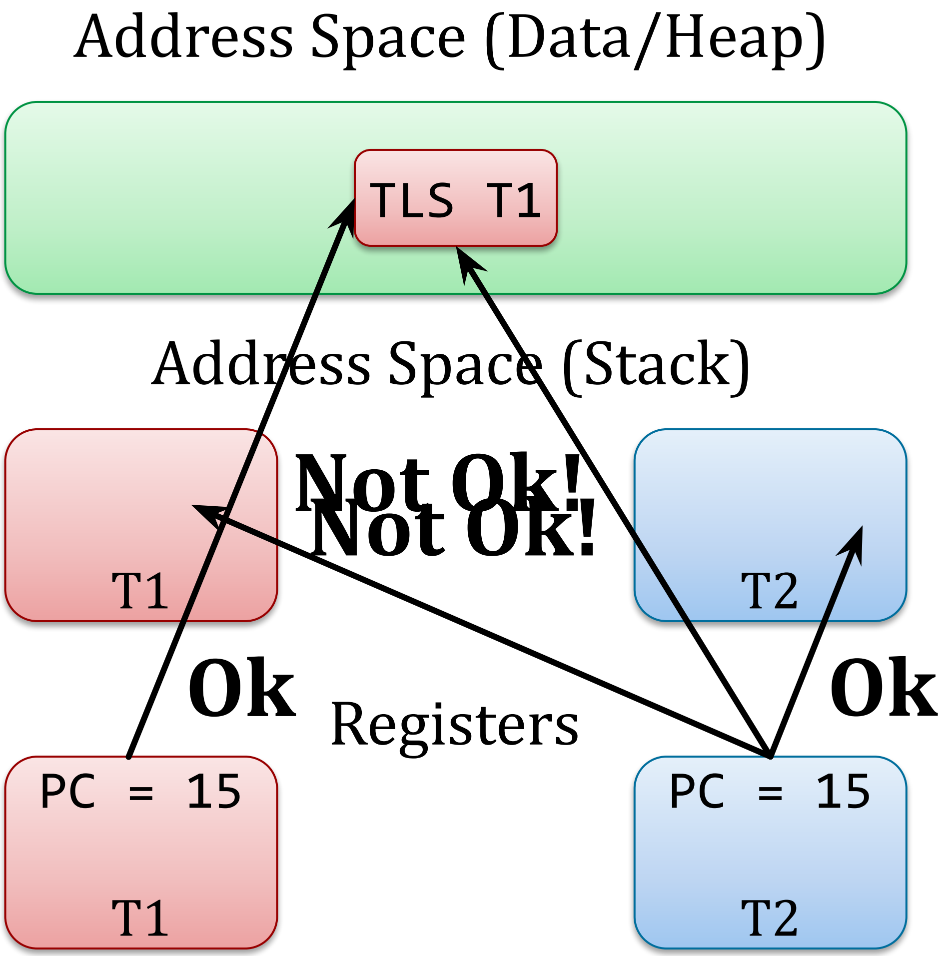
Local

- Each storage area is *local* (i.e., private) to one thread
- i.e., Thread T1 cannot read/write the TLS area of Thread T2

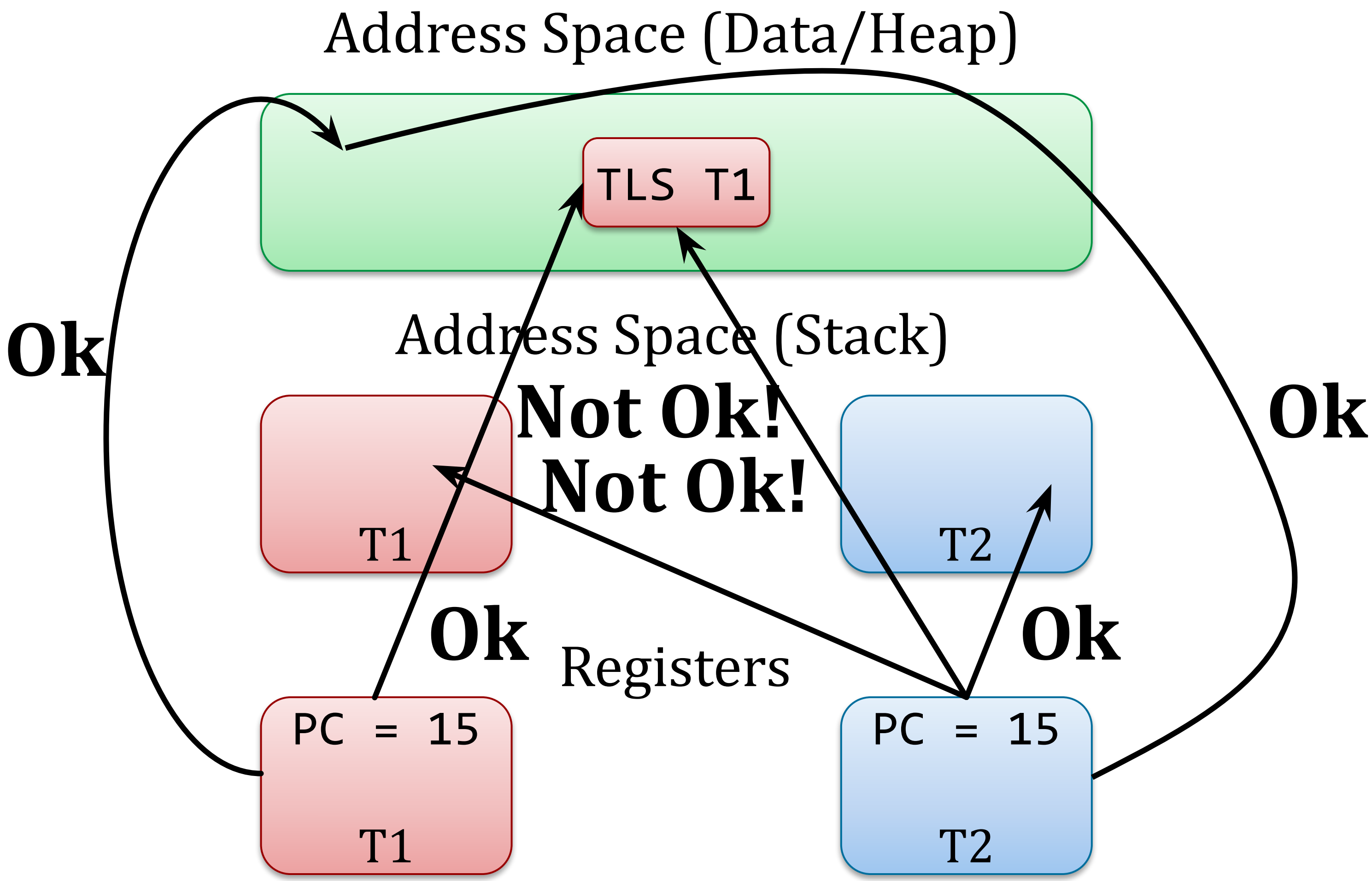
Copy-on-Write (COW) Semantics

- … more on that later …

Threads – TLS


Address Space (Data/Heap)
TLS T1
Address Space (Stack)
T1
Not Ok!
Not Ok!
T2
Ok
Registers
Ok
PC = 15
T1
PC = 15
T2

Threads – TLS



Address Space (Data/Heap)
TLS T1
Address Space (Stack)
Ok
Not Ok!
Not Ok!
Ok
T1
T2
Ok
Registers
Ok
PC = 15
T1
PC = 15
T2

TLS – Local Storage

***Local* Storage – Protection**

- Protect data tampering from other threads (i.e., no other thread can write to my TLS)
- Protect data "stealing" from other threads (i.e., no other thread can read from my TLS)

What if thread violates the protection?

- Terminate the offending thread!

How can we detect protection violations?

Protection, Violation, & Detection

- **Need to protect against read & write**
  – Remember the page-permission bits from the lecture, esp. R(ead) and W(rite)
- **How can violations occur?**
  – If R (W) bit is cleared reading (writing) from (to) the corresponding memory area will trigger a segmentation fault
  – But: segfault kills the process
- **How to detect violations?**
  – Segfault is just another signal, i.e., we can catch it with a signal handler

Enabling Protection

- All TLS sections have R/W bits cleared unless they're actively in use:
  - (i.e., only during calls to tls_read, tls_write)
- We need memory for the TLS sections. How do we allocate that memory?
  - use mmap()!
  - why not simply malloc()?
    - Granularity of <u>protection bits</u> is per virtual memory page (e.g., 4k)
    - malloc() allocates memory <u>w/o regard for page boundaries</u> and might put two different TLSs into the same page
    - mmap() allocates memory with page granularity and <u>aligned to page boundaries</u> (i.e., exactly what we need!)
    - All TLS areas are rounded up to the next page-size

Shades of Segfaults

A segfault happened, now what?

Two cases:

1. Thread Ti (i != 1) accesses T1's TLS
   a. Kill Ti (pthread_exit())
2. A regular segfault, Ti tries to access memory that's not a TLS but the access is inconsistent with page permission settings
   a. Raise segfault to the process (i.e., process will die)

How do we know which case happened?

Which Thread Caused the SEGV For What Address?

Caused a SEGV

- Our signal handler for SIGSEGV is invoked

Which thread?

- pthread_self()

What address?

- Signal handler (man sigaction, esp. fields in siginfo_t)
- `sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
-
```
struct sigaction { …
  void (*sa_sigaction)(int, siginfo_t *, void *)
… }
```

<u>Important</u>: This is different from sa_handler
Make sure you use the SA_SIGINFO value in *sa_flags*

siginfo_t struct with these fields

```
siginfo_t {
    int      si_signo;     /* Signal number */
    int      si_errno;     /* An errno value */
    int      si_code;      /* Signal code */
    int      si_trapno;    /* Trap number that caused hardware-generated signal (unused on most architectures) */
    pid_t    si_pid;       /* Sending process ID */
    uid_t    si_uid;       /* Real user ID of sending process */
    int      si_status;    /* Exit value or signal */
    clock_t  si_utime;     /* User time consumed */
    clock_t  si_stime;     /* System time consumed */
    sigval_t si_value;     /* Signal value */
    int      si_int;       /* POSIX.1b signal */
    void    *si_ptr;       /* POSIX.1b signal */
    int      si_overrun;   /* Timer overrun count; POSIX.1b timers */
    int      si_timerid;   /* Timer ID; POSIX.1b timers */
    void    *si_addr;      /* Memory location which caused fault */
    long     si_band;      /* Band event (was int in glibc 2.3.2 and earlier) */
    int      si_fd;        /* File descriptor */
    short    si_addr_lsb;  /* Least significant bit of address (since Linux 2.6.32) */
    void    *si_lower;     /* Lower bound when address violation occurred (since Linux 3.19) */
    void    *si_upper;     /* Upper bound when address violation occurred (since Linux 3.19) */
    int      si_pkey;      /* Protection key on PTE that caused fault (since Linux 4.6) */
    void    *si_call_addr; /* Address of system call instruction (since Linux 3.5) */
    int      si_syscall;   /* Number of attempted system call (since Linux 3.5) */
    unsigned int si_arch;  /* Architecture of attempted system call (since Linux 3.5) */
}
```

API

Create/Destroy TLS

```
int tls_create(unsigned int size);
int tls_destroy();
int tls_clone(pthread_t tid);
```

… later

Write to a TLS

```
int tls_write(
    unsigned int offset,
    unsigned int length,
    const char *buffer);
```

Read from a TLS

```
int tls_read(
    unsigned int offset,
    unsigned int length,
    char *buffer);
```

tls_create

`int tls_create(unsigned int size)`

- Creates a local storage area of a given *size* for the **current** thread
- Returns 0 on success
- Error: return -1
  - if current thread already has a LSA
  - size <= 0

tls_write

```
int tls_write(
    unsigned int offset,     Start at this location within TLS
    unsigned int length,     Copy this many bytes
    const char *buffer);     Copy into TLS from here
```

– Reads *length* bytes from the memory location pointed to by *buffer* and writes them into the local storage area of the currently executing thread, starting at position *offset*.
– Returns 0 on success
– Error: return -1
  • if current thread does not have an LSA
  • if offset+length > size of LSA

tls_read

```
int tls_read(
    unsigned int offset,    Start at this location within TLS
    unsigned int length,    Copy this many bytes
    char *buffer);          Copy from TLS to here
```

- Reads *length* bytes from the LSA of the currently executing thread, starting at position *offset* and writes into memory location pointed to by *buffer*.
- Returns 0 on success
- Error: return -1
  - if current thread does not have an LSA
  - if offset+length > size of LSA

tls_destroy

```
int tls_destroy();
```

– Frees local storage area for **current** thread.
– Returns 0 on success
– Error: return -1
  - if current thread does not have an LSA

tls_clone

`int tls_clone(pthread_t` *`tid`*`);`

- Clones the LSA of the target thread *`tid`* as CoW.
- Copy on Write (CoW):
  - Storage areas of both threads initially refer to the same memory pages
  - When a write happens ("<u>on write</u>") in a shared TLS, first <u>copy</u> the region that contains the written data
  - Other areas must remain shared!
- Returns 0 on success
- Error: return -1
  - if target thread does not have a LSA
  - if current thread already has a LSA

Assumptions

1. Whenever a thread calls tls_read or tls_write, you can temporarily unprotect this thread's local storage area
   – i.e., It is okay if there is a race condition where other threads can tamper with TLS
2. We do not require more fine granularity than per-page CoW

For example, if T2 clones T1's TLS, and T2 <u>writes one byte</u> to it's own (CoW) TLS
Instead of copying one byte, we <u>copy the entire page</u> that contains the target byte

One complexity

It is possible for more than two threads to share the same LSA

- Example
  T1.tls_create(8192)
  T2.clone(T1)
  T3.clone(T1)
- T1, T2, and T3 share the same local storage area
- CoW applies to all three threads.

Useful Library Functions

`mmap(2)`

- Helps to create local storage that cannot be accessed directly by thread
- Can create without read/write permission
- Memory obtained is aligned to start of page
- Allocates memory in multiples of page size

`mprotect(2)`

- Threads cannot access memory assigned by mmap
- Use mprotect to <u>unprotect</u> the memory before read/write
- <u>Re-protect</u> memory when the operation is complete

Need Some Data Structures

```
typedef struct thread_local_storage
{
  pthread_t tid;
  unsigned int size;      /* size in bytes                    */
  unsigned int page_num;  /* number of pages                  */
  struct page **pages;    /* array of pointers to pages       */
} TLS;


struct page {
  unsigned int address;   /* start address of page            */
  int ref_count;          /* counter for shared pages         */
};
```

Need Some Data Structures

```
typedef struct thread_local_storage
{
  pthread_t tid;
  unsigned int size;       /* size in bytes              */
  unsigned int page_num;   /* number of pages            */
  struct page **pages;     /* array of pointers to pages */
} TLS;
```

Why not just an array of pages?

```
struct page {
  unsigned int address;    /* start address of page      */
  int ref_count;           /* counter for shared pages   */
};
```

Need Some Data Structures

```
typedef struct thread_local_storage
{
  pthread_t tid;
  unsigned int size;       /* size in bytes                    */
  unsigned int page_num;   /* number of pages                  */
  struct page **pages;     /* array of pointers to pages       */
} TLS;


struct page {
  unsigned int address;    /* start address of page            */
  int ref_count;           /* counter for shared pages         */
};
```

Why would this be useful?

Mapping a Thread to a TLS

Need a global data structure to keep this mapping (e.g., fixed-sized array (limited number of concurrent threads inherited from HW2), linked list, hash map, etc.)

```
struct tid_tls_pair
{
  pthread_t tid;
  TLS *tls;
};

static struct tid_tls_pair tid_tls_pairs[MAX_THREAD_COUNT];
```

Initialize on First Create

```
void tls_init()
{
  struct sigaction sigact;
  page_size = getpagesize();

  /* Handle page faults (SIGSEGV, SIGBUS) */
  sigemptyset(&sigact.sa_mask);
  /* Give context to handler */
  sigact.sa_flags = SA_SIGINFO;
  sigact.sa_sigaction = tls_handle_page_fault;
  sigaction(SIGBUS, &sigact, NULL);
  sigaction(SIGSEGV, &sigact, NULL);
}
```

Difference between SIGSEGV and SIGBUS?

Handle SIGSEGV

Does what, exactly?

```
void tls_handle_page_fault(int sig, siginfo_t *si, void *context)
{
  p_fault = ((unsigned int) si->si_addr) & ~(page_size - 1);
```

1. check whether it is a "real" segfault or because a thread has touched forbidden memory

   a. Make a brute force scan through all allocated TLS regions

   b. Exit *just the current thread* if the faulting page is or isn't? found

2. Otherwise, a normal fault - install default handler and re-raise signal

```
    signal(SIGSEGV, SIG_DFL);
    signal(SIGBUS, SIG_DFL);
    raise(sig);
}
```

tls_create

```
int tls_create(unsigned int size) {
  if not initialized, call
tls_init()
```

1. Error handling:
   – check if current thread already has a LSA
   – check size > 0 or not
2. Allocate TLS using malloc/calloc
3. Initialize TLS thread ID, size, page count

tls_create

4. Allocate TLS->pages
   - array of pointers, convenient case for <u>calloc</u>
5. Allocate all pages for this TLS

```
for each page pointer in tls->pages:
    malloc a new page to the pointer
    p->address = mmap(0, page_size, PROT_NONE,
                      MAP_ANON | MAP_PRIVATE,
                      0, 0);
    also init p->ref_count
```

6. Add the (thread id→TLS) mapping to global data structure

} /* end of tls_create */

tls_destroy

```
int tls_destroy(){
```

1. Error handling:
   - check if current thread has LSA
2. Clean up all pages

   How do we know if it is shared?
   Hint: No new variables needed.

   Check each page whether it's shared
   a) If not shared, free the page
   b) If shared, can't free as other threads still rely on it. But…?
3. Clean up TLS
4. Remove the (tid→TLS) from global structure

```
}
```

Helper Function: tls_protect()

```
void tls_protect(struct page *p)
{
  if (mprotect((void *) p->address, page_size, ???)) {
    fprintf(stderr, "tls_protect: could not protect page\n");
    exit(1);
  }
}
```

```
See man mprotect:
int mprotect(void *addr, size_t len, int prot);
prot is a combination of the following access flags:
PROT_NONE or a bitwise-or of the other values:
       PROT_NONE  The memory cannot be accessed at all.
       PROT_READ  The memory can be read.
       PROT_WRITE The memory can be modified.
       ...
```

Helper Function: tls_unprotect()

```
void tls_unprotect(struct page *p)
{
  if (mprotect((void *) p->address, page_size, ???)) {
    fprintf(stderr, "tls_unprotect: could not unprotect page\n");
    exit(1);
  }
}
```

How can you check whether protect/unprotect are working?

tls_read()

```
int tls_read(unsigned int offset,
    unsigned int length, char *buffer)
```

{

1. Error handling:
   - check if current thread has a LSA
   - check if offset+length > size
2. Unprotect all pages belonging to thread's TLS
3. Perform <u>read operation</u> (next slide)
4. Reprotect all pages belonging to thread's TLS

}

Read Operation

- Need to copy:
  - from tls->pages[...]->address
    - But, start at *offset*
  - to *buffer*
  - Only up to *length* bytes
- *Buffer* is contiguous. TLS contents may not be.

```
// Example for one byte at a time:
for (cnt = 0, idx = offset;
     idx < (offset + length);
     ++cnt, ++idx) {
    // Calculate page number and offset within that page,
    // from idx and page size (hint: consider quotients and
    // remainders in division). Then:
    buffer[cnt] = byte in {that page, at that offset}
}
```

tls_write()

```
int tls_write(unsigned int offset,
unsigned int length, char *buffer) {
```

1. Error handling:
   - check if current thread has a LSA
   - check if offset+length > size
2. Unprotect all pages belonging to thread’s TLS
3. Perform <u>write operation</u> (next slide)
4. Reprotect all pages belonging to thread’s TLS

```
}
```

Write Operation

```
/* per-byte example */
for (cnt = 0, idx = offset;
     idx < (offset + length);
     ++cnt, ++idx) {
   // Calculate page number and offset within
   // that page, from idx and page size. Then:
   if (page is shared) {
     // create a private copy (COW)
     (next slide)
   }

   byte in {page, at offset} = buffer[cnt]
  }
```

CoW Implementation

```
// create a private copy (COW)
copy = (struct page *) calloc(1, sizeof(struct page));
copy->address = mmap(0, page_size, PROT_WRITE,
                     MAP_ANON | MAP_PRIVATE, 0, 0);
update the TLS page pointer = copy;

// We are messing around with references by doing a copy.
// So which pages need their ref counts modified? To what values?

tls_protect(/* old page or new page? */);

// Recall: this slide example is inside a loop over pages
// from the previous slide. AFTER this bit of code, there
// is a copy from buffer to a page. Make sure you copy to
// the right one of these pages.
```

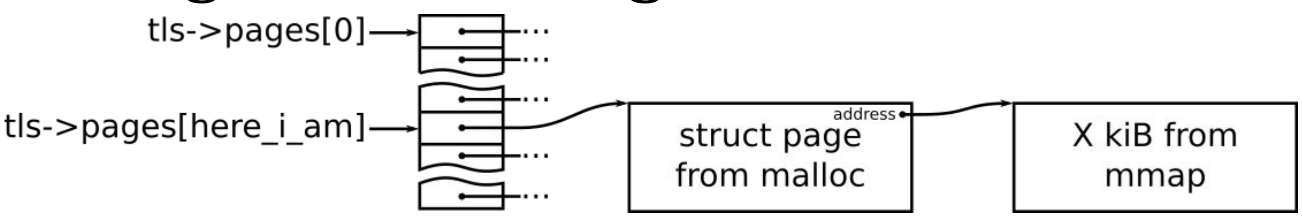
tls_clone

```
int tls_clone(pthread_t tid) {
```

1. Error handling
   - check if current thread already has a LSA
   - check whether target thread has a LSA
2. Do Cloning, allocate TLS
3. Copy page pointers (not contents!), adjust reference counts
   - Note, per proj. description and CoW semantics do not create a copy of the data itself!
   - Make cloned' page entries point to original data-pages
   - CoW is handled in tls_write
4. Add this thread/TLS mapping to global structure

High-Level Hints

1. Do your error handling first, then simple read/write. Cloning/CoW/signal handling last.
2. There is a lot of pointer management, including pointers-to-pointers. Draw diagrams when it gets confusing.



3. Write focused unit tests. There are a lot of moving parts.
4. Memory management bugs are inevitable. Use valgrind, sanitizers, static analyzers, ...

File systems

File Systems

Essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. Information must survive termination of process using it.
3. Multiple processes must be able to access information concurrently.

File Systems

Think of a disk as a linear sequence of fixed-size blocks and supporting two operations:

1. Read block *k*.
2. Write block *k*

File Systems

Questions that quickly arise:

1. How do you find information?
2. How do you keep one user from reading another user's data?
3. How do you know which blocks are free?

File Naming

Extension	Meaning
.bak	Backup file
.c	C source program
.gif	Compuserve Graphical Interchange Format image
.hlp	Help file
.html	World Wide Web HyperText Markup Language document
.jpg	Still picture encoded with the JPEG standard
.mp3	Music encoded in MPEG layer 3 audio format
.mpg	Movie encoded with the MPEG standard
.o	Object file (compiler output, not yet linked)
.pdf	Portable Document Format file
.ps	PostScript file
.tex	Input for the TEX formatting program
.txt	General text file
.zip	Compressed archive

Figure 4-1. Some typical file extensions.

File Structure

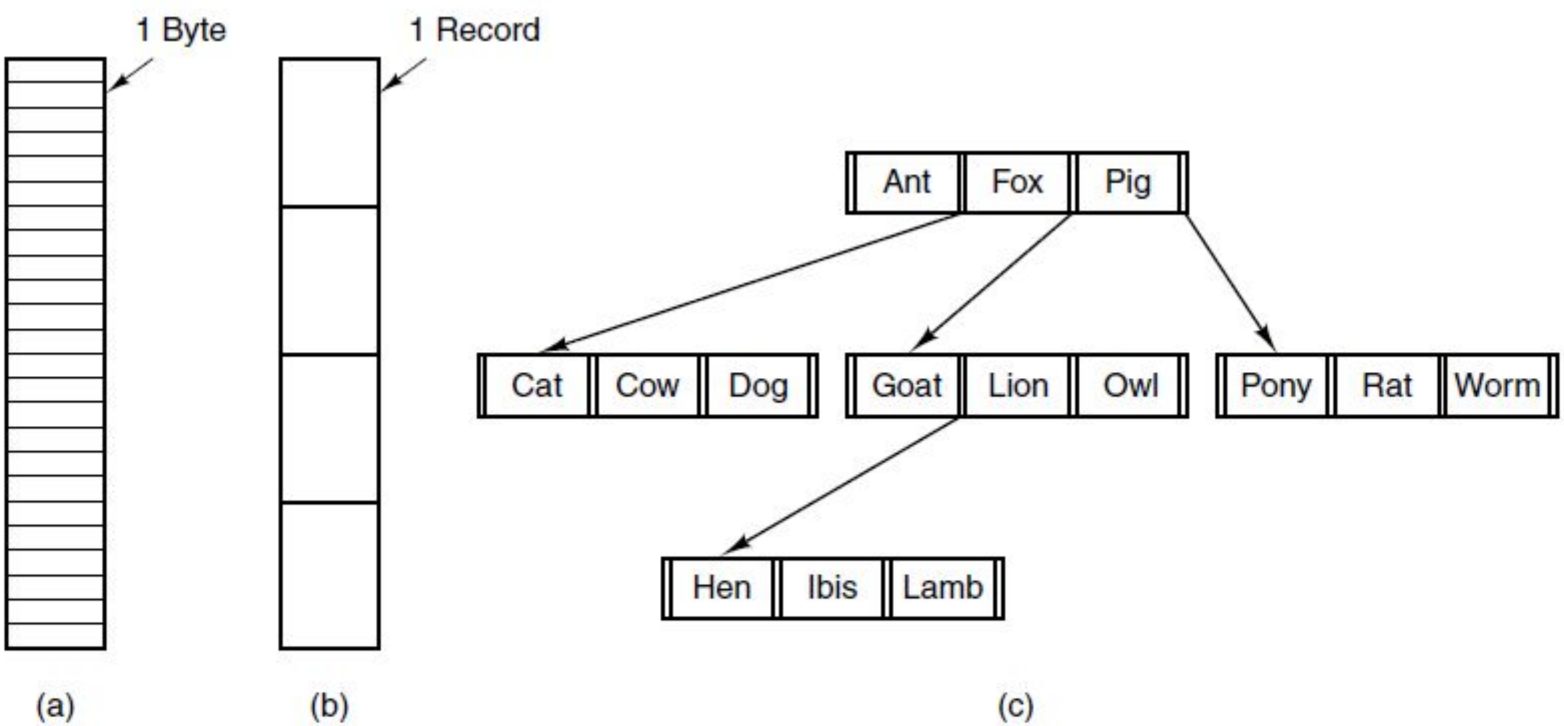


Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Binary File Types

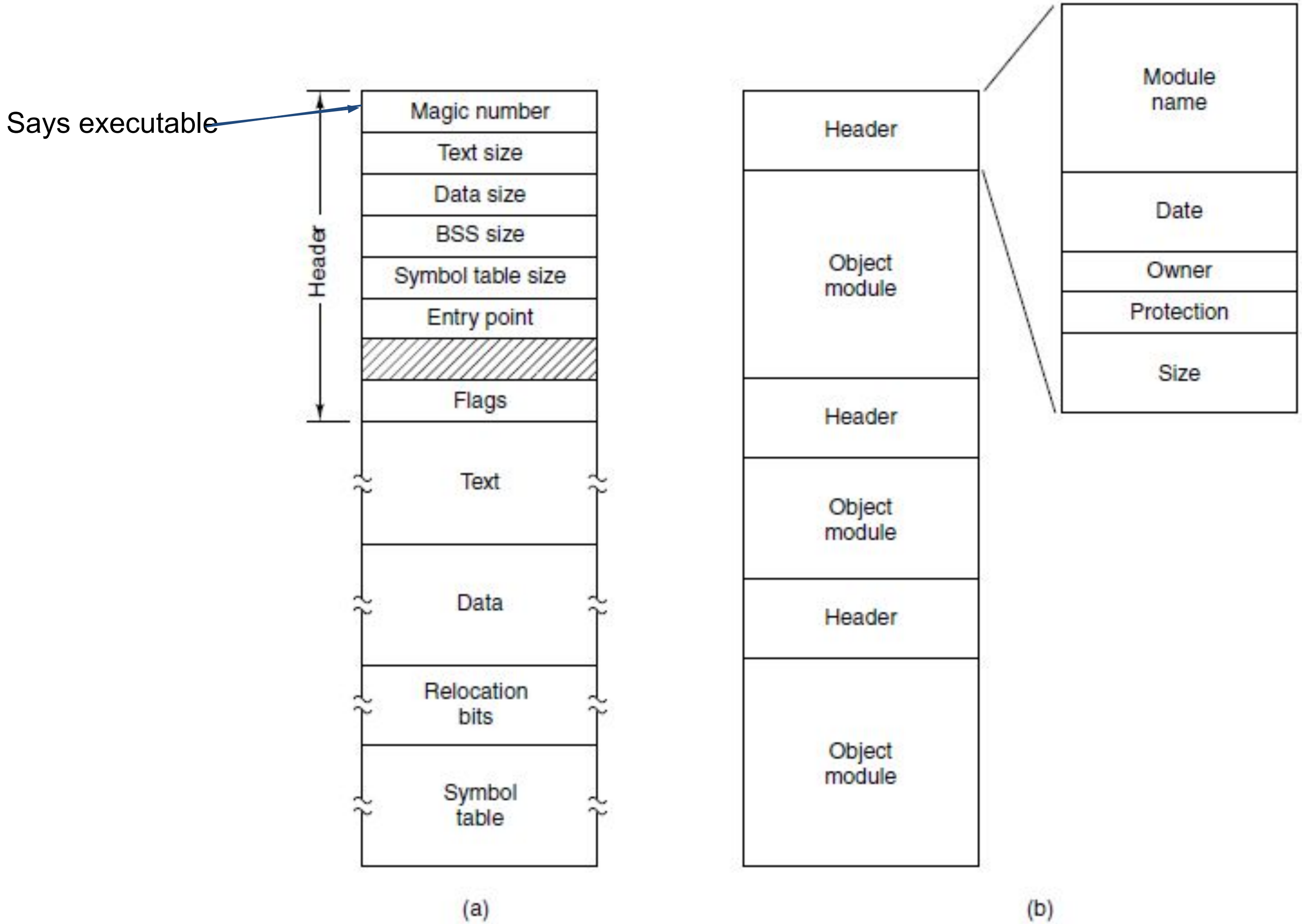


Figure 4-3. (a) An executable file. (b) An archive

File Attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 4-4. Some possible file attributes.

Linux “stat” structure

```
struct stat {
        dev_t     st_dev;         /* ID of device containing file */
        ino_t     st_ino;         /* Inode number */
        mode_t    st_mode;        /* File type and mode */ - RWX for usr/group/all
        nlink_t   st_nlink;       /* Number of hard links */
        uid_t     st_uid;         /* User ID of owner */
        gid_t     st_gid;         /* Group ID of owner */
        dev_t     st_rdev;        /* Device ID (if special file) */
        off_t     st_size;        /* Total size, in bytes */
        blksize_t st_blksize;     /* Block size for filesystem I/O */
        blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
        struct timespec st_atim;  /* Time of last access */
        struct timespec st_mtim;  /* Time of last modification */     e.g. used make
        struct timespec st_ctim;  /* Time of last status change */
    };
```

Types of files in unix

- Regular files - ascii or binary (FS doesn’t care)
- Directories
- Character files - serial I/O devices
- Block special - disks

File Operations

1. Create
2. Delete
3. Open
4. Close
5. Read
6. Write
7. Append
8. Seek
9. Get attributes
10. Set attributes
11. Rename

File System Code Demo by Orran

Example Program Using File System Calls (1)

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                     /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);          /* ANSI prototype */

#define BUF_SIZE 4096                      /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                   /* protection bits for output file */

int main(int argc, char *argv[])
{
     int in_fd, out_fd, rd_count, wt_count;
     char buffer[BUF_SIZE];

     if (argc != 3) exit(1);               /* syntax error if argc is not 3 */

     /* Open the input file and create the output file */
```

Figure 4-5. A simple program to copy a file.

Example Program Using File System Calls (2)

```
if (argc != 3) exit(1);                          /* syntax error if argc is not 3 */

/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY);                 /* open the source file */
if (in_fd < 0) exit(2);                          /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE);            /* create the destination file */
if (out_fd < 0) exit(3);                         /* if it cannot be created, exit */

/* Copy loop */
while (TRUE) {
     rd_count = read(in_fd, buffer, BUF_SIZE);   /* read a block of data */
if (rd_count <= 0) break;                        /* if end of file or error, exit loop */
     wt_count = write(out_fd, buffer, rd_count); /* write data */
```

Figure 4-5. A simple program to copy a file.

Example Program Using File System Calls (3)

```
    /* Copy loop */
    while (TRUE) {
         rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;                /* if end of file or error, exit loop */
         wt_count = write(out_fd, buffer, rd_count); /* write data */
         if (wt_count <= 0) exit(4);         /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                       /* no error on last read */
         exit(0);
    else
         exit(5);                            /* error on last read */
}
```

Figure 4-5. A simple program to copy a file.

Single-Level Directory Systems

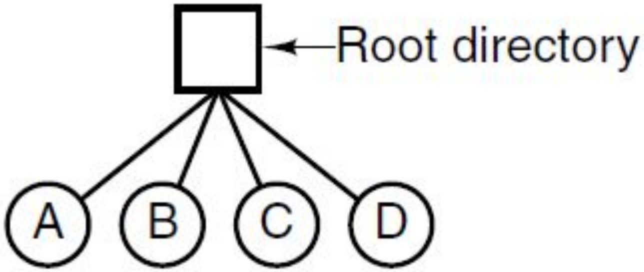


Figure 4-6. A single-level directory system containing four files.

Hierarchical Directory Systems

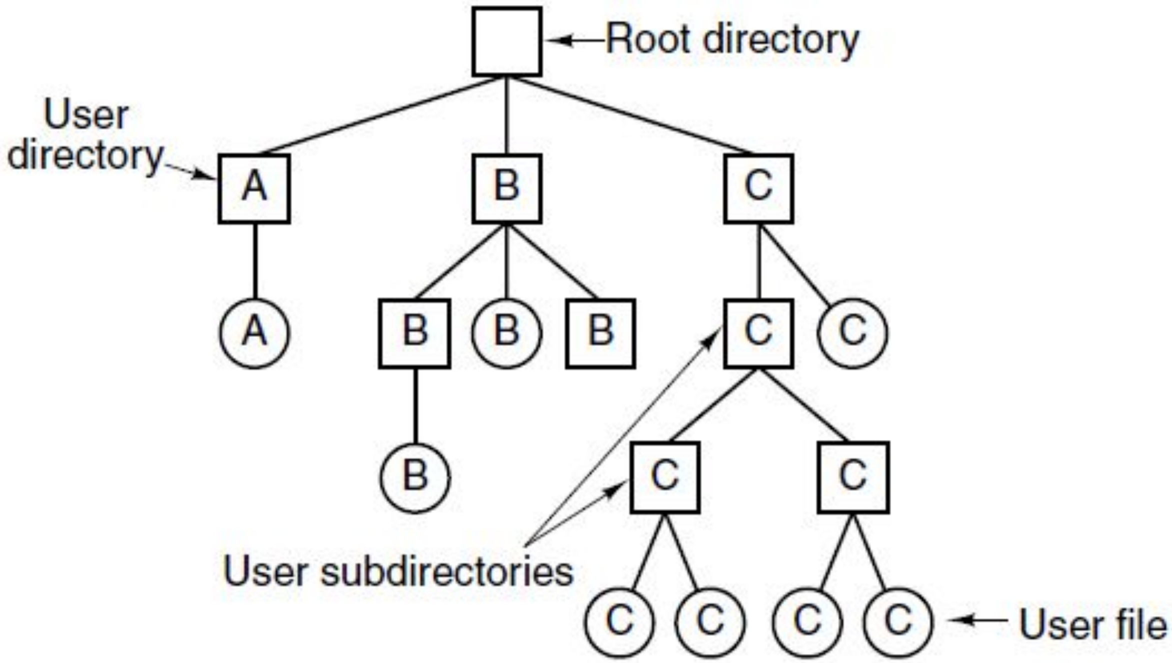
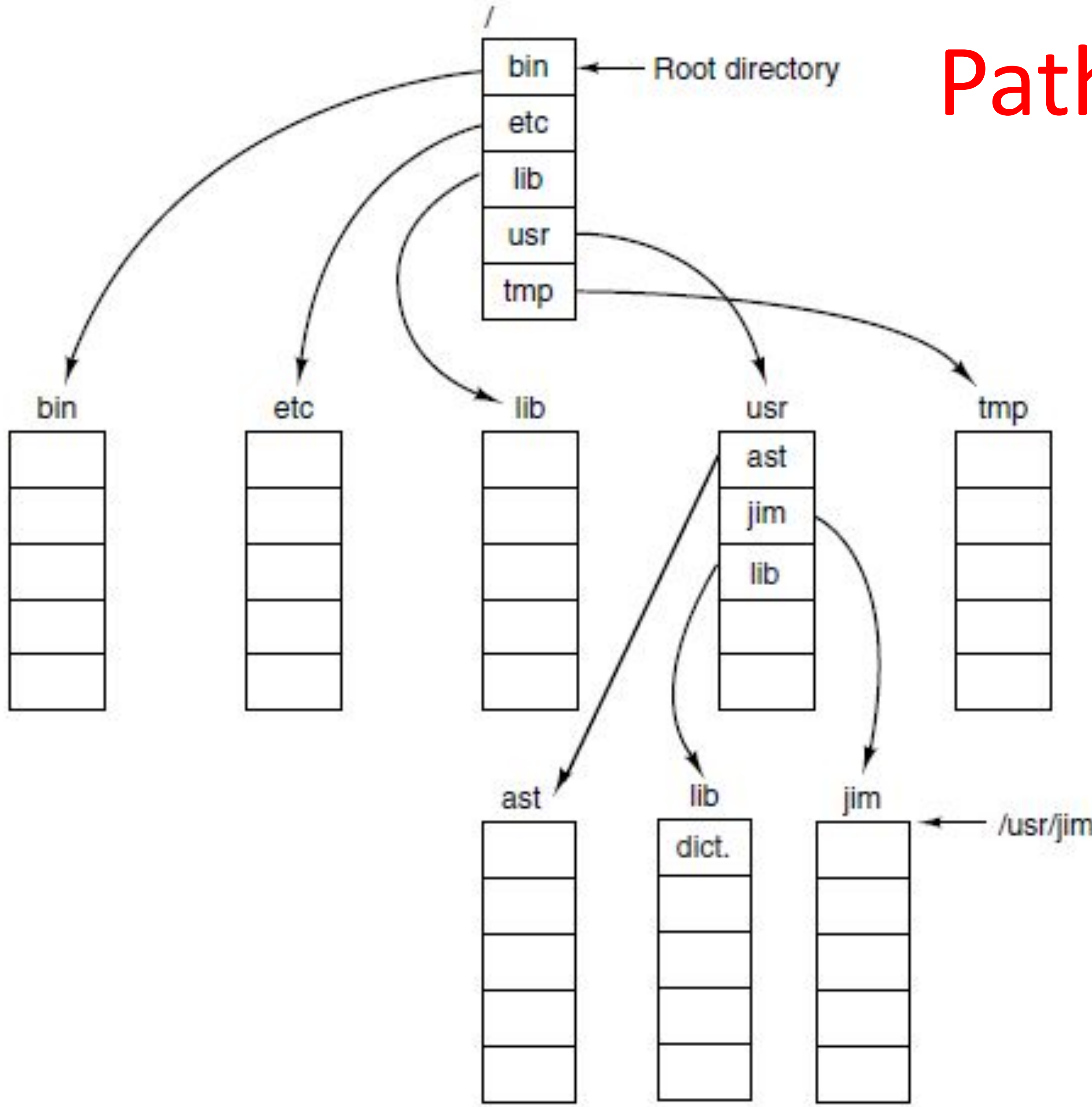


Figure 4-7. A hierarchical directory system.

Path Names


/
bin
etc
lib
usr
tmp
Root directory
bin
etc
lib
usr
tmp
ast
jim
lib
ast
lib
jim
dict.
/usr/jim

Referring to files

- Absolute path, e.g., /usr/jim/foo/bar
- relative path foo/bar (cwd is /usr/jim)
- ".." refers to parent directory
  - e.g., ../foo/bar (cwd is /usr/jim/src)

Directory Operations

1. Create
2. Delete
3. Opendir
4. Closedir
5. Readdir
6. Rename
7. Link
8. Unlink

File system implementation

Disk Layout

- MBR - Master boot record
- Partition table
  - Start/end of each partition
  - Marker for active partition used to boot OS
- Contents of partition is file system specific

Disk Layout

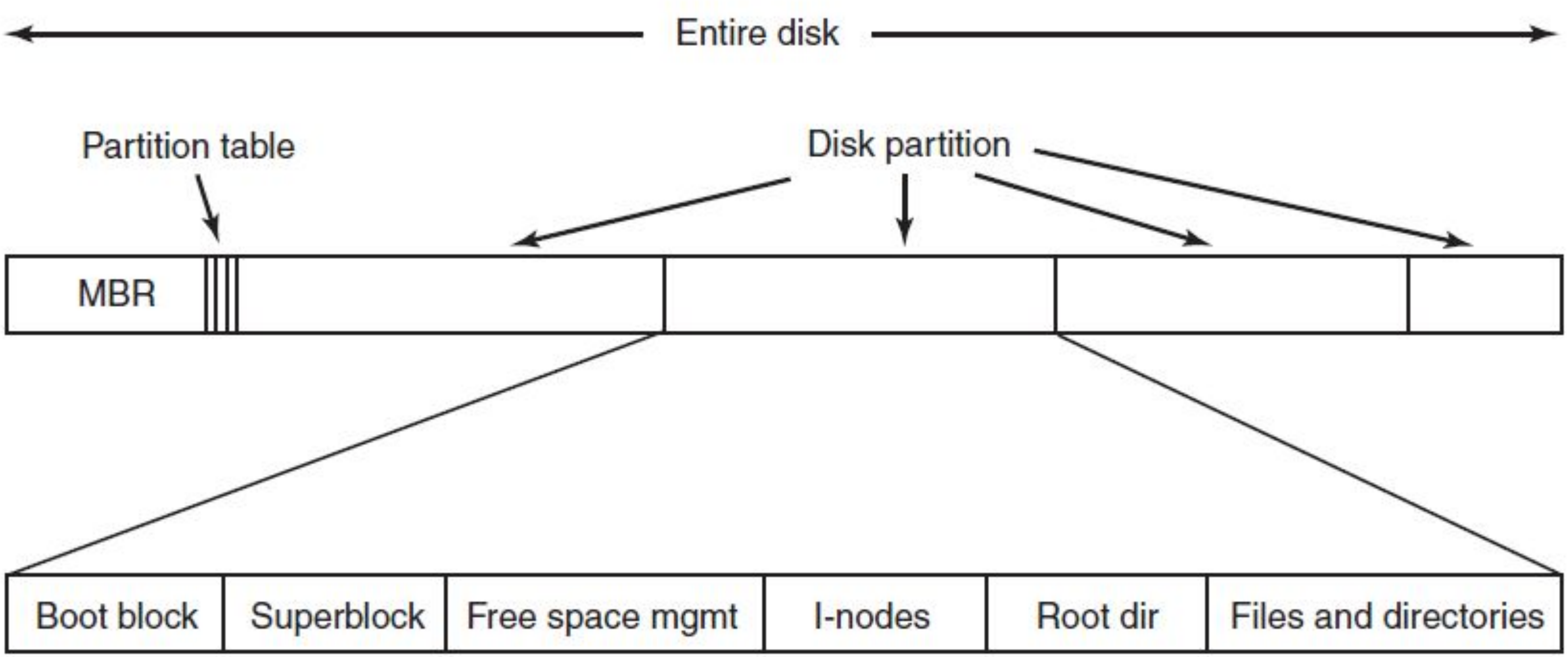


Figure 4-9. A possible file system layout.

Implementing Files Contiguous Layout

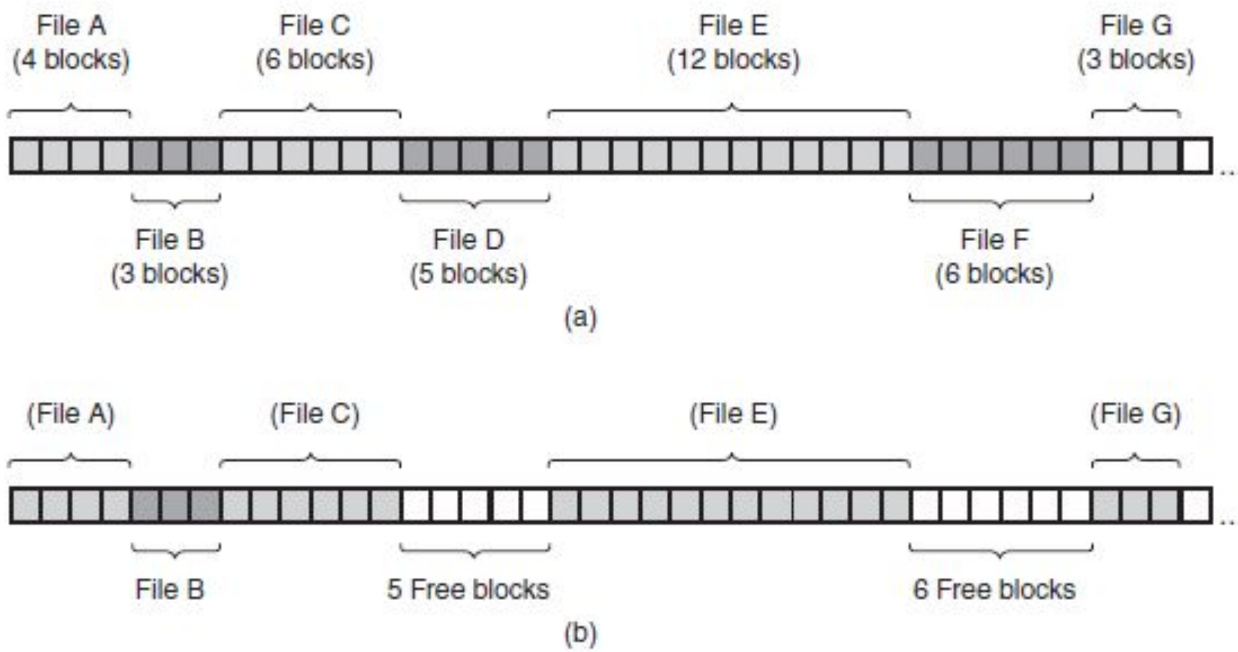


Figure 4-10. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

Tradeoffs

- Contiguous layout advantages
  - very dense meta-data
  - very fast read - one seek operation
- Disadvantages
  - rapidly fragments disks - external fragmentation
  - can’t extend files
- Used CD-ROMs
- Varient extent-based file systems

Implementing Files
Linked List Allocation

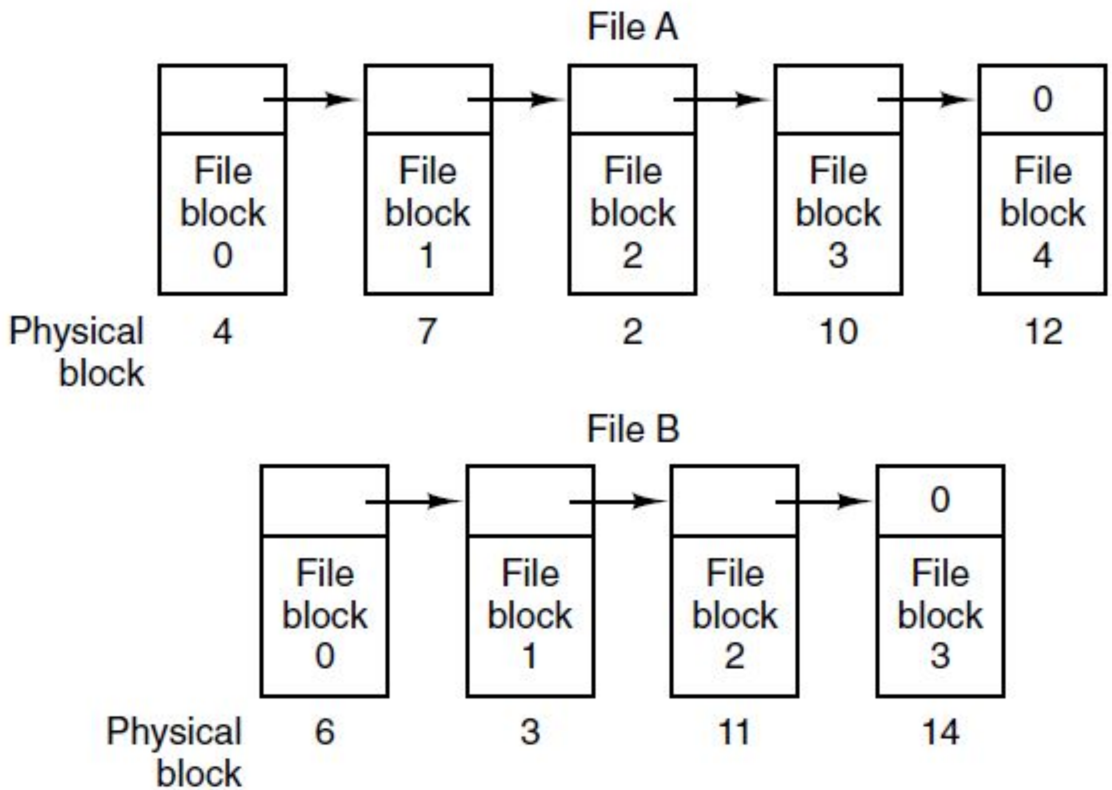


Figure 4-11. Storing a file as a linked list of disk blocks.

Tradeoffs

- No external fragmentation
- Disadvantages
  - block size no longer power of 2
  - need to read sequentially

Implementing Files
Linked List – Table in Memory

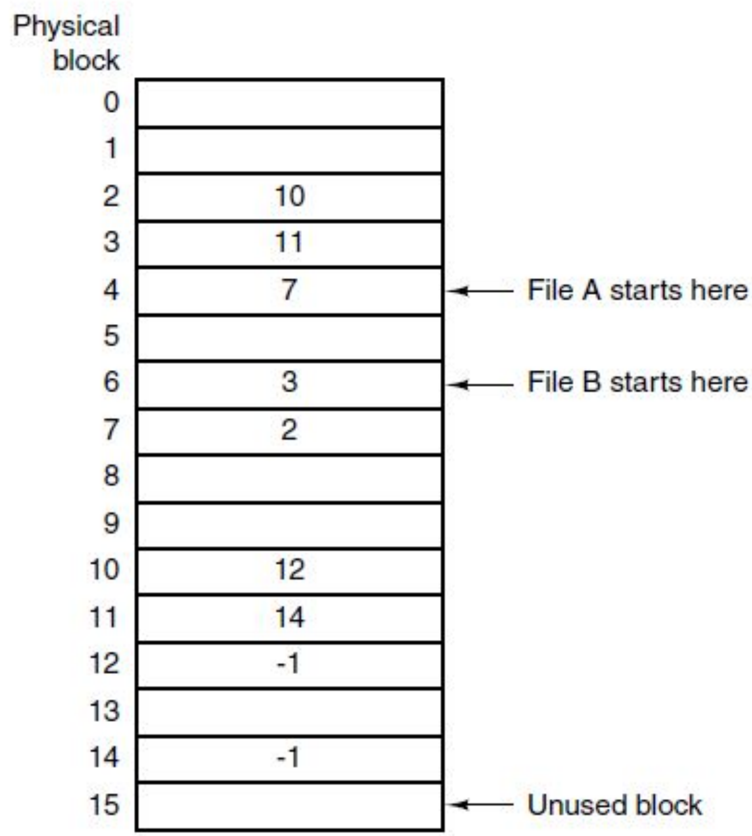


Figure 4-12. Linked list allocation using a file allocation table in main memory.

Implementing Files I-nodes

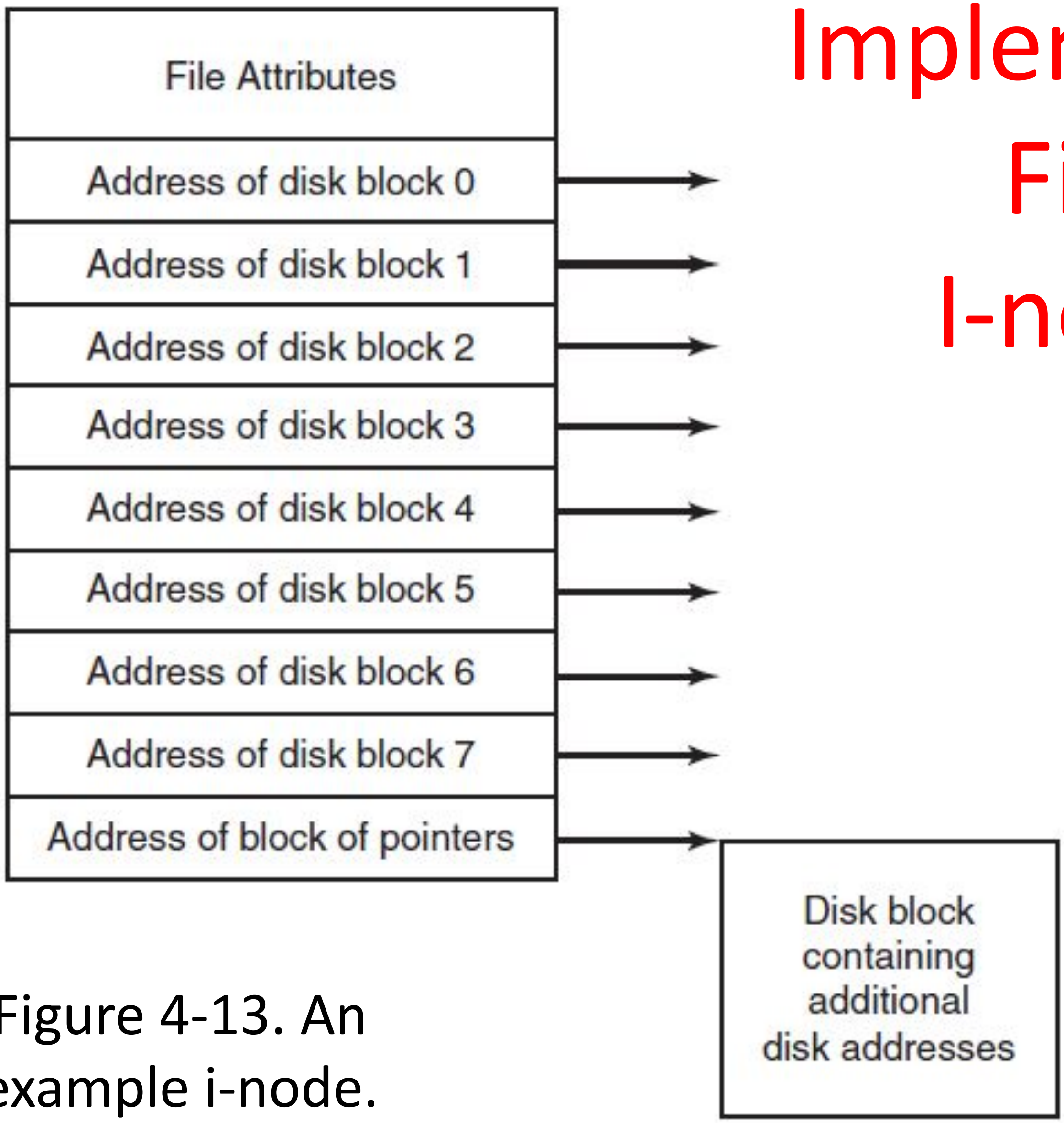


Figure 4-13. An example i-node.