Memory Mapped Files & Transactions

David E. Culler

CS162 – Operating Systems and Systems Programming

Lecture 26

October 27, 2014

Reading: A&D 14.1
HW 5 out
Proj 2 final 11/07

File System Summary (1/2)

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for size, access and usage patterns
  - Maximize sequential access, allow efficient random access
  - Projects the OS protection and security regime (UGO vs ACL)
- File defined by header, called “inode”
- Multilevel Indexed Scheme
  - inode contains file info, direct pointers to blocks, indirect blocks, doubly indirect, etc..
  - NTFS uses variable extents, rather than fixed blocks, and tiny files data is in the header
- 4.2 BSD Multilevel index files
  - Inode contains pointers to actual blocks, indirect blocks, double indirect blocks, etc.
  - Optimizations for sequential access: start new files in open ranges of free blocks, rotational Optimization

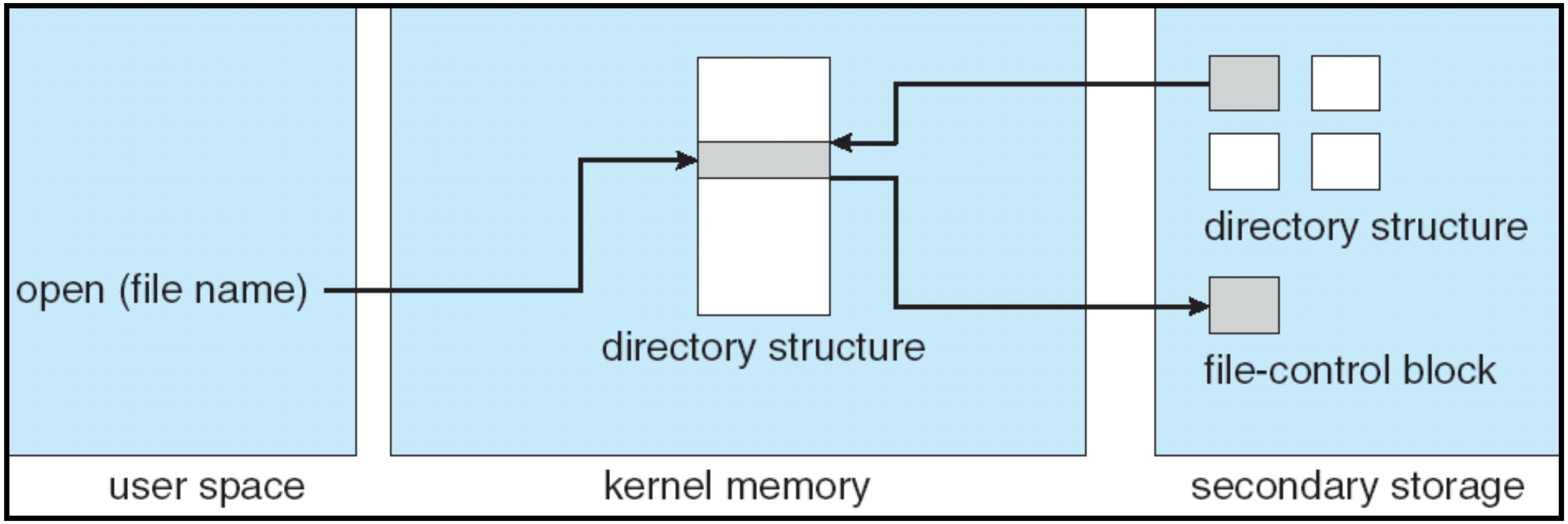
File System Summary (2/2)

- Naming: act of translating from user-visible names to actual system resources
  - Directories used for naming for local file systems
  - Linked or tree structure stored in files
- File layout driven by freespace management
  - Integrate freespace, inode table, file blocks and directories into block group
- Copy-on-write creates new (better positioned) version of file upon burst of writes
- Deep interactions between memory management, file system, and sharing

Bring Distant Concepts together

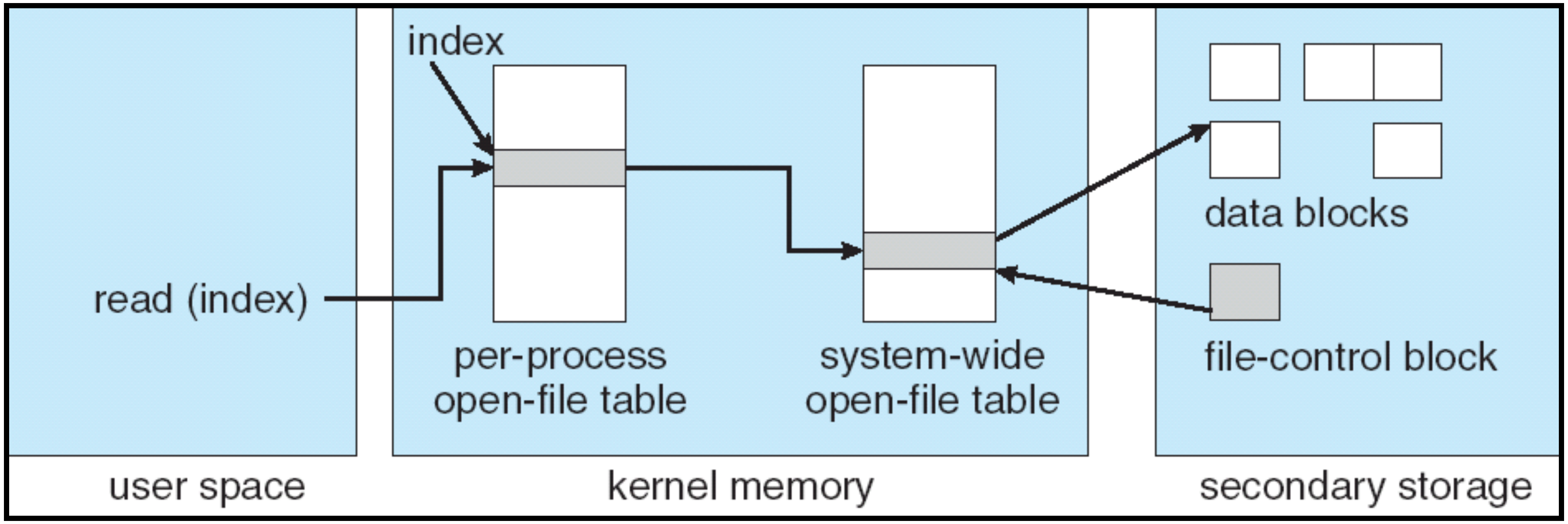
- Can we use files for interprocess communication?
  - Yes, but want flock, in addition to fflush!
- Can we use the file namespace, operations, etc. at the performance of memory?
  - Without the durability
- Can we use the virtual memory machinery to access files with load/store instructions?
  - Map files into the virtual address space
  - Controlled sharing between processes by using file for rendezvous

In-Memory File System Structures



- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called “file handle”) in open-file table

In-Memory File System Structures

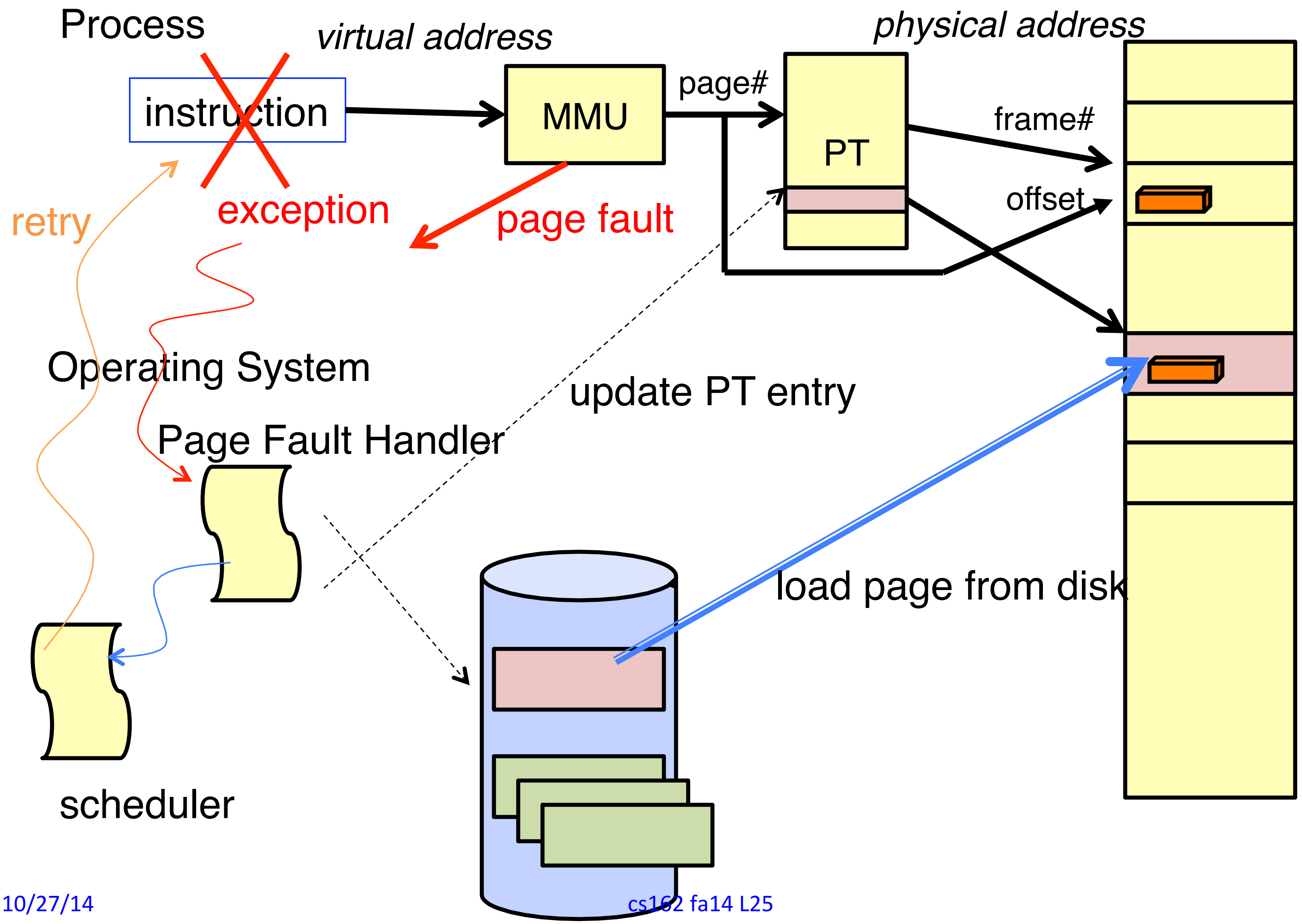


- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes

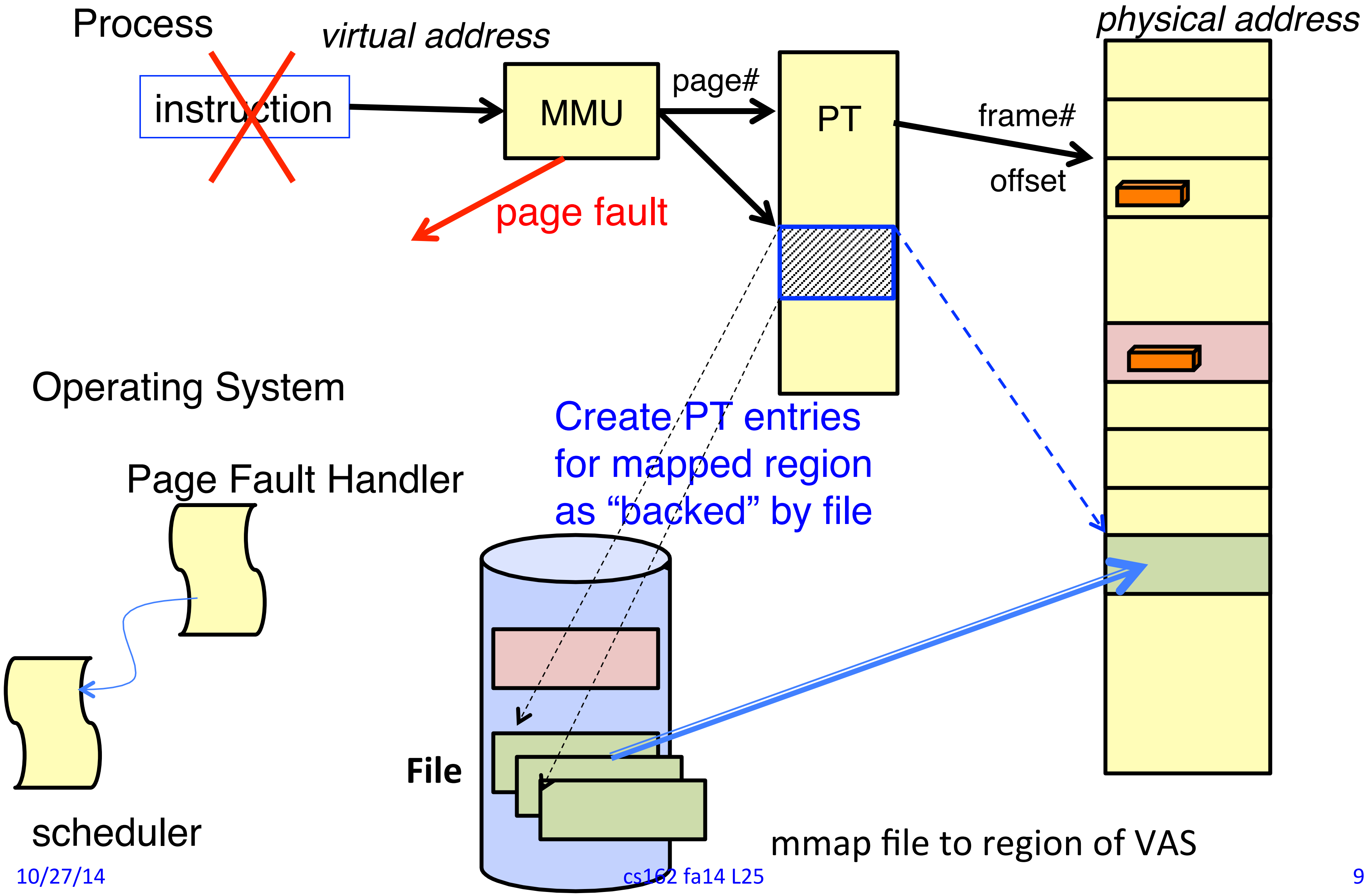
Memory Mapped Files

- Traditional I/O involves explicit transfers between buffers in process address space to regions of a file
  - This involves multiple copies into caches in memory, plus system calls
- What if we could "map" the file directly into an empty region of our address space
  - Implicitly "page it in" when we read it
  - Write it and "eventually" page it out
- Executable file is treated this way when we exec the process !!

Recall: Who does what when ?

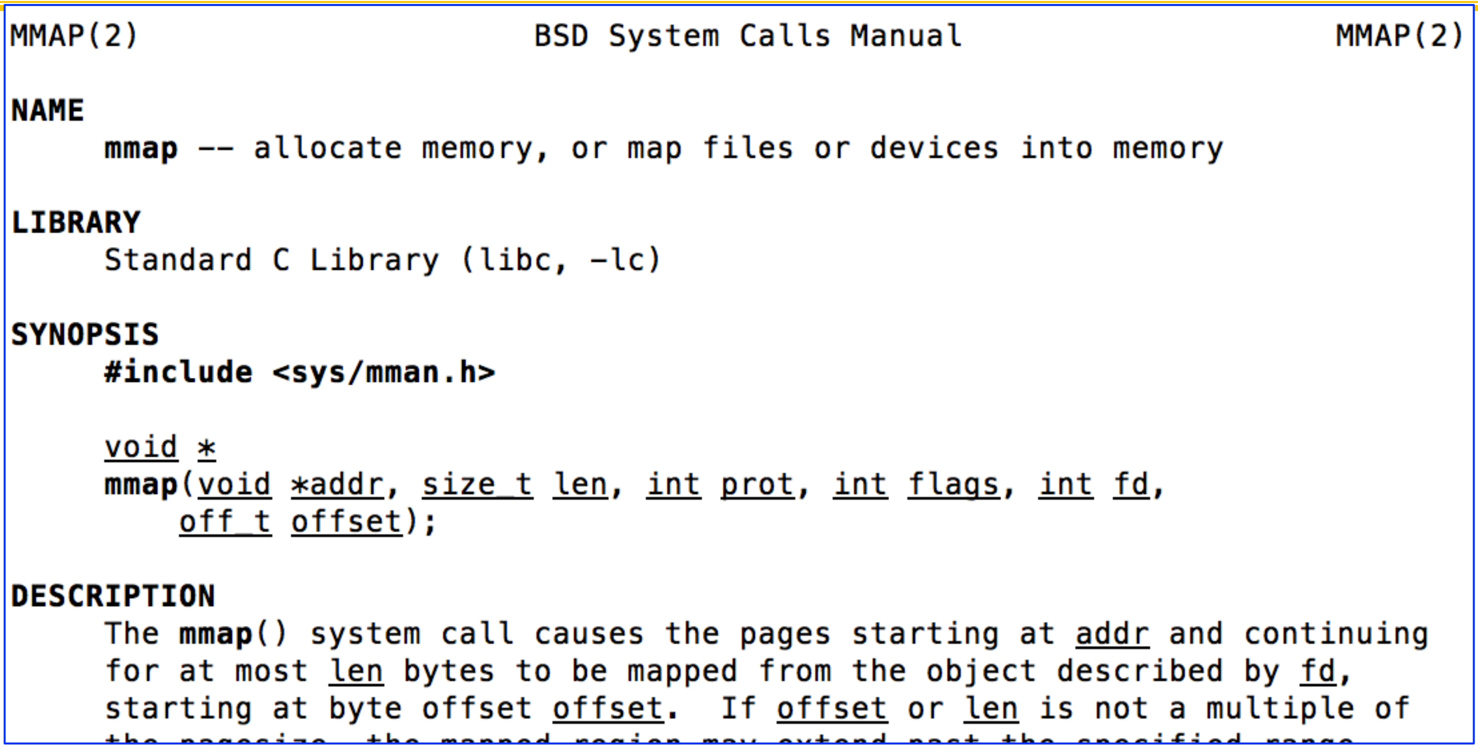


Using Paging to mmap files



mmap file to region of VAS

mmap system call



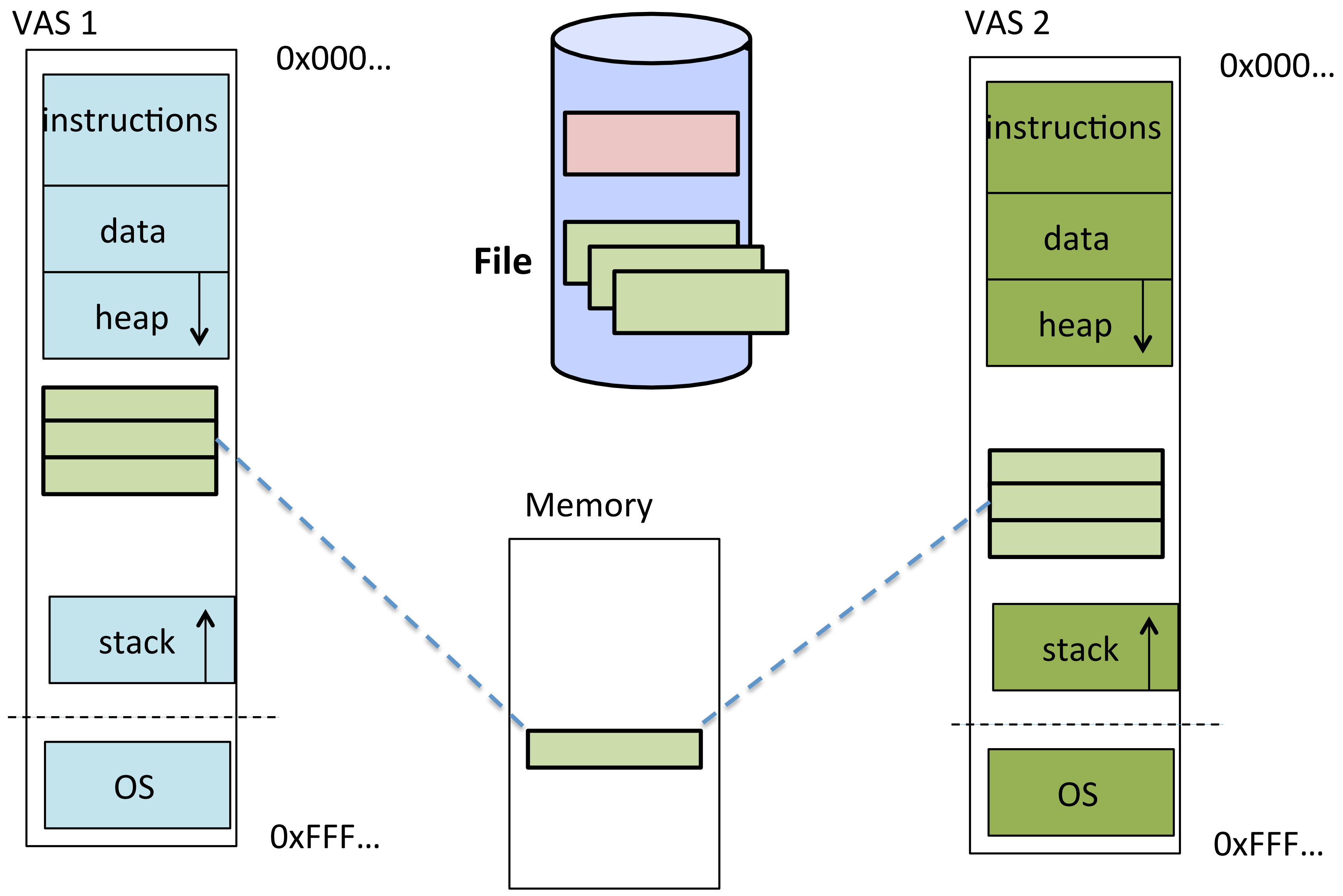
```
MMAP(2)                     BSD System Calls Manual                    MMAP(2)

NAME
     mmap -- allocate memory, or map files or devices into memory

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <sys/mman.h>

     void *
     mmap(void *addr, size_t len, int prot, int flags, int fd,
         off_t offset);

DESCRIPTION
     The mmap() system call causes the pages starting at addr and continuing
     for at most len bytes to be mapped from the object described by fd,
     starting at byte offset offset.  If offset or len is not a multiple of
```

- May map a specific region or let the system find one for you
  - Tricky to know where the holes are
- Used both for manipulating files and for sharing between processes

An example

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

int something = 162;

int main (int argc, char *argv[]) {
  int infile;
  char *mfile;
  void *sadddr = 0;
  something++;
  printf("Data  at: %16lx\n", (long unsigned int) &something);
  printf("Heap at : %16lx\n", (long unsigned int) malloc(1));
  printf("Stack at: %16lx\n", (long unsigned int) &mfile);

  mfile = mmap(0, 10000, PROT_READ|PROT_WRITE, MAP_FILE|MAP_SHARED, infile, 0);
  if (mfile == MAP_FAILED) {perror("mmap failed"); exit(1);}

  printf("mmap at : %16lx\n", (long unsigned int) mfile);

  puts(mfile);
  strcpy(mfile+20,"Let's write over it");
  close(infile);
  return 0;
}
```

Sharing through Mapped Files



Admin Breaks

- Next week: guest lectures
  - Prof. Stoica – kev/val store
  - Kaifei – RPC, Vaishaal – NFS
- Prof. Culler will not be available for office hours
  - This W/Th and Next Week
- Midterm II on 11/14 6-7:30 Pimintel

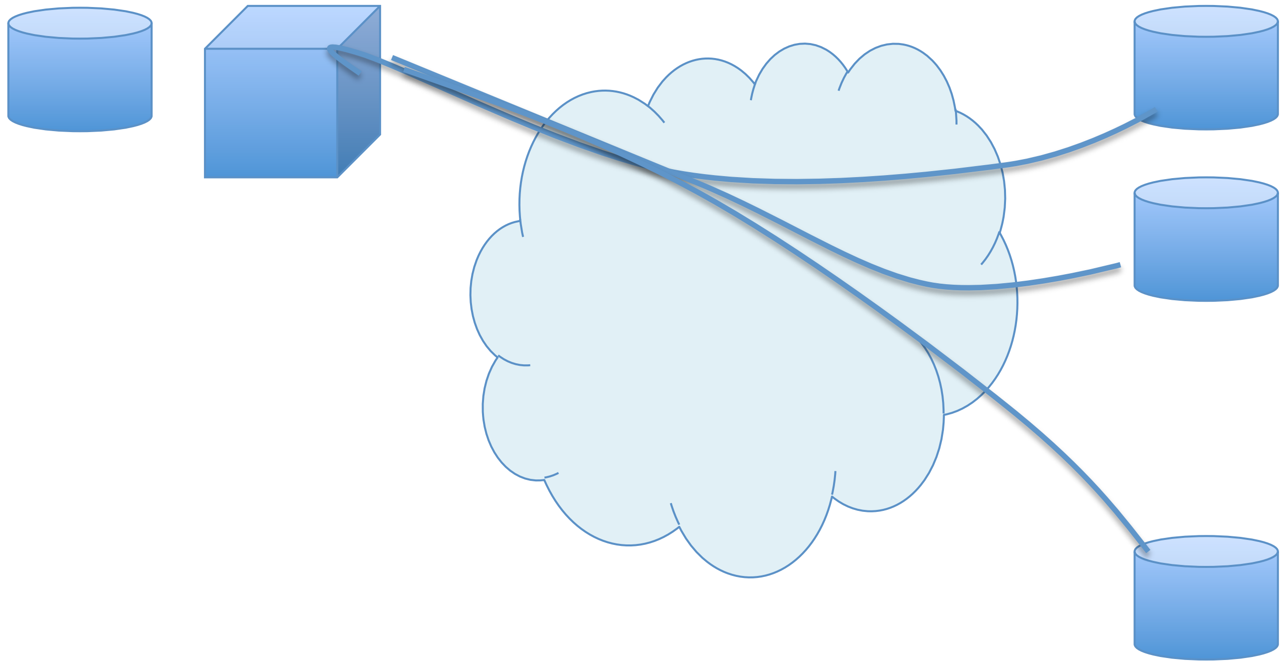
Reliable Storage

- How can we make a storage system more reliable than the physical devices that it is built out of?
  - Disks fail
  - SSDs wear out

- Redundancy

Example: Replicated Storage

- Suppose we write each data block to disks on 100 machines spread around the planet.
- Are we likely to be able to read the data even if disk(s) crash?



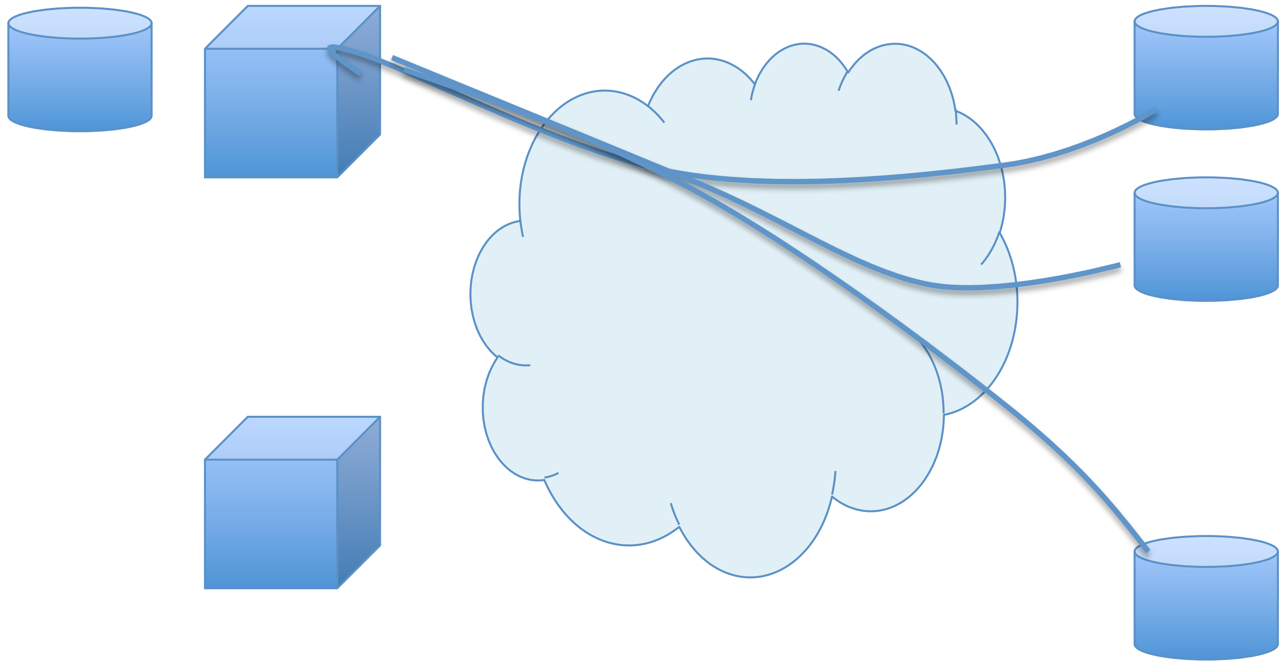
Definitions

- A system is *reliable* if it performs its intended function.
- A system is *available* if it currently can respond to a request.

- A storage system's *reliability* is the probability that it will continue to be reliable for some specified period of time.
- Its *availability* is the probability that it will be available at any given time.

Replicated Storage Example

- Highly reliable
- Highly available for reads
- Low availability for writes
  - Can’t write if any one is not up



Threats to Reliability

- Interrupted Operation
  - Crash or power failure in the middle of a series of related updates may leave stored data in an *inconsistent state*.
  - e.g.: transfer funds from BofA to Schwab. What if transfer is interrupted after withdrawal and before deposit
- Loss of stored data
  - Failure of non-volatile storage media may cause previously stored data to disappear or be corrupted

Solutions

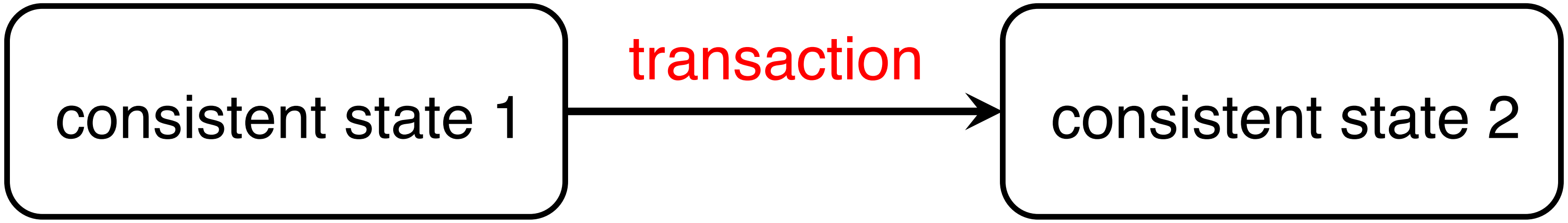
- Transactions for Atomic Updates
  - Ensure that multiple related updates are performed atomically
  - i.e., if a crash occurs in the middle, the state of the systems reflects either *all or none* of the updates
  - Most modern file systems use transactions internally to update the many pieces
  - Many applications implement their own transactions
- Redundancy for media failures
  - Redundant representation (error correcting codes)
  - Replication
  - E.g., RAID disks

Transactions

- Closely related to critical sections in manipulating shared data structures
- Extend concept of atomic update from memory to stable storage
  - Atomically update multiple persistent data structures
- Like flags for threads, many ad hoc approaches
  - FFS carefully ordered the sequence of updates so that if a crash occurred while manipulating directory or inodes the disk scan on reboot would detect and recover the error, -- fsck
  - Applications use temporary files and rename

Key concept: Transaction

- An atomic sequence of actions (reads/ writes) on a storage system (or database)
- That takes it from one consistent state to another



Typical Structure

- Begin a transaction – get transaction id
- Do a bunch of updates
  - If any fail along the way, roll-back
- Commit the transaction

“Classic” Example: Transaction

```
BEGIN;     --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';

 UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts
    WHERE name = 'Alice');

 UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';

 UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts
    WHERE name = 'Bob');

COMMIT;     --COMMIT WORK
```

Transfer $100 from Alice’s account to Bob’s account

The ACID properties of Transactions

- **Atomicity:** all actions in the transaction happen, or none happen
- **Consistency:** transactions maintain data integrity, e.g.,
  - Balance cannot be negative
  - Cannot reschedule meeting on February 30
- **Isolation:** execution of one transaction is isolated from that of all others; no problems from concurrency
- **Durability:** if a transaction commits, its effects persist despite crashes