Final Review

Tyler Harter
12/14/14
Chapters 4+5: Processes
How do we share?

CPU?

Memory?

Disk?
How do we share?

CPU? (a: time sharing)

Memory? (a: space sharing)

Disk? (a: space sharing)
How do we share?

CPU? (a: time sharing)  TODAY

Memory? (a: space sharing)

Disk? (a: space sharing)

Goal: processes should NOT even know they are sharing (each process will get its own virtual CPU)
What to Do with Processes That Are Not Running?

A: store context in OS struct

Look in kernel/proc.h

context (CPU registers)
ofile (file descriptors)
state (sleeping, running, etc)
State Transitions

Running \[\xrightarrow{\text{Descheduled}}\] Ready

Scheduled

I/O: initiate

Blocked

I/O: done
Chapters 6: LDE
CPU Time Sharing

Goal 1: **efficiency**
   OS should have minimal overhead

Goal 2: **control**
   Processes shouldn’t do anything bad
   OS should decide when processes run

Solution: **limited direct execution**
What to limit?

General memory access

Disk I/O

Special x86 instructions like `lidt`

How? Get HW help, put processes in “user mode”
What to limit?

General memory access

Disk I/O

Special x86 instructions like \texttt{lidt}

How? Get HW help, put processes in “user mode”
lidt example

Process P

RAM

trap-table index

syscall-table index
Process P

P tries to call `lidt`!
lidt example

goodbye, P

CPU warns OS, OS kills P
Problem: when to switch process contexts?

Direct execution $\Rightarrow$ OS can’t run while process runs

How can the OS do anything while it’s not running? 
A: it can’t

Solution: switch on interrupts. But which interrupt?
Chapters 7: Scheduling
# Scheduling Basics

<table>
<thead>
<tr>
<th>Workloads:</th>
<th>Schedulers:</th>
<th>Metrics:</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrival_time</td>
<td>FIFO</td>
<td>turnaround_time</td>
</tr>
<tr>
<td>run_time</td>
<td>SJF</td>
<td>response_time</td>
</tr>
<tr>
<td></td>
<td>STCF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RR</td>
<td></td>
</tr>
</tbody>
</table>
Workloads

Arrival: time at which scheduler is aware of job

Run time: how long does it take if run beginning to end?
Schedulers

FIFO: first in, first out

SJF: shortest job first (not preemptive)

STCF: shortest time to completion first

RR: round robin
What is the average turnaround time? (Q1)

\[
\frac{10 + 20 + 30}{3} = 20\text{s}
\]
FIFO vs. RR (Q5) — which is each?

Avg Response Time?

Q5
FIFO vs. RR (Q5) — which is each?

Avg Response Time?
\[(0+1+2)/3 = 1\]

Avg Response Time?
\[(0+5+10)/3 = 5\]
Chapters 16: Segmentation
int x;
int main(int argc, char *argv[]) {
    int y;
    int *z = malloc(sizeof(int));
}

x code
main data
y heap
z stack
```c
int x;
int main(int argc, char *argv[]) {
    int y;
    int *z = malloc(sizeof(int));
}
```

Match that Segment!

- `x` is a variable declared in the code segment.
- `main` is the entry point of the program.
- `y` and `z` are variables declared within the `main` function.
- The heap is used to store dynamically allocated memory.
- The stack is used for temporary storage during function calls.

The diagram shows the relationships between `x`, `y`, `z`, code, data, heap, and stack.
same code
P1 is running
P2 is running

base register
Program Code
Heap
(free)
Stack
wasted space
Multi-segment translation

One (correct) approach:
- break virtual addresses into two parts
- one part indicates segment
- one part indicates offset within segment
Chapters 18: Paging
Paging

Segmentation is too coarse-grained. Either waste space OR memcpy often.

We need a fine-grained alternative!

Paging idea:
- break mem into small, fix-sized chunks (aka pages)
- each virt page is independently mapped to a phys page
- grow memory segments however we please!
For segmentation, we used a formula (e.g., \( \text{phys} = \text{virt+offset} + \text{base_reg} \))

Now, we need a more general mapping mechanism.

What data structure is good? Big array, called a \textit{pagetable}
Where Are Pagetable’s Stored?

How big is a typical page table?
- assume 32-bit address space
- assume 4 KB pages
- assume 4 byte entries (or this could be less)
- \(2^{(32 - \log(4\text{KB}))} \times 4 = 4 \text{ MB}\)

Store in memory.
CPU finds it via register (e.g., CR3 on x86)
Other PT info

What other data should go in pagetable entries besides translation?

- valid bit
- protection bits
- present bit
- reference bit
- dirty bit
Chapters 19: TLBs
Translation Steps

H/W: for each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr) (cheap)
2. calculate addr of **PTE** (page table entry) (cheap)
3. fetch **PTE** (expensive)
4. extract **PFN** (page frame num) (cheap)
5. build **PA** (phys addr) (cheap)
6. fetch **PA** to register (expensive)

Which expensive step can we avoid?
Array Iterator

```c
int sum = 0;
for (i=0; i<N; i++) {
    sum += a[i];
}
```
Array Iterator

load 0x3000
load 0x3004
load 0x3008
load 0x300C
...

Virt
## Array Iterator

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x3000</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>load 0x3004</td>
<td>load 0x7000</td>
</tr>
<tr>
<td>load 0x3008</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>load 0x300C</td>
<td>load 0x7004</td>
</tr>
<tr>
<td>...</td>
<td>load 0x7008</td>
</tr>
<tr>
<td>...</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>...</td>
<td>load 0x700C</td>
</tr>
</tbody>
</table>
## Array Iterator

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x3000</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>load 0x3004</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>load 0x3008</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>load 0x300C</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>...</td>
<td>load 0x7000</td>
</tr>
<tr>
<td>...</td>
<td>load 0x7004</td>
</tr>
<tr>
<td>...</td>
<td>load 0x7008</td>
</tr>
<tr>
<td>...</td>
<td>load 0x700C</td>
</tr>
</tbody>
</table>
## Array Iterator

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x3000</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>load 0x3004</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>load 0x3008</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>load 0x300C</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>...</td>
<td>load 0x100C</td>
</tr>
<tr>
<td></td>
<td>load 0x7000</td>
</tr>
<tr>
<td></td>
<td>load 0x7004</td>
</tr>
<tr>
<td></td>
<td>load 0x7008</td>
</tr>
<tr>
<td></td>
<td>load 0x700C</td>
</tr>
<tr>
<td>Virt</td>
<td>Phys</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>load 0x3000</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>load 0x3004</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>load 0x3008</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>load 0x300C</td>
<td>load 0x100C</td>
</tr>
<tr>
<td>...</td>
<td>load 0x700C</td>
</tr>
<tr>
<td>...</td>
<td>load 0x7008</td>
</tr>
<tr>
<td>...</td>
<td>load 0x700C</td>
</tr>
</tbody>
</table>
Spatial Locality

Temporal Locality
Address Space Identifier

Tag each TLB entry with an 8-bit ASID
- how many ASIDs to we get?
- why not use PIDs?
- what if there are more PIDs than ASIDs?
Security

Modifying TLB entries is privileged
- otherwise what could you do?

Need same protection bits in TLB as pagetable
- rwx
Chapters 20: multi-level PTs
Motivation

Why do we want big virtual address spaces?
- programming is easier
- applications need not worry (as much) about fragmentation

Paging goals:
- space efficiency (don’t waste on invalid data)
- simplicity (no bookkeeping should require contiguous pages)
Waste!
Many invalid PT entries

<table>
<thead>
<tr>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>r-x</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>rw-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>rw-</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>rw-</td>
</tr>
</tbody>
</table>

...many more invalid...
Multi-Level Page Tables

Idea: break PT itself into pages
- a page directory refers to pieces
- only have pieces with >0 valid entries

Used by x86.
Problem: page directories may not fit in a page

Solution: split page directories into pieces. Use another page dir to refer to the page dir pieces.
Chapters 22: cache policy
Cache

Upon access, we must load the desired page.

Do we **prefetch** other adjacent pages? (remember disks have high fixed costs)

Prefetching more means we will have to **evict** more.

What to **evict**?
FIFO

Items are evicted in the order they are inserted
#### Belady’s Anomaly

**(a) size 3**

<table>
<thead>
<tr>
<th>Access</th>
<th>Hit</th>
<th>State (after)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>1,2</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>1,2,3</td>
</tr>
<tr>
<td>4</td>
<td>no</td>
<td>2,3,4</td>
</tr>
<tr>
<td>1</td>
<td>no</td>
<td>3,4,1</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td><strong>4,1,2</strong></td>
</tr>
<tr>
<td>5</td>
<td>no</td>
<td>1,2,5</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>1,2,5</td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td>1,2,5</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>2,5,3</td>
</tr>
<tr>
<td>4</td>
<td>no</td>
<td>5,3,4</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>5,3,4</td>
</tr>
</tbody>
</table>

**Access Hit State (after)**

<table>
<thead>
<tr>
<th>Access</th>
<th>Hit</th>
<th>State (after)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>1,2</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>1,2,3</td>
</tr>
<tr>
<td>4</td>
<td>no</td>
<td>1,2,3,4</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td><strong>1,2,3,4</strong></td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td><strong>1,2,3,4</strong></td>
</tr>
<tr>
<td>5</td>
<td>no</td>
<td>1,2,3,4</td>
</tr>
<tr>
<td>1</td>
<td>no</td>
<td>3,4,5,1</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>4,5,1,2</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>5,1,2,3</td>
</tr>
<tr>
<td>4</td>
<td>no</td>
<td>1,2,3,4</td>
</tr>
<tr>
<td>5</td>
<td>no</td>
<td>2,3,4,5</td>
</tr>
</tbody>
</table>

**(b) size 4**

<table>
<thead>
<tr>
<th>Access</th>
<th>Hit</th>
<th>State (after)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>1,2</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>1,2,3</td>
</tr>
<tr>
<td>4</td>
<td>no</td>
<td>1,2,3,4</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td><strong>1,2,3,4</strong></td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td><strong>1,2,3,4</strong></td>
</tr>
<tr>
<td>5</td>
<td>no</td>
<td>2,3,4,5</td>
</tr>
<tr>
<td>1</td>
<td>no</td>
<td>3,4,5,1</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>4,5,1,2</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>5,1,2,3</td>
</tr>
<tr>
<td>4</td>
<td>no</td>
<td>1,2,3,4</td>
</tr>
<tr>
<td>5</td>
<td>no</td>
<td>2,3,4,5</td>
</tr>
</tbody>
</table>
LRU: evict least-recently used
  - consider history

MRU: evict most-recently used
Discuss

Can Belady’s anomaly happen with LRU?

Stack property: smaller cache always subset of bigger
LRU Hardware Support

What is needed?

**Timestamps.** Why can’t OS alone track this?
LRU Hardware Support

What is needed?

Timestamps. Why can’t OS alone track this?

Cheap approximation: reference (or use) bits.
- set upon access, cleared by OS
- useful for clock algorithm
Thrashing

A machine is **thrashing** when there is not enough RAM, and we constantly swap in/out pages

Solutions?
- **admission control** (like scheduler project)
- **buy** more memory
- Linux **out-of-memory killer**!
Chapters 26+27: threads
Strategy 2

New abstraction: the **thread**.

Threads are just like **processes**, but they share the address space (e.g., using same PT).
CPU 1
running thread 1

CPU 2
running thread 2

RAM
CPU 1
running thread 1
PTBR

CPU 2
running thread 2
PTBR

RAM
PageDir A
PageDir B
...

---

[Diagram showing the relationship between CPUs and RAM with PTBR connections.]
CPU 1
running thread 1
PTBR
IP

CPU 2
running thread 2
PTBR
IP

RAM
PageDir A
PageDir B
...

---
CPU 1
running thread 1
PTBR
IP

CPU 2
running thread 2
PTBR
IP

RAM
PageDir A
PageDir B
...

Virt Mem (PageDir A)
CODE
HEAP
...
Each thread may be executing different code at the same time.
threads executing different functions need different stacks
Chapters 28: spinlocks
Lock Goals

Correctness

Fairness

Performance
void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1)
        ; // spin
}

void SpinUnlock(volatile unsigned int *lock) {
    xchg(lock, 0);
}
Test-and-set Spinlock (optimized)

```c
void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1)
        ; // spin
}

void SpinUnlock(volatile unsigned int *lock) {
    *lock = 0;
}
```
Test-and-set Spinlock (optimized)

```c
void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1)
        ; // spin
}

void SpinUnlock(volatile unsigned int *lock) {
    *lock = 0;
}

Works on newer x86 processors.
Not on all CPUs (sometimes due to CPU bugs!)
Basic Spinlocks are Unfair

The diagram illustrates the unfairness of basic spinlocks. Each block represents a process, and the green arrows indicate the lock/unlock transitions. The unfairness is evident as process A spins more than process B due to the lock/unlock mechanism.

- A: Process A
- B: Process B
CPU Scheduler is Ignorant

CPU scheduler may run B instead of A even though B is waiting for A.
Chapters 30: condition variables
(and sleeping locks)
Queue Lock

RUNNABLE: A, B, C, D

RUNNING: <empty>

WAITING: <empty>
Queue Lock

RUNNABLE:  B, C, D
RUNNING:   A
WAITING:   <empty>
<table>
<thead>
<tr>
<th>Status</th>
<th>Process(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runnable</td>
<td>C, D, A</td>
</tr>
<tr>
<td>Running</td>
<td>B</td>
</tr>
<tr>
<td>Waiting</td>
<td>&lt;empty&gt;</td>
</tr>
</tbody>
</table>

![Queue Lock Diagram](image_url)
Queue Lock

RUNNABLE: C, D, A

RUNNING: 

WAITING: B

lock

try lock (sleep)
Queue Lock

RUNNABLE: D, A

RUNNING: C

WAITING: B

A
B
C

lock

try lock (sleep)
Queue Lock

RUNNABLE: A, C

RUNNING: D

WAITING: B

Diagram showing the lock and try lock (sleep) actions.
Queue Lock

RUNNABLE: A, C

RUNNING:

WAITING: B, D
Queue Lock

RUNNABLE:  C

RUNNING:  A

WAITING:  B, D

lock
try lock (sleep)
try lock (sleep)
Queue Lock

RUNNABLE: A

RUNNING: C

WAITING: B, D

lock  try lock (sleep)  try lock (sleep)
Queue Lock

RUNNABLE: C

RUNNING: A

WAITING: B, D

lock
try lock (sleep)
try lock (sleep)
Queue Lock

RUNNABLE:  B, C

RUNNING:  A

WAITING:  D
Queue Lock

RUNNABLE: B, C

RUNNING: A

WAITING: D

A

lock

try lock (sleep)

try lock (sleep)

unlock

A

B

C

D

A

C

A
Queue Lock

RUNNABLE:  C, A

RUNNING:  B

WAITING:  D
Concurrency Objectives

**Mutual exclusion** (e.g., A and B don’t run at same time)
- solved with *locks*

**Ordering** (e.g., B runs after A)
- solved with *condition variables*
Correct CV’s

wait(cond_t *cv, mutex_t *lock)
- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

signal(cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything

requires kernel support!
Produce/Consumer

Pipes
Web servers
Memory allocators
Device I/O
...

General strategy: use condition variables to make consumers wait when there is nothing to consume, and make producers wait when buffers are full.
What about 2 consumers (v1)?

Can you find a problematic timeline?

Producer:        p1 p2 p4 p5 p6 p1 p2 p3
Consumer1: c1 c2 c3         signal()
Consumer2: c1 c2 c3    p1 p2 p3
 wait()         wait() wait() signal()
What about 2 consumers (v1)?

Can you find a problematic timeline?

wait()  wait()  signal()  wait()  signal()

Producer:  c1  c2  c3  p1  p2  p4  p5  p6  p1  p2  p3
Consumer1:  c1  c2  c3  
Consumer2:  c1  c2  c3  c2  c4  c5

does this wake producer or consumer2?
How to wake the right thread?

One solution:

wake all the threads!

Better solution (usually): use two condition variables.
Chapters 31: semaphores
CV’s vs. Semaphores

CV rules of thumb:
- Keep state in addition to CV’s
- Always do wait/signal with lock held
- Whenever you acquire a lock, recheck state

How do semaphores eliminate these needs?
Thread Queue:

Condition Variable (CV)

Thread Queue: Signal Queue:

Semaphore
Thread Queue:

Condition Variable (CV)

Semaphore

Thread Queue: A

Signal Queue: A

signal()
Condition Variable (CV)

Thread Queue:

Semaphore

Signal Queue:

Thread Queue:
signal()
Thread Queue:

Semaphore

Thread Queue:  Signal Queue:
Thread Queue: Signal Queue: 

Semaphore

Condition Variable (CV)
Condition Variable (CV)

Thread Queue:
- B

Semaphore

wait()

Thread Queue: B

Signal Queue: signal
Thread Queue:

B

wait()

Thread Queue:  Signal Queue:

Semaphore

Condition Variable (CV)
Condition Variable (CV)

Thread Queue:

Semaphore

Thread Queue: B

Signal Queue:
Thread Queue:

| B |

may wait forever (if not careful)

Thread Queue:  Signal Queue:

Semaphore
Thread Queue:

Condition Variable (CV)

B may wait forever (if not careful)

Semaphore

Thread Queue:  Signal Queue:

just use counter
Chapters 37: disks
Platter
Platter is covered with a magnetic film.
Many platters may be bound to the spindle.
Each surface is divided into rings called **tracks**. A stack of tracks (across platters) is called a **cylinder**.
The tracks are divided into numbered sectors.
Heads on a moving arm can read from each surface.
Spindle/platters rapidly spin.
Workload

So…
- seeks are slow
- rotations are slow
- transfers are fast

What kind of workload is fastest for disks?

**Sequential**: access sectors in order (transfer dominated)

**Random**: access sectors arbitrarily (seek+rotation dominated)
Other Improvements

Track Skew

Zones

Cache
Other Improvements

Track Skew

Zones

Cache
When reading 16 after 15, the head won’t settle quick enough, so we need to do a rotation.
enough time to settle now
Other Improvements

Track Skew

Zones

Cache
Other Improvements

Track Skew

Zones

Cache
Drive Cache

Drives may cache both reads and writes.

OS does this to.

What advantage does drive have for reads?

What advantage does drive have for writes?
Schedulers

os

Disk
Schedulers

Where should the scheduler go?
SPTF (Shortest Positioning Time First)

**Strategy**: always choose the request that will take the least time for seeking and rotating.

How to implement in disk?
How to implement in OS?
SPTF (Shortest Positioning Time First)

**Strategy**: always choose the request that will take the least time for seeking and rotating.

How to implement in **disk**?
How to implement in **OS**?

Disadvantages?
SCAN

Sweep back and forth, from one end of disk to the other, serving requests as you go.

Pros/Cons?
SCAN

Sweep back and forth, from one end of disk to the other, serving requests as you go.

Pros/Cons?

Better: C-SCAN (circular scan)
- only sweep in one direction
Chapters 38: RAID
<table>
<thead>
<tr>
<th>RAID</th>
<th>Reliability</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAID-0</td>
<td>0</td>
<td>C*N</td>
</tr>
<tr>
<td>RAID-1</td>
<td>1</td>
<td>C*N/2</td>
</tr>
<tr>
<td>RAID-4</td>
<td>1</td>
<td>N-1</td>
</tr>
<tr>
<td>RAID-5</td>
<td>1</td>
<td>N-1</td>
</tr>
<tr>
<td>RAID Level</td>
<td>Read Latency</td>
<td>Write Latency</td>
</tr>
<tr>
<td>------------</td>
<td>--------------</td>
<td>---------------</td>
</tr>
<tr>
<td>RAID-0</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>RAID-1</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>RAID-4</td>
<td>D</td>
<td>2D</td>
</tr>
<tr>
<td>RAID-5</td>
<td>D</td>
<td>2D</td>
</tr>
<tr>
<td></td>
<td>Read Latency</td>
<td>Write Latency</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td>---------------</td>
</tr>
<tr>
<td>RAID-0</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>RAID-1</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>RAID-4</td>
<td>D</td>
<td>2D</td>
</tr>
<tr>
<td>RAID-5</td>
<td>D</td>
<td>2D</td>
</tr>
</tbody>
</table>

but RAID-5 can do more in parallel
## All RAID

<table>
<thead>
<tr>
<th></th>
<th>Seq Read</th>
<th>Seq Write</th>
<th>Rand Read</th>
<th>Rand Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAID-0</td>
<td>N * S</td>
<td>N * S</td>
<td>N * R</td>
<td>N * R</td>
</tr>
<tr>
<td>RAID-1</td>
<td>N/2 * S</td>
<td>N/2 * S</td>
<td>N * R</td>
<td>N/2 * R</td>
</tr>
<tr>
<td>RAID-4</td>
<td>(N-1)*S</td>
<td>(N-1)*S</td>
<td>(N-1)*R</td>
<td>R/2</td>
</tr>
<tr>
<td>RAID-5</td>
<td>(N-1)*S</td>
<td>(N-1)*S</td>
<td>N * R</td>
<td>N/4 * R</td>
</tr>
</tbody>
</table>
## All RAID

<table>
<thead>
<tr>
<th></th>
<th>Seq Read</th>
<th>Seq Write</th>
<th>Rand Read</th>
<th>Rand Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAID-0</td>
<td>$N \times S$</td>
<td>$N \times S$</td>
<td>$N \times R$</td>
<td>$N \times R$</td>
</tr>
<tr>
<td>RAID-1</td>
<td>$N/2 \times S$</td>
<td>$N/2 \times S$</td>
<td>$N \times R$</td>
<td>$N/2 \times R$</td>
</tr>
<tr>
<td>RAID-4</td>
<td>$(N-1)\times S$</td>
<td>$(N-1)\times S$</td>
<td>$(N-1)\times R$</td>
<td>$R/2$</td>
</tr>
<tr>
<td>RAID-5</td>
<td>$(N-1)\times S$</td>
<td>$(N-1)\times S$</td>
<td>$N \times R$</td>
<td>$N/4 \times R$</td>
</tr>
</tbody>
</table>

RAID-5 is strictly better than RAID-4
<table>
<thead>
<tr>
<th>RAID</th>
<th>Seq Read</th>
<th>Seq Write</th>
<th>Rand Read</th>
<th>Rand Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAID-0</td>
<td>(N \times S)</td>
<td>(N \times S)</td>
<td>(N \times R)</td>
<td>(N \times R)</td>
</tr>
<tr>
<td>RAID-1</td>
<td>(N/2 \times S)</td>
<td>(N/2 \times S)</td>
<td>(N \times R)</td>
<td>(N/2 \times R)</td>
</tr>
<tr>
<td>RAID-5</td>
<td>((N-1)\times S)</td>
<td>((N-1)\times S)</td>
<td>(N \times R)</td>
<td>(N/4 \times R)</td>
</tr>
</tbody>
</table>
RAID-0 is always fastest and has best capacity.
(but at cost of reliability)
<table>
<thead>
<tr>
<th>RAID</th>
<th>Seq Read</th>
<th>Seq Write</th>
<th>Rand Read</th>
<th>Rand Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAID-0</td>
<td>$N \times S$</td>
<td>$N \times S$</td>
<td>$N \times R$</td>
<td>$N \times R$</td>
</tr>
<tr>
<td>RAID-1</td>
<td>$N/2 \times S$</td>
<td>$N/2 \times S$</td>
<td>$N \times R$</td>
<td>$N/2 \times R$</td>
</tr>
<tr>
<td>RAID-5</td>
<td>$(N-1) \times S$</td>
<td>$(N-1) \times S$</td>
<td>$N \times R$</td>
<td>$N/4 \times R$</td>
</tr>
</tbody>
</table>
**All RAID**

<table>
<thead>
<tr>
<th>RAID Type</th>
<th>Seq Read</th>
<th>Seq Write</th>
<th>Rand Read</th>
<th>Rand Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAID-0</td>
<td>$N \times S$</td>
<td>$N \times S$</td>
<td>$N \times R$</td>
<td>$N \times R$</td>
</tr>
<tr>
<td>RAID-1</td>
<td>$N/2 \times S$</td>
<td>$N/2 \times S$</td>
<td>$N \times R$</td>
<td>$N/2 \times R$</td>
</tr>
<tr>
<td>RAID-5</td>
<td>$(N-1)\times S$</td>
<td>$(N-1)\times S$</td>
<td>$N \times R$</td>
<td>$N/4 \times R$</td>
</tr>
</tbody>
</table>

RAID-5 better than RAID-1 for sequential.
<table>
<thead>
<tr>
<th>RAID</th>
<th>Seq Read</th>
<th>Seq Write</th>
<th>Rand Read</th>
<th>Rand Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAID-0</td>
<td>N * S</td>
<td>N * S</td>
<td>N * R</td>
<td>N * R</td>
</tr>
<tr>
<td>RAID-1</td>
<td>N/2 * S</td>
<td>N/2 * S</td>
<td>N * R</td>
<td>N/2 * R</td>
</tr>
<tr>
<td>RAID-5</td>
<td>(N-1)*S</td>
<td>(N-1)*S</td>
<td>N * R</td>
<td>N/4 * R</td>
</tr>
</tbody>
</table>
## All RAID

<table>
<thead>
<tr>
<th></th>
<th>Seq Read</th>
<th>Seq Write</th>
<th>Rand Read</th>
<th>Rand Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAID-0</td>
<td>$N \times S$</td>
<td>$N \times S$</td>
<td>$N \times R$</td>
<td>$N \times R$</td>
</tr>
<tr>
<td>RAID-1</td>
<td>$N/2 \times S$</td>
<td>$N/2 \times S$</td>
<td>$N \times R$</td>
<td>$N/2 \times R$</td>
</tr>
<tr>
<td>RAID-5</td>
<td>$(N-1)\times S$</td>
<td>$(N-1)\times S$</td>
<td>$N \times R$</td>
<td>$N/4 \times R$</td>
</tr>
</tbody>
</table>

RAID-1 better than RAID-4 for random write.
Chapters 39: File-System API
File Names

Three types of names:
- inode
- path
- file descriptor
Atomic File Update

Say we want to update file.txt.

1. write new data to new file.txt.tmp file
2. fsync file.txt.tmp
3. rename file.txt.tmp over file.txt, replacing it
Chapters 41: FFS

Treat a disk like a disk!

Place related data together: hopefully makes future reads faster.
Technique 2: Groups

![Diagram showing the layout of superblock, bitmaps, inodes, and data blocks.]

Before: whole disk

Fast
Technique 2: Groups

- super block
- bitmaps
- inodes
- Data Blocks

slow

before: whole disk
Technique 2: Groups

before: whole disk

super block  bitmaps  inodes  Data Blocks

slower
Technique 2: Groups

Before: whole disk

- Super block
- Bitmaps
- Inodes
- Data Blocks

Slowest
Technique 2: Groups

super block | bitmaps | inodes | Data Blocks

before: whole disk
Technique 2: Groups

- super block
- bitmaps
- inodes
- Data Blocks

now: one (smallish) group
Technique 2: Groups

zoom out

- Group 1
- Group 2
- Group 3

...
Technique 2: Groups

strategy: allocate **inodes** and **data blocks** in same group.
Chapters 42: Journaling
Definition: if $A$ and $B$ are two pieces of data, and knowing $A$ eliminates some or all the values $B$ could take, there is redundancy between $A$ and $B$.

RAID examples:
- mirrored disk (complete redundancy)
- parity blocks (partial redundancy)
Problem 3

Give 5 examples of redundancy in FFS (or files systems in general).
Problem 3

Give 5 examples of redundancy in FFS (or files systems in general).

Dir entries AND inode table.
Dir entries AND inode link count.
Data bitmap AND inode pointers.
Data bitmap AND group descriptor.
Inode file size AND inode/indirect pointers.
...

fsck

FSCK = file system checker.

Strategy: after a crash, scan whole disk for contradictions.

For example, is a bitmap block correct?

Read every valid inode+indirect. If an inode points to a block, the corresponding bit should be 1
Journal: General Strategy

Never delete ANY old data, until, ALL new data is safely on disk.

Ironically, this means we’re adding redundancy to fix the problem caused by redundancy.
transaction: write A to block 5; write B to block 2
transactions: write A to block 5; write B to block 2
transaction: write A to block 5; write B to block 2
transaction: write A to block 5; write B to block 2
transaction: write A to block 5; write B to block 2
transaction: write A to block 5; write B to block 2
New Layout

journal
Optimizations

1. Reuse small area for journal
2. Barriers
3. Checksums
4. Circular journal
5. Logical journal
Chapters 43: LFS

Write data fastest way possible… Sequentially!

Reads may be slower later (scattered).
Big Picture

buffer:  

disk:  

Big Picture

buffer: □

disk: □
Big Picture

buffer: 

disk: 

Big Picture

buffer: [Median]

disk: [Upper Quartile]
Big Picture

buffer: 

disk:
Problem: for every data update, we need to do updates all the way up the tree.

Why? We change inode number when we copy it.

Solution: keep inode numbers constant. Don’t base on offset.

Before we found inodes with math. How now?
Data Structures (attempt 2)

What can we get rid of from FFS?
- allocation structs: data + inode bitmaps

Inodes are no longer at fixed offset.
- use imap struct to map number => inode.
Garbage Collection

Is data alive? Use segment summary.

How to clean? Copy clean data out of M segments into N new segments (N < M).

Which segments to clean? Cold, invalid, etc.
Chapters 44: Integrity

Checksums...
Chapters 47: Distributed Systems
Channels

UDP: unreliable

TCP: reliable
- seq numbers, buffering, retry
RPC

Machine A

```c
int main(...)
{
}
```

Machine B

```c
int foo(char *msg)
{
...
}
```
int main(…) {
    int x = foo();
}

int foo(char *msg) {
...
}
int main(…) {
    int x = foo();
}

int foo(char *msg) {
    ...
}

RPC

Machine A

Machine B

Want main() on A to call foo() on B.
RPC

Machine A

```c
int main(...) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

Machine B

```c
int foo(char *msg) {
    ...
}
```

Want main() on A to call foo() on B.
RPC

Machine A

```c
int main(...) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

Machine B

```c
int foo(char *msg) {
    ...
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```

Want main() on A to call foo() on B.
int main(...) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
RPC

Machine A

```c
int main(...) {
  int x = foo();
}

int foo(char *msg) {
  send msg to B
  recv msg from B
}
```

Machine B

```c
int foo(char *msg) {
  ...
}

void foo_listener() {
  while(1) {
    recv, call foo
  }
}
```

What it feels like for programmer.
```c
int main(...) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```
RPC Tools

RPC packages help with this with two components.

(1) **Stub generation**
   - create wrappers automatically

(2) **Runtime library**
   - thread pool
   - socket listeners call functions on server
Chapters 48: NFS
General Strategy: Export FS

Client

Local FS   NFS

Server

Local FS

mount
General Strategy: Export FS

Client

- Local FS
- NFS

Server

- Local FS
General Strategy: Export FS

Client

```
Local FS  NFS
```

Server

```
Local FS
```
General Strategy: Export FS

Client

Local FS  NFS

Server

Local FS

read
Stateless

Requests understandable without any context about clients.

No fds!
Idempotent

Design API so that there is no harm is executing a call more than once.

An API call that has this is “idempotent”. If f() is idempotent, then:
f() has the same effect as f(); f(); … f(); f()
Cache Consistency

Know update visibility, stale cache.
Chapters 49: AFS
AFS Goals

Primary goal: scalability! (many clients per server)

More reasonable semantics for concurrent file access.

Not good about handling some failure scenarios.
AFS Design

NFS: export local FS

AFS: present big file tree, store across many machines.

Break tree into “volumes.”
I.e., partial sub trees.
Update Visibility

Clients updates not seen on servers yet.

AFS solution:
- flush on close
- buffer whole files on local disk

Concurrent writes? Last writer (i.e., closer) wins.

Never get mixed data.
Stale Cache

AFS solution: tell clients when data is overwritten.

When clients cache data, ask for “callback” from server.

No longer stateless!
Callbacks

What if client crashes?

What if server runs out of memory?

What if server crashes?
GFS
Architecture

Master

(metadata)

(one)

RPC

RPC

Local FS’s

large capacity

metadata consistency easy

Client

(many)

Worker
Chunk Layer

Break GFS files into large chunks (e.g., 64MB).

Workers store physical chunks in Linux files.

Master maps logical chunk to physical chunk locations.
File Namespace

**Master**
- file namespace:
  - /foo/bar => 924,813
  - /var/log => 123,999
- chunk map:
<table>
<thead>
<tr>
<th>logical</th>
<th>phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>924</td>
<td>w2, w5, w7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Worker w2**
- Local FS:
  - /chunks/942 => data1
  - /churks/521 => data2
  ...
File Namespace

Master

file namespace:
/foo/bar => 924,813
/var/log => 123,999

chunk map:

<table>
<thead>
<tr>
<th>logical</th>
<th>phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>924</td>
<td>w2, w5, w7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

client

lookup /foo/bar

Worker w2

Local FS

/chunks/942 => data1
/churks/521 => data2
...
File Namespace

**Master**

- **file namespace:**
  - `/foo/bar` => 924,813
  - `/var/log` => 123,999

- **chunk map:**
  - logical
    - 924
    - ...
  - phys
    - w2, w5, w7
    - ...

**Worker w2**

- **Local FS**
  - `/chunks/942` => data1
  - `/chunks/521` => data2
  - ...

**client**

- **file namespace:**
  - `/foo/bar` => 924,813
  - `/var/log` => 123,999

- **chunk map:**
  - 924: [w2, w5, w7]
  - 813: [...]

...
File Namespace

**Master**

- file namespace:
  - /foo/bar => 924,813
  - /var/log => 123,999

  chunk map:
  - logical: 924
  - phys: w2, w5, w7

**Worker w2**

- Local FS:
  - /chunks/942 => data1
  - /churks/521 => data2
  - ...

**Client**

- ...

- file namespace:
  - /foo/bar => 924,813
  - /var/log => 123,999
**File Namespace**

**Master**
- **file namespace:**
  - /foo/bar => 924,813
  - /var/log => 123,999

  **chunk map:**
<table>
<thead>
<tr>
<th>logical</th>
<th>phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>924</td>
<td>w2, w5, w7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Worker w2**
- **Local FS**
  - /chunks/942 => data1
  - /chunks/521 => data2
  - ...  

**client**
- read 942:
  - offset = 0MB
  - size = 1MB
Master: Crashes + Consistency

File namespace and chunk map are 100% in RAM.  
- allows master to work with 1000's of workers  
- what master crashes?
MapReduce
public void map(LongWritable key, Text value) {
    String line = value.toString();
    StringTokenizer st = new StringTokenizer(line);
    while (st.hasMoreTokens())
        output.collect(st.nextToken(), 1);
}

public void reduce(Text key,
    Iterator<IntWritable> values) {
    int sum = 0;
    while (values.hasNext())
        sum += values.next().get();
    output.collect(key, sum);
}
Single- vs. Multi-Level Cell

SLC: expensive, robust

MLC: cheap, sensitive
Wearout

Problem: flash cells wear out after being overwritten too many times.

MLC: ~10K times
SLC: ~100K times

Usage strategy: wear leveling.
- prevents some cells from wearing out while others still fresh.
Block

one page
Flash Hierarchy

**Plane**: 1024 to 4096 blocks  
- planes accessed in parallel

**Block**: 64 to 256 pages  
- unit of *erase*

**Page**: 2 to 8 KB  
- unit of *read and program*
<table>
<thead>
<tr>
<th></th>
<th>disk</th>
<th>flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read sector</td>
<td>read page</td>
</tr>
<tr>
<td>write</td>
<td>write sector</td>
<td>program page (0's) erase block (1's)</td>
</tr>
</tbody>
</table>
Flash Translation Layer

logical:

0 1 2 3 4 5 6 7

physical:

block 0

00 00 00 00
01 10 11 00

block 1

10 11 11 11
01 11 11 11

11 11 11
Flash Translation Layer

write 1101

logical:

0 1 2 3 4 5 6 7

physical:

block 0

00 00 00 00
01 10 11 00

block 1

10 11 11 11 11
01 11 11 11 11
Flash Translation Layer

logical:

write 1101

physical:

block 0
00 00 00 00
01 10 11 00

block 1
10 11 11 11
01 01 11 11
11 11 11 11
Flash Translation Layer

write 1101

logical:

```
0 1 2 3 4 5 6 7
```

physical:

```
block 0
00 00 00 00 01 10 11 00

block 1
10 11 11 11 01 01 11 11
```
Flash Translation Layer

logical:  

0 1 2 3 4 5 6 7

physical:  

block 0
00 00 00 00
01 10 11 00

block 1
10 11 11 11
01 01 11 11
11 11
Flash Translation Layer

logical:

must eventually be garbage collected

physical:

block 0

block 1
Search Engines

PageRank: important?

Inverted index: relevant?
Strategy: Count Backlinks

Importance:
A = 1
B = 3.5
C = 0.5 (from B’s vote)
D = 0
E = 0.5 (from A’s vote)
F = 0.5

Why do A and B get same votes? B is more important.
Circular Votes

Want: number of votes you get determines number of votes you give.

Problem: changing A’s votes changes B’s votes changes A’s votes…

Fortunately, if you just keep updating every PageRank, it eventually converges.
Intuition: Random Surfer

Imagine!

1. a bunch of web surfers start on various pages
2. they randomly click links, forever
3. you measure webpage visit frequency

Visit frequency will be proportional to PageRank.
Graph 1

A --- B --- C

0.25 0.5 0.25
Graph 1

Rank(B) = (0.25 / 1) + (0.25 / 1) = 0.5
Rank(A) = (0.5 / 2) = 0.25
Rank(C) = (0.5 / 2) = 0.25

\[ \text{Rank}(x) = c \sum_{y \in \text{LinksTo}(x)} \frac{\text{Rank}(y)}{N_y} \]
Graph 3

Problem: ???
Problem: Surfers get stuck in C and D. C+D called a rank “sink”. A and B get 0 rank.
Inverted Index

forward index

<table>
<thead>
<tr>
<th>docID</th>
<th>wordID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>1442</td>
<td>922</td>
</tr>
<tr>
<td>1442</td>
<td>2</td>
</tr>
<tr>
<td>1442</td>
<td>66</td>
</tr>
<tr>
<td>1442</td>
<td>42</td>
</tr>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Inverted Index

forward index

<table>
<thead>
<tr>
<th>docID</th>
<th>wordID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>1442</td>
<td>922</td>
</tr>
<tr>
<td>1442</td>
<td>2</td>
</tr>
<tr>
<td>1442</td>
<td>66</td>
</tr>
<tr>
<td>1442</td>
<td>42</td>
</tr>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>docID</th>
<th>wordID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>1442</td>
<td>922</td>
</tr>
<tr>
<td>1442</td>
<td>2</td>
</tr>
<tr>
<td>1442</td>
<td>66</td>
</tr>
<tr>
<td>1442</td>
<td>42</td>
</tr>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
## Inverted Index

### forward index

<table>
<thead>
<tr>
<th>docID</th>
<th>wordID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>1442</td>
<td>922</td>
</tr>
<tr>
<td>1442</td>
<td>2</td>
</tr>
<tr>
<td>1442</td>
<td>66</td>
</tr>
<tr>
<td>1442</td>
<td>42</td>
</tr>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### swap columns

<table>
<thead>
<tr>
<th>wordID</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1442</td>
</tr>
<tr>
<td>922</td>
<td>1442</td>
</tr>
<tr>
<td>2</td>
<td>1442</td>
</tr>
<tr>
<td>66</td>
<td>1442</td>
</tr>
<tr>
<td>42</td>
<td>1442</td>
</tr>
<tr>
<td>5</td>
<td>1442</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Inverted Index

#### Forward Index

<table>
<thead>
<tr>
<th>docID</th>
<th>wordID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>1442</td>
<td>922</td>
</tr>
<tr>
<td>1442</td>
<td>2</td>
</tr>
<tr>
<td>1442</td>
<td>66</td>
</tr>
<tr>
<td>1442</td>
<td>42</td>
</tr>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#### Sort by wordID

<table>
<thead>
<tr>
<th>wordID</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>244</td>
</tr>
<tr>
<td>2</td>
<td>1442</td>
</tr>
<tr>
<td>5</td>
<td>1442</td>
</tr>
<tr>
<td>5</td>
<td>1442</td>
</tr>
<tr>
<td>5</td>
<td>999</td>
</tr>
<tr>
<td>6</td>
<td>133</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Inverted Index

<table>
<thead>
<tr>
<th>docID</th>
<th>wordID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>1442</td>
<td>922</td>
</tr>
<tr>
<td>1442</td>
<td>2</td>
</tr>
<tr>
<td>1442</td>
<td>66</td>
</tr>
<tr>
<td>1442</td>
<td>42</td>
</tr>
<tr>
<td>1442</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

forward index

<table>
<thead>
<tr>
<th>wordID</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>244</td>
</tr>
<tr>
<td>2</td>
<td>1442</td>
</tr>
<tr>
<td>5</td>
<td>1442,1442,999</td>
</tr>
<tr>
<td>6</td>
<td>133,411</td>
</tr>
<tr>
<td>7</td>
<td>1442,133,999</td>
</tr>
<tr>
<td>9</td>
<td>411,875</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

inverted index