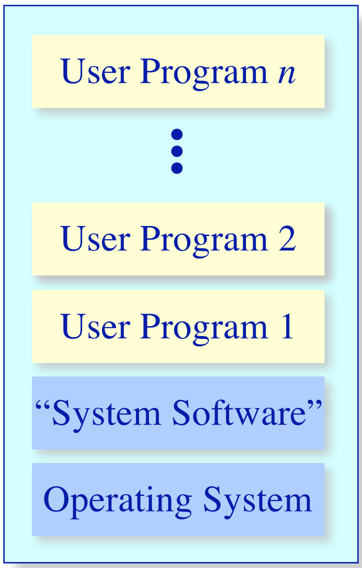
Page Replacement Algorithms

Virtual Memory Management

Fundamental issues : A Recap

- Key concept: Demand paging
  - Load pages into memory only when a page fault occurs
- Issues:
  - Placement strategies
    - Place pages anywhere – no placement policy required
  - Replacement strategies
    - What to do when there exist more jobs than can fit in memory
  - Load control strategies
    - Determining how many jobs can be in memory at one time



Memory

Page Replacement Algorithms
Concept

- Typically Σ_i VAS_i >> *Physical Memory*
- With demand paging, physical memory fills quickly
- When a process faults & memory is full, some page must be swapped out
  - Handling a page fault now requires ***2*** disk accesses not 1!

Which page should be replaced?

Local replacement — Replace a page of the faulting process
Global replacement — Possibly replace the page of another process

Page Replacement Algorithms
Evaluation methodology

- Record a *trace* of the pages accessed by a process
  - Example: (Virtual) address trace...
    (3,0), (1,9), (4,1), (2,1), (5,3), (2,0), (1,9), (2,4), (3,1), (4,8)
  - generates page trace
    3, 1, 4, 2, 5, 2, 1, 2, 3, 4 (represented as *c*, *a*, *d*, *b*, *e*, *b*, *a*, *b*, *c*, *d*)

Simulate the behavior of a page replacement algorithm on the trace and record the number of page faults generated

fewer faults ⇒ *better performance*

Optimal Page Replacement
Clairvoyant replacement

- Replace the page that won't be needed for the longest time in the future

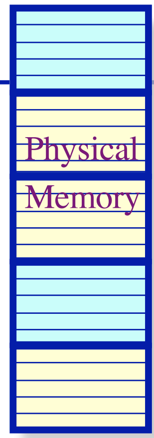
Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*										
	1	*b*										
	2	*c*										
	3	*d*										
Faults												
Time page needed next												

Optimal Page Replacement
Clairvoyant replacement

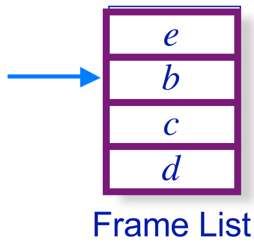
- Replace the page that won't be needed for the longest time in the future

		0	1	2	3	4	5	6	7	8	9	10
Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	***d***
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*
	3	*d*	*d*	*d*	*d*	*d*	***e***	*e*	*e*	*e*	*e*	*e*
Faults							•					•
Time page needed next							$a = 7$ $b = 6$ $c = 9$ $d = 10$					$a = 15$ $b = 11$ $c = 13$ $e = 14$

Local Page Replacement
FIFO replacement



- Simple to implement
  - A single pointer suffices

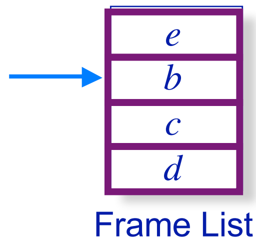


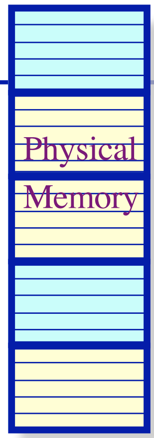
- Performance with 4 page frames:

	Time	0	1	2	3	4	5	6	7	8	9	10
	Requests		*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*										
	1	*b*										
	2	*c*										
	3	*d*										
Faults												

Local Page Replacement
FIFO replacement

- Simple to implement
  - A single pointer suffices





- Performance with 4 page frames:

	Time	0	1	2	3	4	5	6	7	8	9	10
	Requests		c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	**e**	e	e	e	e	**d**
	1	b	b	b	b	b	b	b	**a**	a	a	a
	2	c	c	c	c	c	c	c	c	**b**	b	b
	3	d	d	d	d	d	d	d	d	d	**c**	c
Faults							•		•	•	•	•

Least Recently Used Page Replacement

Use the recent past as a predictor of the near future

- Replace the page that hasn't been referenced for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*										
	1	*b*										
	2	*c*										
	3	*d*										
Faults												
Time page last used												

Least Recently Used Page Replacement

Use the recent past as a predictor of the near future

- Replace the page that hasn't been referenced for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	***e***	*e*	*e*	*e*	*e*	***d***
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							•				•	•
Time page last used						*a* = 2 *b* = 4 *c* = 1 *d* = 3				*a* = 7 *b* = 8 *e* = 5 *d* = 3	*a* = 7 *b* = 8 *e* = 5 *c* = 9	

Least Recently Used Page Replacement
Implementation

- Maintain a "stack" of recently used pages

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	**e**	e	e	e	e	**d**
	3	d	d	d	d	d	d	d	d	d	**c**	c
Faults							•				•	•

LRU
page stack

Page to replace

Least Recently Used Page Replacement
Implementation

- Maintain a "stack" of recently used pages

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	***e***	*e*	*e*	*e*	*e*	***d***
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							•				•	•

LRU page stack	*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
		c	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*
			c	*a*	*d*	*d*	*e*	*e*	*a*	*b*
				c	*a*	*a*	*d*	*d*	*e*	*a*
Page to replace					***c***				***d***	***e***

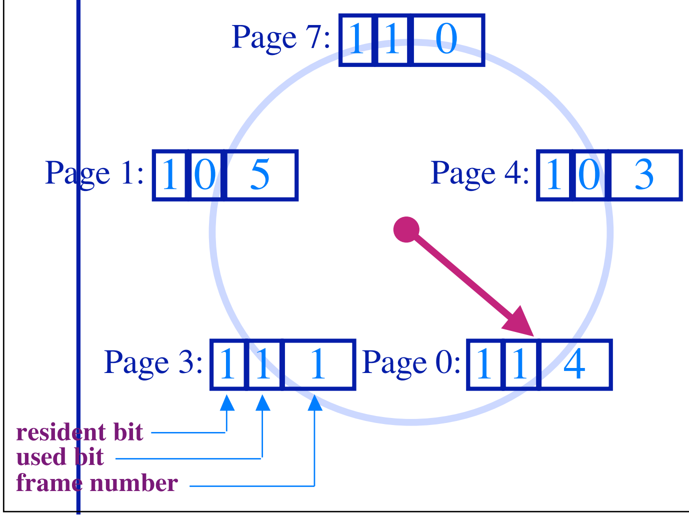
Least Recently Used Page Replacement

Alternate Implementation --- Aging Register

- Maintain an n-bit aging register $R = R_{n-1}R_{n-2}...R_0$ for each page frame
  - On a page reference, set R_{n-1} to 1
  - Every T units of time, shift the aging vector right by one bit
- Key idea:
  - Aging vector can be interpreted as a positive binary number
  - Value of R decreases periodically unless the page is referenced
- Page replacement algorithm:
  - On a page fault, replace the page with the smallest value of R

Approximate LRU Page Replacement
The Clock algorithm

- Maintain a circular list of pages resident in memory
  - Use a *clock* (or *used/referenced*) bit to track how often a page is accessed
  - The bit is set whenever a page is referenced
- Clock hand sweeps over pages looking for one with *used* bit = 0
  - Replace pages that haven't been referenced for one complete revolution of the clock




```
func Clock_Replacement
begin
  while (victim page not found) do
    if (used bit for current page = 0) then
      replace current page
    else
      reset used bit
    end if
    advance clock pointer
  end while
end Clock_Replacement
```

Clock Page Replacement
Example

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*						
	1	*b*	*b*	*b*	*b*	*b*						
	2	*c*	*c*	*c*	*c*	*c*						
	3	*d*	*d*	*d*	*d*	*d*						
Faults												

Page table entries for resident pages:

1	*a*
1	*b*
1	*c*
1	*d*

Clock Page Replacement
Example

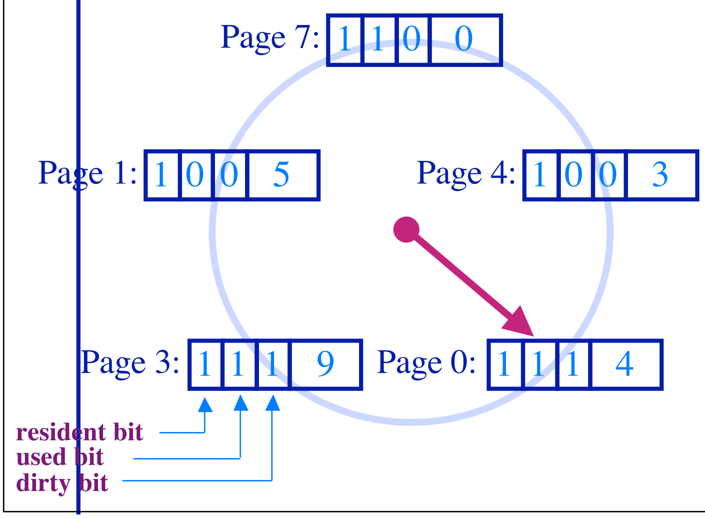
Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	***e***	*e*	*e*	*e*	*e*	***d***
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	***a***	*a*	*a*	*a*
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							•		•		•	•

Page table entries for resident pages:

1	*a*	1	*e*	1	*e*	1	*e*	1	*e*	1	*e*	1	*d*
1	*b*	0	*b*	1	*b*	0	*b*	1	*b*	1	*b*	0	*b*
1	*c*	0	*c*	0	*c*	1	*a*	1	*a*	1	*a*	0	*a*
1	*d*	0	*d*	0	*d*	0	*d*	0	*d*	1	*c*	0	*c*

Optimizing Approximate LRU Replacement
The Second Chance algorithm

- There is a significant cost to replacing "dirty" pages
- Modify the Clock algorithm to allow dirty pages to always survive one sweep of the clock hand
  - Use both the *dirty bit* and the *used bit* to drive replacement



Second Chance Algorithm

Before clock sweep: *used*	Before clock sweep: *dirty*	After clock sweep: *used*	After clock sweep: *dirty*
0	0	*replace page*	
0	1	0	0
1	0	0	0
1	1	0	1

The Second Chance Algorithm
Example

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a^w	d	b^w	e	b	a^w	b	c	d
Page Frames	0	a	a	a	a	a						
	1	b	b	b	b	b						
	2	c	c	c	c	c						
	3	d	d	d	d	d						
Faults												

Page table entries for resident pages:

10	a
10	b
10	c
10	d

The Second Chance Algorithm

Example

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a^w	d	b^w	e	b	a^w	b	c	d
Page Frames 0	a	a	a	a	a	a	a	a	a	a	a
1	b	b	b	b	b	b	b	b	b	b	d
2	c	c	c	c	c	e	e	e	e	e	e
3	d	d	d	d	d	d	d	d	d	c	c
Faults						•				•	•

Page table entries for resident pages:

10 a	11 a	00 a^*	00 a^*	11 a	11 a	00 a^*
10 b	11 b	00 b^*	10 b^*	10 b^*	10 b^*	10 d
10 c	10 c	10 e	10 e	10 e	10 e	00 e
10 d	10 d	00 d	00 d	00 d	10 c	00 c

The Problem With Local Page Replacement

How much memory do we allocate to a process?

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*

Page Frames		
	0	*a*
	1	*b*
	2	*c*
Faults		

Page Frames		
	0	*a*
	1	*b*
	2	*c*
	3	–
Faults		

The Problem With Local Page Replacement

How much memory do we allocate to a process?

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*

Page Frames													
0	*a*	*a*	*a*	*a*	***d***	*d*	*d*	***c***	*c*	*c*	***b***	*b*	*b*
1	*b*	*b*	*b*	*b*	*b*	***a***	*a*	*a*	***d***	*d*	*d*	***c***	*c*
2	*c*	*c*	*c*	*c*	*c*	*c*	***b***	*b*	*b*	***a***	*a*	*a*	***d***
Faults					•	•	•	•	•	•	•	•	•

Page Frames													
0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*
3	–				***d***	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*
Faults					•								

Page Replacement Algorithms

Performance

- Local page replacement
  - LRU — *Ages pages based on when they were last used*
  - FIFO — *Ages pages based on when they're brought into memory*
- Towards global page replacement ... with variable number of page frames allocated to processes

The principle of locality

- 90% of the execution of a program is sequential
- Most iterative constructs consist of a relatively small number of instructions
- When processing large data structures, the dominant cost is sequential processing on individual structure elements
- Temporal vs. physical locality

Optimal Page Replacement

For processes with a variable number of frames

- *VMIN* — Replace a page that is not referenced in the *next* τ accesses
- Example: τ = 4

		0	1	2	3	4	5	6	7	8	9	10
Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory	Page *a*	• $t = 0$										
	Page *b*	-										
	Page *c*	-										
	Page *d*	• $t = -1$										
	Page *e*	-										
Faults												

Optimal Page Replacement
For processes with a variable number of frames

- *VMIN* — Replace a page that is not referenced in the *next* τ accesses
- Example: τ = 4

		0	1	2	3	4	5	6	7	8	9	10
Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory	Page *a*	• $t = 0$	-	-	-	-	-	-	-	-	*F*	-
	Page *b*	-	-	-	-	*F*	-	-	-	-	-	-
	Page *c*	-	*F*	•	•	•	•	•	•	-	-	-
	Page *d*	• $t = -1$	•	•	•	-	-	-	-	-	-	*F*
	Page *e*	-	-	-	-	-	-	*F*	•	•	-	-
Faults			•			•		•			•	•

Explicitly Using Locality

The working set model of page replacement

- Assume recently referenced pages are likely to be referenced again soon…
- … and *only* keep those pages recently referenced in memory (called *the working set*)
  - Thus pages may be removed even when no page fault occurs
  - The number of frames allocated to a process will vary over time
- A process is allowed to execute only if its working set fits into memory
  - The working set model performs implicit load control

Working Set Page Replacement
Implementation

- Keep track of the last τ references
  - The pages referenced during the last τ memory accesses are the working set
  - τ is called the *window size*
- Example: Working set computation, τ = 4 references:

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory	Page *a*	• $t = 0$										
	Page *b*	-										
	Page *c*	-										
	Page *d*	• $t = -1$										
	Page *e*	• $t = -2$										
Faults												

Working Set Page Replacement
Implementation

- Keep track of the last τ references
  - The pages referenced during the last τ memory accesses are the working set
  - τ is called the *window size*
- Example: Working set computation, τ = 4 references:

	Time	0	1	2	3	4	5	6	7	8	9	10
	Requests		*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory	Page *a*	• $t = 0$	•	•	•	-	-	-	-	-	*F*	•
	Page *b*	-	-	-	-	*F*	•	•	•	-	-	-
	Page *c*	-	*F*	•	•	•	•	•	•	•	•	•
	Page *d*	• $t = -1$	•	•	•	•	•	•	-	-	-	*F*
	Page *e*	• $t = -2$	•	-	-	-	-	*F*	•	•	•	•
	Faults		•			•		•			•	•

Page-Fault-Frequency Page Replacement
An alternate working set computation

- Explicitly attempt to minimize page faults
  - When page fault frequency is high — *increase working set*
  - When page fault frequency is low — *decrease working set*

Algorithm:

Keep track of the rate at which faults occur

When a fault occurs, compute the time since the last page fault

Record the time, t_{last}, of the last page fault

If the time between page faults is "large" then reduce the working set

If $t_{current} - t_{last} > \tau$, then remove from memory all pages not referenced in $[t_{last}, \ t_{current}]$

If the time between page faults is "small" then increase working set

If $t_{current} - t_{last} \leq \tau$, then add faulting page to the working set

Page-Fault-Frequency Page Replacement

Example, window size = 2

- If $t_{current} - t_{last} > 2$, remove pages not referenced in $[t_{last}, t_{current}]$ from the working set
- If $t_{current} - t_{last} \le 2$, just add faulting page to the working set

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory	Page *a*	•										
	Page *b*	-										
	Page *c*	-										
	Page *d*	•										
	Page *e*	•										
Faults												
$t_{cur} - t_{last}$												

Page-Fault-Frequency Page Replacement

Example, window size = 2

- If $t_{current} - t_{last} > 2$, remove pages not referenced in $[t_{last}, t_{current}]$ from the working set
- If $t_{current} - t_{last} \leq 2$, just add faulting page to the working set

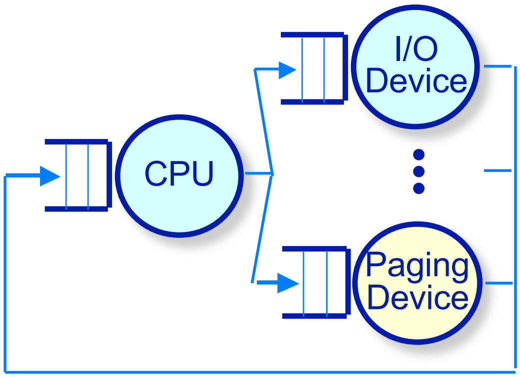
		0	1	2	3	4	5	6	7	8	9	10
Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory	Page *a*	•	•	•	•	-	-	-	-	-	*F*	•
	Page *b*	-	-	-	-	*F*	•	•	•	•	-	-
	Page *c*	-	*F*	•	•	•	•	•	•	•	•	•
	Page *d*	•	•	•	•	•	•	•	•	•	-	*F*
	Page *e*	•	•	•	•	-	-	*F*	•	•	•	•
Faults			•			•		•			•	•
$t_{cur} - t_{last}$			1			3		2			3	1

Load Control
Fundamental tradeoff

- High multiprogramming level
  - $MPL_{max} = \dfrac{\textit{number of page frames}}{\textit{minimum number of frames required for a process to execute}}$
- Low paging overhead
  - MPL_{min} = 1 process
- Issues
  - What criterion should be used to determine when to increase or decrease the *MPL*?
  - Which task should be swapped out if the *MPL* must be reduced?

Load Control

How not to do it: Base load control on CPU utilization



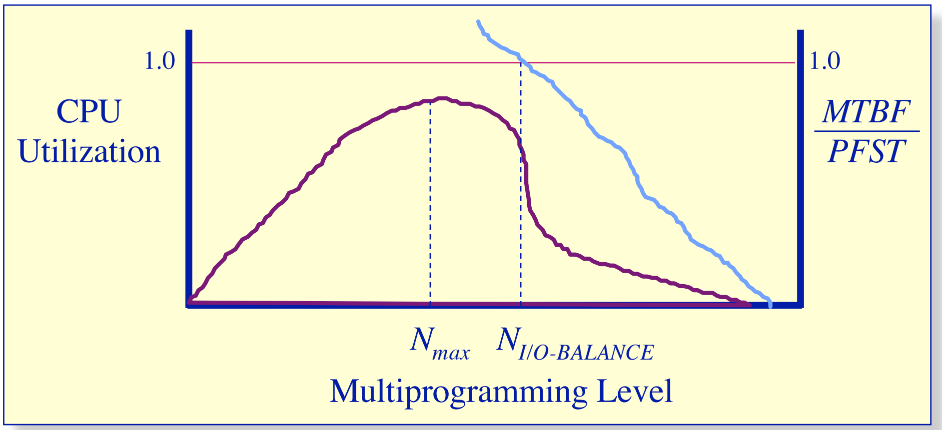
- Assume memory is nearly full
- A chain of page faults occur
  - A queue of processes forms at the paging device
- CPU utilization falls
- Operating system increases *MPL*
  - New processes fault, taking memory away from existing processes
- CPU utilization goes to 0, the OS increases the *MPL* further...

System is *thrashing* — spending all of its time paging

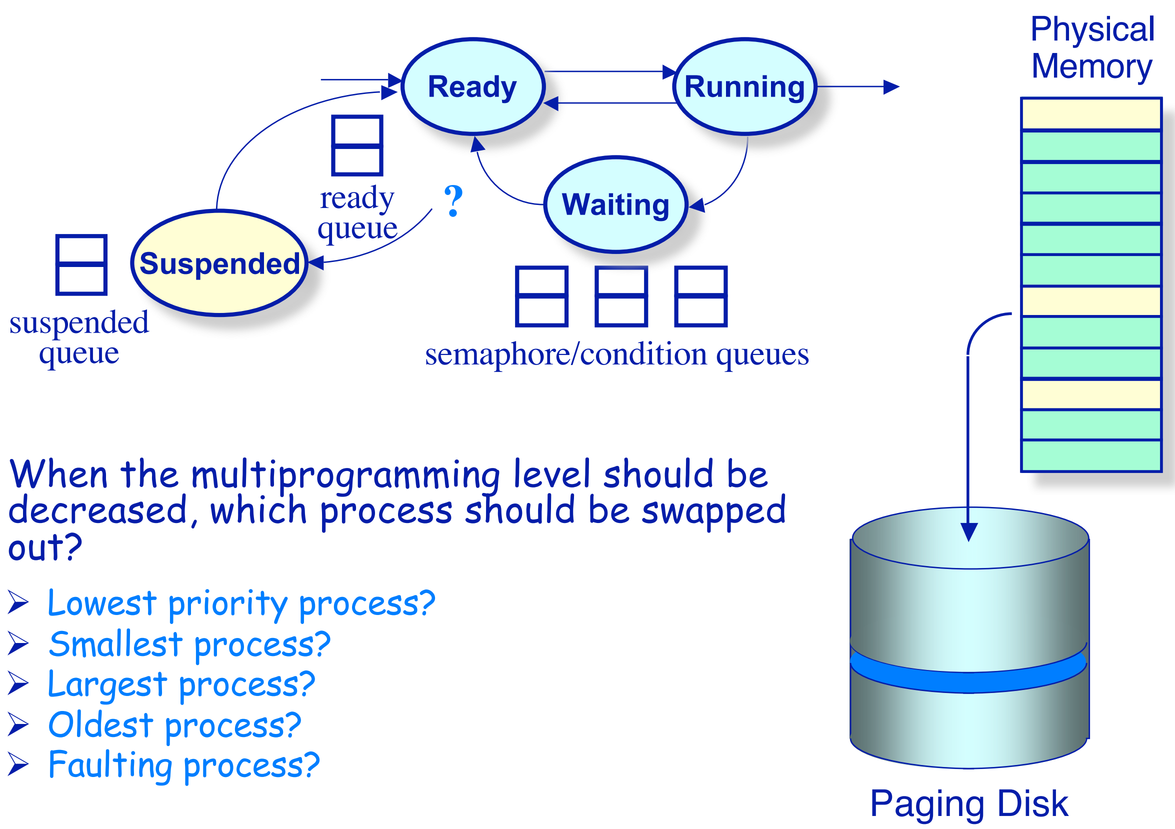
Load Control

Thrashing

- Thrashing can be ameliorated by *local* page replacement
- Better criteria for load control: Adjust MPL so that:
  - *mean time between page faults (MTBF) = page fault service time (PFST)*
  - $\Sigma\ WS_i$ *= size of memory*



Load Control
Thrashing



- When the multiprogramming level should be decreased, which process should be swapped out?
  - Lowest priority process?
  - Smallest process?
  - Largest process?
  - Oldest process?
  - Faulting process?

Virtual Memory: Examples

CSE 231

Instructor: Arani Bhattacharya

Example 1(a)

Suppose we have a page table scheme as follows:

10-bit	8-bit	6-bit	8 bit

- First level takes 10 bits
- Second level takes 8 bits
- Third level takes 6 bits
- Offset takes 8 bits

What is the size of a page, *assuming that the memory is byte-addressable?*

Solution: Because the offset is of size 8 bits, a single page has size 2^8 = 256 bytes

Example 1

Suppose we have a page table scheme as follows:

10-bit	8-bit	6-bit	8 bit

- What is the size of a page table for a process that has 256K of memory starting at address 0, assuming a single entry has a size of 2 bytes?

- 256 KB of memory = 256 KB / 256 byes = 1024 pages
- Number of 2nd level page table entries needed = 1024 / 2^6 = 2^4 = 16
- Thus, we need 1 first level + 16 second level + 1024 third level = 1041 page table entries
- Thus, size of page table = 1041 * 2 = 2042 bytes

Example 2(a)

A computer system has a 36-bit virtual address space with a page size of 8K, and 4 bytes per page table entry.

How many pages are in the virtual address space?

Total memory in virtual address space = 2^36

Total number of pages = 2^36 / 8K = 2^36 / 2^13 = 2^23

Example 2(b)

A computer system has a 36-bit virtual address space with a page size of 8K, and 4 bytes per page table entry.

What is the maximum size of addressable physical memory in this system?

Because there are 4 bytes per page table entry, assuming no extra flags, the physical address is of size 32 bits. So, size of physical address = 2^32 = 4GB

Example 2(c)

A computer system has a 36-bit virtual address space with a page size of 8K, and 4 bytes per page table entry.

If the average process size is 8GB, would you use a one-level, two-level, or three-level page table? Why?

If we take single-level page table, then size of page table will be 2^23 * 4 = 32 MB, which is very high

If we take two-level page table, and assume that the division is 12 | 11 | 13, then the process accesses a total of 2^20 pages. This can be accommodated by 2^(20 - 11) = 2^9 entries in 2nd level page table. Size = (2^12 + 2^9 * 2^11) * 4 = 4 MB

Similarly, three-level page table would also take 4 MB, which is similar in size. So two-level paging is sufficient

Example 2(d)

A computer system has a 36-bit virtual address space with a page size of 8K, and 4 bytes per page table entry.

If the average process size is 8GB, what would be the size of inverted page table?

Since physical address is 4GB = 2^32, number of frames in inverted page table = 2^32 / 2^13 = 2^19

Therefore, a total of 2^19 entries are needed in inverted page table. Assuming the same size of 4 bytes per entry, size of inverted page table = 4 * 2^19 = 2MB

Tradeoffs of multilevel page table

What are the advantages and disadvantages of multilevel page table?

Advantages: Reduces memory requirement

Disadvantages: Too many memory references

How to mitigate this disadvantage?

Use another cache -- Translation Lookaside Buffer (provided by hardware)

If physical address available in cache, then use it straight away. Otherwise, need to use the standard technique. X86-64 machines use this extensively

Tradeoffs of Inverted Page Table

What are the advantages and disadvantages of multilevel page table?

Advantages: Reduces memory requirement

Disadvantages: What if data is NOT found?

> The address has to be translated in software, which can be very very slow

Filesystem: The Basics

CSE 231

Instructor: Arani Bhattacharya

Files are organized in a hierarchy

- What is a file?
    - Logically contiguous space for storing any type of data

- All file systems follow the hierarchical model
- Files are organized hierarchically into folders/directories
- The files themselves have a name
    - Optionally consists of two parts
    - First part is used to denote the actual name used
    - Second part is used to denote the type of file (such as image, slide, etc.)
    - Note that this division of filenames is ONLY a convention on Linux

Windows Filesystem Hierarchy

Windows uses a very simple hierarchy, with a single disk divided into three filesystems by default -- named C:, D:, and E:

Advantage: The hierarchy is easy to understand

Disadvantages:

- Most files tend to go straight to C:, since both system files and user files go there
- User files are kept very deep in the hierarchy, making it cumbersome to find them

Linux Filesystem Hierarchy

Linux uses a much more complex hierarchy

- **/ – The Root Directory**
  - All content is stored in this directory
- **/bin -- Essential User Binaries**
  - Like Firefox, ls, bash
- **/boot -- File needed to boot the system**
  - All GRUB files

Linux Filesystem Hierarchy(2)

- **/dev -- Device files**
  - Linux has a somewhat non-intuitive system of dealing with I/O devices
  - For Linux, all I/O devices are handled as “special files”
  - Example: /dev/sda represents one hard disk, /dev/sr0 represents CD-ROM
  - There are also “pseudo-files” to deal with “virtual devices”, such as /dev/null, /dev/random and so on
- **/etc -- Configuration files**
  - All configuration files are stored here as text files
  - Example: network configuration, sudo permissions of individual users, etc.

Linux Filesystem Hierarchy(3)

- **/home -- Home folders for each user**
  - Personal files of each user
- **/lib -- Libraries installed by users**
  - Libraries (both static and dynamic) needed by different installed programs
- **/media -- Removable Media**
- **/mnt – Temporary Mount Points**

Linux Filesystem Hierarchy(4)

- **/proc – Kernel & Process Files**
  - Shows data about the kernel
  - Example: What is the processor and memory configuration?
  - How much memory is taken by each process?
- **/root – Home directory of root user**
- **/tmp – Temporary Files**
- **/var – Variable Data Files**
  - Used to store data required by different packages
- **/opt – Optional Packages**
  - Used by proprietary software packages

Disks and Partitions

- Disk space is divided into partitions. Each partition consists of its own file system
- A partition decides how data should be organized in the space
- Common file systems used include:
    - Linux: ext4, BtrFS
    - Windows: NTFS
    - CDROM: iso9660
    - USB drive: FAT32
- We will discuss the internal organization in the next class

Mount Points

Since all files under all file systems must come under the overall file system hierarchy, how do we map the content?

Mountpoint: An empty directory where a disk is mounted, i.e. all the contents of the disk can be accessed by entering this directory. Process of mapping the disk to this directory is called “mounting”.

Example: Suppose you insert a CD. How do you access it?

You create an empty directory called (say) mydir, and then mount the disk using mount command (internally uses mount system call).

~> mkdir mydir && mount -t iso9660 /dev/sr0 mydir

How does system keep track of mounted filesystems?

The /etc/fstab keeps track of available disk partitions, mount points and filesystem type

```
suse1:~ # cat /etc/fstab
/dev/sda1            swap                 swap       defaults          0 0
/dev/sda2            /                    ext3       acl,user_xattr    1 1
proc                 /proc                proc       defaults          0 0
sysfs                /sys                 sysfs      noauto            0 0
debugfs              /sys/kernel/debug    debugfs    noauto            0 0
usbfs                /proc/bus/usb        usbfs      noauto            0 0
devpts               /dev/pts             devpts     mode=0620,gid=5   0 0
# /dev/sr0      /cdrom  iso9660 ro,nosuid,nodev,uid=0    0 0
/dev/sdc1            /novi_disk                    ext3       acl,user_xattr,usr
quota,grpquota        2 0
```

How does system keep track of mounted filesystems?

The /etc/mtab keeps track of only mounted file systems

```
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
devpts /dev/pts devpts rw,gid=5,mode=620 0 0
/dev/sda1 /boot ext3 rw 0 0
tmpfs /dev/shm tmpfs rw 0 0
none /proc/sys/fs/binfmt_misc binfmt_misc rw 0 0
sunrpc /var/lib/nfs/rpc_pipefs rpc_pipefs rw 0 0
```

File Attributes

```
struct stat {
    dev_t     st_dev;         /* IDs of device on which file resides */
    ino_t     st_ino;         /* I-node number of file */
    mode_t    st_mode;        /* File type and permissions */
    nlink_t   st_nlink;       /* Number of (hard) links to file */
    uid_t     st_uid;         /* User ID of file owner */
    gid_t     st_gid;         /* Group ID of file owner */
    dev_t     st_rdev;        /* IDs for device special files */
    off_t     st_size;        /* Total file size (bytes) */
    blksize_t st_blksize;     /* Optimal block size for I/O (bytes) */
    blkcnt_t  st_blocks;      /* Number of (512B) blocks allocated */
    time_t    st_atime;       /* Time of last file access */
    time_t    st_mtime;       /* Time of last file modification */
    time_t    st_ctime;       /* Time of last status change */
};
```

File Ownership & Permissions

File has its user id and group id, and permissions for three different operations, each represented by an octal number:

1. Read -- 4
2. Write -- 2
3. Execute -- 1

Also, by looking at user and group, three types of users:

1. User
2. Group
3. Others

The three octal numbers together represent the overall permissions of the file

Do directories also have permissions?

Yes, for directories, the permissions imply the following:

1. Read -- ls and reading files and subdirectories is allowed
2. Write -- creating and/or removing files is possible (but only if execute permission is also set)
3. Execute -- existing files in directory can be accessed

Changing file owner and permission

In both bash shell and C:

1. Changing file owner -- chown command/system call
2. Changing file permission -- chmod command/system call

Finding all the file attributes

1. stat(const char *pathname, struct stat *statbuf)
2. lstat(const char *pathname, struct stat *statbuf)
3. fstat(int fd, struct stat *statbuf)

Directories

- A special file kept in file system
- Directory itself stores the list of files and subdirectories
- File information are stored internally in i-nodes
- The exact organization of i-nodes depends on the filesystem
- Having a file within a directory is simply mentioning the i-node corresponding to a file within the directory file

Soft Links & Hard Links

- Soft link is similar to a shortcut
    - Internally, create a new file (i.e. inode) and point to the original file’s inode
    - ln -s original_file_name link_name
    - Changing the location of the original file makes the soft link invalid

- Hard link is different
    - The inode itself should store two or more distinct locations
    - Changing the location of one location does not affect the hard links

File Locking

- Note that reading and writing to files can lead to race conditions
  - Relatively common

- Different OSes use different techniques
  - Windows -- Locks all files whenever a process opens it for reading or writing
  - Linux -- Provides special system calls called flock and fcntl

    int flock(int fd, int operation);

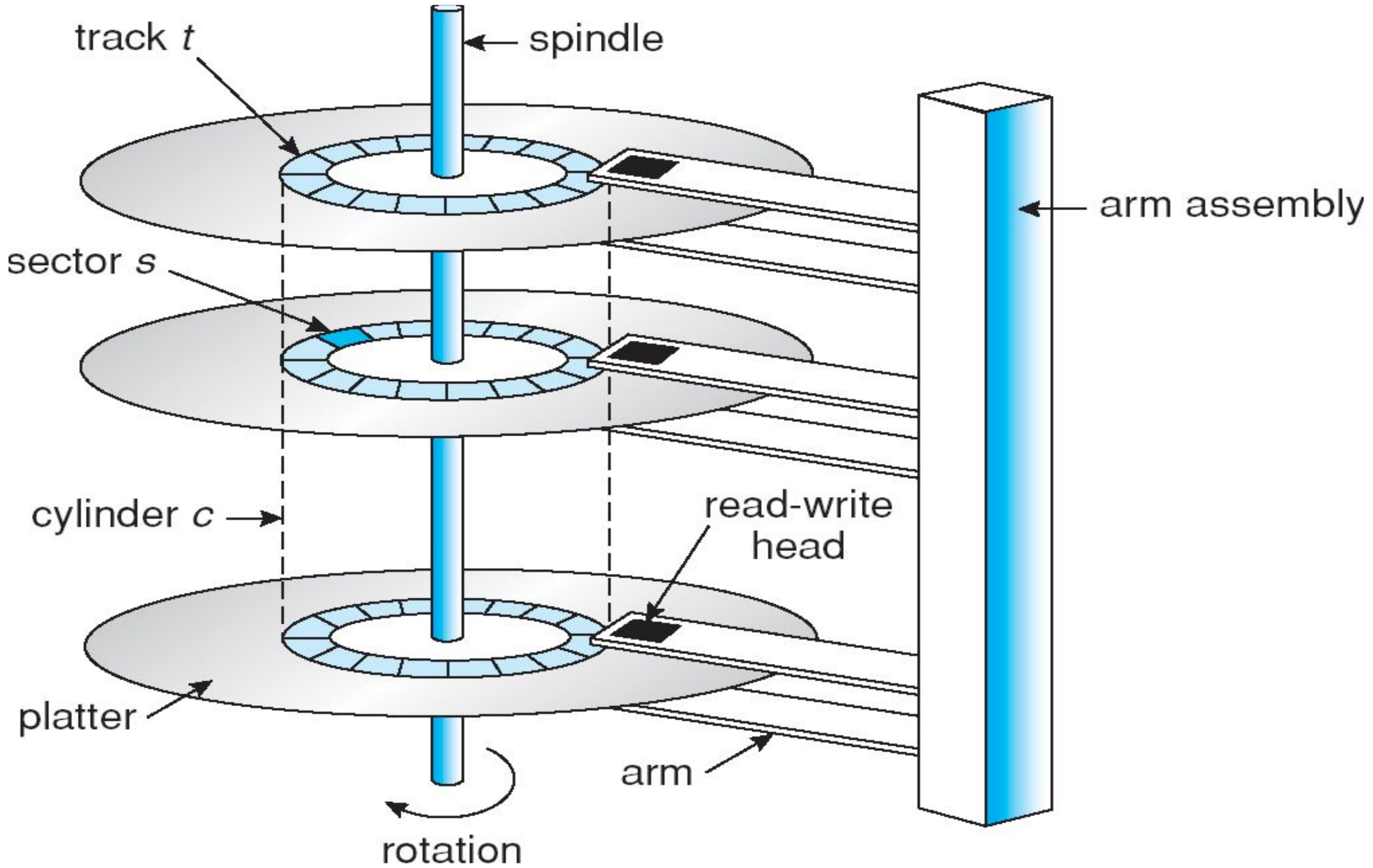
    fcntl(fd, cmd, &flockstr);

Operating Systems

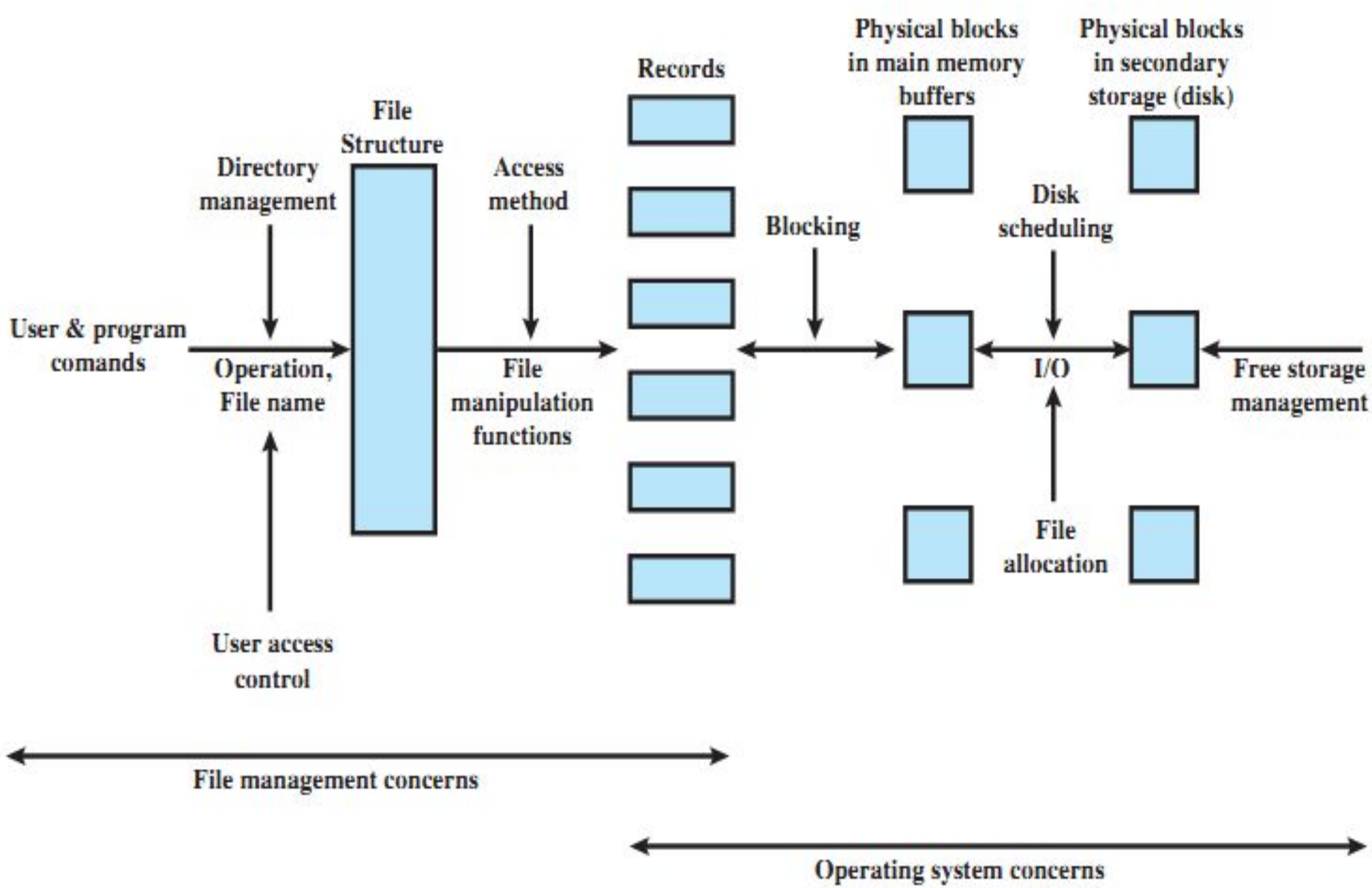
Disk Scheduling

A. Frank - P. Weisberg

Moving-head Disk Mechanism



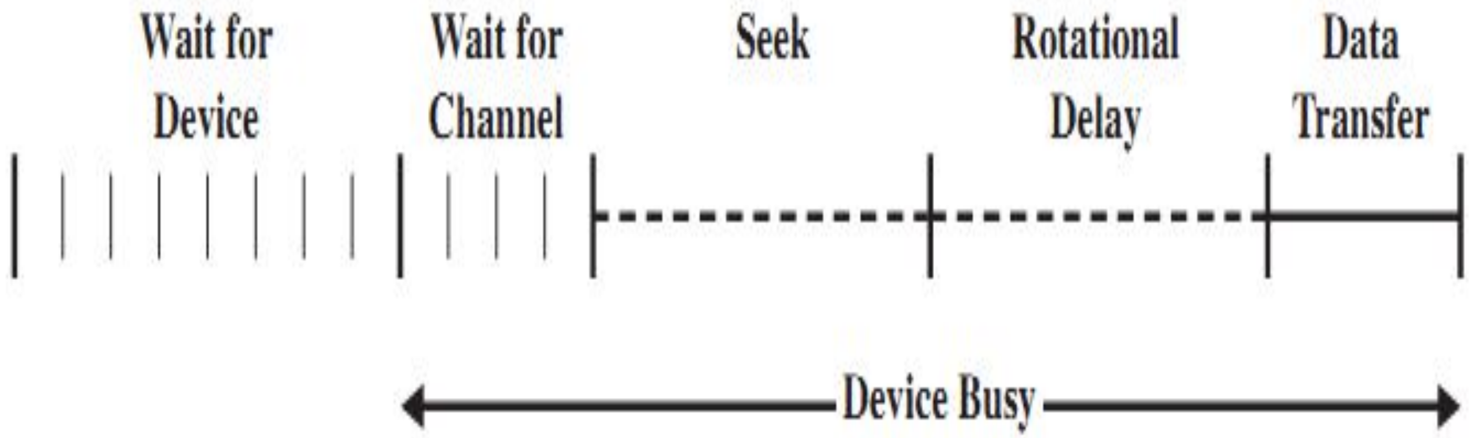
Elements of File Management



(Disk Scheduling (1

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components:
  - Seek time is the time for the disk are to move the heads to the cylinder containing the desired sector.
  - Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.
- Minimize seek time ≈ seek distance.
- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of last transfer.

Components of Disk I/O Transfer



(Disk Scheduling (2

- There are many sources of disk I/O request:
  - OS
  - System processes
  - Users processes
- I/O request includes input/output mode, disk address, memory address, number of sectors to transfer.
- OS maintains queue of requests, per disk or device.
- Idle disk can immediately work on I/O request, busy disk means work must queue:
  - Optimization algorithms only make sense when a queue exists.

Disk Scheduling Algorithms

- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”).
- Several algorithms exist to schedule the servicing of disk I/O requests.
- The analysis is true for one or many platters.
- We illustrate them with a I/O request queue (cylinders are between 0-199):

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

First Come First Serve (FCFS) Example

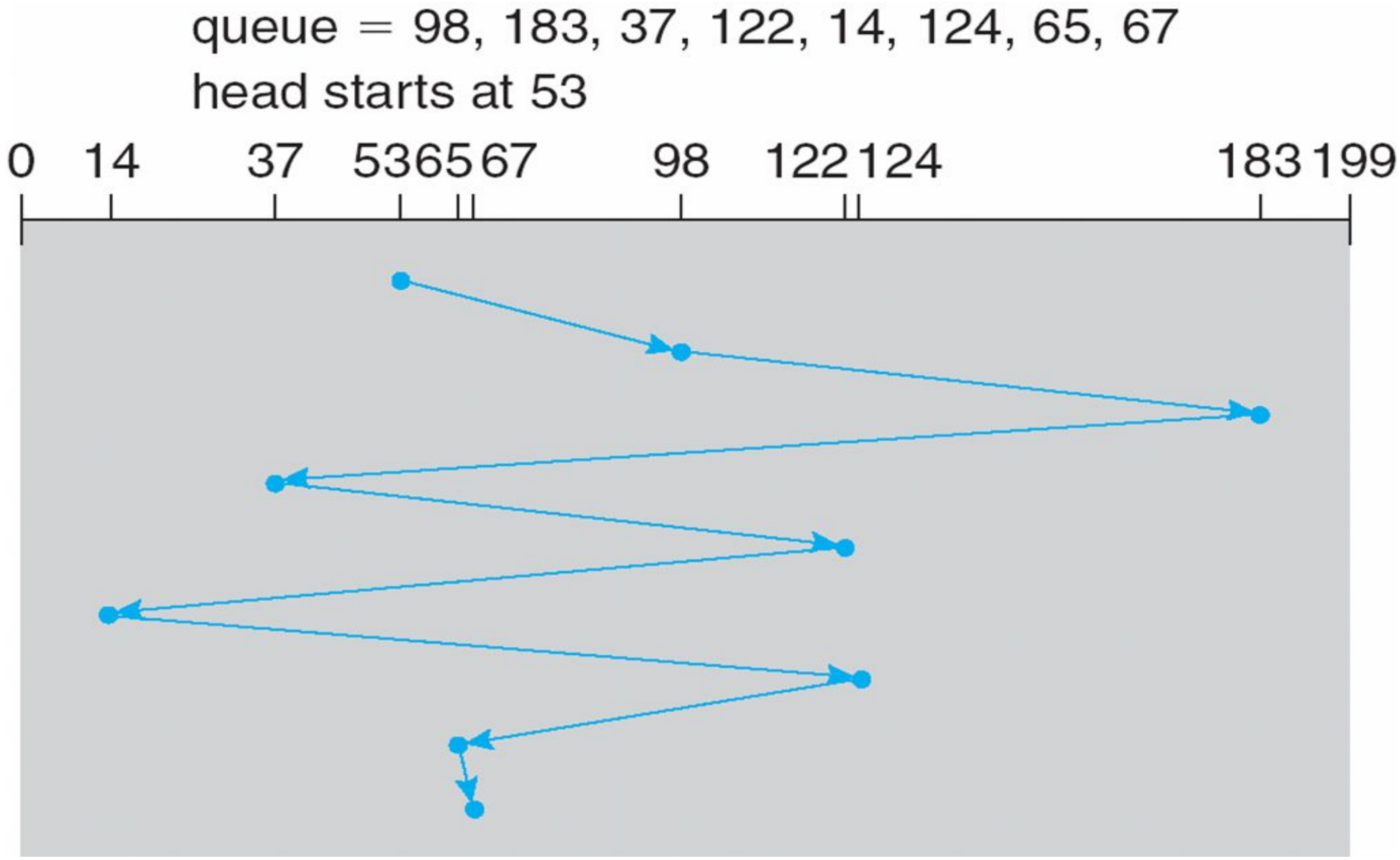


Illustration shows total head movement of 640 cylinders.

(First Come First Serve (FCFS

- Handle I/O requests sequentially.
- Fair to all processes.
- Approaches random scheduling in performance if there are many processes/requests.
- Suffers from global zigzag effect.

Shortest Seek Time First (SSTF) Example

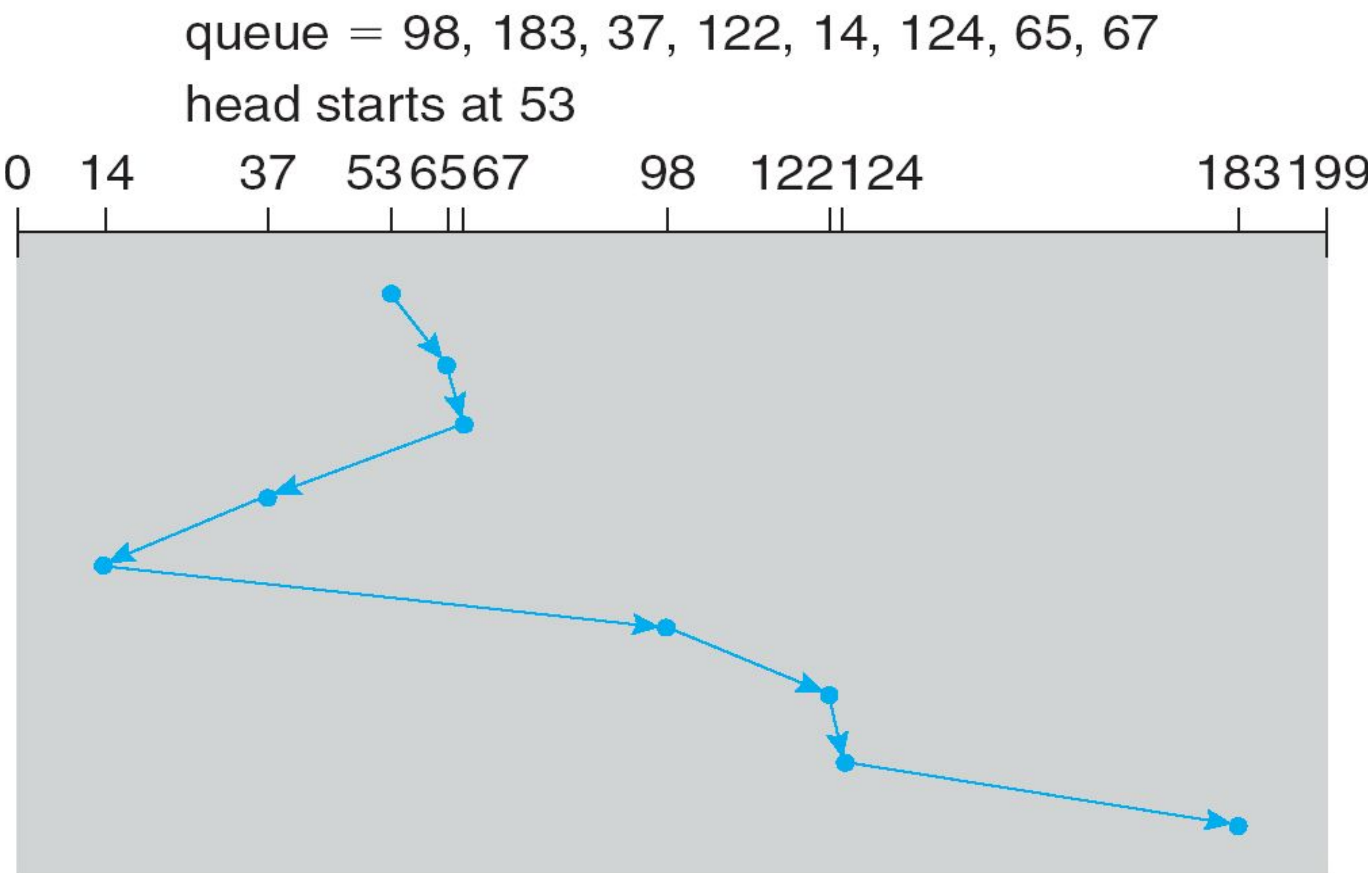


Illustration shows total head movement of 236 cylinders.

(Shortest Seek Time First (SSTF

- Selects the request with the minimum seek time from the current head position.
- Also called Shortest Seek Distance First (SSDF) – It's easier to compute distances.
- It's biased in favor of the middle cylinders requests.
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.

Elevator Algorithms

- Algorithms based on the common elevator principle.
- Four combinations of Elevator algorithms:

.Service in both directions or in only one direction –

.Go until last cylinder or until last I/O request –

Direction / Go until	Go until the last cylinder	Go until the last request
Service both directions	**Scan**	**Look**
Service in only one direction	**C-Scan**	**C-Look**

Scan Example

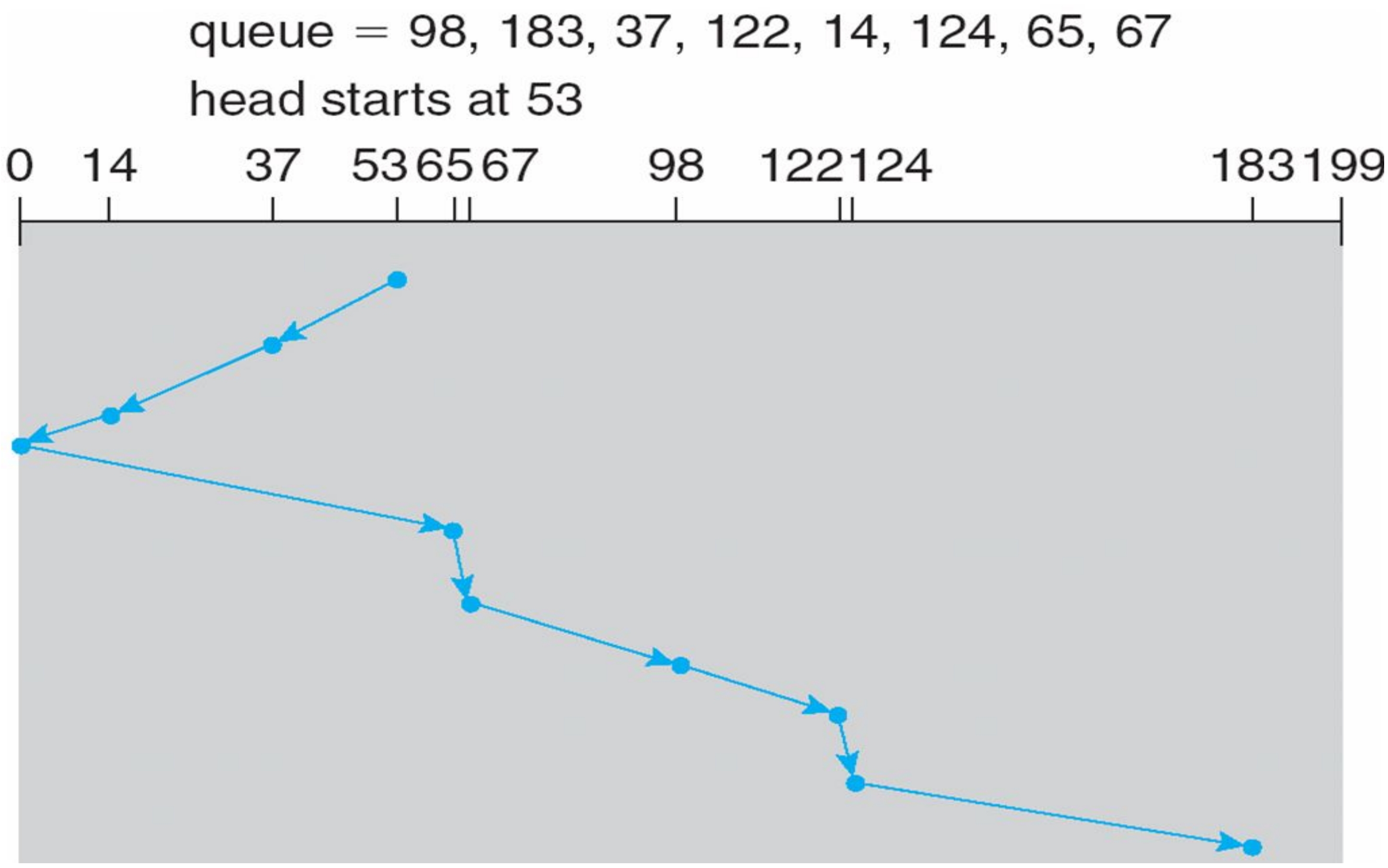


Illustration shows total head movement of 208 cylinders.

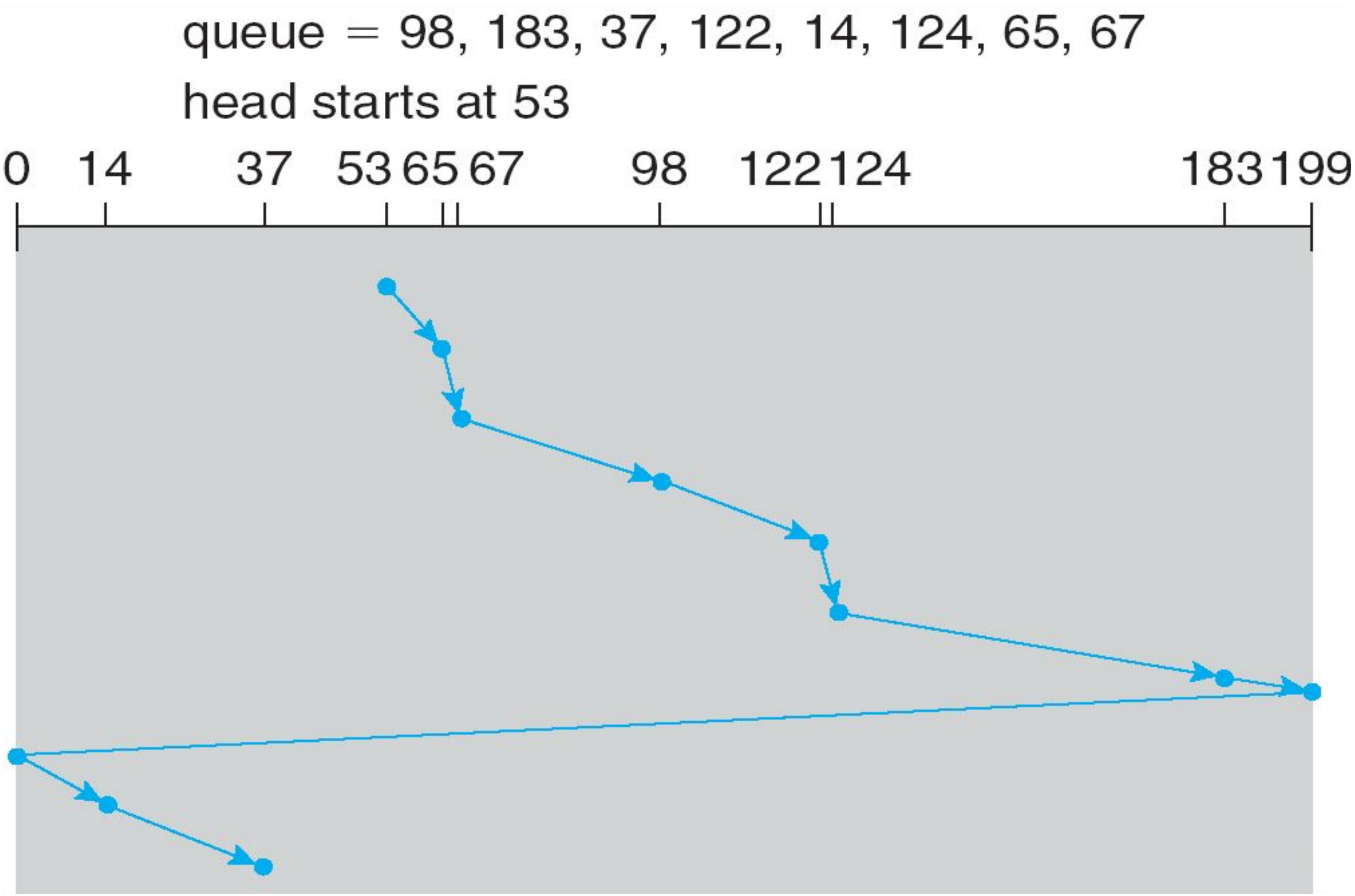
Scan

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- It moves in both directions until both ends.
- Tends to stay more at the ends so more fair to the extreme cylinder requests.

Look

- The disk arm starts at the first I/O request on the disk, and moves toward the last I/O request on the other end, servicing requests until it gets to the other extreme I/O request on the disk, where the head movement is reversed and servicing continues.
- It moves in both directions until both last I/O requests; more inclined to serve the middle cylinder requests.

C-Scan Example



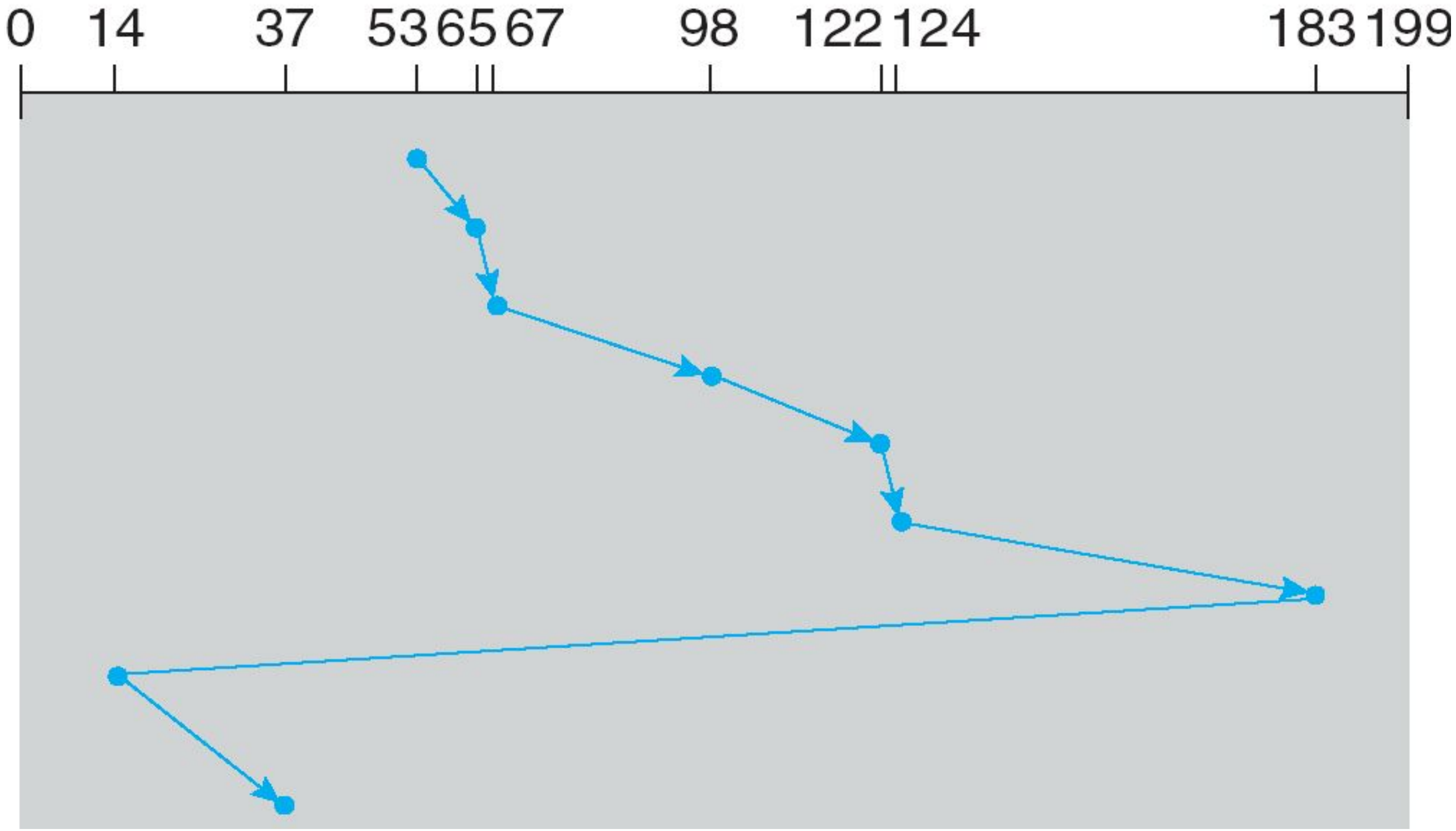
C-Scan

- The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.
- Provides a more uniform wait time than SCAN; it treats all cylinders in the same manner.

C-Look Example

queue 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



C-Look

- Look version of C-Scan.
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.
- In general, Circular versions are more fair but pay with a larger total seek time.
- Scan versions have a larger total seek time than the corresponding Look versions.

Another Example

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) LOOK (starting at track 100, in the direction of increasing track number)		(d) C-LOOK (starting at track 100, in the direction of increasing track number)	
Next track accessed	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	**Average seek length**	27.5	**Average seek length**	27.8	**Average seek length**	35.8

Linux Scheduler

- Merging of successive requests
  - Front merge: if a new request immediately precedes an existing request
  - Back merge: if a new request immediately succeeds an existing request
- Inserting
  - Scheduling

Linux Scheduler

- Merging of successive requests
  - Front merge: if a new request immediately precedes an existing request
  - Back merge: if a new request immediately succeeds an existing request
- Inserting
  - Scheduling
  - Primarily uses elevator algorithms

Linux Scheduler

- Merging of successive requests
  - Front merge: if a new request immediately precedes an existing request
  - Back merge: if a new request immediately succeeds an existing request
- Inserting
  - Scheduling
  - Primarily uses elevator algorithms

(Linus Elevator (Used in v2.4

- Suppose you get a request R
- Try to merge a request
- If merge fails:
  - Is there any request that is older than T?
    - If yes, then add R to the end of queue
    - If no, then insert at a position so that it is in the "right" sequence
    - If right sequence not found, then add R to the end of queue

(Linus Elevator (Used in v2.4

- Advantages
  - Relatively simple
  - Usually provides decent throughput
- Disadvantages
  - Age checking is very random
  - Starvation is possible

(Deadline Scheduler (Used in v2.6

- One feature:
  - Reads are usually more urgent than writes
  - Why?
- Traditional elevator scheduling do not consider this aspect

(Deadline Scheduler (Used in v2.6

- One feature:
  - Reads are usually more urgent than writes
  - Why?
- Traditional elevator scheduling do not consider this aspect

(Deadline Scheduler (Used in v2.6

- All requests are assigned a deadline
  - Read requests: 500 ms
  - Write requests: 5 s
- Maintains three queues
  - For normal sector-wise operation
  - For FIFO read operation
  - For FIFO write operation
- If any FIFO queue's request expires, then handle all the requests of that queue

Disadvantage of Deadline Scheduler

- If there are frequent writes
  - Immediately after handling FIFO write request, can switch to FIFO read queue
  - Can lead to a major fall in throughput
- Anticipatory Scheduler
  - Waits for some few milliseconds after handling a read request
  - If a read/write request comes close to this sector during waiting period, handle it

Disadvantage of Deadline Scheduler

- If there are frequent writes
  - Immediately after handling FIFO write request, can switch to FIFO read queue
  - Can lead to a major fall in throughput
- Anticipatory Scheduler
  - Waits for some few milliseconds after handling a read request
  - If a read/write request comes close to this sector during waiting period, handle it

Completely Fair Scheduling

- In modern desktop systems, usually operations are not very disk intensive
- Maintain one queue for each process
- Within a queue, try to merge and insert operations
- Default for desktop systems

(Selecting a Disk-Scheduling Algorithm (2

- With low load on the disk, It's FCFS anyway.
- SSTF is common and has a natural appeal – good for medium disk load.
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk; Less starvation.
- Performance depends on number and types of requests.
- Requests for disk service can be influenced by the file-allocation method and metadata layout.
- Either SSTF or LOOK (as part of an Elevator package) is a reasonable choice for the default algorithm.

More on Linux Filesystems

CSE 231

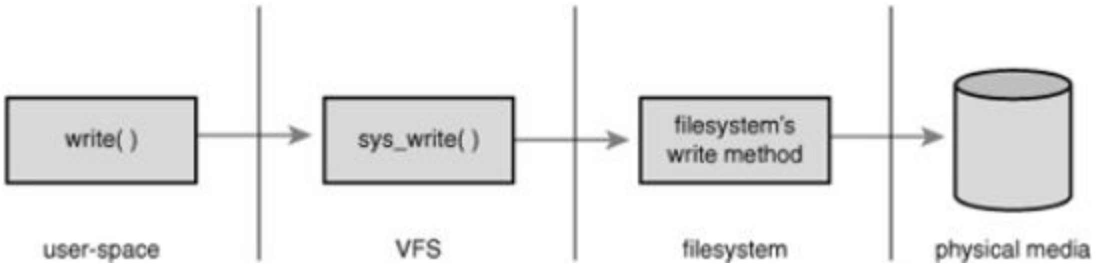
Instructor: Arani Bhattacharya

What is the goal of Linux file management?

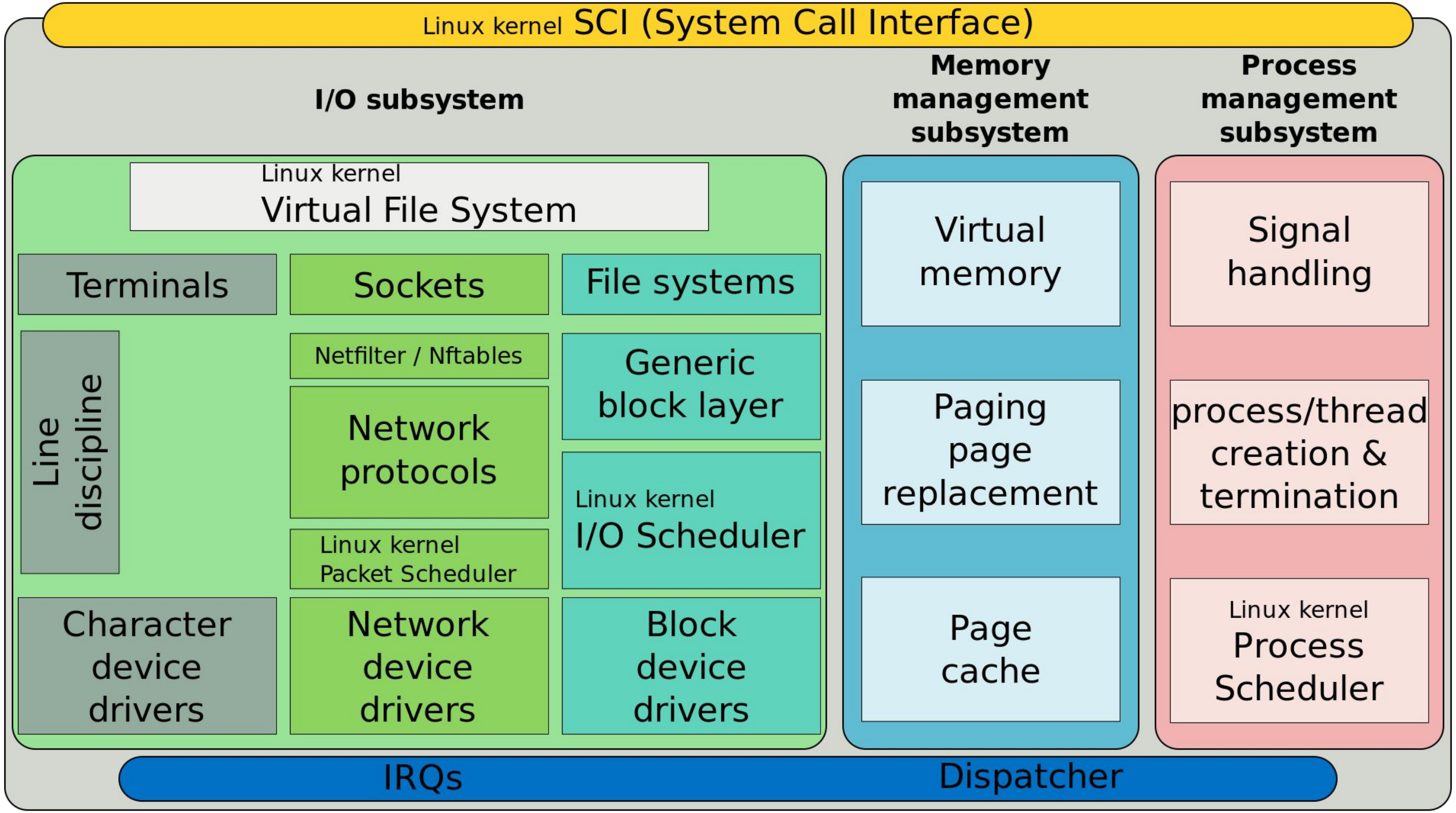
- Support a wide variety of devices
  - Hard disks -- even if Windows is installed on it
  - CD-ROM
  - Thumb drive
  - SD Cards
- Easy movement of data across these devices
- Yet the data itself might be organized physically in different ways
- Each of these filesystems must support reading and writing using exactly the same system calls

Linux uses an abstraction called Virtual File System (VFS)

- VFS provides a common file model
- Standard system calls all write only to the VFS, and not to the actual filesystem



Where does VFS fall in the picture?



Organization of Data in VFS

- VFS provides a common file model

- File Path (represented by dentry object)
  - Consider the file path /home/arani/cse231.ppt
  - There will be **four** dentry objects for this path: /, home, arani, cse231.ppt
- Open file (represented by file object)
- Inode or index node, to store file metadata
- Superblock, to store filesystem metadata

Current state of Linux filesystems

- Most Linux installations use ext4 or btrfs
  - Can support very large files
  - Does not easily lead to file fragmentation (why?)

- FAT/UbiFS for SD-cards/SSDs (why?)
- Very resistant to data loss on power/disk failure (why?)

Filesystem structure (ext2)

The entire disk consists of a set of blocks of length 1KB, 2KB or 4KB. Note that this is different from a disk sector of size 512 bytes. A block group can contain 8192-32768 blocks.

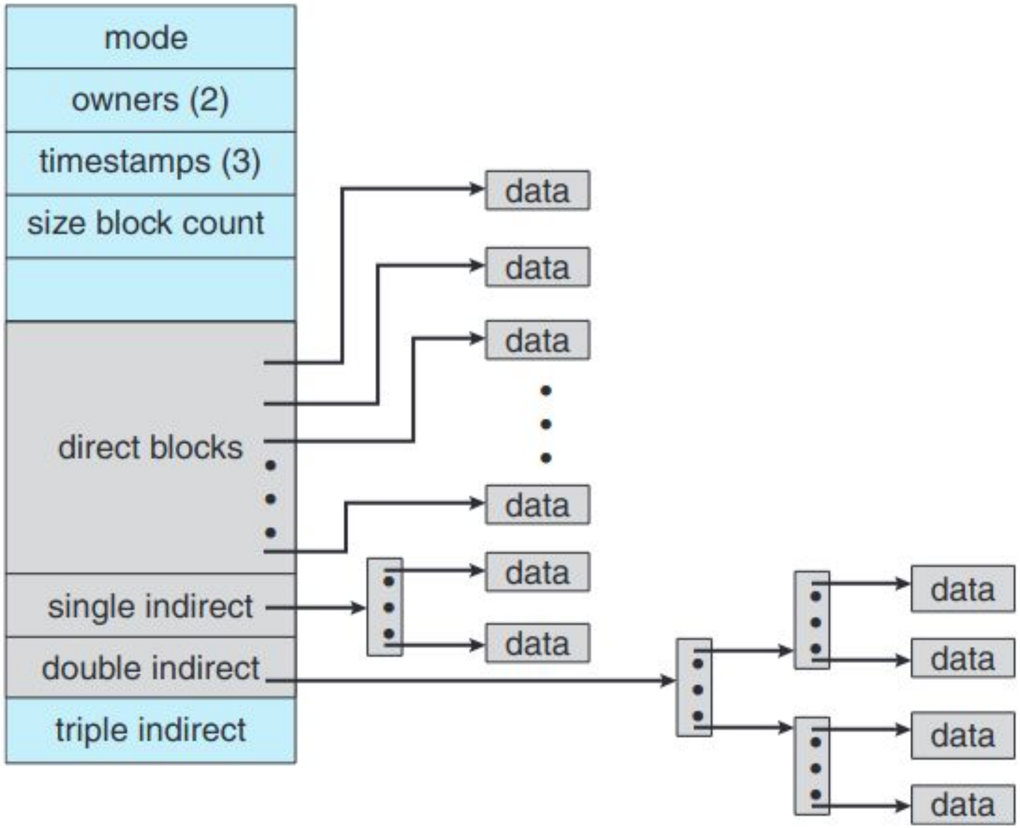
Boot Sector	Block Group 1	Block Group 2	... ...	Block Group N

Each block group contains the following information:

Super Block	FS descriptors	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

Why is so much information repeated in each block group?

Organization of an inode

- 

Some Linux commands related to inodes

- ls -il
  Shows the inode number along with other details of the files

- Showing a file with a specific inode number
  ```
  find /var/ -inum 3634906 -exec ls -l {}
  ```

Advantages of Inode Structure

1. Can handle very large files
2. Very versatile

Disadvantages of Inode Structure

1. No protection from power failure -- entire disk's inode structure can be in inconsistent state
2. Possibility of data fragmentation

Problem 1: Data Fragmentation

1. Hard disks are mechanical devices
2. The disk head needs to actually move to the point where the data is accessed, so that it can be actually read or written
3. Such movement takes time
4. If disk accesses are close to each other, then it becomes much faster
5. But inodes do not provide any such guarantee -- a single file can have data blocks located far from each other

One Possible Solution: Disk Defragmentation

1. Disk fragmentation leads to slower accesses to files

2. **Solution**: Periodically use a tool that reorganizes all the files on the disk. This process is called disk defragmentation

3. Time-consuming, but acceptable in many cases

One Possible Solution: Disk Defragmentation

1. Disk fragmentation leads to slower accesses to files

2. **Solution**: Periodically use a tool that reorganizes all the files on the disk. This process is called disk defragmentation

3. Time-consuming, but acceptable in many cases

Problem 2: Loss of Data on Power Failure

1. Notice that there is a bitmap in the group blocks

2. If power failure happens while writing to a file, the inode bitmap and inode table may be in inconsistent state

3. The entire integrity of the file system depends on the bitmap and inode table, so it is possible to lose data present on the entire disk
4. Need to fix using fsck, which can take a very long time

Solution to Data Loss: Journaled File Systems

Keep a log of what you are about to do. Effectively there are three steps in writing:

1. Mention in the log (“journal”) about where you are going to write. **Do not touch the inode table at this point.**

2. Do the actual writing on the data blocks

3. Update the log specifying that the writing has been completed

Integrating this together: ext4

1. The ext4 filesystem implements all these features, and so is considered to be a much better filesystem than ext2 and ext3
2. Supports file size of up to 16 TB
3. Supports disks of upto 1000 TB
4. Can deal with a total of 30 billion files
5. Supports file timestamp of nanosecond range
6. Not suited for SD cards/SSDs because of journal

File Superblock

```
struct super_block {
        struct list_head          s_list;            /* list of all superblocks */
        dev_t                     s_dev;             /* identifier */
        unsigned long             s_blocksize;       /* block size in bytes */
        unsigned char             s_blocksize_bits;  /* block size in bits */
        unsigned char             s_dirt;            /* dirty flag */
        unsigned long long        s_maxbytes;        /* max file size */
        struct file_system_type   s_type;            /* filesystem type */
        struct super_operations   s_op;              /* superblock methods */
        struct dquot_operations   *dq_op;            /* quota methods */
        struct quotactl_ops       *s_qcop;           /* quota control methods */
        struct export_operations  *s_export_op;      /* export methods */
        unsigned long              s_flags;          /* mount flags */
        unsigned long              s_magic;          /* filesystem's magic number */
        struct dentry              *s_root;          /* directory mount point */
        struct rw_semaphore        s_umount;         /* unmount semaphore */
        struct semaphore           s_lock;           /* superblock semaphore */
        int                        s_count;          /* superblock ref count */
        int                        s_need_sync;      /* not-yet-synced flag */
        atomic_t                   s_active;         /* active reference count */
        void                       *s_security;      /* security module */
        struct xattr_handler       **s_xattr;        /* extended attribute handlers */
```

File Superblock

```
        struct list_head      s_inodes;        /* list of inodes */
        struct list_head      s_dirty;         /* list of dirty inodes */
        struct list_head      s_io;            /* list of writebacks */
        struct list_head      s_more_io;       /* list of more writeback */
        struct hlist_head     s_anon;          /* anonymous dentries */
        struct list_head      s_files;         /* list of assigned files */
        struct list_head      s_dentry_lru;    /* list of unused dentries */
        int                   s_nr_dentry_unused; /* number of dentries on list */
        struct block_device   *s_bdev;         /* associated block device */
        struct mtd_info       *s_mtd;          /* memory disk information */
        struct list_head      s_instances;     /* instances of this fs */
        struct quota_info     s_dquot;         /* quota-specific options */
        int                   s_frozen;        /* frozen status */
        wait_queue_head_t     s_wait_unfrozen; /* wait queue on freeze */
        char                  s_id[32];        /* text name */
        void                  *s_fs_info;      /* filesystem-specific info */
        fmode_t               s_mode;          /* mount permissions */
        struct semaphore      s_vfs_rename_sem; /* rename semaphore */
        u32                   s_time_gran;     /* granularity of timestamps */
        char                  *s_subtype;      /* subtype name */
        char                  *s_options;      /* saved mount options */
};
```

Superblock Operations

```
struct super_operations {
  struct inode *(*alloc_inode)(struct super_block *sb);
  void (*destroy_inode)(struct inode *);
  void (*dirty_inode) (struct inode *);
  int (*write_inode) (struct inode *, int);
  void (*drop_inode) (struct inode *);
  void (*delete_inode) (struct inode *);
  void (*put_super) (struct super_block *);
  void (*write_super) (struct super_block *);
  int (*sync_fs)(struct super_block *sb, int wait);
  int (*freeze_fs) (struct super_block *);
  int (*unfreeze_fs) (struct super_block *);
  int (*statfs) (struct dentry *, struct kstatfs *);
  int (*remount_fs) (struct super_block *, int *, char *);
  void (*clear_inode) (struct inode *);
  void (*umount_begin) (struct super_block *);
  int (*show_options)(struct seq_file *, struct vfsmount *);
  int (*show_stats)(struct seq_file *, struct vfsmount *);
  ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
  ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
  int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
};
```

Inode Structure

```
struct inode {
        struct hlist_node       i_hash;                 /* hash list */
        struct list_head        i_list;                 /* list of inodes */
        struct list_head        i_sb_list;              /* list of superblocks */
        struct list_head        i_dentry;               /* list of dentries */
        unsigned long           i_ino;                  /* inode number */
        atomic_t                i_count;                /* reference counter */
        unsigned int            i_nlink;                /* number of hard links */
        uid_t                   i_uid;                  /* user id of owner */
        gid_t                   i_gid;                  /* group id of owner */
        kdev_t                  i_rdev;                 /* real device node */
        u64                     i_version;              /* versioning number */
        loff_t                  i_size;                 /* file size in bytes */
        seqcount_t              i_size_seqcount;        /* serializer for i_size */
        struct timespec         i_atime;                /* last access time */
        struct timespec         i_mtime;                /* last modify time */
        struct timespec         i_ctime;                /* last change time */
        unsigned int            i_blkbits;              /* block size in bits */
        blkcnt_t                i_blocks;               /* file size in blocks */
        unsigned short          i_bytes;                /* bytes consumed */
        umode_t                 i_mode;                 /* access permissions */
```

Inode Structure

```
        spinlock_t              i_lock;             /* spinlock */
        struct rw_semaphore     i_alloc_sem;        /* nests inside of i_sem */
        struct semaphore        i_sem;              /* inode semaphore */
        struct inode_operations *i_op;              /* inode ops table */
        struct file_operations  *i_fop;             /* default inode ops */
        struct super_block      *i_sb;              /* associated superblock */
        struct file_lock        *i_flock;           /* file lock list */
        struct address_space    *i_mapping;         /* associated mapping */
        struct address_space    i_data;             /* mapping for device */
        struct dquot            *i_dquot[MAXQUOTAS]; /* disk quotas for inode */
        struct list_head        i_devices;          /* list of block devices */
        union {
            struct pipe_inode_info  *i_pipe;        /* pipe information */
            struct block_device     *i_bdev;        /* block device driver */
            struct cdev             *i_cdev;        /* character device driver */
        };
        unsigned long           i_dnotify_mask;     /* directory notify mask */
        struct dnotify_struct   *i_dnotify;         /* dnotify */
        struct list_head        inotify_watches;    /* inotify watches */
        struct mutex            inotify_mutex;      /* protects inotify_watches */
        unsigned long           i_state;            /* state flags */
        unsigned long           dirtied_when;       /* first dirtying time */
        unsigned int            i_flags;            /* filesystem flags */
        atomic_t                i_writecount;       /* count of writers */
        void                    *i_security;        /* security module */
        void                    *i_private;         /* fs private pointer */
};
```

Inode Operations

```
struct inode_operations {
    int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
    struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct inode *,struct dentry *,const char *);
    int (*mkdir) (struct inode *,struct dentry *,int);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,int,dev_t);
    int (*rename) (struct inode *, struct dentry *,
                   struct inode *, struct dentry *);
    int (*readlink) (struct dentry *, char __user *,int);
    void * (*follow_link) (struct dentry *, struct nameidata *);
    void (*put_link) (struct dentry *, struct nameidata *, void *);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int);
    int (*setattr) (struct dentry *, struct iattr *);
    int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
    int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
    ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int (*removexattr) (struct dentry *, const char *);
    void (*truncate_range)(struct inode *, loff_t, loff_t);
    long (*fallocate)(struct inode *inode, int mode, loff_t offset,
                      loff_t len);
    int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
                  u64 len);
};
```

File structure

```
struct file {
        union {
            struct list_head   fu_list;       /* list of file objects */
            struct rcu_head    fu_rcuhead;    /* RCU list after freeing */
        } f_u;
        struct path            f_path;        /* contains the dentry */
        struct file_operations *f_op;         /* file operations table */
        spinlock_t             f_lock;        /* per-file struct lock */
        atomic_t               f_count;       /* file object's usage count */
        unsigned int           f_flags;       /* flags specified on open */
        mode_t                 f_mode;        /* file access mode */
        loff_t                 f_pos;         /* file offset (file pointer) */
        struct fown_struct     f_owner;       /* owner data for signals */
        const struct cred      *f_cred;       /* file credentials */
        struct file_ra_state   f_ra;          /* read-ahead state */
        u64                    f_version;     /* version number */
        void                   *f_security;   /* security module */
        void                   *private_data; /* tty driver hook */
        struct list_head       f_ep_links;    /* list of epoll links */
        spinlock_t             f_ep_lock;     /* epoll lock */
        struct address_space   *f_mapping;    /* page cache mapping */
        unsigned long          f_mnt_write_state; /* debugging state */
};
```

File Operations

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                             unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                              unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int,
                      unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *,
                             int, size_t, loff_t *, int);
```