University of Pittsburgh

Introduction to Operating Systems
CS 1550

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Some slides are from **Silberschatz, Galvin and Gagne ©2013)**

Announcements

- Upcoming deadlines
  - Homework 10 due this Friday
  - Project 3 is due this Friday at 11:59 pm
  - Lab 4 is due on Tuesday 4/11 at 11:59 pm

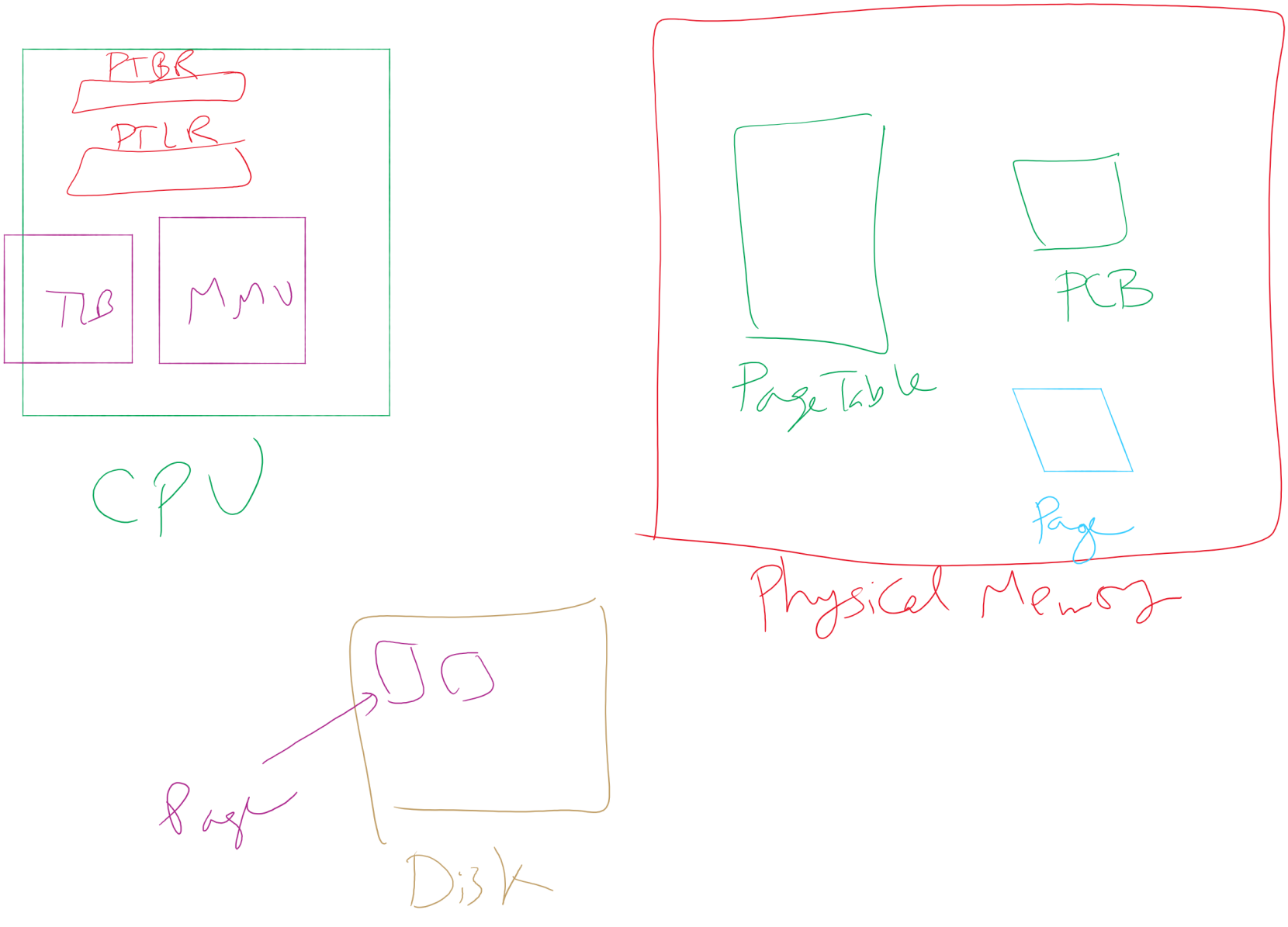
Previous lecture …

- Problem of Too Large Page Tables
  - multi-level page tables + TLB
  - inverted page tables

This lecture …

- Miscellaneous Memory Management Issues
- How to allocate disk blocks to files and directories?
  - contiguous, linked, FAT, indexed, and hybrid

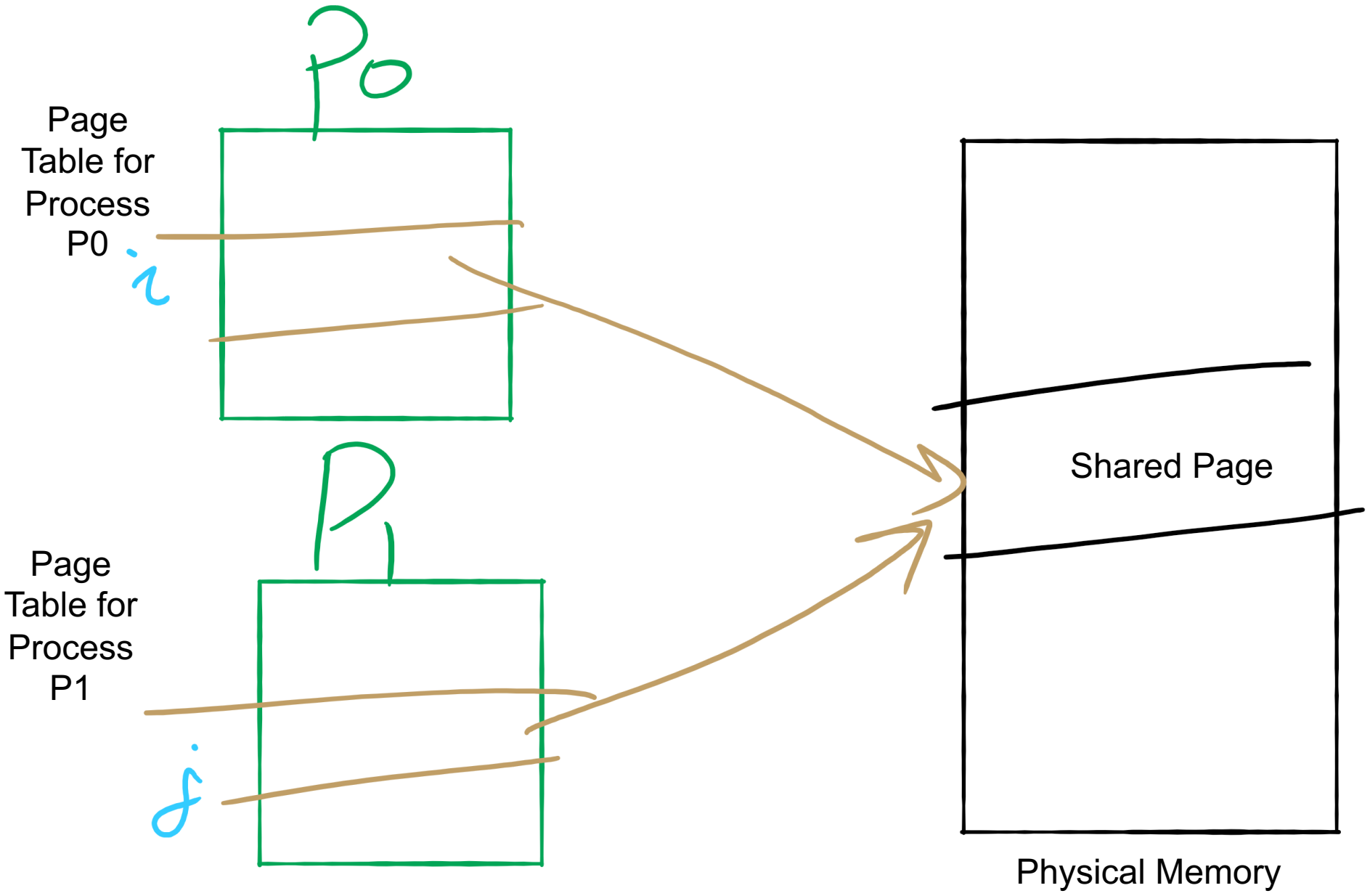
MMU, TLB, Page Tables, etc.



How big should a page be?

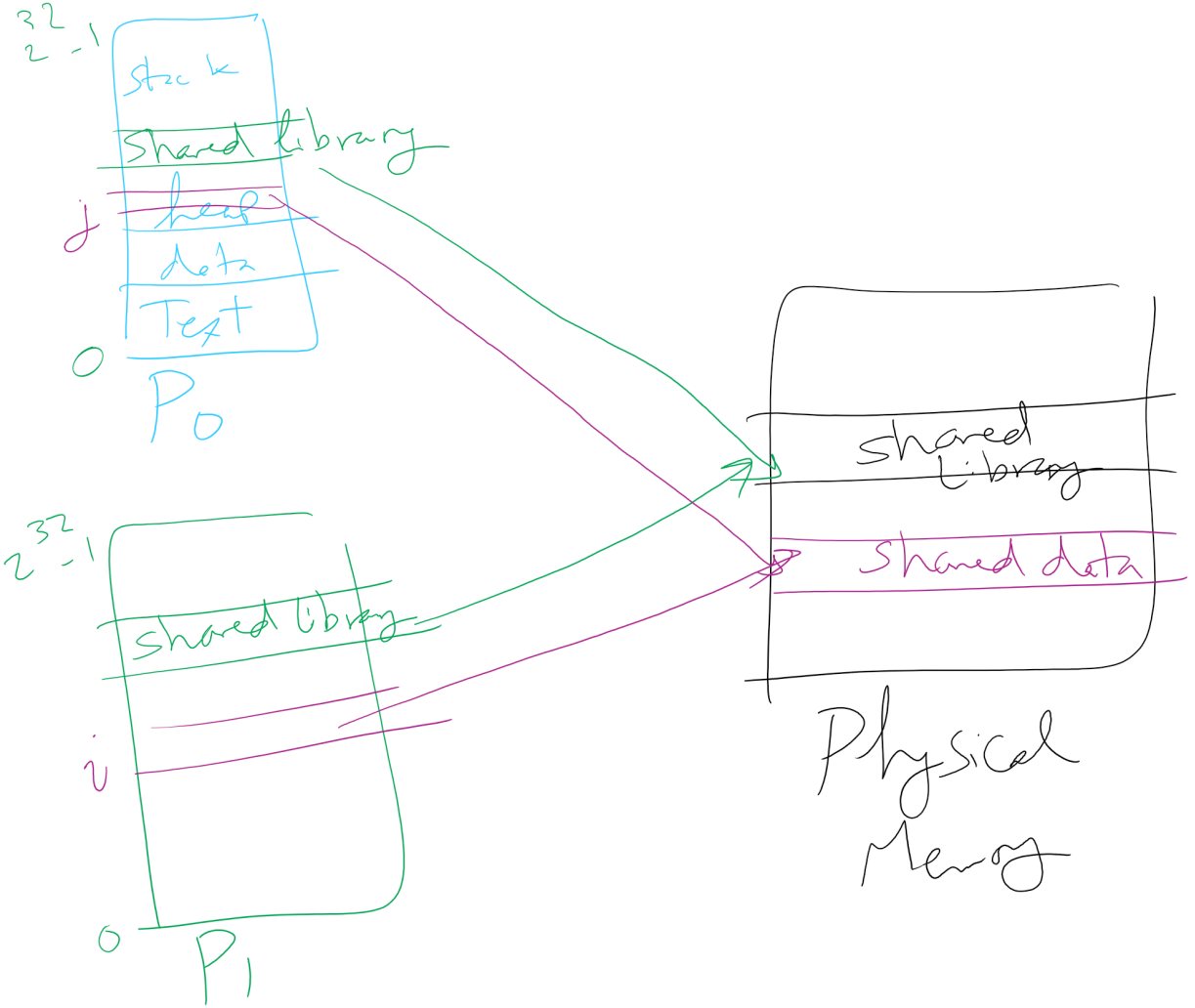
- Smaller pages have advantages
  - **Better fit** for various data structures, code sections
  - **Less unused** physical memory (some pages have few useful bytes and the rest isn't needed currently)
  - Less **internal fragmentation**
  - **Faster** disk write and read
- Larger pages also have advantages
  - Less overhead (**smaller** page tables)
  - **Lower** TLB miss rate
    - TLB covers more memory with same number of entries
  - **Faster** page replacement algorithms (fewer entries to look through)
  - Higher **throughput** of disk transfers

Sharing Pages Between Processes



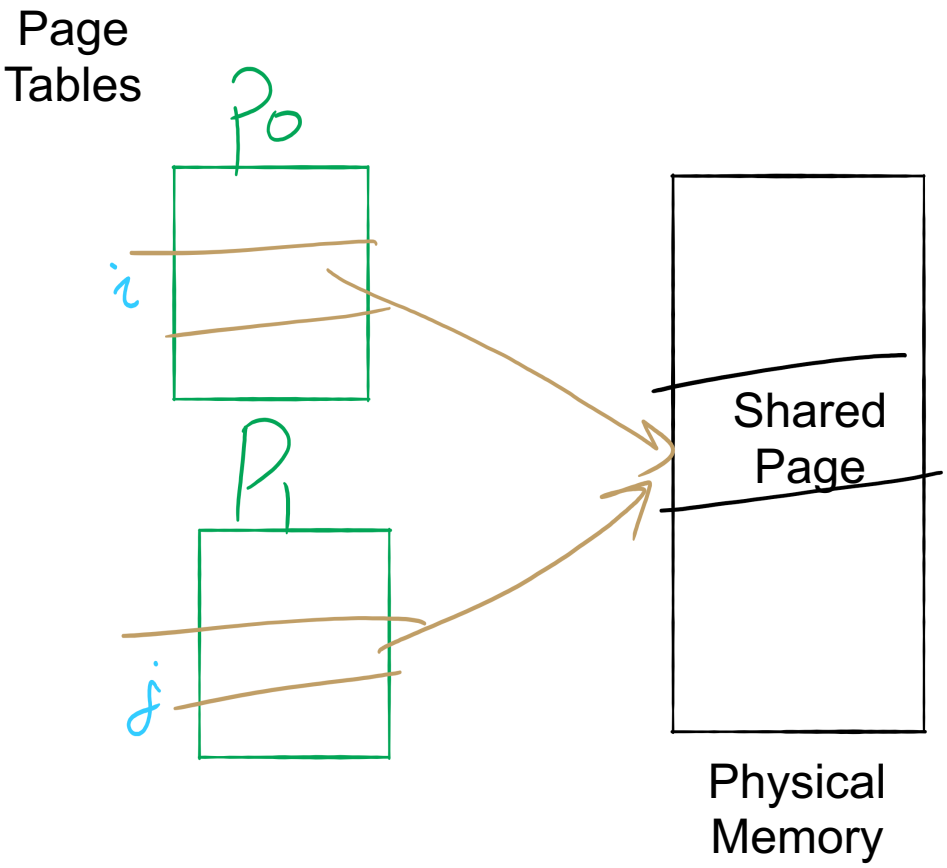
Page Sharing: Why?

- Code sharing (e.g., C standard library)
- Data sharing (need for process synchronization)



Sharing pages: How?

- Processes share pages
  - Entries in page tables point to the same physical frame
  - Easier to share code: no problems with modification
- Virtual addresses for shared pages (i == j?)
  - Same: easier to exchange pointers, keep data structures consistent
    - difficult if many processes share with each other
  - Different: easier to actually implement



When does OS manage memory?

Four times when OS involved with **paging**

- Process creation
  - Determine code size
  - Create **page table**
- During process execution
  - Reset the **MMU** for new process
  - **Flush/Reload** the TLB (if no Address Space IDs)
- Page fault 
  - Determine virtual address causing fault (**faulting address**)
  - **Swap** target page out, needed page in
- Process termination
  - **Release** page table
  - Return pages to the **free** pool

How is a page fault handled? (partially in Lab 4)

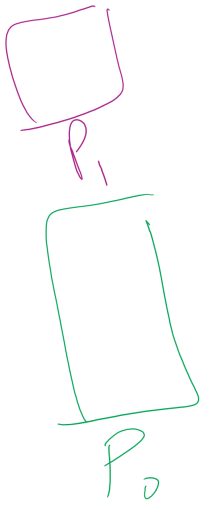
- MMU raises **page fault**
  - General registers **saved**
  - OS determines virtual page needed (**faulting address** in a special register
  - OS checks **validity** of address (Process killed if address illegal)
  - OS empties a frame to put new page
    - If **victim** page dirty, OS requests **write back** to disk
    - OS updates **page table**
  - OS requests new page from disk (if on disk)
    - If not on disk, wipe out frame for security
  - Faulting Process blocked and process state **saved**
  - Registers restored

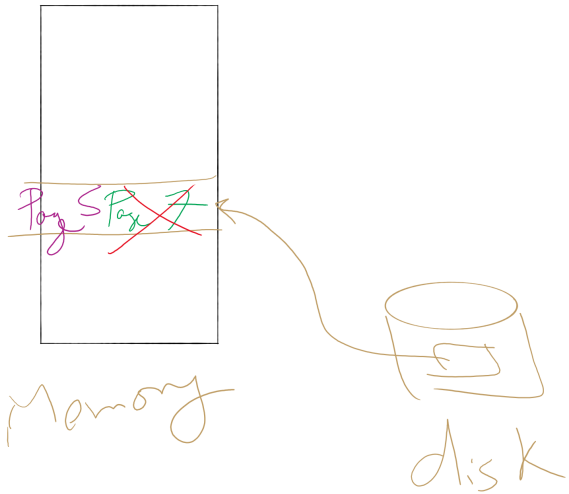
How is a page fault handled? (partially in Lab 4)

- When **frame ready**
  - OS updates **page table**
  - Faulting instruction **backed up** so it can be restarted
- When faulting process **scheduled**
  - Process state **restored**
  - Process **resumes**

Locking pages in memory

- P0 issues call for read from device into P0's I/O buffer (Page 7 of P0)
  - While P0 I/O waiting, P1 runs
  - P1 has a page fault (Page 5)
  - P0's I/O buffer chosen to be paged out
    - I/O device writes to P1's Page 5 instead of P0's Page 7!
- Solution: allow pages to be *locked* into physical memory
  - immune from being replaced
  - stay locked for (relatively) short periods





When are dirty pages written to disk?

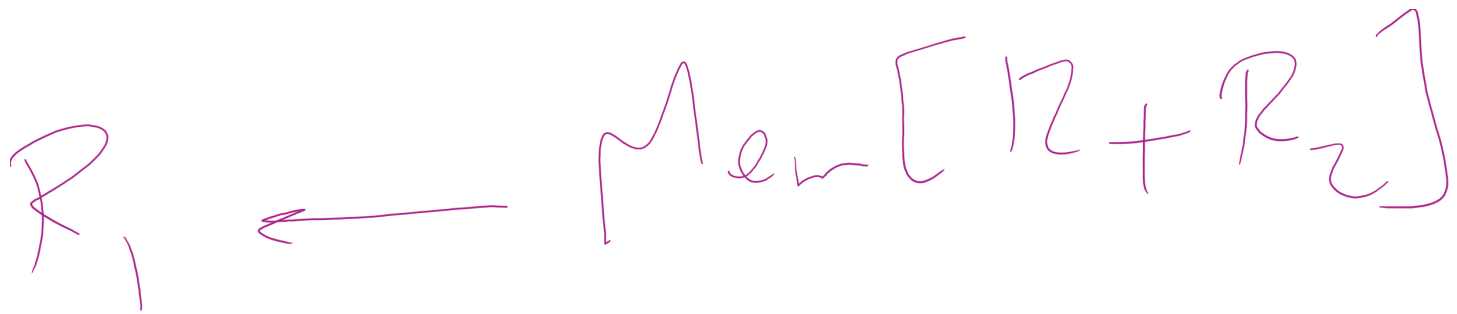
- **On demand** (when they're replaced)
  - **Fewest** writes to disk
  - **Slower** page replacement: takes twice as long (must wait for disk write ***and*** disk read)
- **Periodically** (in the background)
  - Background process (called **Cleaner**) scans through page tables, writes out dirty pages that are pretty old
- Cleaner also keeps a list of pages **ready for replacement**
  - Page faults handled **faster**: no need to find space at page fault time
  - Cleaner may use algorithms discussed earlier (Aging, clock, etc.)

Control overall page fault rate

- Despite good designs, system may still **thrash**
- Solution: **Reallocate** memory frames
- What if most (or all) processes have high page fault rate
  - Some processes need more memory, …
  - but no processes could give up some memory
- No way to reduce page fault rate by reallocation
- Solution: Reduce number of processes in memory
  - **Swap** one or more processed to disk
  - Divide up pages they held
  - Reconsider **degree of multiprogramming**

Backing up an instruction

- Problem: Not always possible to **resume** an instruction if page fault in the **middle of** instruction execution
  - Part of the instruction may have executed
  - Some changes may have already happened
- Solution:
  - **undo** all of the changes made by the instruction
  - **Restart** instruction from the beginning
  - This is easier on some architectures than others
- Example: LW R1, 12(R2)
  - Page fault in fetching instruction: nothing to undo
  - Page fault in getting value at 12(R2): restart instruction



Minimum memory allocation per process

- Enough for every single instruction to eventually finish without page faults
- Example: ADD (Rd)+,(Rs1)+,(Rs2)+

Mem[Rd] ← Mem[Rs1] + Mem[Rs2]

Mem[Rs1]++

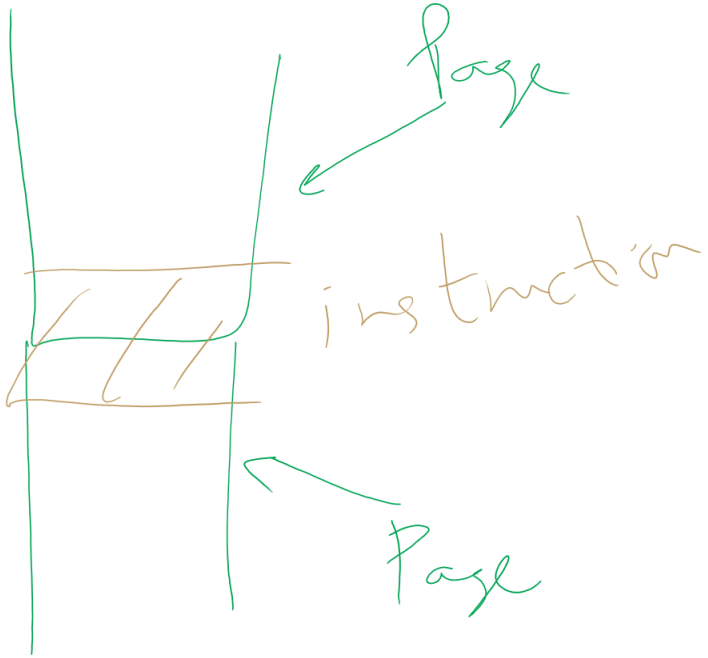
Mem[Rs2]++

Mem[Rd]++

2 pages for instruction

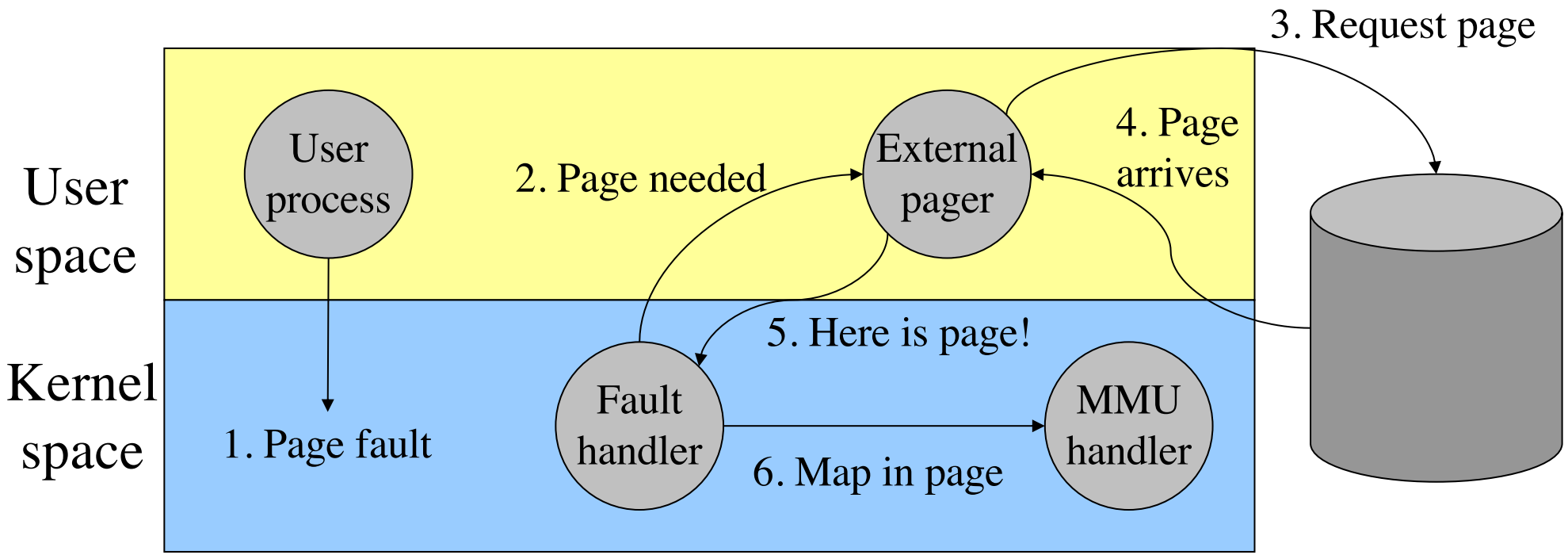
2 * 3 pages for 3 operands

8 pages



Separating policy and mechanism

- Mechanism for page replacement has to be in kernel
  - Reading and writing page table entries
- Policy for deciding which pages to replace or where to store evicted pages could be in user space
  - More flexibility but higher overhed

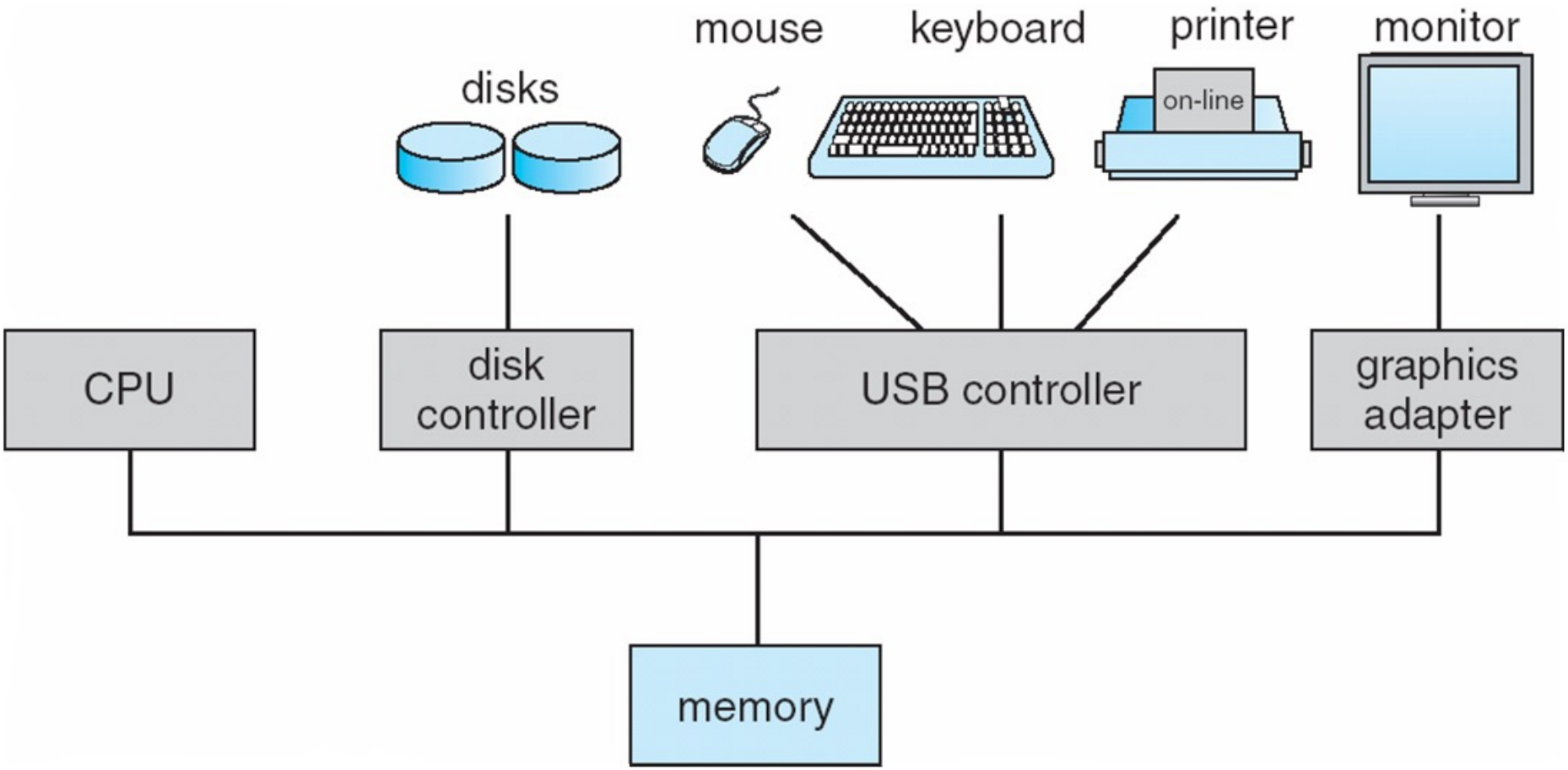


Enter File System!

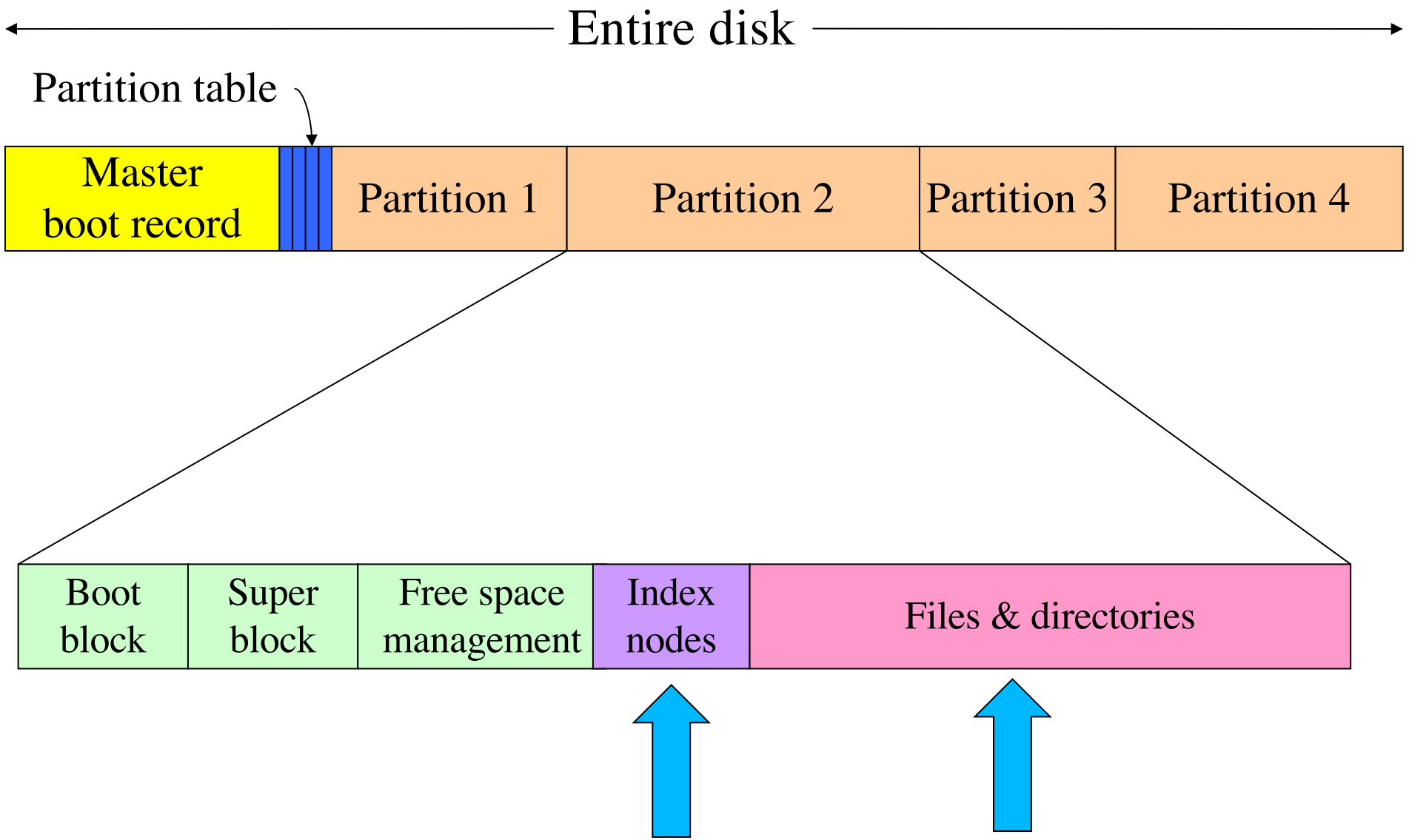
How to allocate disk blocks to **files** and **directories**?

How complex is the OS's job?

Let's look at one of the resources managed by the OS: I/O devices

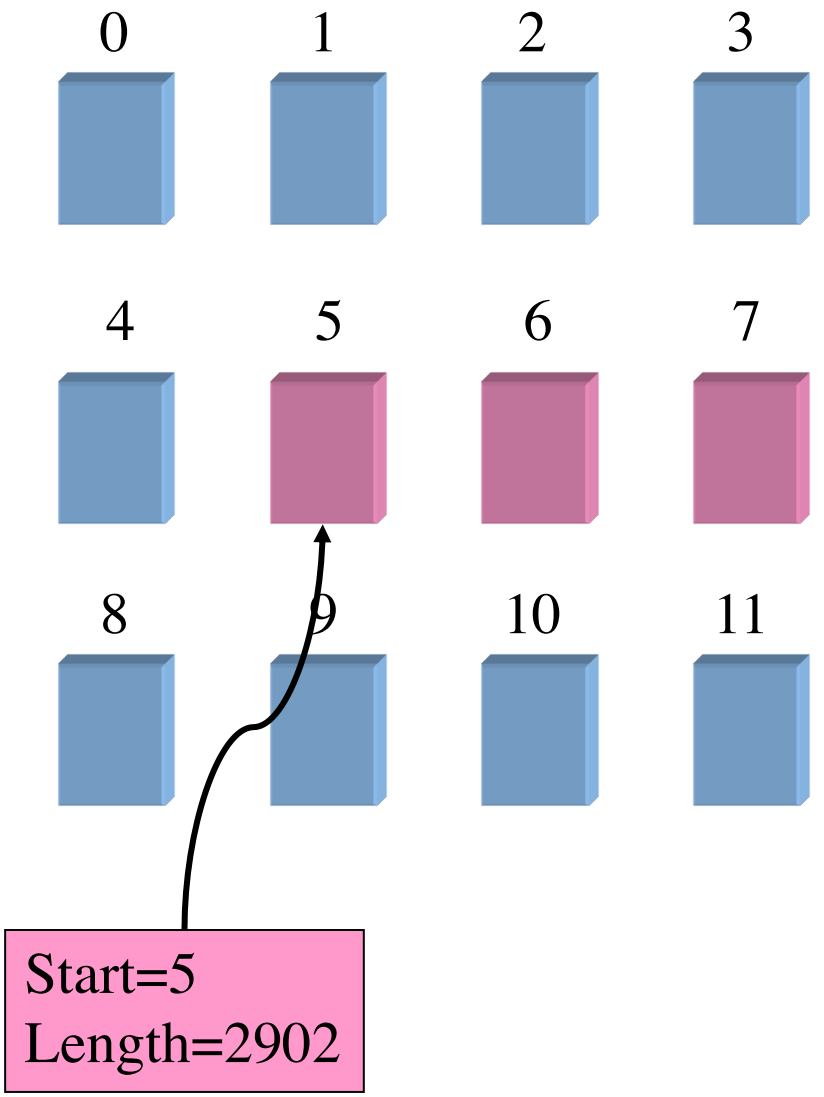


Carving up the disk



Contiguous allocation

- Data in each file in **consecutive blocks** on disk
- Simple & efficient indexing
  - Index node contains
    - Starting location (block #) on disk (Start)
    - Length of file in bytes (Length)
- Random access well-supported



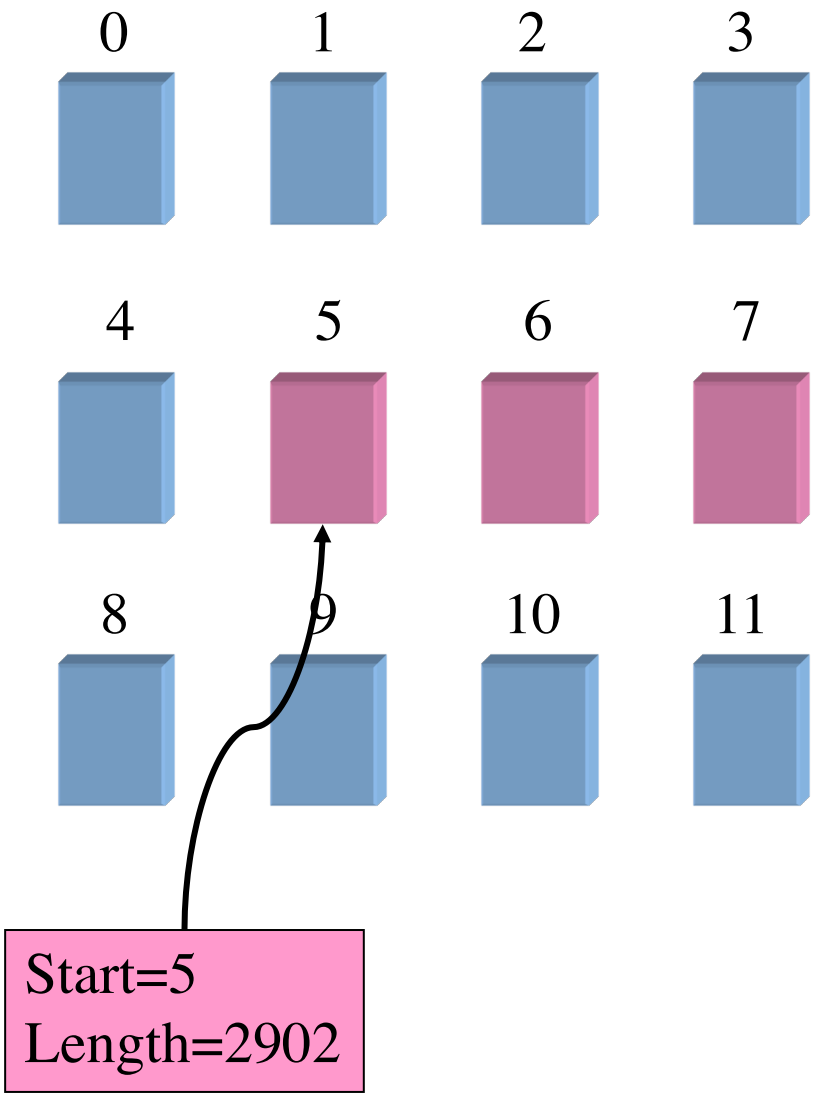
Contiguous allocation: Logical to Phyiscal Mapping

- pos: logical byte number (from the application)
- physical_block_num =

  (pos / block_size) + start
- offset_in_block =

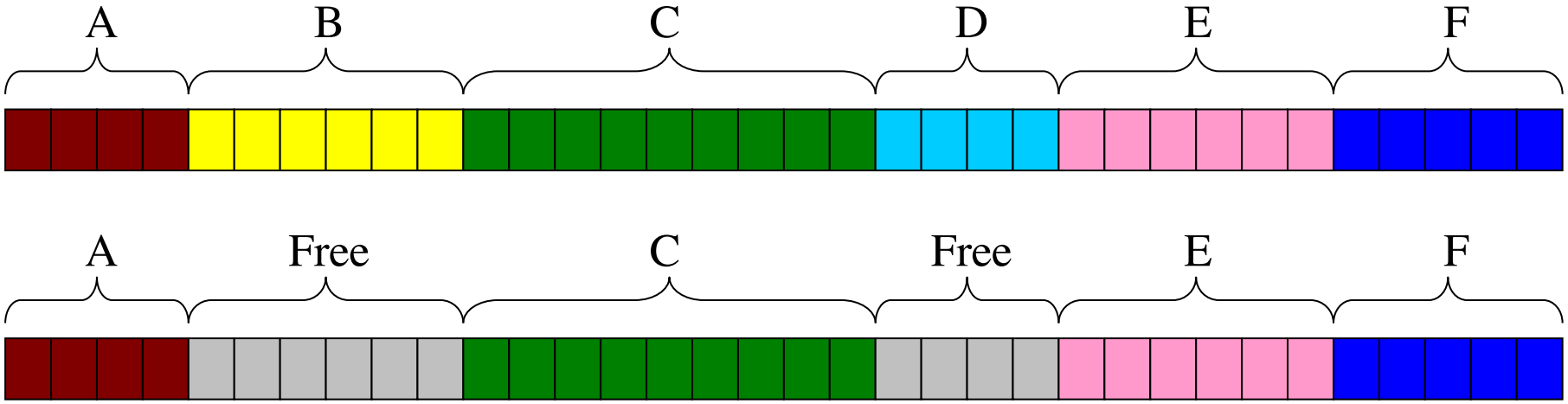
  pos % block_size
- Example: pos = 2000
- physical_block_num = (2000 / 1024) + 5 = 6

  offset_in_block = 2000 % 1024 = 976



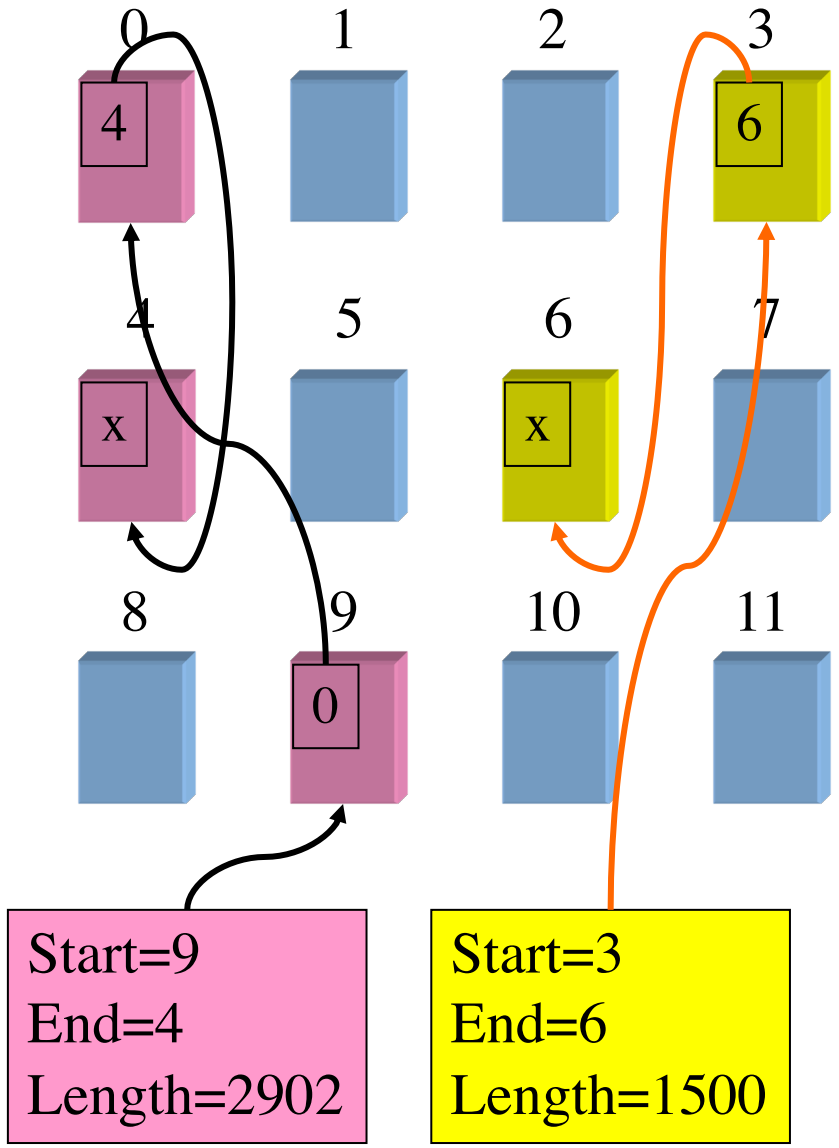
Contiguous allocation: Cons

- Problem: Difficult to grow files
  - Must pre-allocate all needed space
  - Wasteful of storage if file doesn't use all allocated space
- Problem: deleting files leaves "holes"
  - Similar to memory allocation issues (CS 0449 heap project)
  - Compacting the disk is very slow …



Linked allocation

- File blocks form **a linked list**
  - Blocks may be scattered around in disk
  - Block has pointer to next block
  - Files grow as large as needed
- Indexing is simple
  - Index node contains
    - First and last block
    - Length of file in bytes
- Blocks allocated as needed
  - Linked into list of blocks
  - Removed from list of free blocks (or bitmap)
- No wasted blocks - all blocks can be used



Data Structure for Linked Allocation

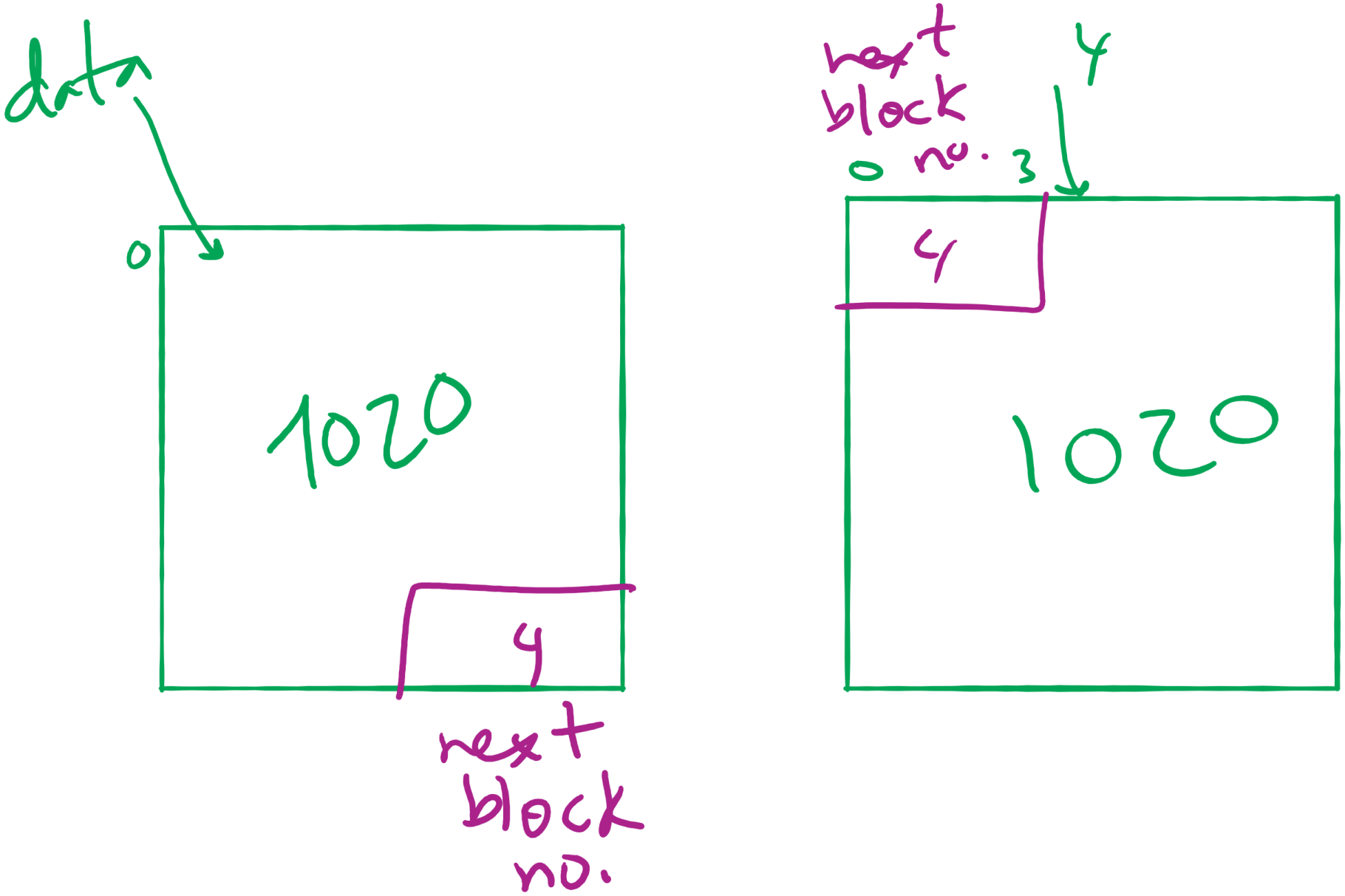
```
struct disk_block {
    unsigned int next_block;
    unsigned char data[1020];
}
```

Logical to Physical Mapping

```
block = start
logical_block_num = pos / block_size
offset_in_block = pos % block_size
for (j = 0; j < logical_block_num; j++) {
   read disk block no. block
   block = block->next_block
}
```

- Assumes next block pointer stored at end of block. **Why?**

Offset Calculation

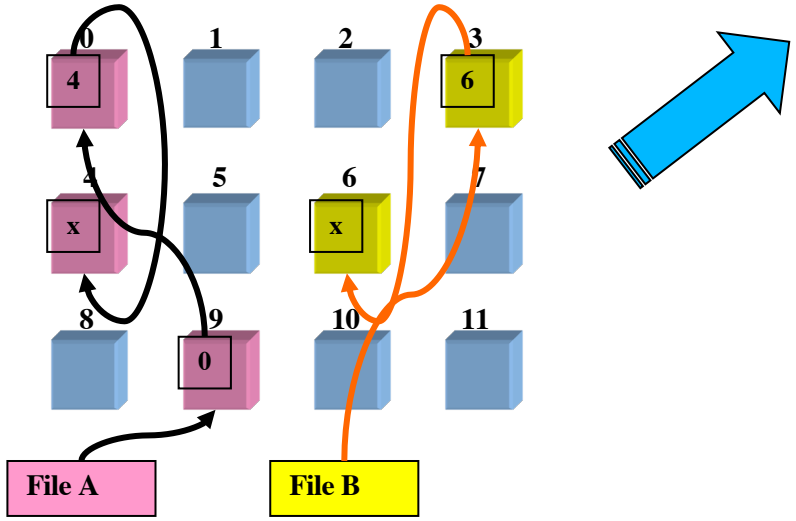


Linked allocation: Cons

- Random access not possible
  - May require a long time for seek to random location in file
- Overhead of next block pointers in each block

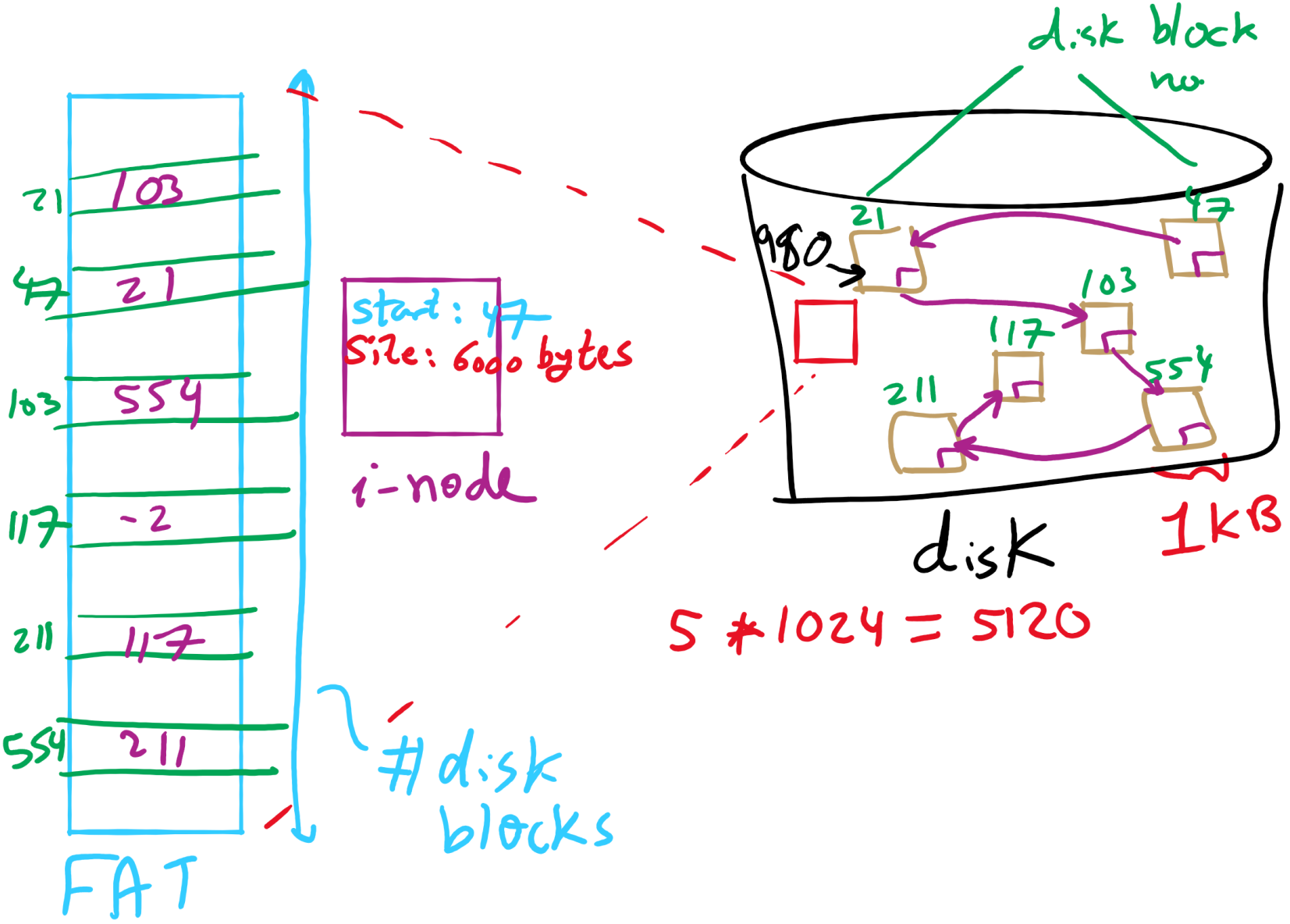
File Allocation Table (FAT)

- Keep linked lists in memory
- Advantage: **faster**
- Disadvantages
  - Have to copy it to disk at some point
  - Have to keep in-memory and on-disk copy **consistent**

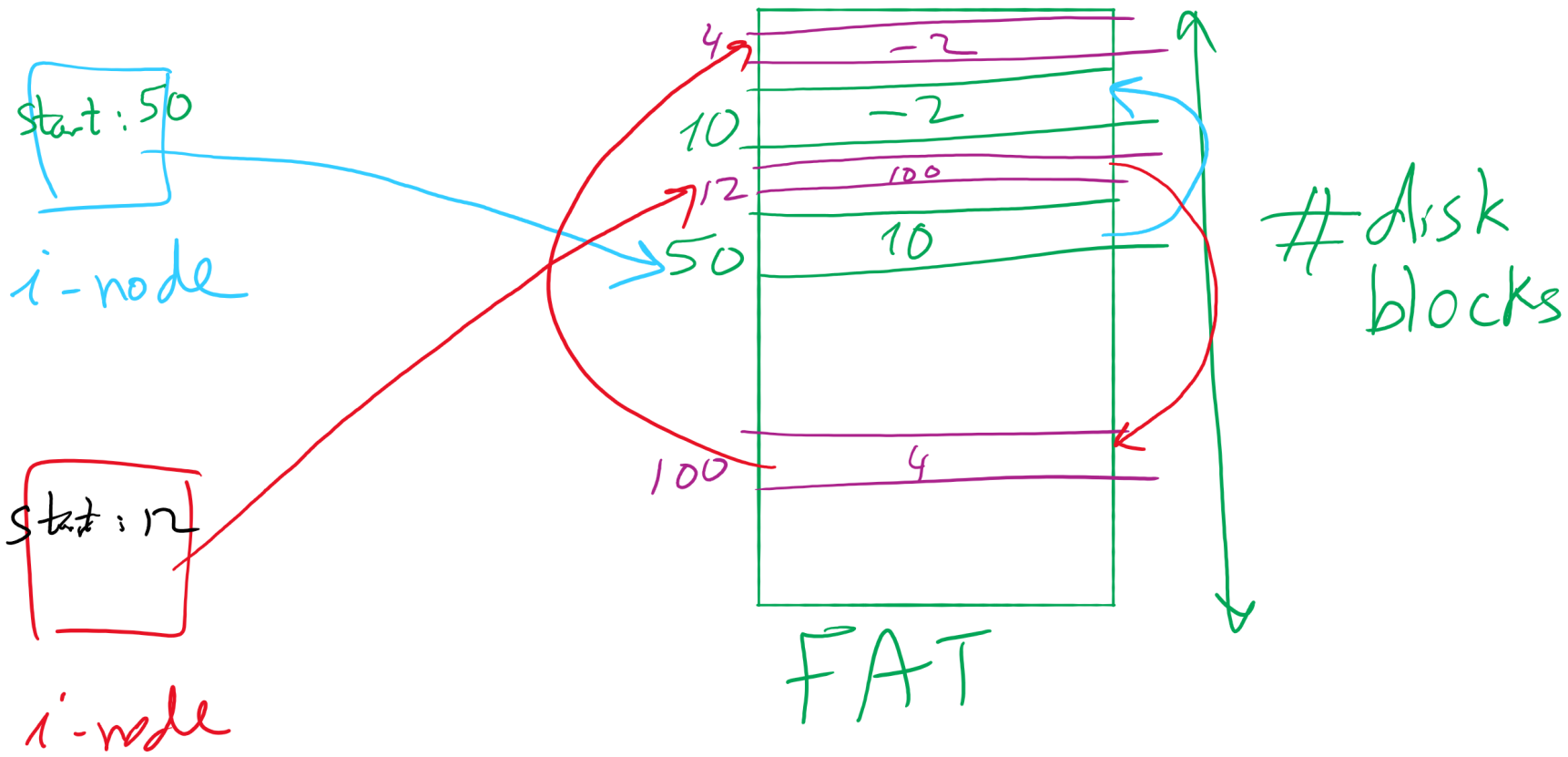


0	4	
1	-1	
2	-1	
3	-2	
4	-2	
5	-1	
6	3	B
7	-1	
8	-1	
9	0	A
10	-1	
11	-1	
12	-1	
13	-1	
14	-1	
15	-1	

FAT Example

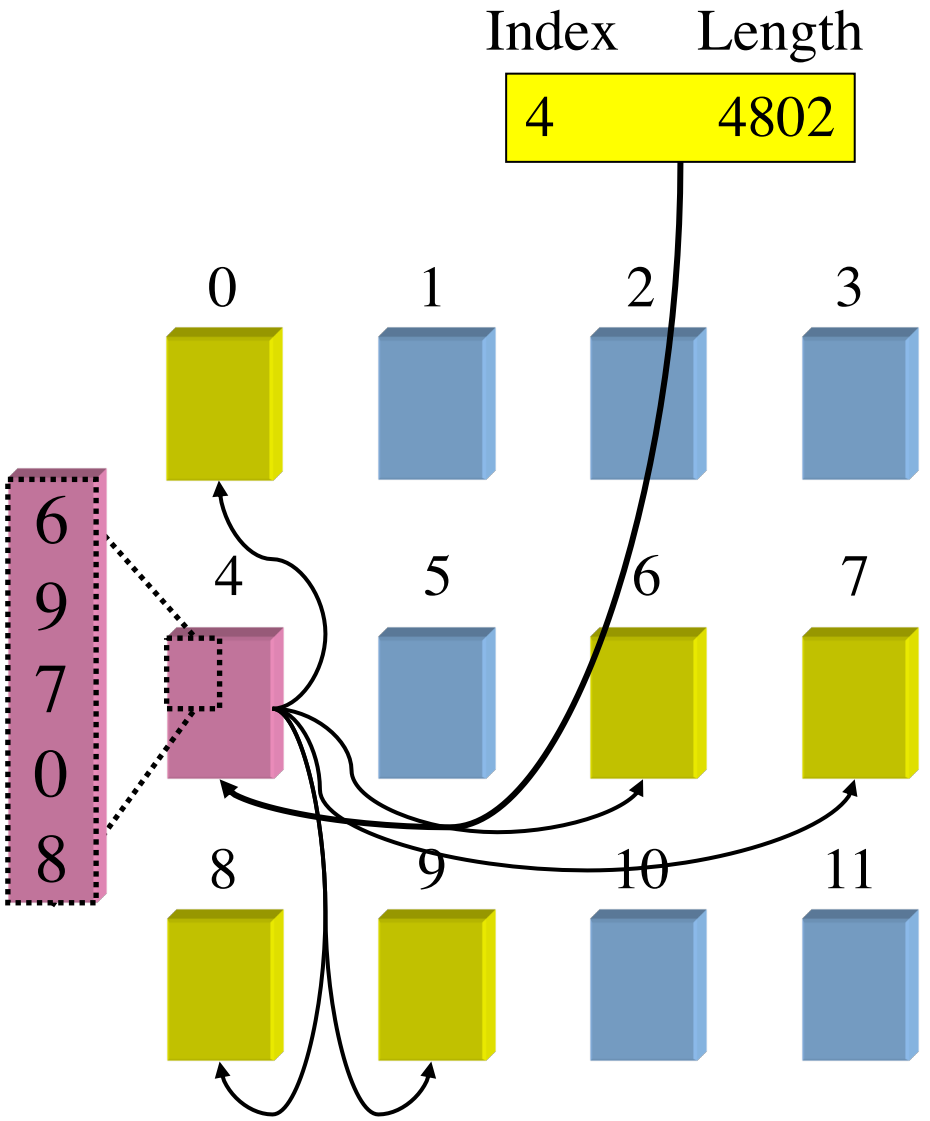

disk block no
21
103
47
21
103
554
117
-2
211
117
554
211
FAT
start: 47
size: 6000 bytes
i-node
disk blocks
980
21
47
103
117
554
211
disk
1kB
5 * 1024 = 5120

index nodes and FAT

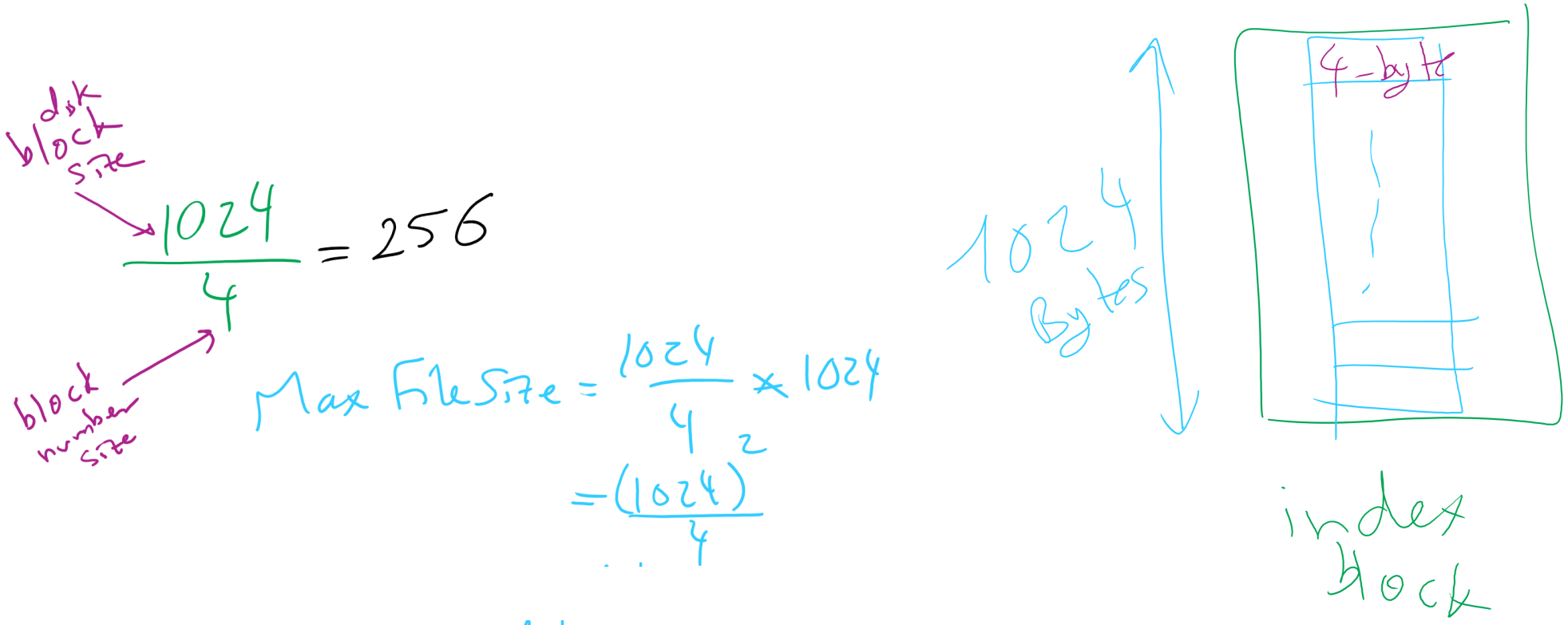


Indexed Allocation

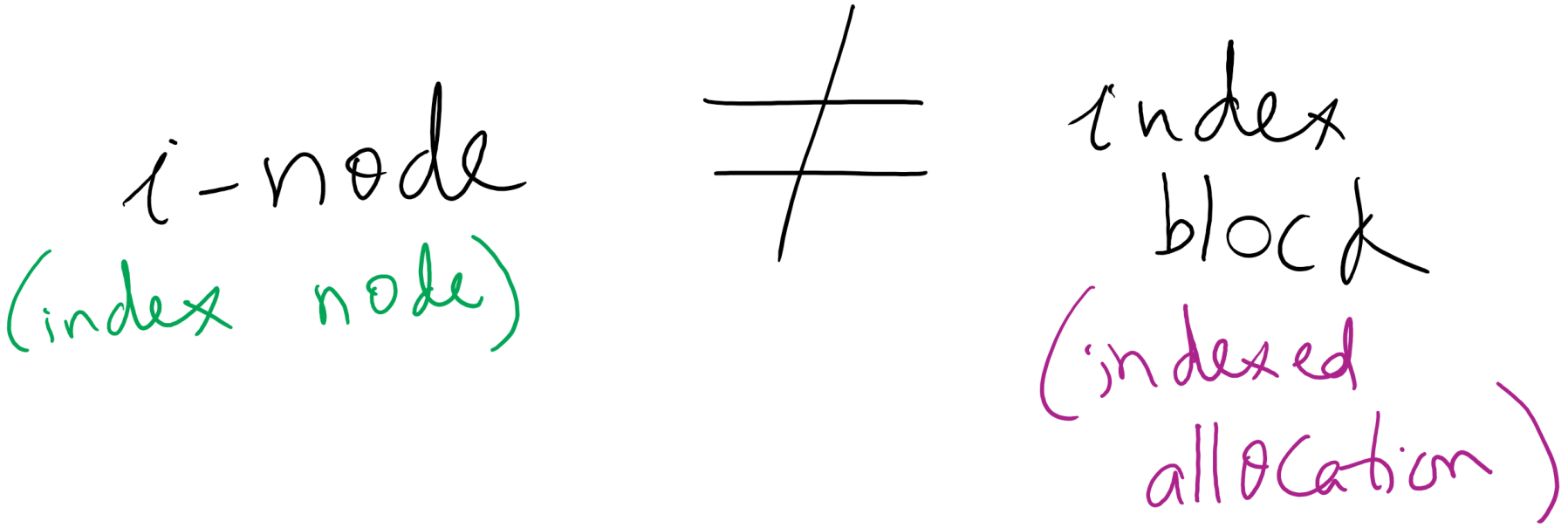
- Store file block addresses in an array (called **index**)
  - Array itself is stored in a disk block
  - Index node has a pointer to index disk block
  - Non-existent blocks indicated by -1
- Random access easy
- Cons: Limit on file size!



Max File Size for Indexed Allocation

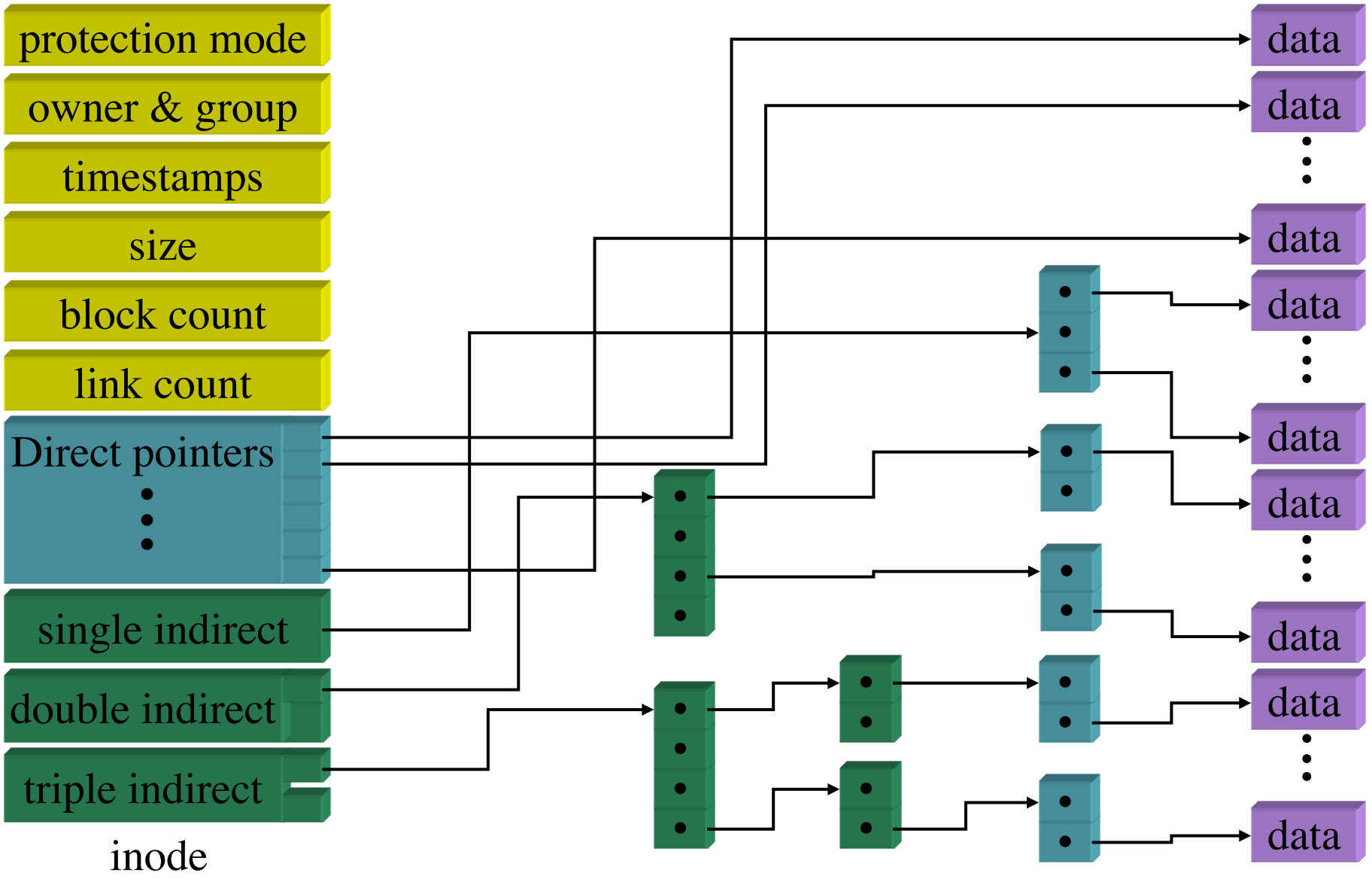


i-node vs index block

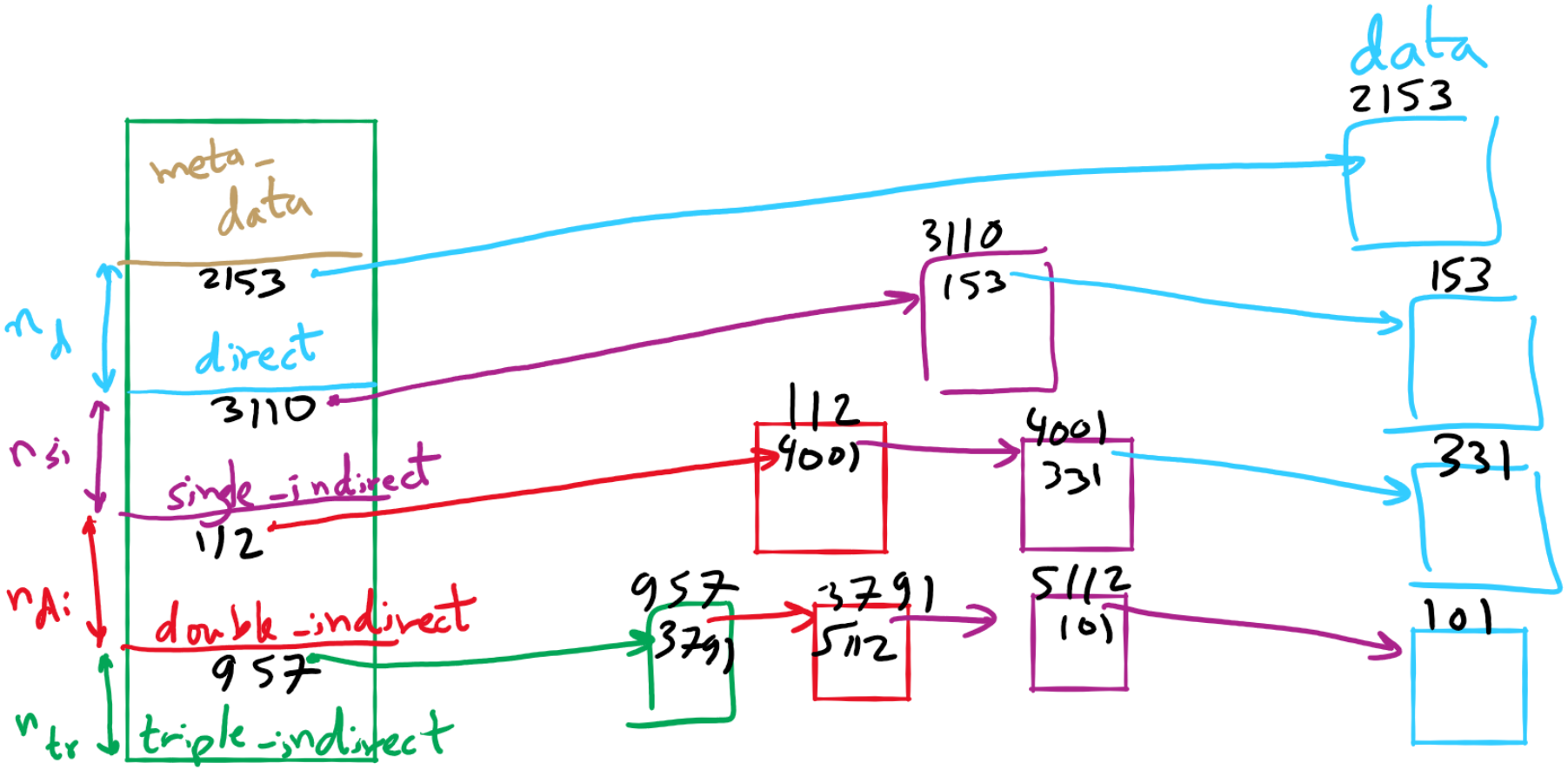


Hybrid Allocation (Lab 5)

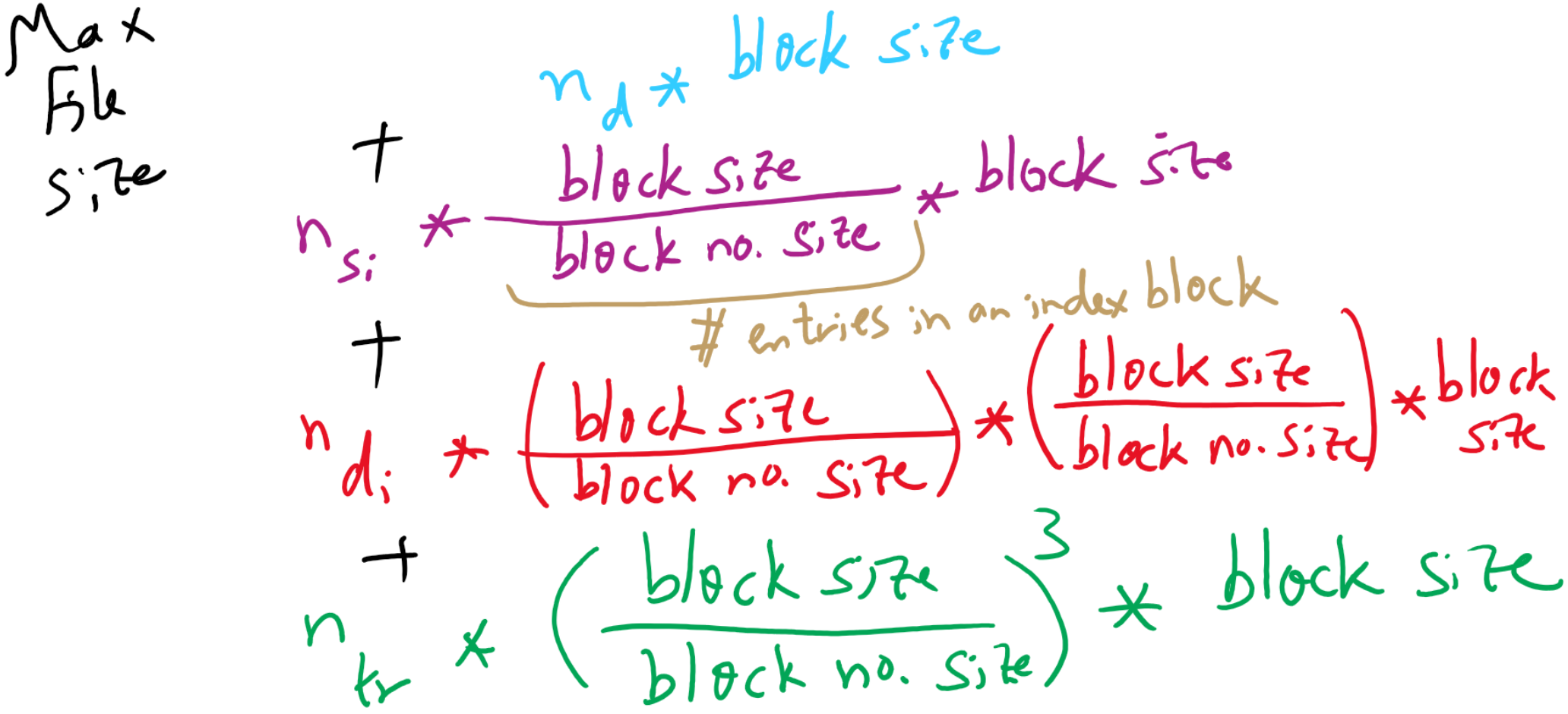
- Unix Fast File System indexing scheme



FFS Example

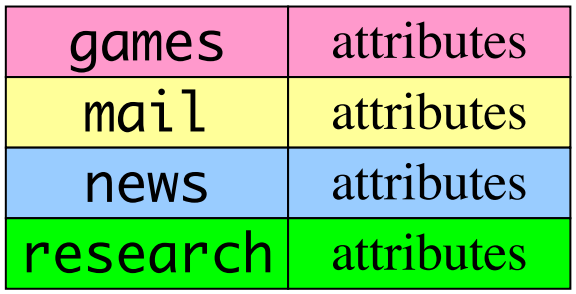

meta-data
2153
direct
3110
single-indirect
112
double-indirect
957
triple-indirect
data
2153
3110
153
153
112
4001
4001
331
331
957
3791
3791
5112
5112
101
101

FFS and Max File Size

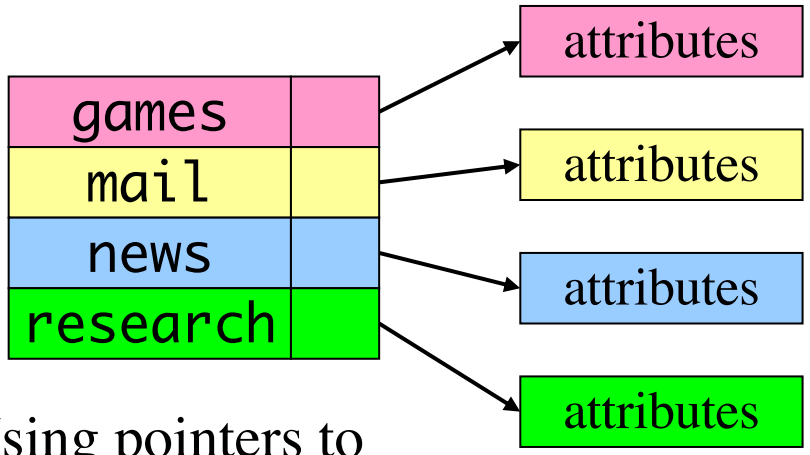


What's in a file directory?

- Two types of information needed about a file
  - File names
  - File metadata (size, timestamps, etc.)
- Basic choices for directory information
  - Store all information in directory
    - Fixed size entries
    - Disk addresses and attributes in directory entry
  - Store names & pointers to index nodes (i-nodes)

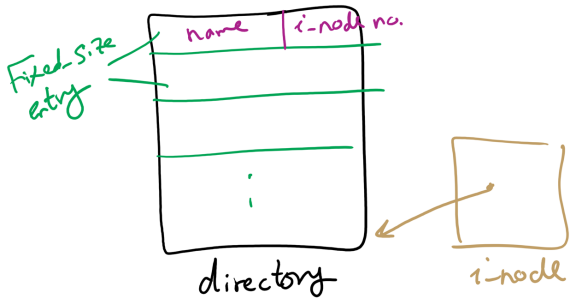


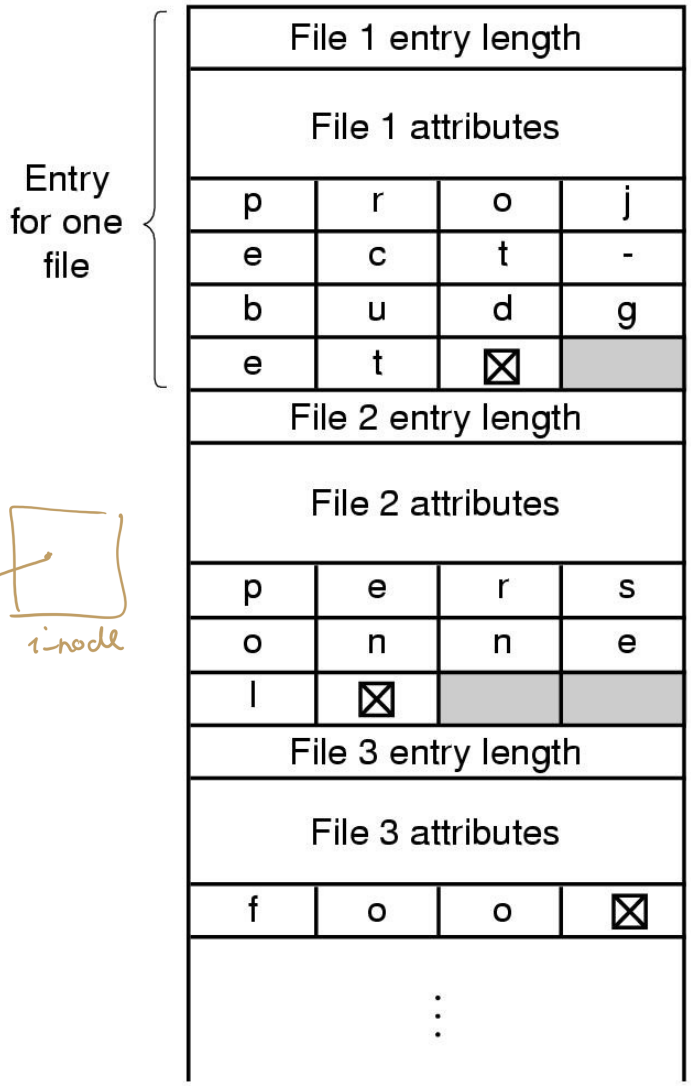
Storing all information in the directory



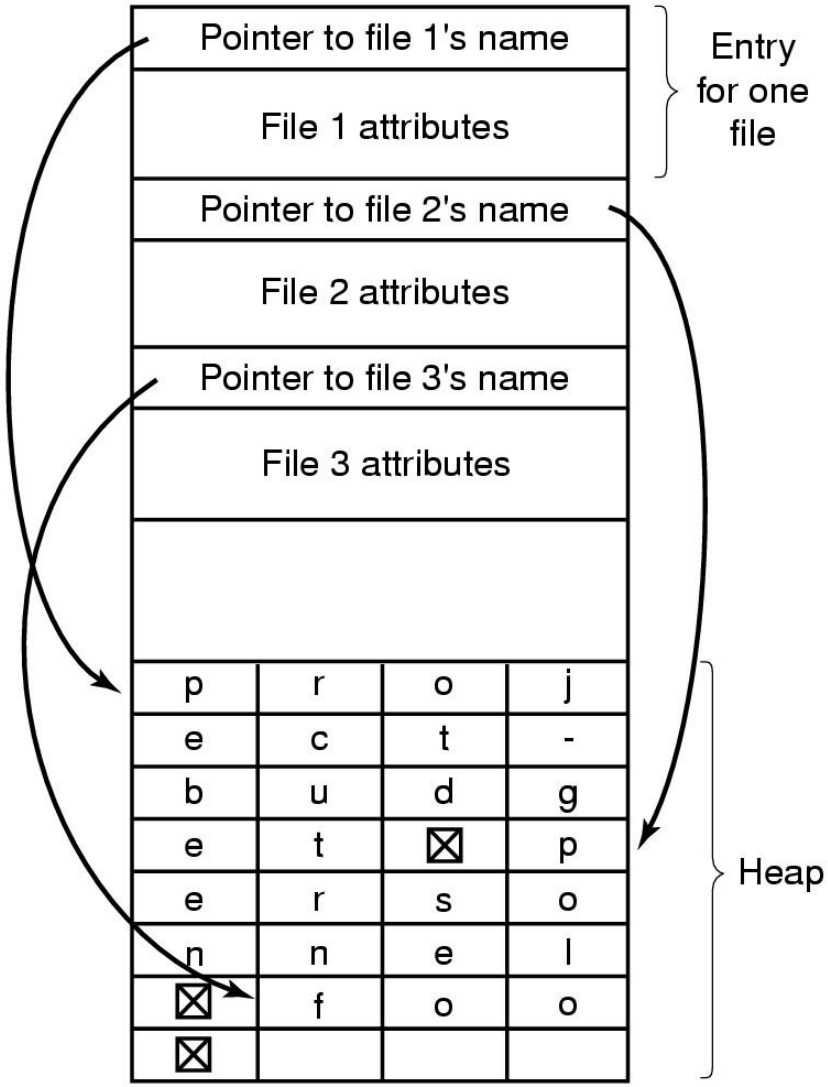
Using pointers to index nodes

Handling long file names in a directory



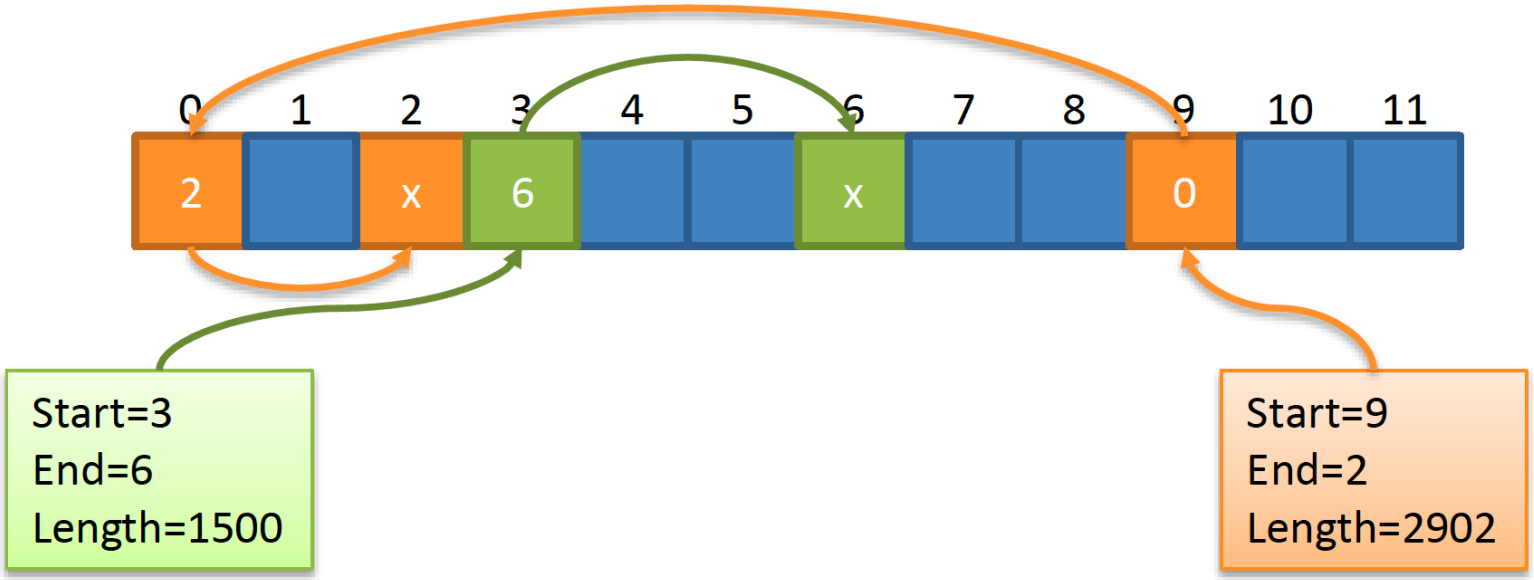


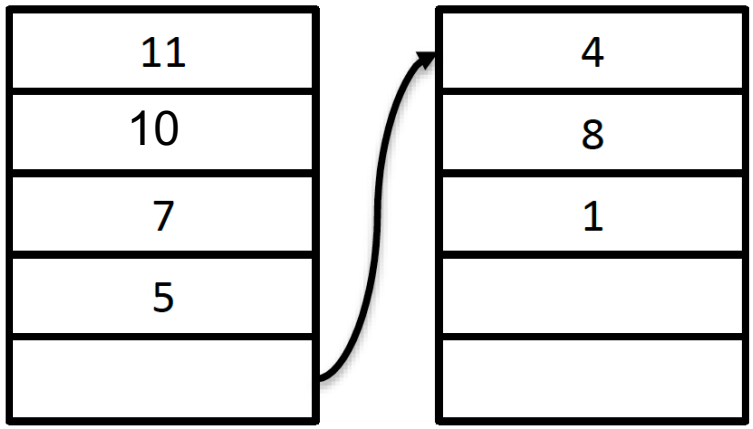
(a)



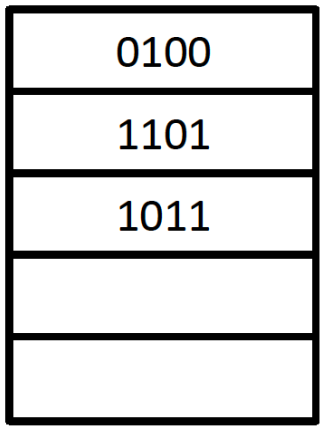
(b)

Free Block Tracking



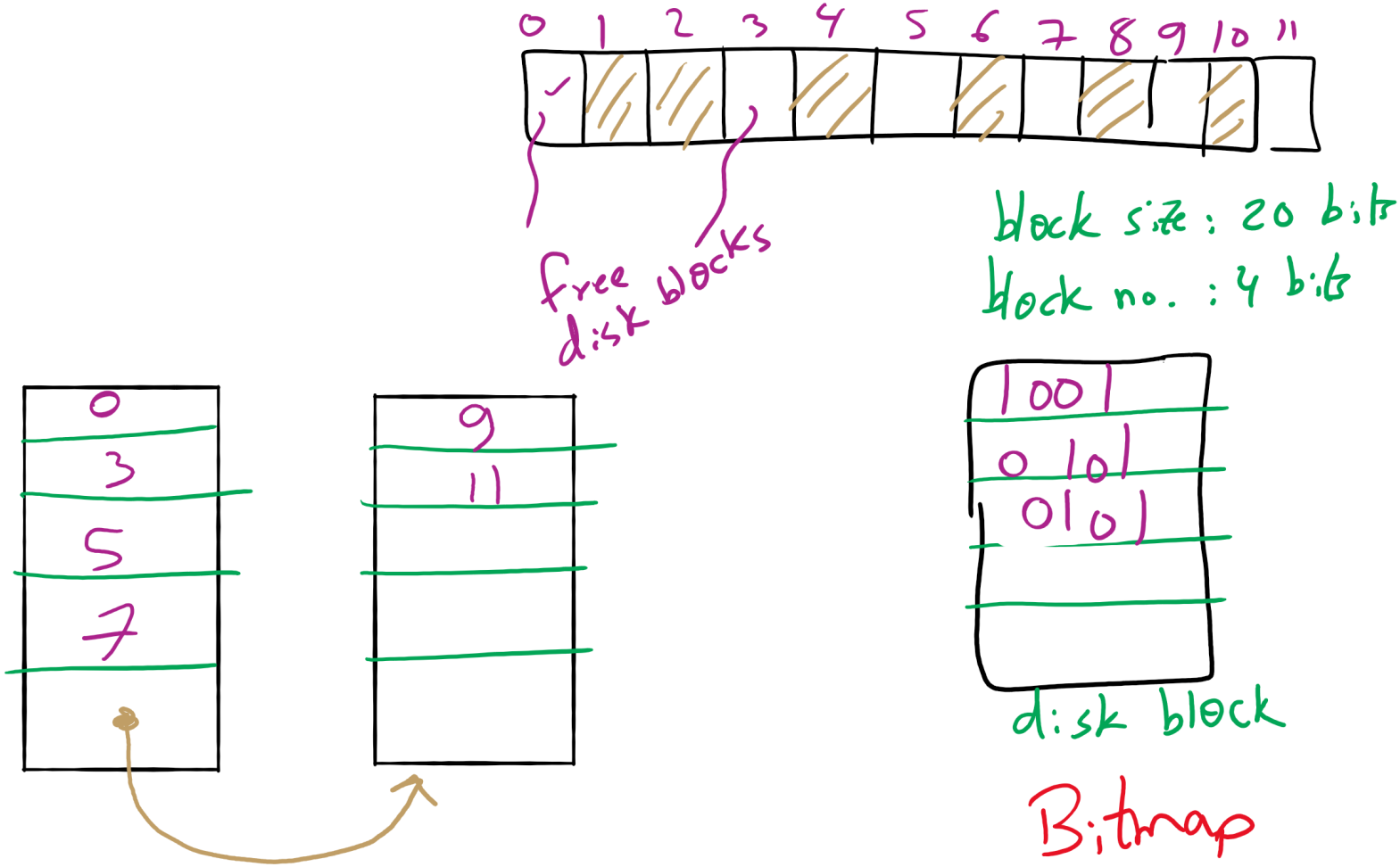


Linked List



Bitmap

Free Block Tracking Example 1



Free Block Tracking Example 2

20-bit disk block

disk block no. : 4 bits