CSE 120
PA4 Discussion

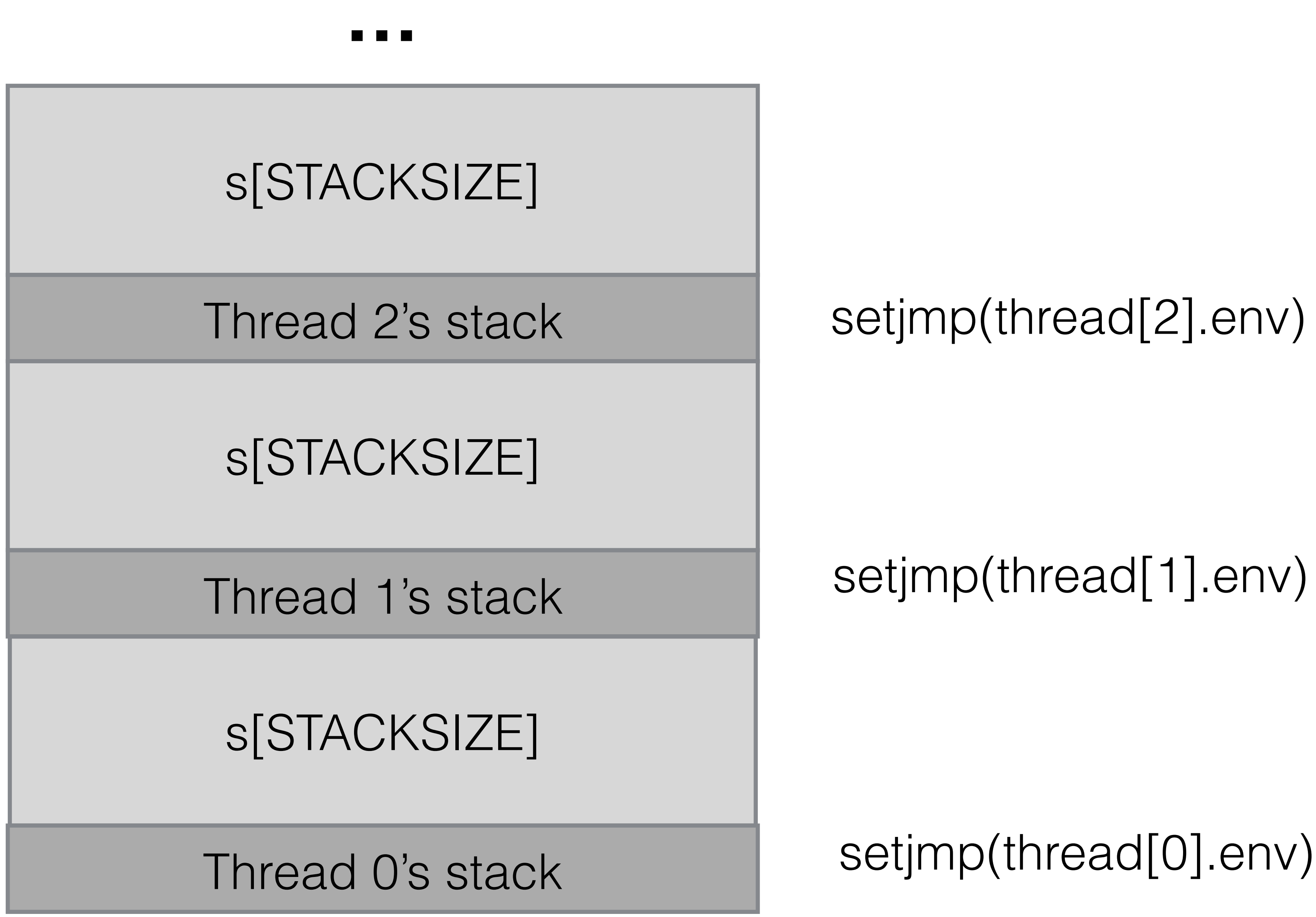
PA4

- Implement a user-level thread package
- Entirely on user-level, no kernel modification
- You will be only turning in mythreads.c, which includes following functions:
    - MyInitThreads()
    - MySpawnThread(func, param)
    - MyYieldThread(t)
    - MyGetThread()
    - MySchedThread()
    - MyExitThread()

InitThreads

- Partition the stack into MAXTHREADS parts
- Make use of dummy array to separate stacks of different threads
- Needs to generalize the code in MySpawnThread(func, param) to support partition into MAXTHREADS parts

After partition



Iterative Solution

```
for (int i=1; i< MAXTHREADS; i++)

{

    char s[i*STACKSIZE]; //dummy array

    Save the environment using setjmp(thread[i].env);

}
```

Common Mistakes

- Forgot to call setjmp(env) for thread 0.
- Referencing local variable after MyInitThread returns.

Iterative Solution

```
for (int i=1; i< MAXTHREADS; i++)

{

    char s[i*STACKSIZE]; //dummy array

    Save the environment using setjmp(thread[i].env);

}
```

Thread 0’s context is not saved!

Example

```
for (int i=1; i< MAXTHREADS; i++)
{
    char s[i*STACKSIZE]; //dummy array
    if (setjmp(thread[i].env) != 0) {
        //access i    May cause segmentation fault!
    }
}
```

Example (cont'd)

- When would setjmp(thread[i].env) return for the second time?

  When some other thread yield to thread i.

- What does the stack look like at that point?

  The activation record for MyInitThread is gone, including the iterating variable i. It is very likely to be overwritten by other activation records.

```
for (int i=1; i<MAXTHREADS; i++) {
    char s[i*STACKSIZE];
    if (setjmp(thread[i].env)!=0) {
        // access i
    }
}
```

Alternative

- Using Recursion

```
stackPartition(int number_partitions) {
    if (number_of_partitions <=1) then return;
    else {
        char s[STACKSIZE];
        int id = MAXTHREADS - number_partitions + 1;
         if (setjmp(thread[id].env)!=0) {
           //access id variable
        }
        …
        stackPartition(number_partitions-1);
    }
}
```

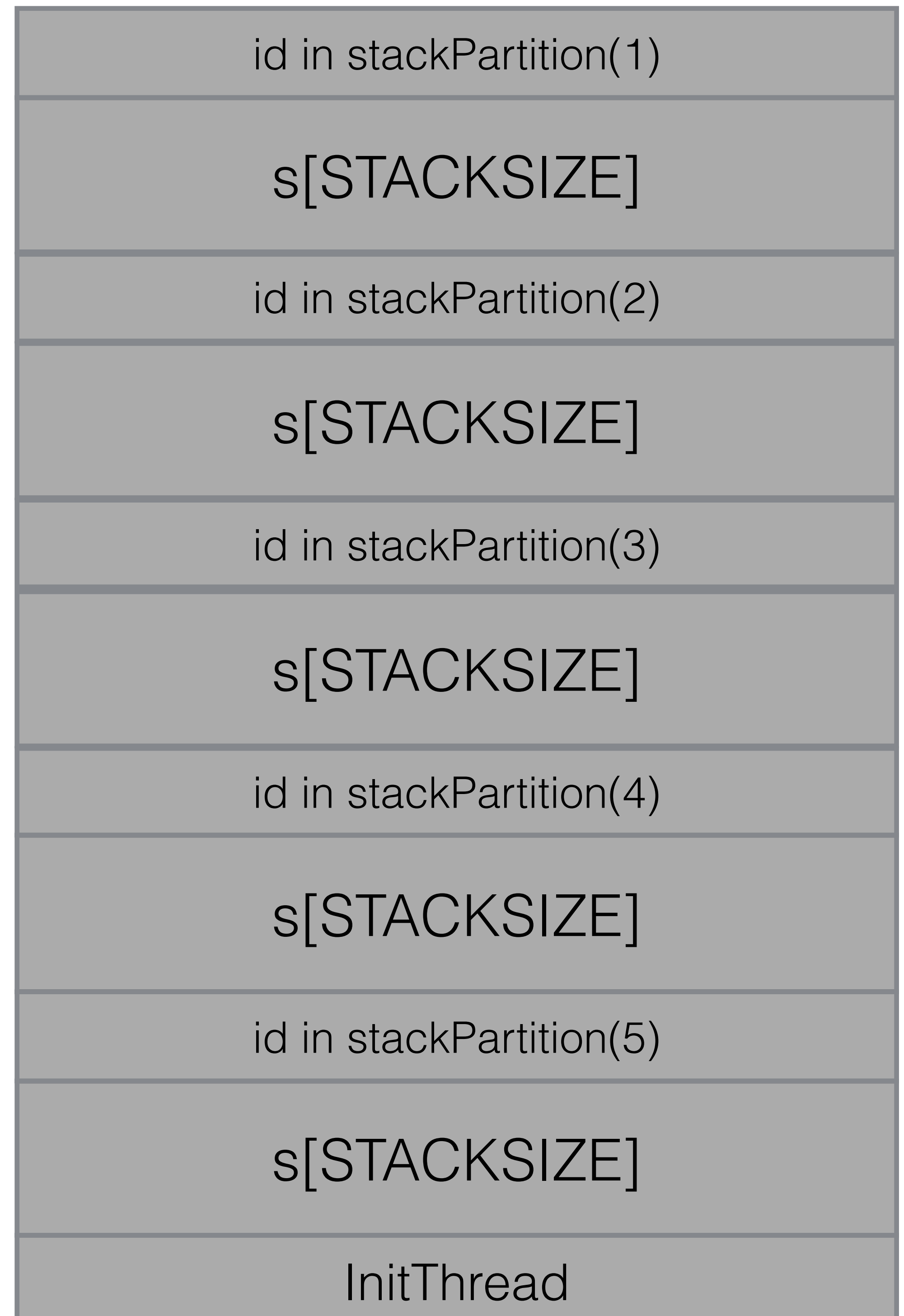
Will we have segmentation fault here?

No

Stack Inspection

- What stack looks like when stackPartition(5) returns

- Each recursive call of stackPartition() have its own instance of ***id*** variable. And they are protected by the “cushion” array **s**.



MySpawnThread

- Incremental assignment of thread id.
- Store the function pointer and parameter in thread table
- No need to call setjmp inside MySpawnThread since you have partitioned the stack inside MyInitThread
- Thread id is the index in thread table

Thread Id Example

Thread Id

0	T0
1	T1
2	T2
3	T3
4	T4
5	T5
6	T6
7	
8	
9	

Thread 0 ~ 6 spawned

Thread Id Example

Thread Id

0	T0
1	T1
2	T2
3	T3
4	T4
5	T5
6	
7	
8	
9	

T6 exits

Thread Id Example

T7 spawned

Thread Id		
0	T0	
1	T1	
2	T2	
3	T3	
4	T4	
5	T5	
6		<— Last Assigned Id
7	T7	
8		
9		

MyYieldThread

- Use setjmp to save the context of current thread
- Use longjmp to give up CPU to the specified thread.
- If you pass the second parameter of longjmp(env, t) as the current running thread id, make sure you handle the case when current thread id is 0.
- Handle the case when yielding to a thread id that is in range but invalid.

MySchedThread

- The FIFO queue in this assignment is different from the one you saw in PA3 and PA2 because interior elements can be removed.
- Reuse MyYieldThread
- Make sure MySchedThread and MyYieldThread cooperates correctly.

Example

- One implementation:

```
MySchedThread() {
    int t  = remove head from queue
    add current to tail;
    MyYieldThread(t);
}
MyYieldThread(t) {
    remove t from queue
    add current to tail;
    if (setjmp(thread[current].env) != 0) {
        longjmp(thread[t].env);
    }
}
```

- Alternative:

```
MySchedThread() {
    int t  = head from queue
    MyYieldThread(t);
}
MyYieldThread(t) {
    remove t from queue
    add current to tail;
    if (setjmp(thread[current].env) != 0) {
        longjmp(thread[t].env);
    }
}
```

Since we are always going to call MyYieldThread, we can put the logic of updating the queue inside MyYieldThread

MyYieldThread is Most Important

- MyYieldThread()
- MySchedThread() -> MyYieldThread()
- MyExitThread() -> MySchedThread() -> MyYieldThread()

MyExitThread

- The function to call when thread finishes execution of the associated function.
- Always call MyExitThread() after the function execution line in mythreads.c
- It releases the slot in thread table so threads spawned later on can make use of that.
- Needs to reset the **env** to initial state (you may need to add some variables inside thread table).

MyExitThread

- Needs to call MySchedThread in the end so other threads can get scheduled.
- When no other threads left, call Exit() to terminate the whole process

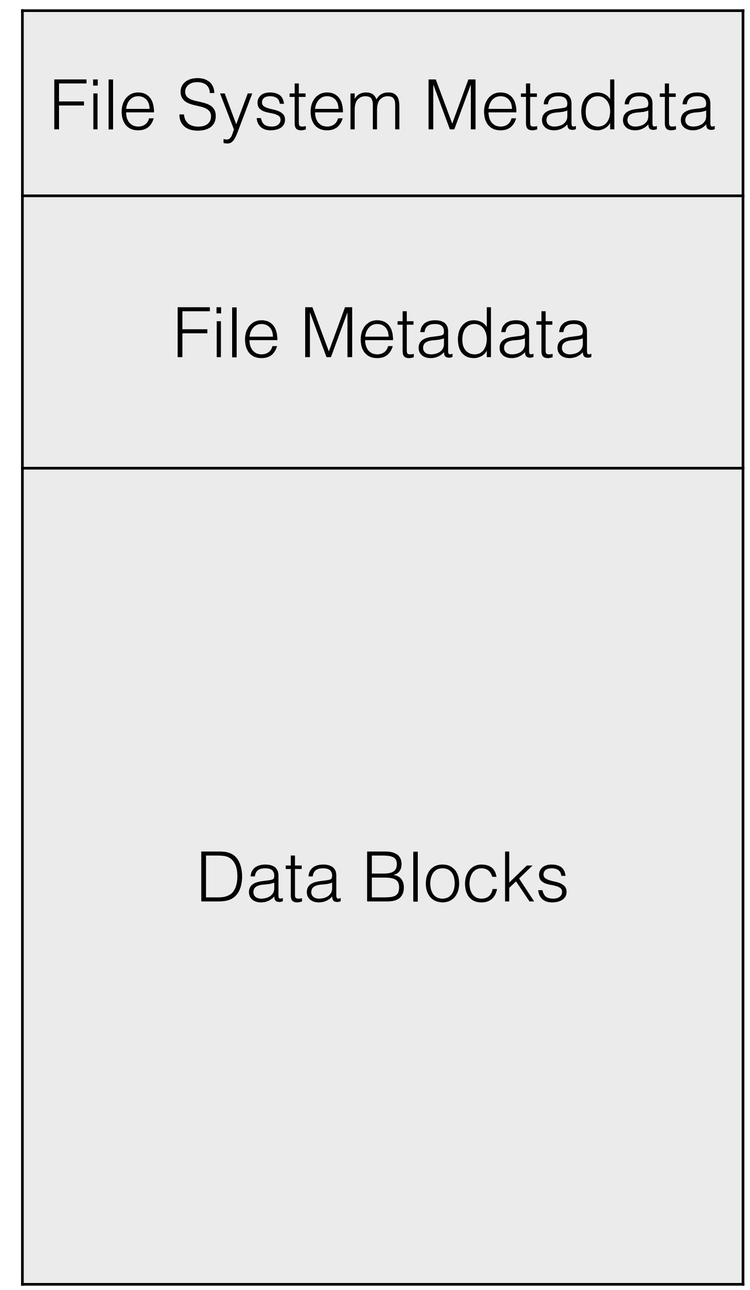
PA4 Verification

- If you are unsure about what the correct behavior may be, you can always remove the “My” prefix and make and check the output.

PA4 Questions?

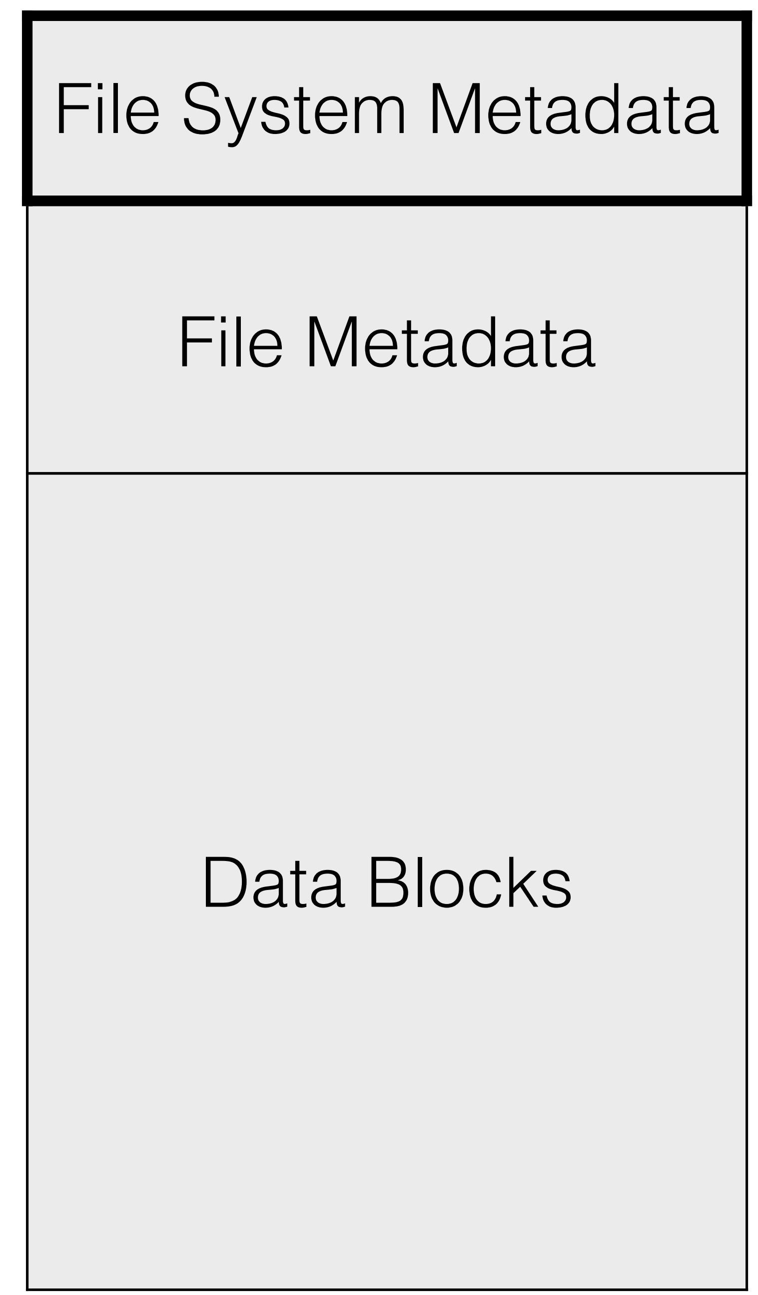
File System Layout

- Divided into three regions:
  - File System Metadata
    - Information about file system
  - File Metadata
    - File control blocks
  - Data Blocks
    - Actual file data



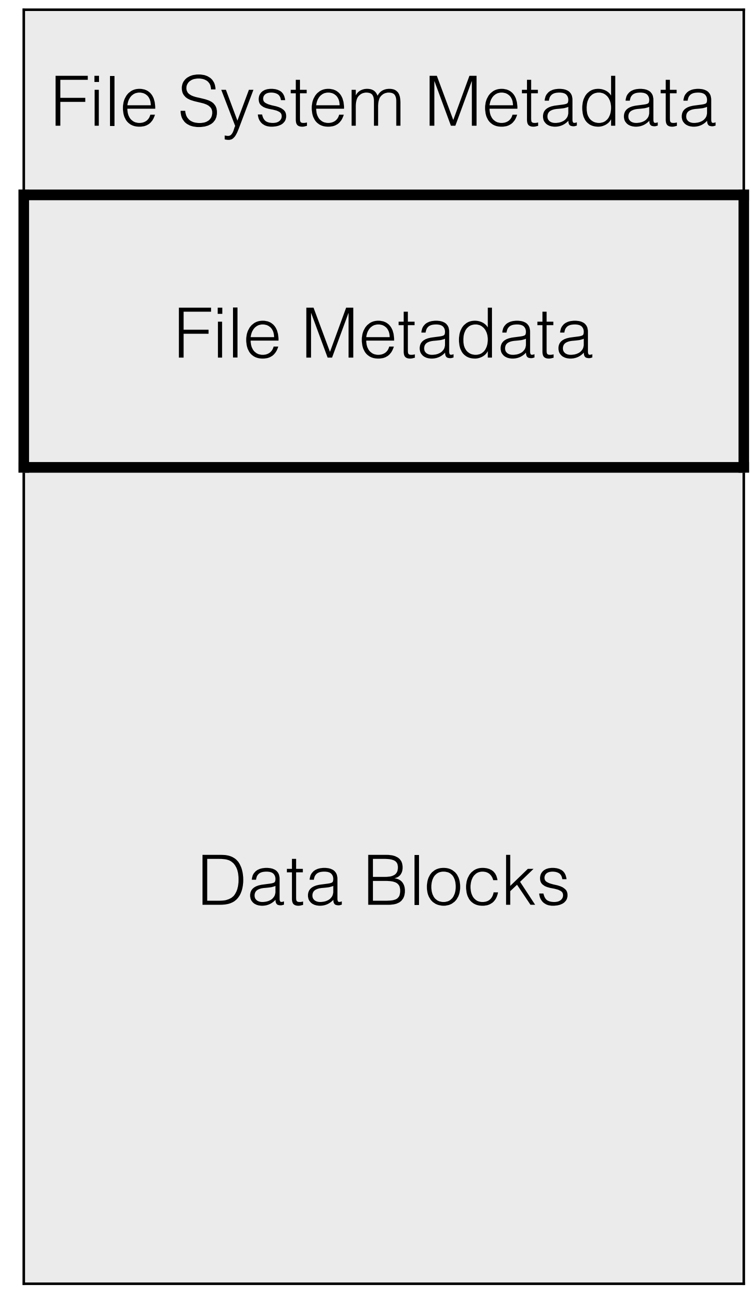
File System Metadata

- Type of the file system
- The location of root directory
- How many free blocks



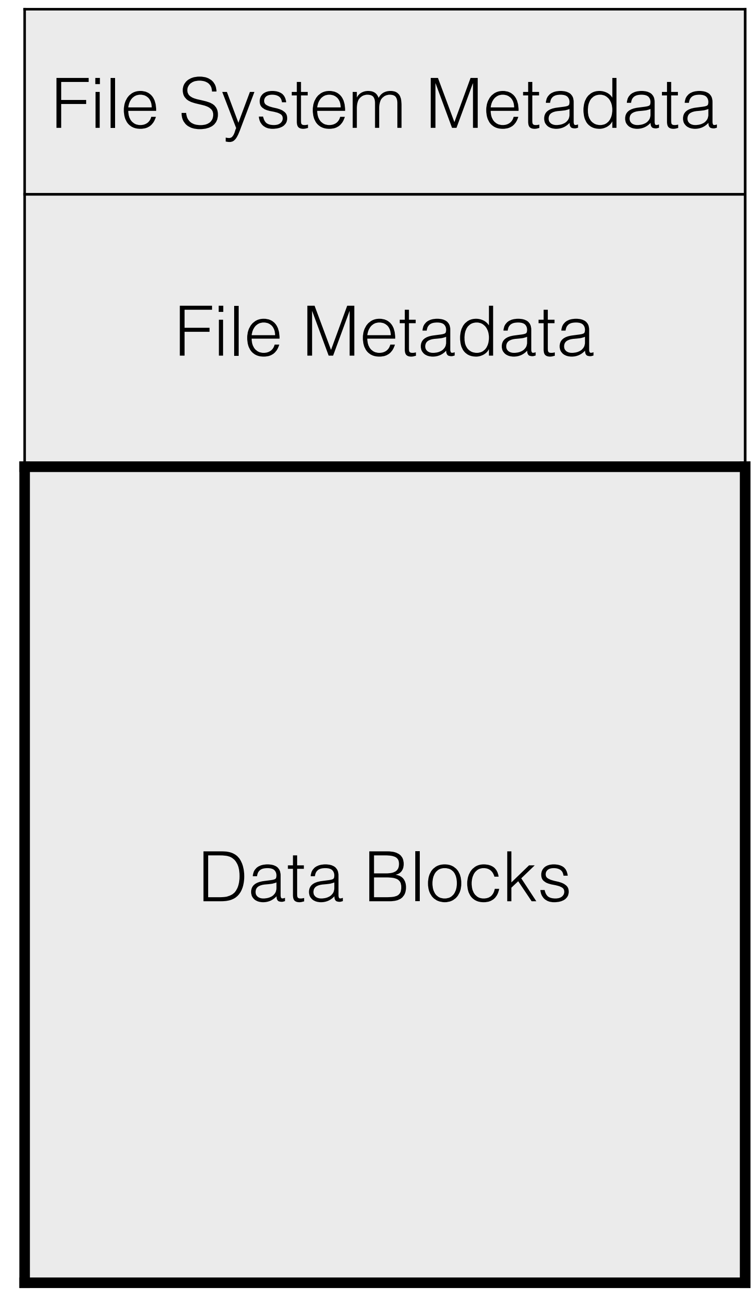
File Metadata

- Divided into entries
- Each entry contains a file's metadata:
  - Attributes: file size, permissions,file type
  - Block map: pointers to file data
  - Filename is not stored here, why?



Data Blocks

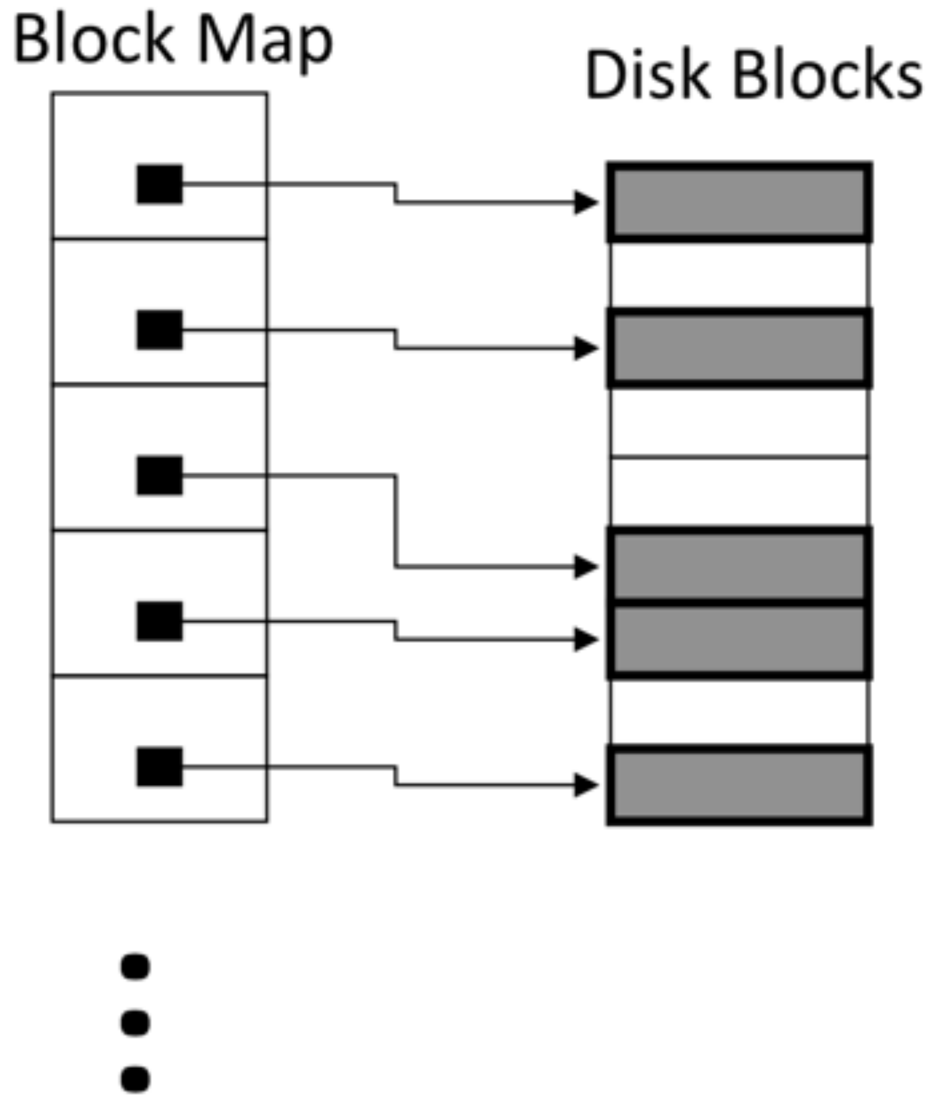
- Actual contents of files



Unix Block Map

- Which blocks on disk contain file data?
- How much metadata do we need to store to find these blocks?

All Direct Pointers



Exercise

- Assuming using only direct pointers, and the address is 32-bit, one block is 1KB, how much size of disk pointers do we need to support a file of 16GB (2^34 bytes)?

16GB means 2^34/2^10 = 2^24 blocks
For each block we need to assign a pointer, which is 4 byte.
So in total, the pointer size is 4*2^24 = 2^26 bytes = 64 MB

Too much space for metadata!

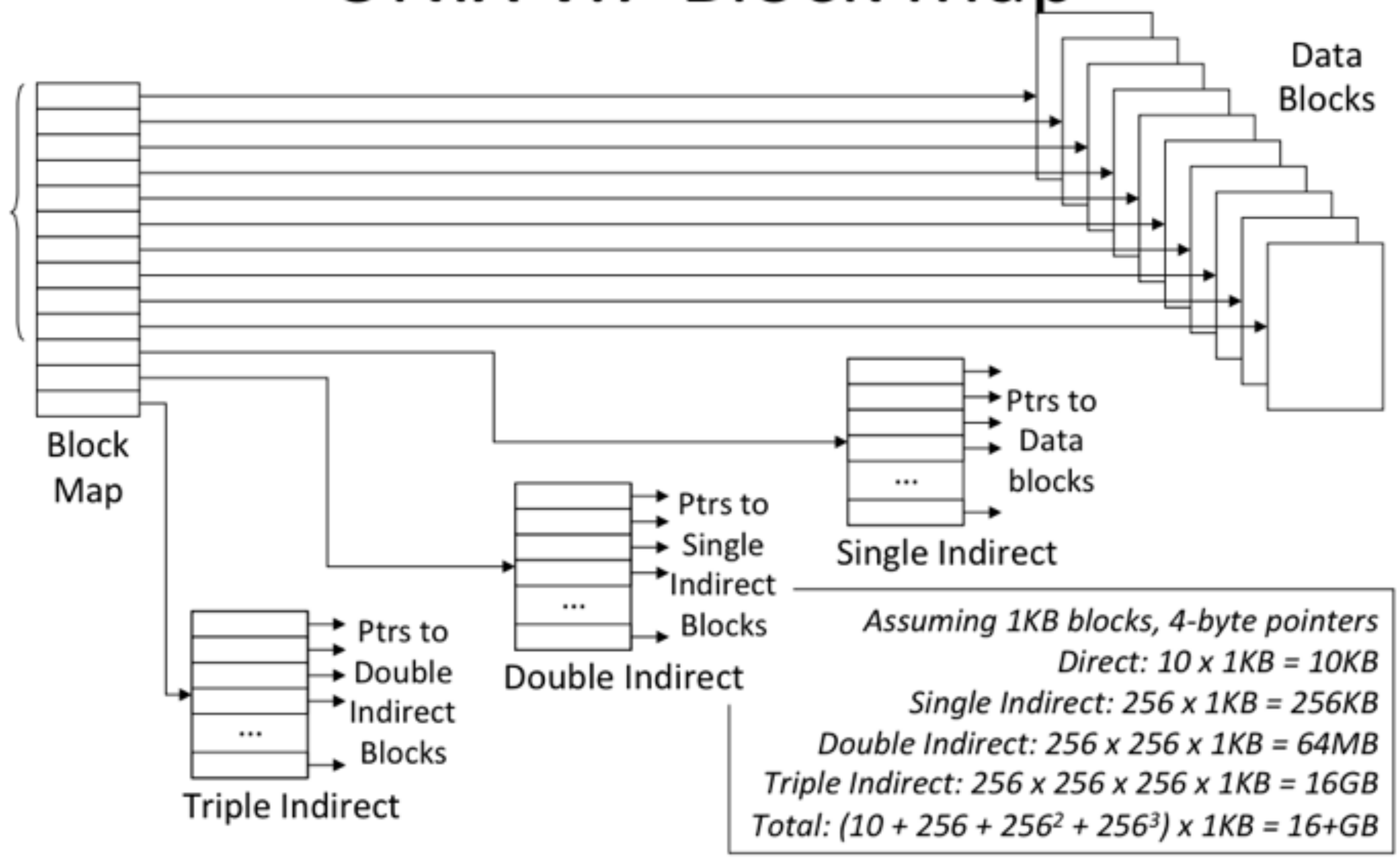
Storage Overhead

- 64MB metadata for one file
- What if the file size is only 1KB?
- Direct block map is a bad idea.
  - Need to handle max file size
  - Most files are not at the max size

Unix Block Map

- 13 Pointers (10 direct + 3 indirect)
  - 10 direct pointers, each of them points to a block
  - 1 points to a block that contains pointers to N blocks. (1st level)
  - 1 points to a block that contains pointers to N 1st level blocks (2nd level)
  - 1 points to a block that contains pointers to N 2nd level blocks
- N is determined by block size

UNIX v.7 Block Map



Now we can use 13 pointers (52 bytes) to support files of size 16GB! This is much better than 64MB only using direct pointer!