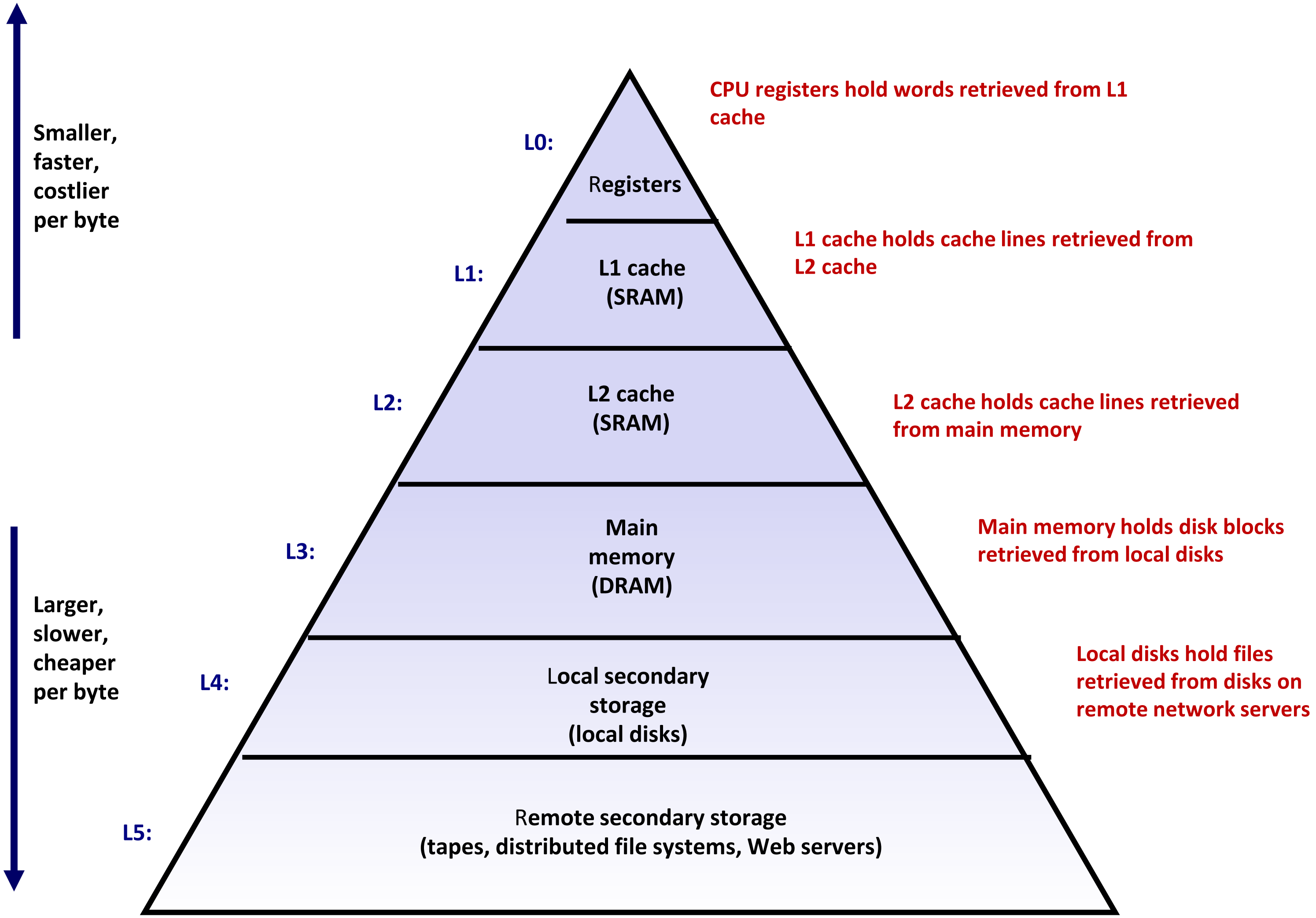
# Cache Lab Implementation and Blocking

# Outline

- **Memory organization**
- **Caching**
  - Different types of locality
  - Cache organization
- **Cache lab**
  - Cache Structure
  - getopt/fscanf/Malloc
- **Page Replacement**
  - LRU algorithm
  - FIFO algorithm

# Memory Hierarchy

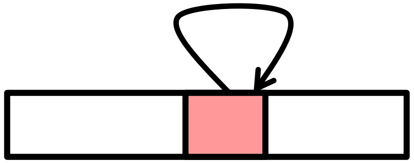


# SRAM vs DRAM tradeoff

- **SRAM (cache)**
  - Faster (L1 cache: 1 CPU cycle)
  - Smaller (Kilobytes (L1) or Megabytes (L2))
  - More expensive and “energy-hungry”
- **DRAM (main memory)**
  - Relatively slower (hundreds of CPU cycles)
  - Larger (Gigabytes)
  - Cheaper

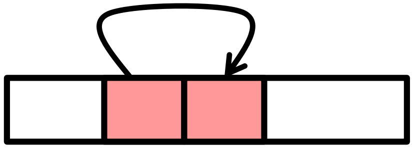
# Locality

- **Temporal locality**
  - Recently referenced items are likely to be referenced again in the near future



  - After accessing address X in memory, save the bytes in cache for future access

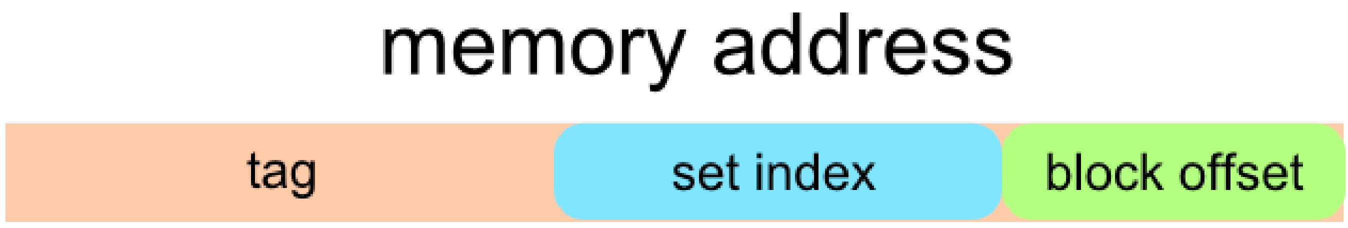
- **Spatial locality**
  - Items with nearby addresses tend to be referenced close together in time



  - After accessing address X, save the block of memory around X in cache for future access

# Memory Address

- **For example, 64-bit on shark machines**




  - Block offset: b bits
  - Set index: s bits
  - Tag Bits: (Address Size – b – s)

# Cache

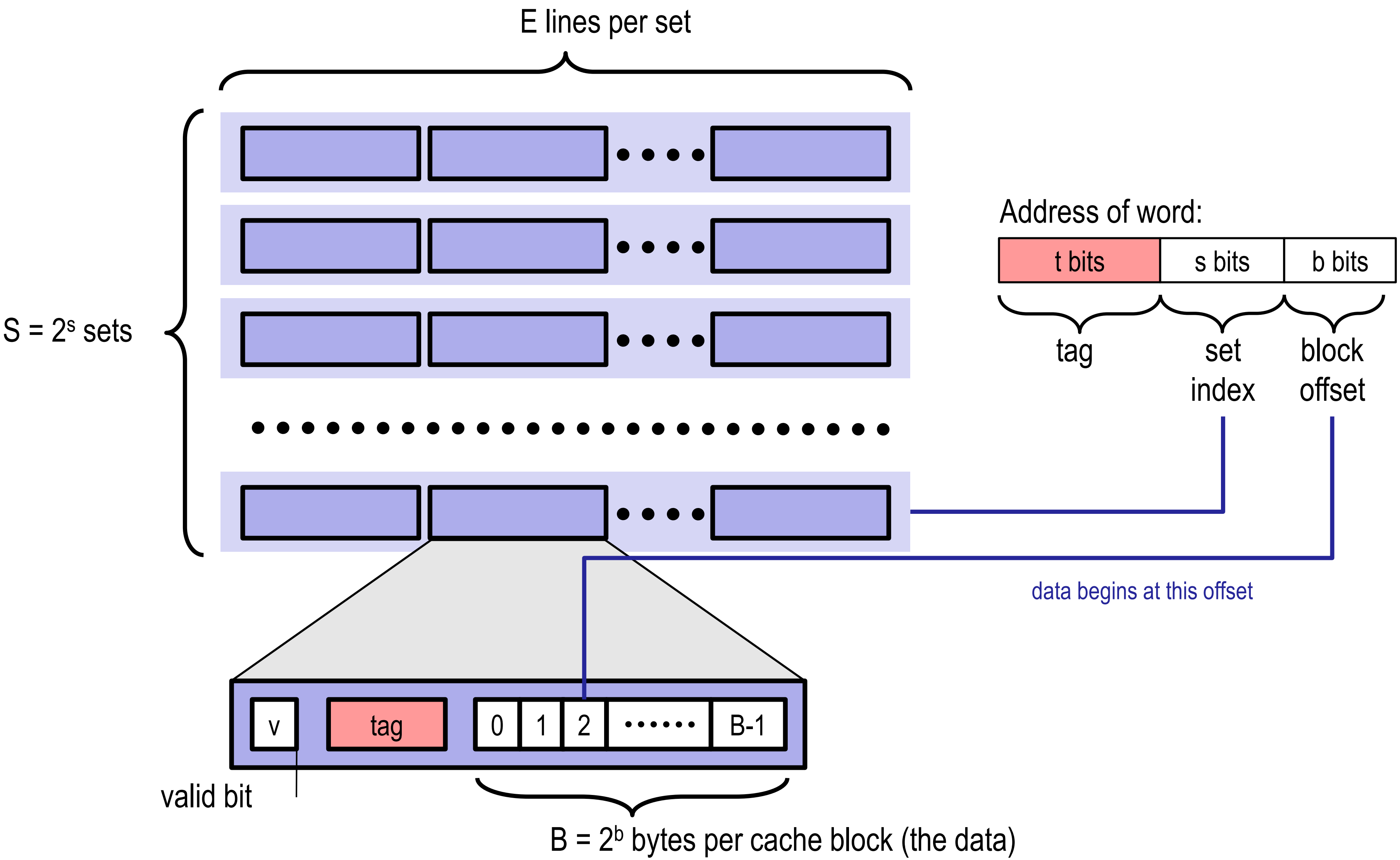
- **A cache is a set of 2^s *cache sets***

- **A *cache set* is a set of E *cache lines***
  - E is called associativity
  - If E=1, it is called “direct-mapped”

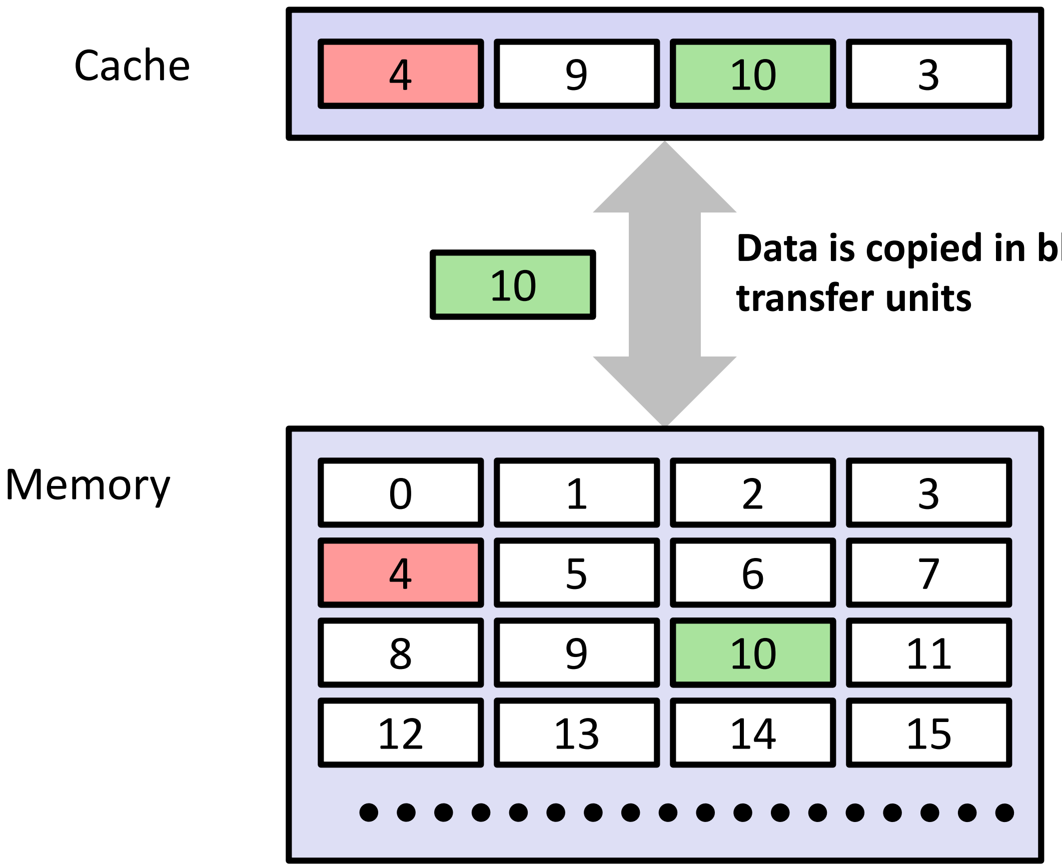
- **Each *cache line* stores a block**
  - Each block has B = 2^b bytes

- **Total Capacity = S*B*E**

# Visual Cache Terminology



# General Cache Concepts

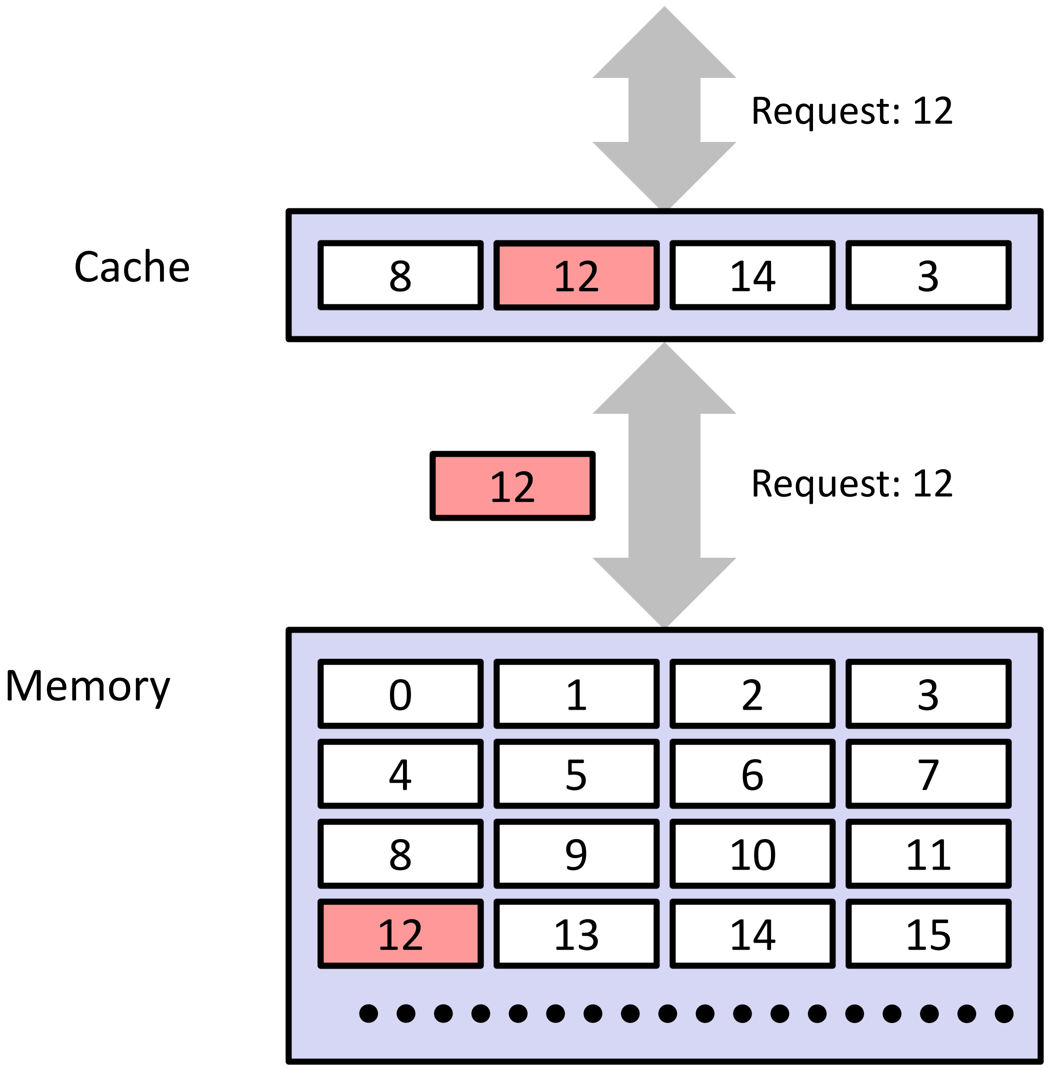



**Smaller, faster, more expensive memory caches a subset of the blocks**

**Data is copied in block-sized transfer units**

**Larger, slower, cheaper memory viewed as partitioned into “blocks”**

# General Cache Concepts: Miss




***Data in block b is needed***

***Block b is not in cache:***
***Miss!***

***Block b is fetched from*** *memory*

***Block b is stored in cache***
- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

# General Caching Concepts: Types of Cache Misses

- **Cold (compulsory) miss**
  - The first access to a block has to be a miss

- **Conflict miss**
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block
    - E.g., Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time

- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache

# Cache Simulator

- **A cache simulator is NOT a cache!**
  - Memory contents NOT stored
  - Block offsets are NOT used – the b bits in your address don't matter.
  - Simply **count** hits, misses, and evictions

- **Your cache simulator needs to work for different s, b, e, given at run time.**

- **Use LRU – Least Recently Used replacement policy**
  - Evict the least recently used block from the cache to make room for the next block.
  - Queues ? Time Stamps ?

# Cache structure

- **A cache is just 2D array of *cache lines*:**
  - struct cache_line cache[S][E];
  - S = 2^s,  is the number of sets
  - E is associativity

- **Each cache_line has:**
  - Valid bit
  - Tag
  - LRU counter ( only if you are not using a queue )

# getopt

- **getopt() automates parsing elements on the unix command line If function declaration is missing**
  - Typically called in a loop to retrieve arguments
  - Its return value is stored in a local variable
  - When getopt() returns -1, there are no more options

- **To use getopt, your program must include the header file**
  **#include <unistd.h>**

- **If not running on the shark machines then you will need #include <getopt.h>.**
  - Better Advice: Run on Shark Machines !

# getopt

- **A switch statement is used on the local variable holding the return value from getopt()**
  - Each command line input case can be taken care of separately
  - “optarg” is an important variable – it will point to the value of the option argument

- **Think about how to handle invalid inputs**

- **For more information,**
  - look at man 3 getopt
  - http://www.gnu.org/software/libc/manual/html_node/Getopt.html

# `getopt` Example

```
int main(int argc, char** argv){
    int opt,x,y;
    /* looping over arguments */
    while(-1 != (opt = getopt(argc, argv, “x:y:"))){
        /* determine which argument it’s processing */
        switch(opt) {
            case 'x':
                x = atoi(optarg);
                break;
            case ‘y':
                y = atoi(optarg);
                break;
            default:
                printf(“wrong argument\n");
                break;
        }
    }
}
```

- Suppose the program executable was called “foo”. Then we would call “./foo -x 1 –y 3“ to pass the value 1 to variable x and 3 to y.

# `fscanf`

- **The fscanf() function is just like scanf() except it can specify a stream to read from (scanf always reads from stdin)**
  - parameters:
    - A stream pointer
    - format string with information on how to parse the file
    - the rest are pointers to variables to store the parsed data
  - You typically want to use this function in a loop. It returns -1 when it hits EOF or if the data doesn't match the format string
- **For more information,**
  - man fscanf
  - http://crasseux.com/books/ctutorial/fscanf.html
- **fscanf will be useful in reading lines from the trace files.**
  - L 10,1
  - M 20,1

# fscanf example

```
FILE * pFile; //pointer to FILE object

pFile = fopen ("tracefile.txt",“r"); //open file for reading

char identifier;
unsigned address;
int size;
// Reading lines like " M 20,1" or "L 19,3"

while(fscanf(pFile,“ %c %x,%d”, &identifier, &address, &size)>0)
{
   // Do stuff
}

fclose(pFile); //remember to close file when done
```

# Malloc/free

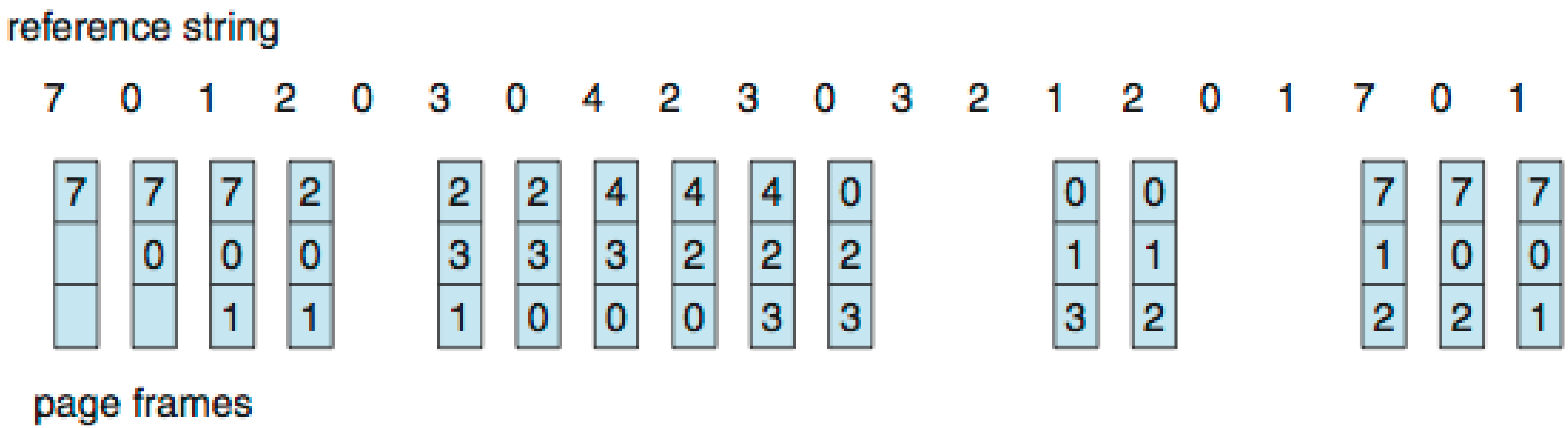
- **Use malloc to allocate memory on the heap**
- **Always free what you malloc, otherwise may get memory leak**
  - some_pointer_you_malloced = malloc(sizeof(int));
  - Free(some_pointer_you_malloced);
- **Don’t free memory you didn’t allocate**

# Page Replacement Algorithms

- **When cache is full, a cached data should be replaced with the new data**
  - The new data is referred right now and just used
  - To find a space to hold the new data, choose a data which was previously cached as a victim
    - The victim data is least likely used
- **Algorithms**
  - First-In-First-Out (FIFO) Algorithm
  - Optimal Algorithm
  - Least Recently Used (LRU) Algorithm
- **Implementation of the algorithms in the cachelab**
  - LRU should be implemented by default
  - FIFO and optimal algorithms could be implemented for extra credits

# First-In-First-Out (FIFO) Algorithm

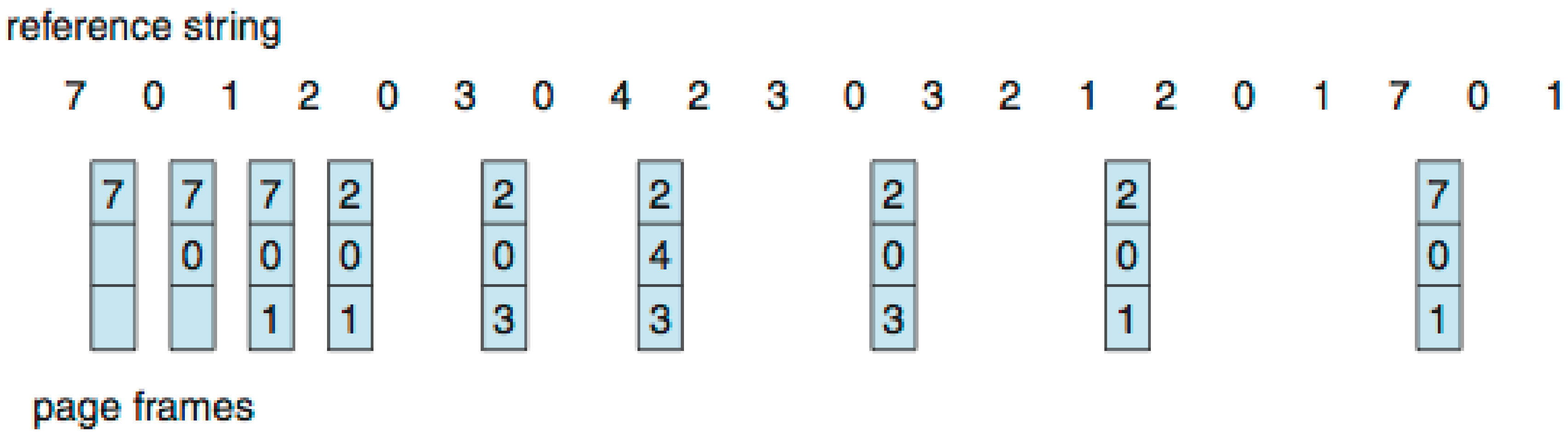
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- Cache: Direct Mapped, 3 sets are instantiated
  - Note that the number of sets must be $2^s$ in your actual cachelab.
  - Page: data block




  - Totally 15 cache misses occurred

# Optimal Algorithm

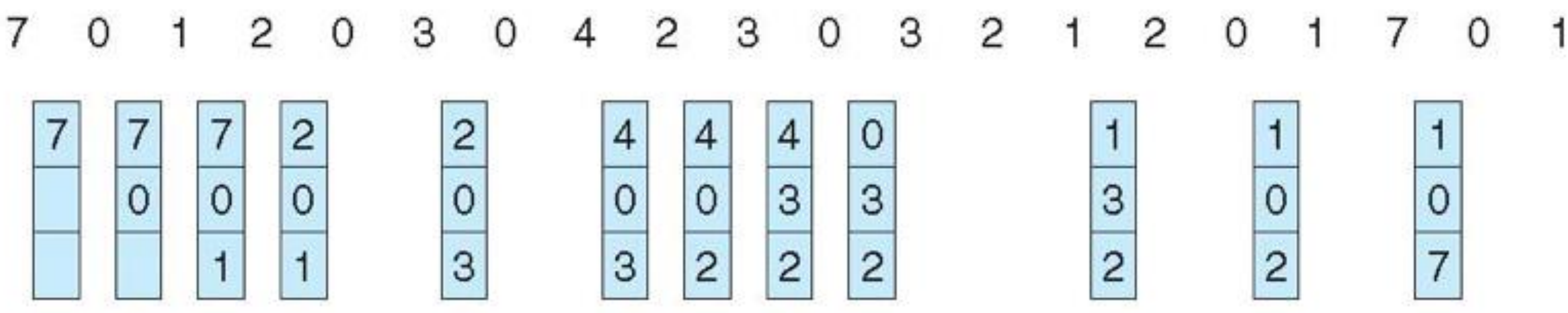
- **Replace page(data block) that will not be used for longest period of time**
- Used for measuring how well your algorithm performs




  - Totally 9 cache misses occurred
- **Issue: how do you know this?**
  - Can't read the future

# Least Recently Used (LRU) Algorithm

- **Use past knowledge rather than future**
  - Replace page that has not been used in the most amount of time
- Associate time of last use with each page




  - 12 misses – better than FIFO but worse than OPT
- **Generally good algorithm and frequently used**
- **But how to implement?**

# LRU Implementation

- **Counter implementation**
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- **Stack implementation**
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

# LRU Implementation Example with Stack