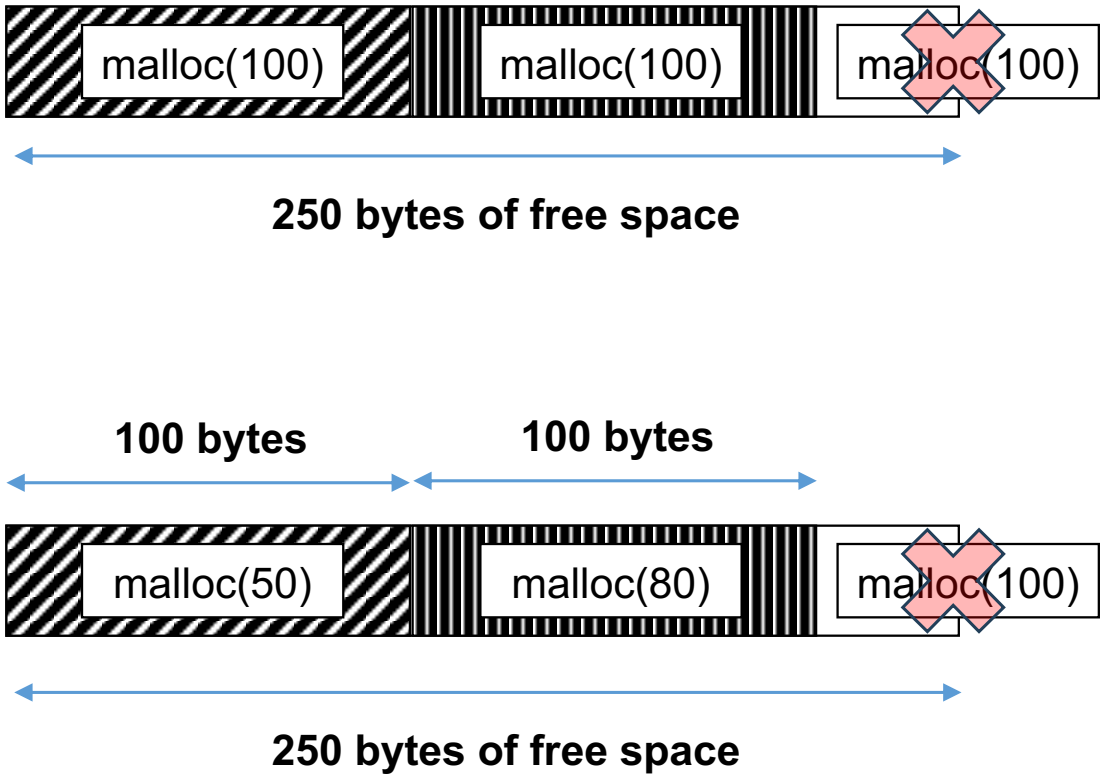
Lecture 26: End Semester Review

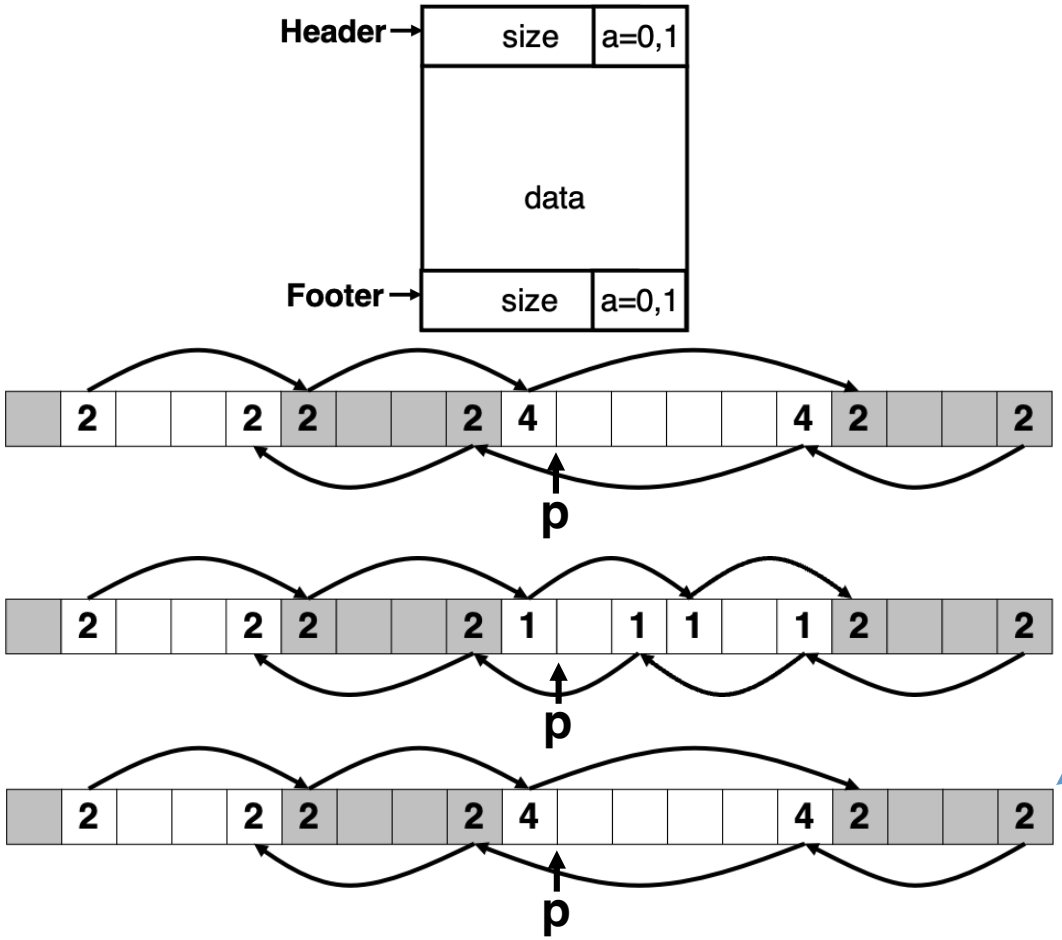
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Fragmentation



- Segments of memory can become unusable due to the result of allocation scheme
- Two types of fragmentation
  - External fragmentation
    - Memory remains unallocated
  - Internal fragmentation
    - Memory is allocated but unused
    - Fixed allocation sizes

Malloc using Implicit List



- Each memory block has a header/footer
- Blocks connected as linked list
- Block splitting during malloc
  - E.g., p=malloc(1)
- Block coalescing during free
  - E.g., free(p)

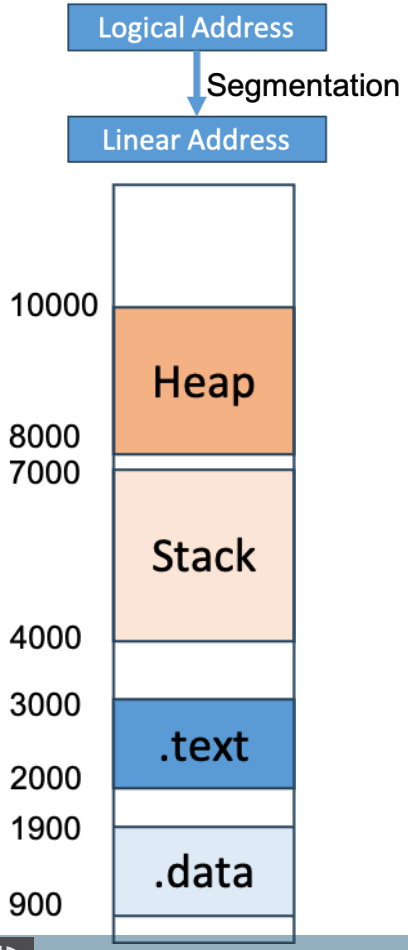
Address Spaces

- Using some mapping technique to translate the address seen by the user to the actual address on the RAM gives two different views of addresses
  - Virtual address space (as seen by the process)
  - Physical address space (as seen by the OS depending on DRAM size)

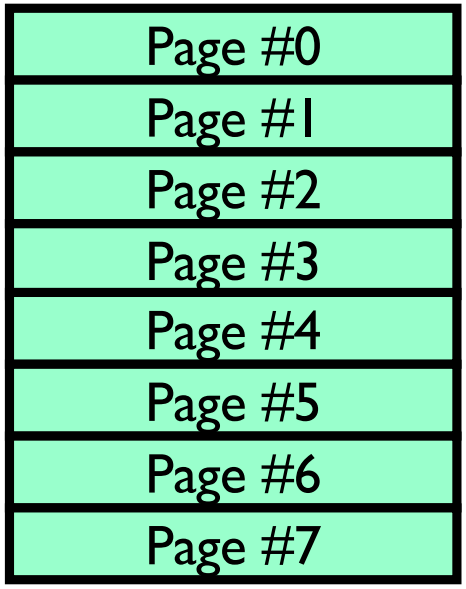
- Virtual address space: Set of $N = 2^n$ virtual addresses
  {0, 1, 2, 3, …, N-1}

- Physical address space: Set of $M = 2^m$ physical addresses
  {0, 1, 2, 3, …, M-1}

Logical Address to Linear Address



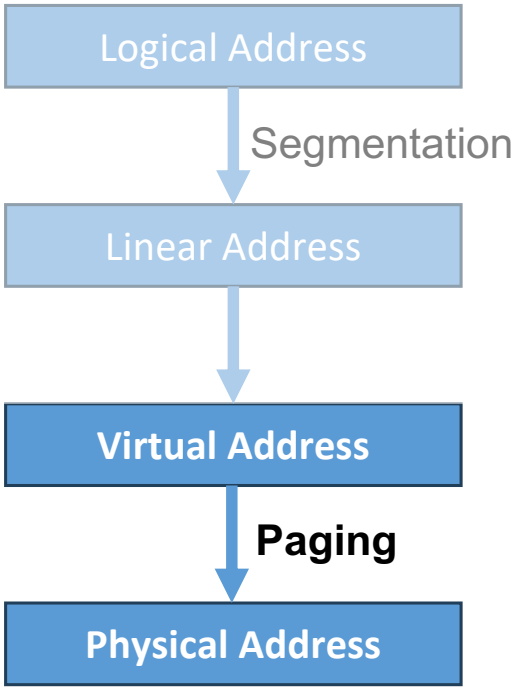
Paging



Physical Memory

- Segmentation is easy to implement but it has major drawbacks because segments could have variable sized memory allocation
  - Can lead to **external** fragmentation
  - To allocate space for a new process, segments of some other process has to be swapped to the disk (huge overhead)
- Paging solves both the above issues by dividing the available memory into small and fixed size blocks (pages)
  - The memory can then be viewed as an array of fixed sized slots
- Default page size on Linux is 4096 bytes (4KB)
  - Total number of pages in **32-bit** addressing mode is 2^{32}/4KB, i.e., 2^{20}
  - Total number of pages in **64-bit** addressing mode is 2^{64}/4KB, i.e., 2^{52}

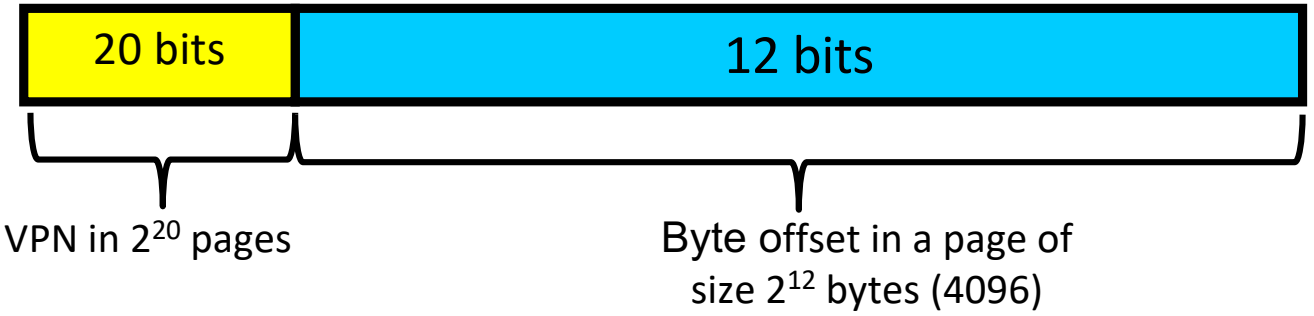
Segmentation Along with Paging



- Segmentation cannot be disabled in x86 processors, but address translation is complex using the segmentation-based approach
  - Hence, to simplify the address translation, segment registers are set to have the same base (0) and bound (2^{32}-1) values (Lecture #16)
- Segmentation converts the logical address to linear address, which is then converted into virtual address to support paging
- However, as the base and bound of each segment registers maps to entire 2^{32} memory size (4GB), logical address will be the same as virtual address

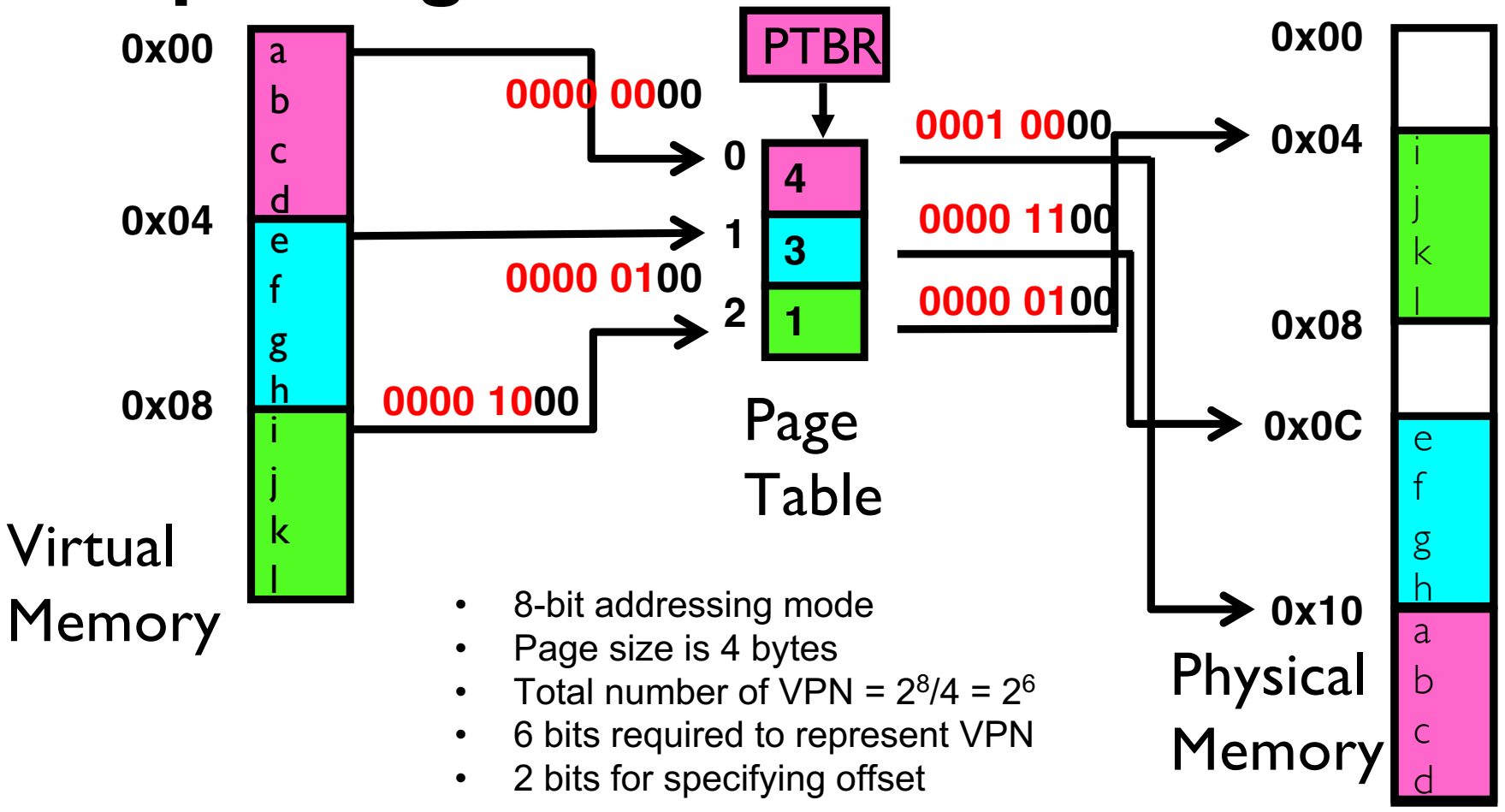
Two Components of a Virtual Address

- Dedicate few bits to identify VPN and the remaining bits to identify the offset into that VPN
- How many bits required to represent a VPN for pages of N bytes in a virtual memory of M bytes (total M/N pages)?
  - M=4GB (2^{32}) and N=4KB
    - Pages = $2^{32}/4096 = 2^{20}$ and bits required to identify 2^{20} pages are 20



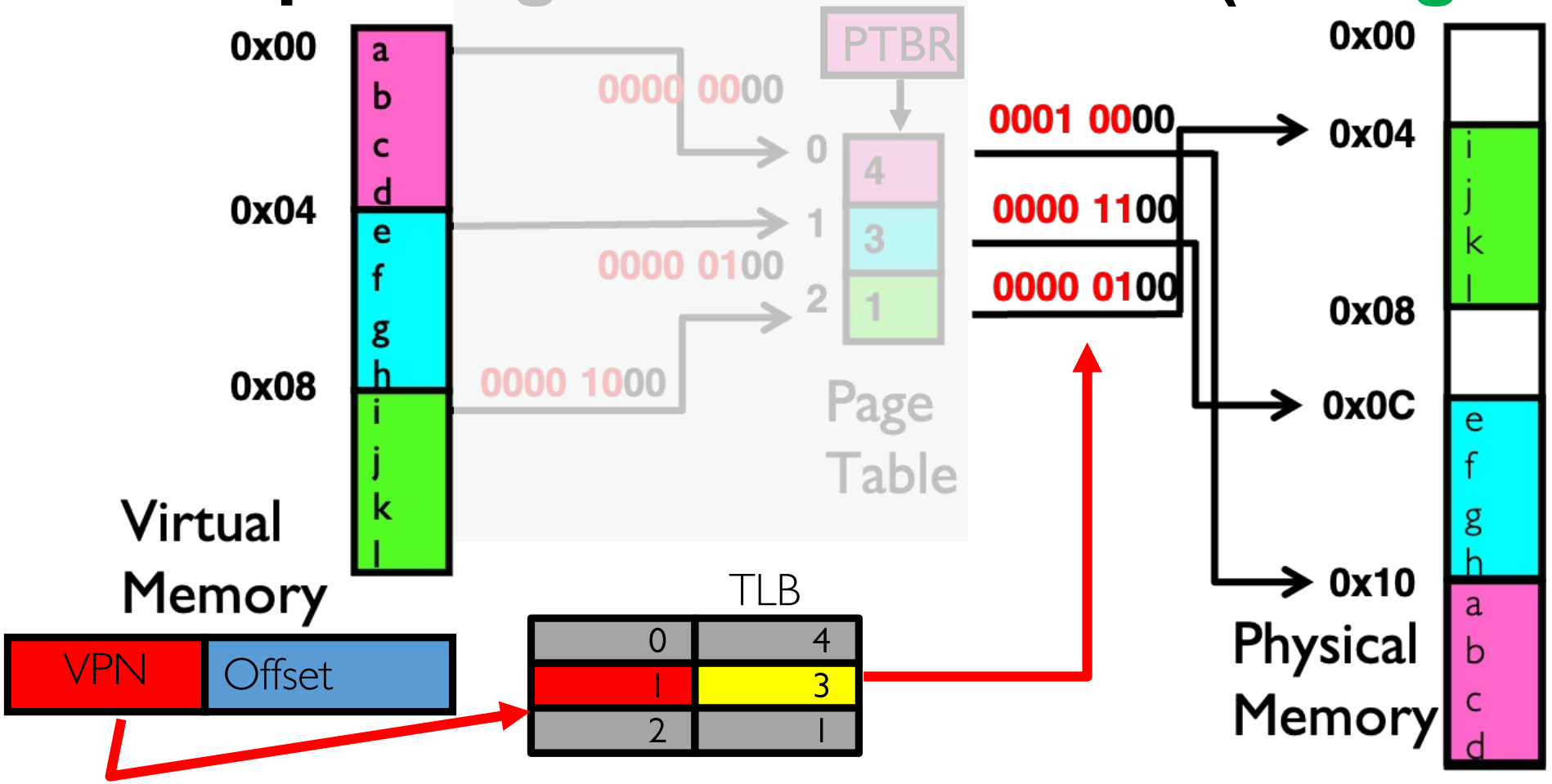
32-bit addressing with page size of 4KB

A Simple Page Table in Action

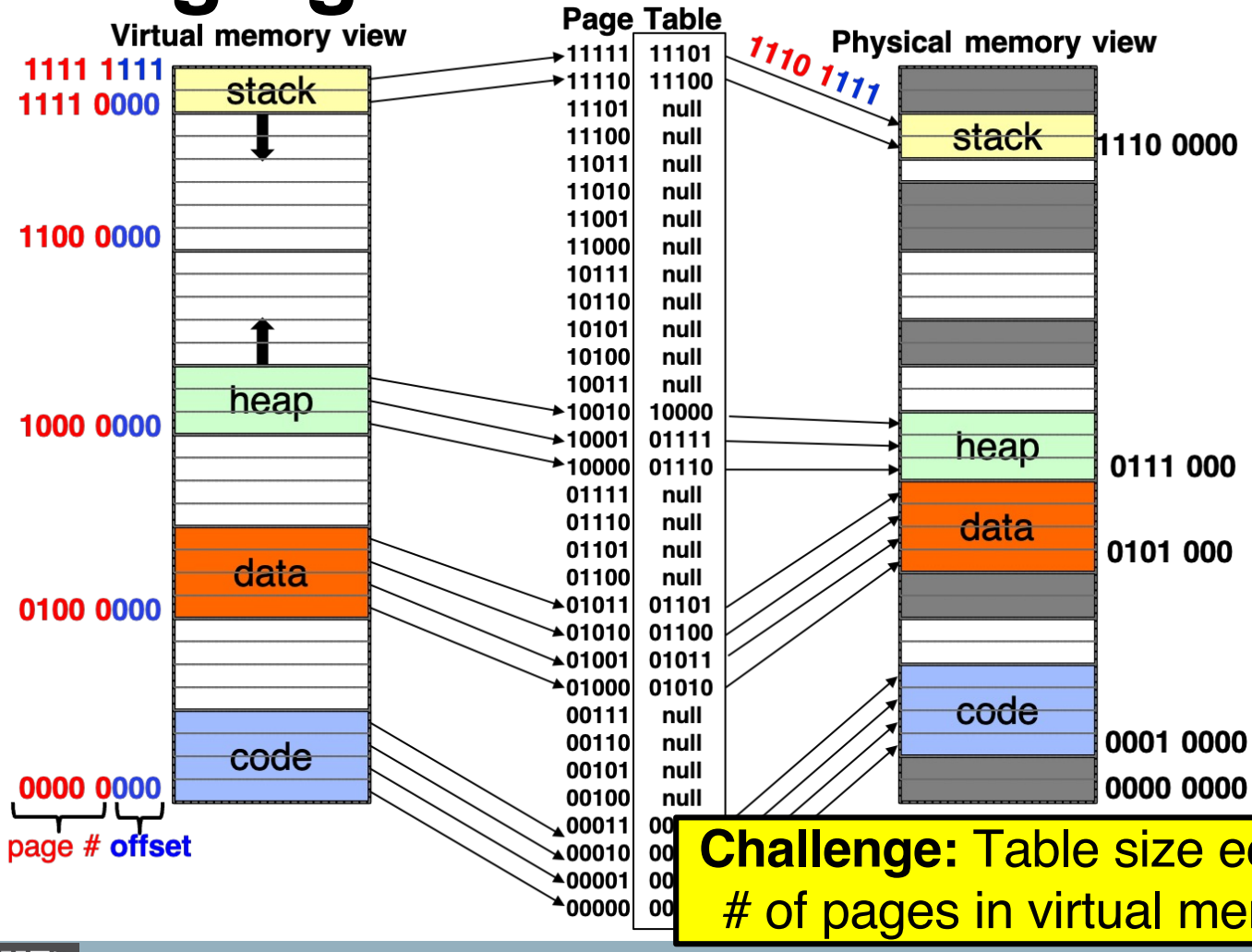


- 8-bit addressing mode
- Page size is 4 bytes
- Total number of VPN = $2^8/4 = 2^6$
- 6 bits required to represent VPN
- 2 bits for specifying offset

A Simple Page Table in Action (Using TLB)



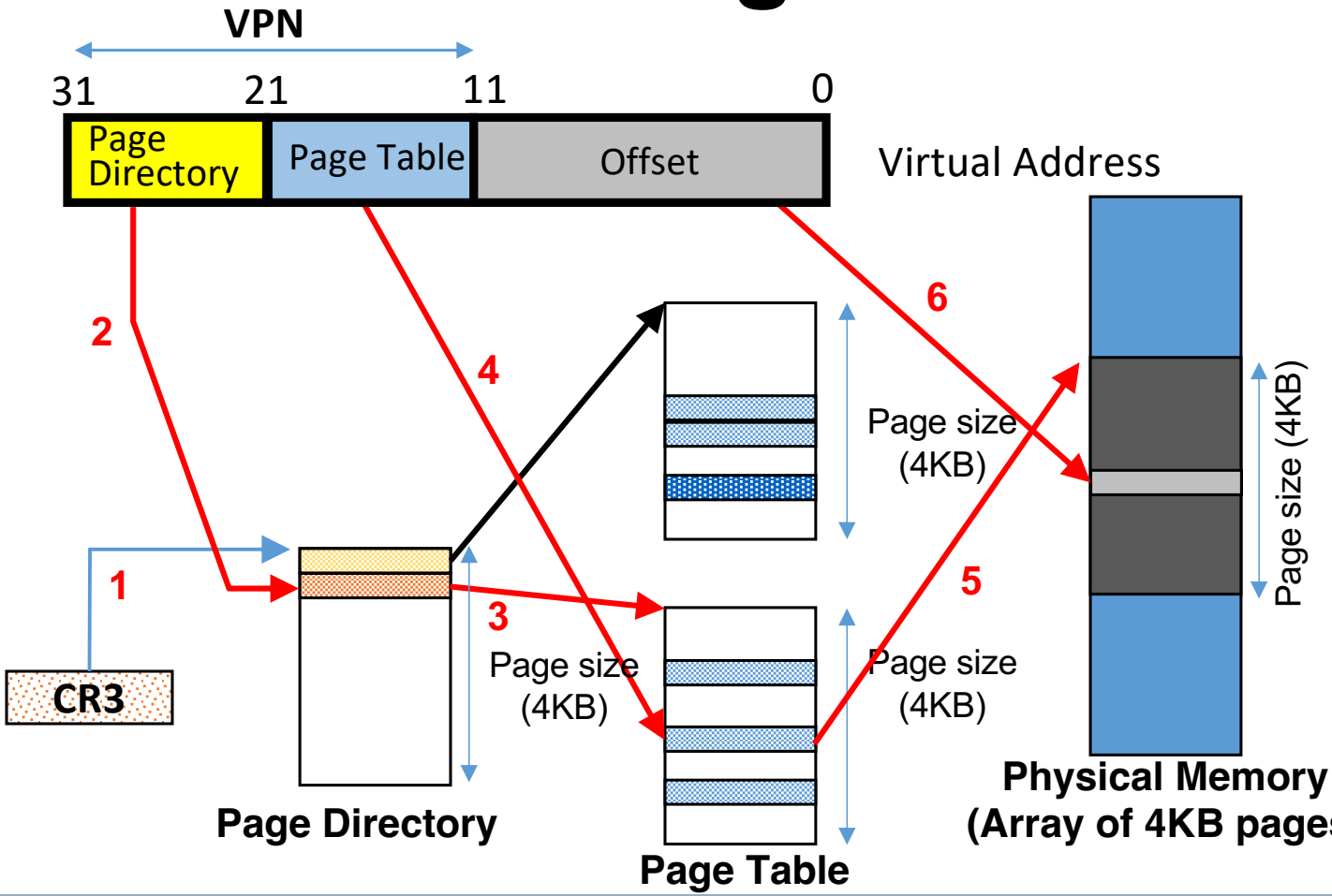
Paging



Challenge: Table size equal to # of pages in virtual memory!

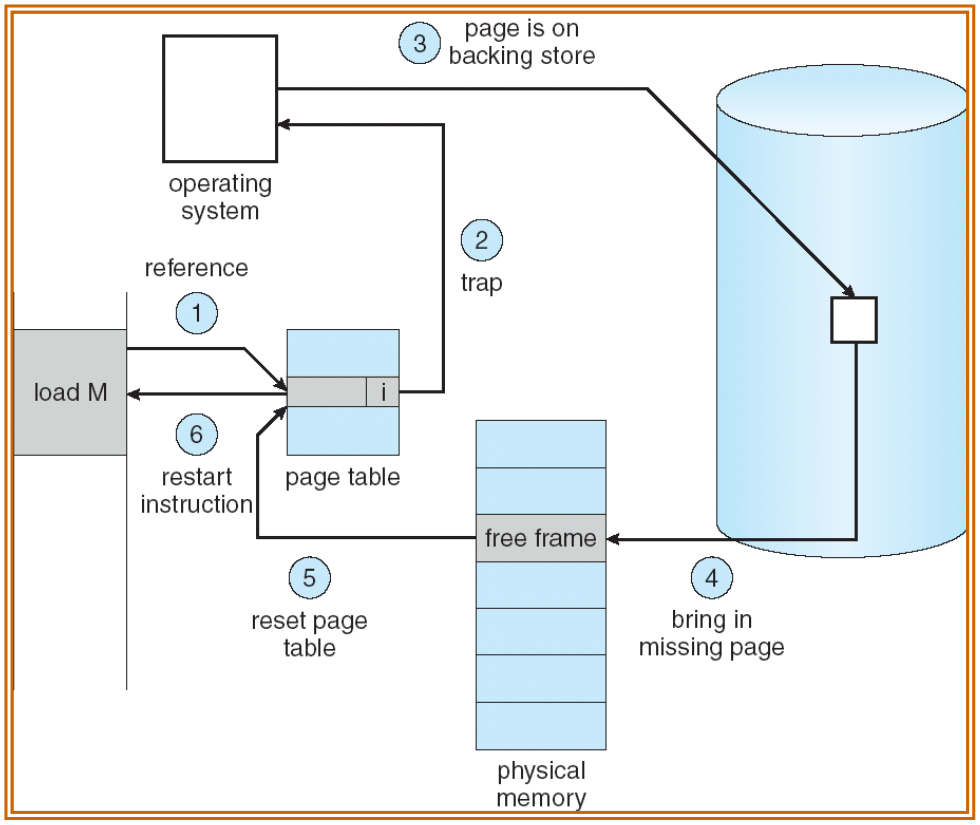
- Recall, process address space is not contiguous
  - Inter-segment memory is not contiguous
  - Only intra-segment virtual memory is contiguous (the physical memory may/may-not be contiguous)
- Page table until now is an **array** where the length of the array will be the total number of VPN being used in the process address space
  - Not all the VPNs will be used as the segments are not contiguous (there could be gaps between them)

Two-Level Page Table in x86



- VA stores page directory index, page table index, and address offset in PP
- CR3 register stores the starting address of the Page Directory
- Each PDE contains the Page Frame Number and a status bit
  - Status=0 if there is not a single valid entry in the PFN, otherwise status =1 (valid PFN only if status=1)
- Page Table Entry (PTE) is found from the PDE
- Physical Page Frame Number (PFN) is found from the PTE

Demand Paging



- It the mechanism to provide the illusion of infinite memory
- OS keeps some amount of DRAM always free by deciding some upper cap
  - Whenever the upper cap memory level is breached, unused pages are moved out to disk
- OS brings the pages into memory back into DRAM from disk when it is accessed
  - Happens during page fault
- PTE makes demand paging implementable using the Present bit
  - P=Valid ⇒ Page in memory, PTE points at physical page
  - P=Not Valid ⇒ Page not in memory

IPC Within a Multicore Processor

- Inter-process communication in **shared memory**
  - Transfer of control from user space to kernel space and vice-versa
- Complicated IPC mechanism for communication
- OS has to reserve extra memory / resources
  - Separate heap, stack, .text segment, etc. for each process
  - Same copy of .text segment in each process
- Separate page table for each process
- Cost of IPC may exceed the cost of actual computation!

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Producer Consumer using Pthreads

```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

Consumer(s)

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

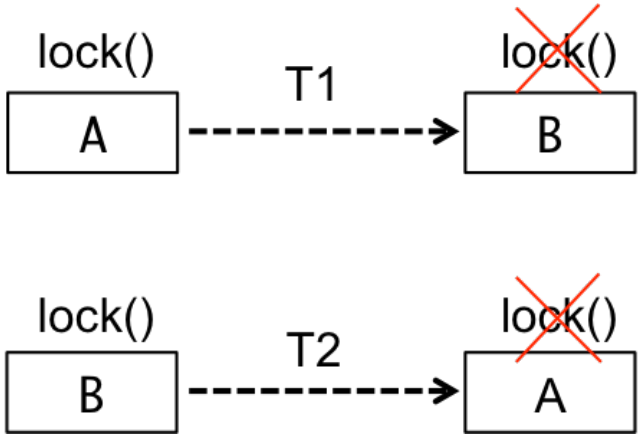
Producer

- pthread_cond_**wait** causes the current thread to relinquish the CPU and wait until another thread invokes the **signal** or the **broadcast**
- Upon call for **wait**, the thread releases ownership of the mutex and waits until another thread signals the waiting threads

Deadlock

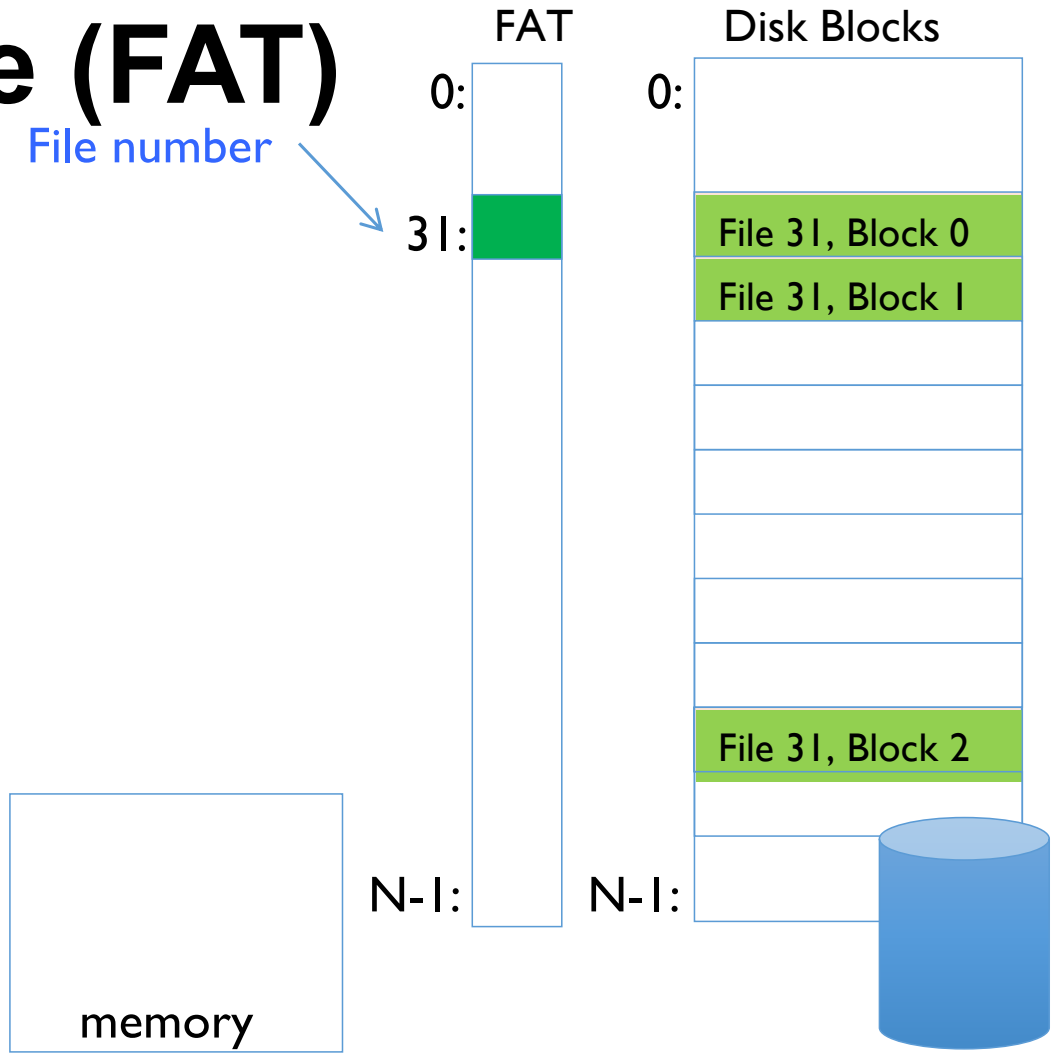
```
class Transfer {
    Account A, B;
    int amount;

    void run() {
        A.lock();
        B.lock();
        A.debit(amount);
        B.credit(amount);
        A.unlock();
        B.unlock();
    }
}
```



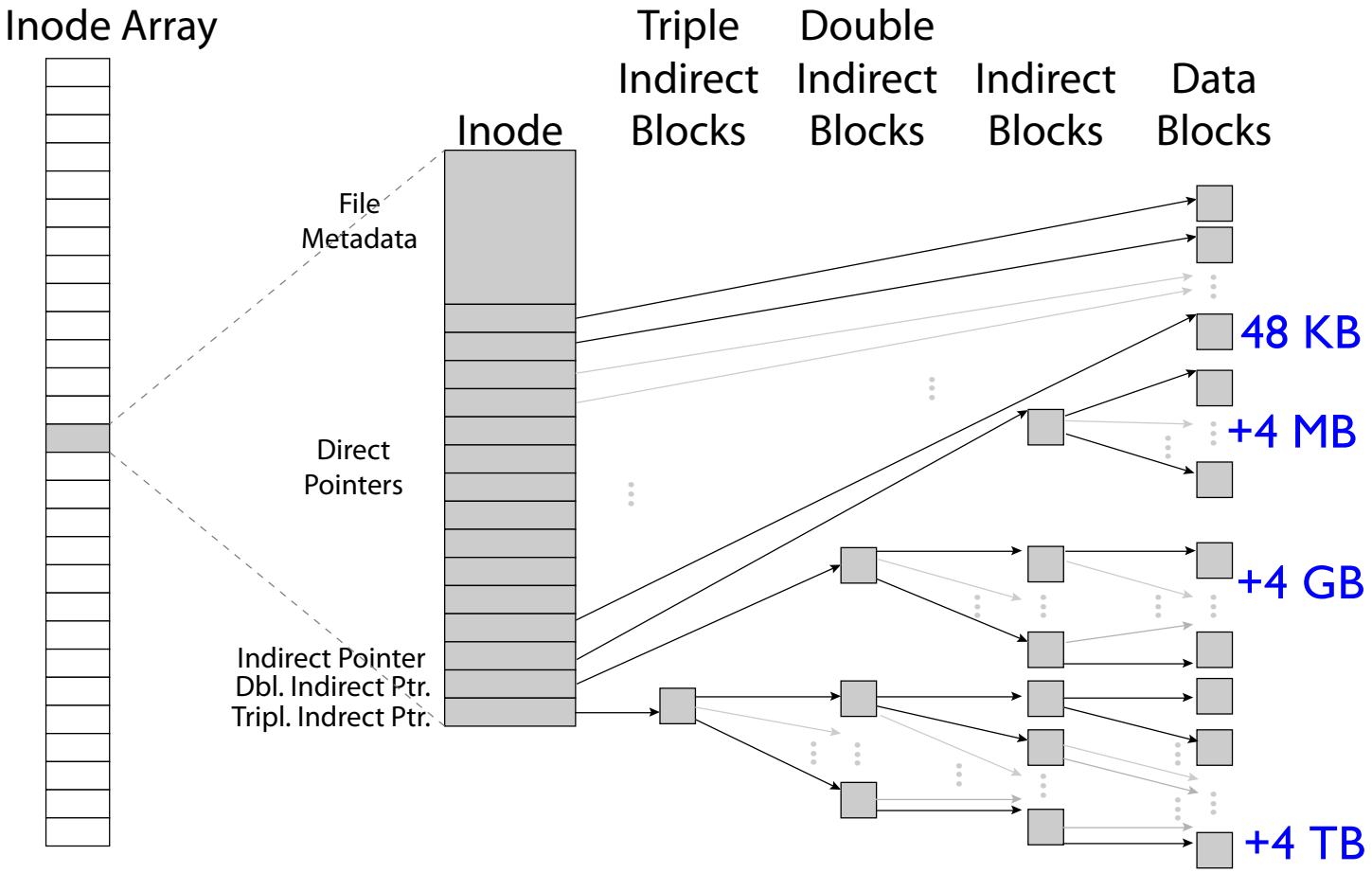
- Deadlock occurs when multiple threads need the same locks but obtain them in different order
  - It could be avoided by using lock ordering
    - Ensure that all locks are taken in same order by any thread

File Allocation Table (FAT)



Inode Structure

Large files: 2,3,4 level indirect pointers



I hope you enjoyed the course..

All the best for your exams !