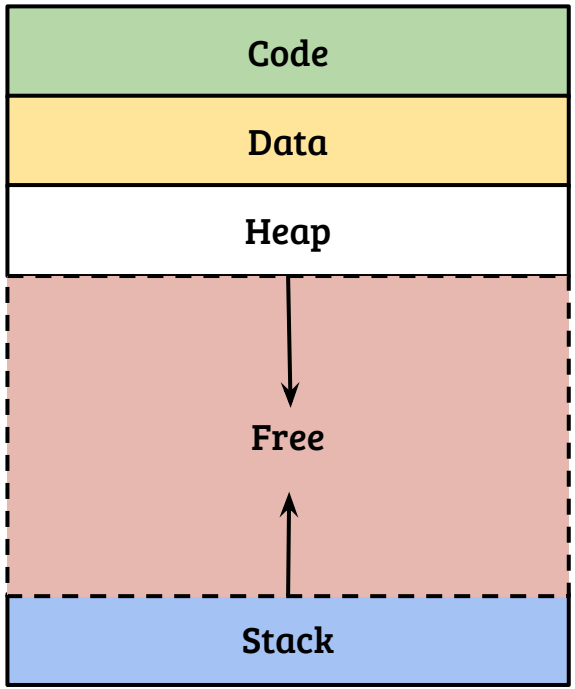
CS330: Operating Systems

Virtual Memory: Address translation

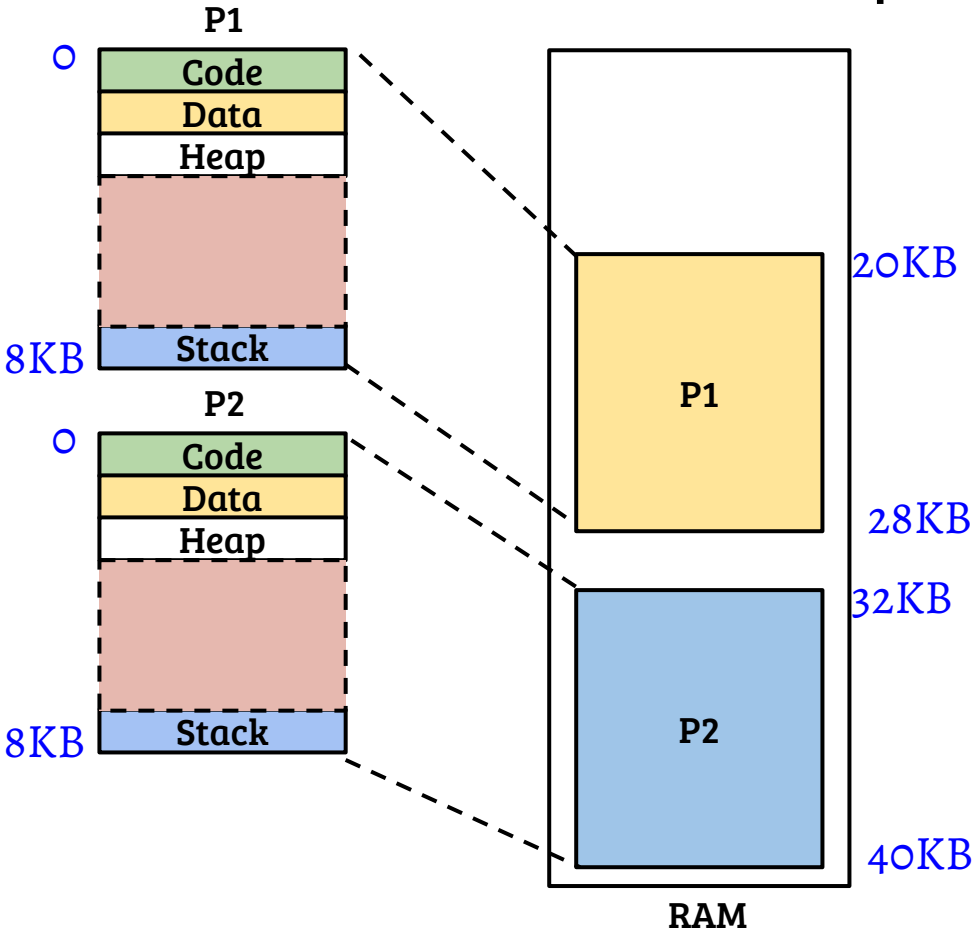
Recap: Process address space



- Address space abstraction provides the same view of memory to *all processes*
  - Address space is virtual
  - OS enables this virtual view
- User can organize/manage virtual memory using OS APIs
  - No control on physical memory!

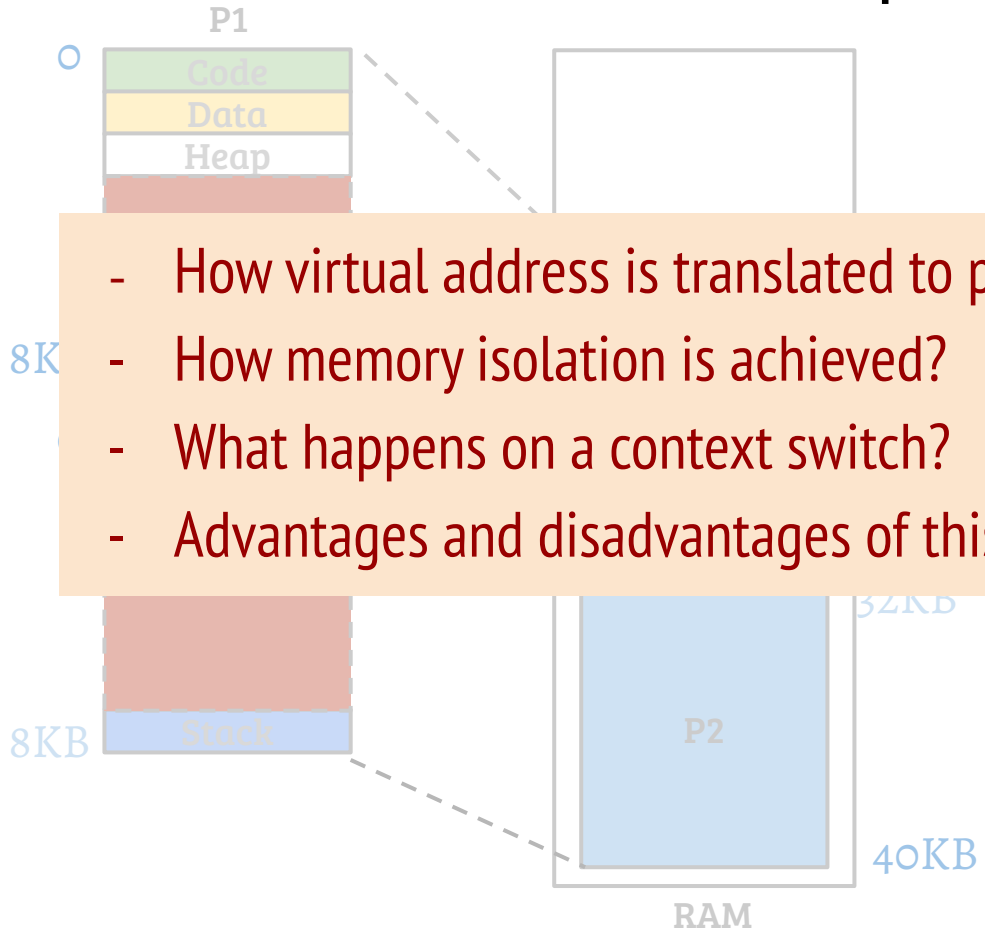
Agenda: Virtual to physical address translation

Translation at address space granularity



- Physical memory of same size as the address space size is allocated to each process
- Physical memory for a process can be at any address, but should be contiguous

Translation at address space granularity



- Physical memory of same size as

- How virtual address is translated to physical address?
- How memory isolation is achieved?
- What happens on a context switch?
- Advantages and disadvantages of this scheme

contiguous

ISA: commonly used addressing modes (x86)

- At a high-level, instructions contains two parts: opcode and operand
  - ISA defines binary encoding of opcodes, mode and register operands (more complex in practice)
- Operands can be specified in multiple ways
  - Register: mov %rcx, %rax
  - Immediate: mov $5, %rax
  - Absolute: mov 8000000, %rax
  - Indirect: mov (%rcx), %rax
  - Displacement: mov -16(%rbp), %rax

X86 ISA: examples

- Access local variables using %rbp (examples)
- long a = 100, b =20, c;
  - mov $100, -8(%rbp); mov $20, -16(%rbp)
- c = a + b;
  - mov -8(%rbp), %rax; mov -16(%rbp), %rcx;
  - add %rcx, %rax; mov %rax, -24(%rbp)
- PC relative jump/call
  - jmp 0x20(%rip)
  - call -0x20(%rip)

Role of the compiler

Simple function

```
func( )
{
  long a = 100;
  a+ = 10;
}
```

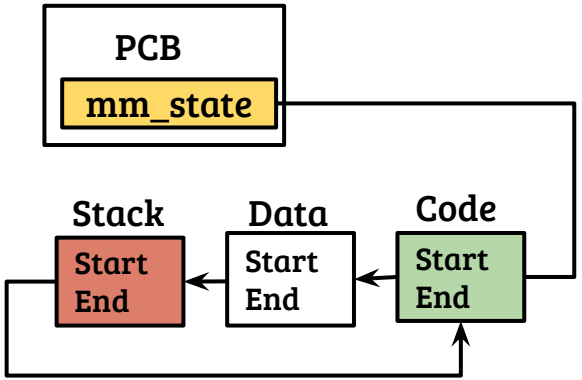
Compiled assembly

```
func:
10:    push %rbp;
12:    mov %rsp, %rbp;
16:    mov $100, -8(%rbp);
20:    mov -8(%rbp), %rax
24:    add $10, %rax
29:    mov %rax, -8(%rbp)
33:    pop %rbp;
35:    ret;
```

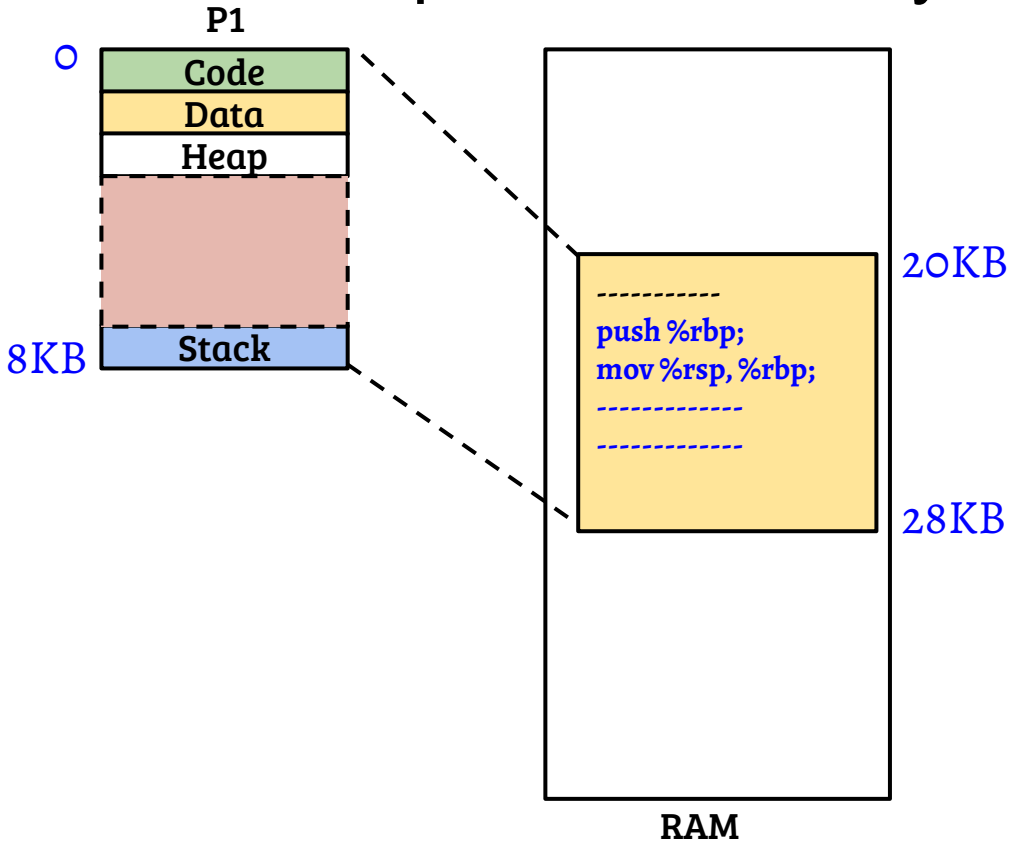
- Compiler can genera
  code assuming starti
  the code address as
- Compiler does not k
  the stack address, bl
  uses the registers (rb

OS during binary load (simplified fork + exec)

```
load_new_executable( PCB *current, File *exe)
{
  verify_executable(exe);
  reinit_address_space(current → mm_state);
  allocate_phys_mem(current);
  load_exe_to_physmem(current, exe);
  set_user_sp(current → mm_state → stack_start);
  set_user_pc(current → mm_state → code_start);
  return_to_user;
}
```

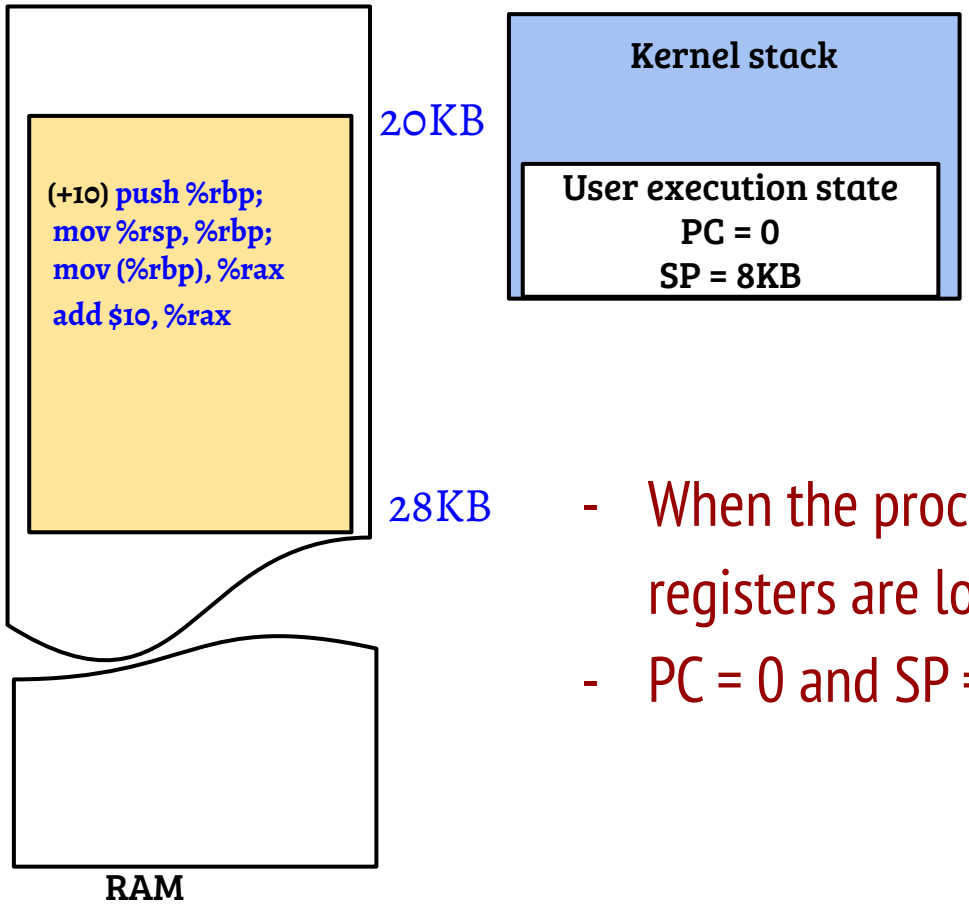


Address space to memory translation



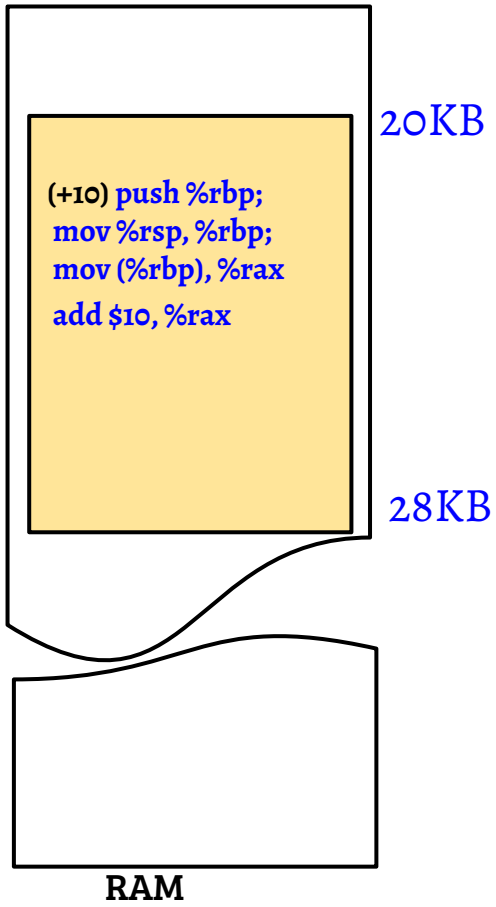
- Physical memory of 8KB is allocated and the code is loaded
- The PCB memory state is updated based on the executable format

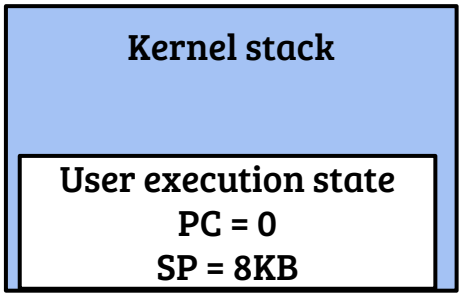
Process state after exec( )

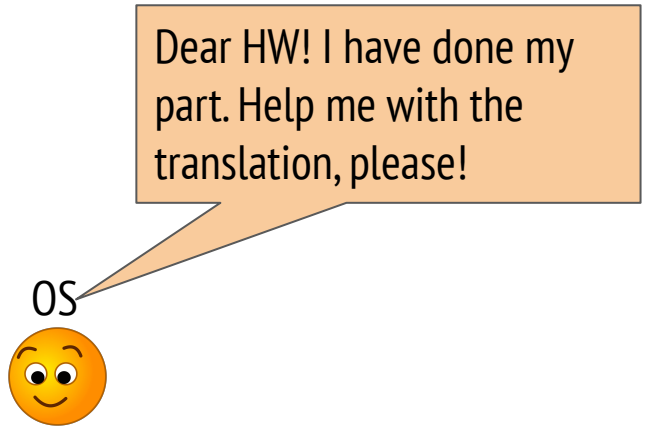


- When the process returns to user space, the registers are loaded with virtual addresses
- PC = 0 and SP = 8KB

Process state after exec( )

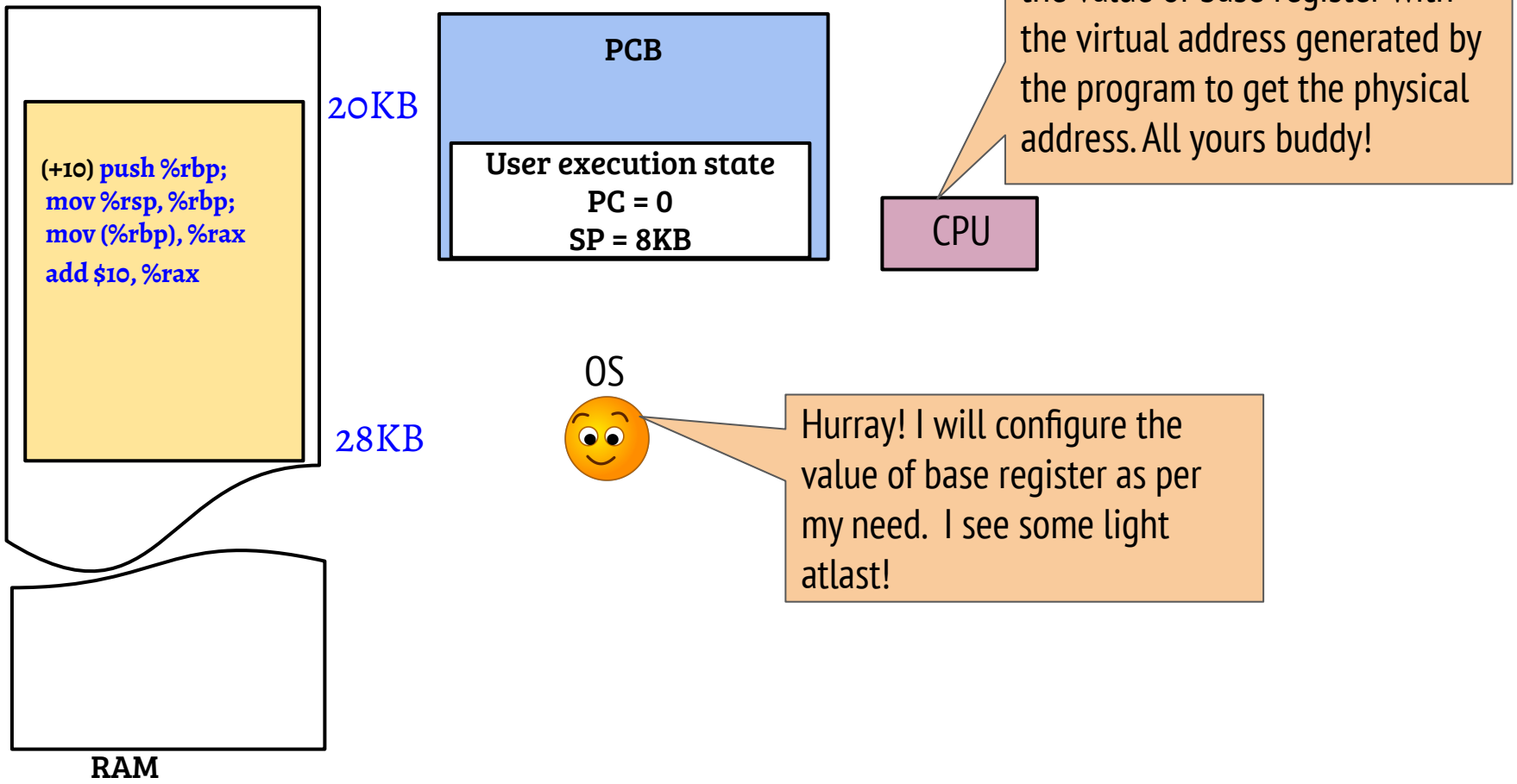




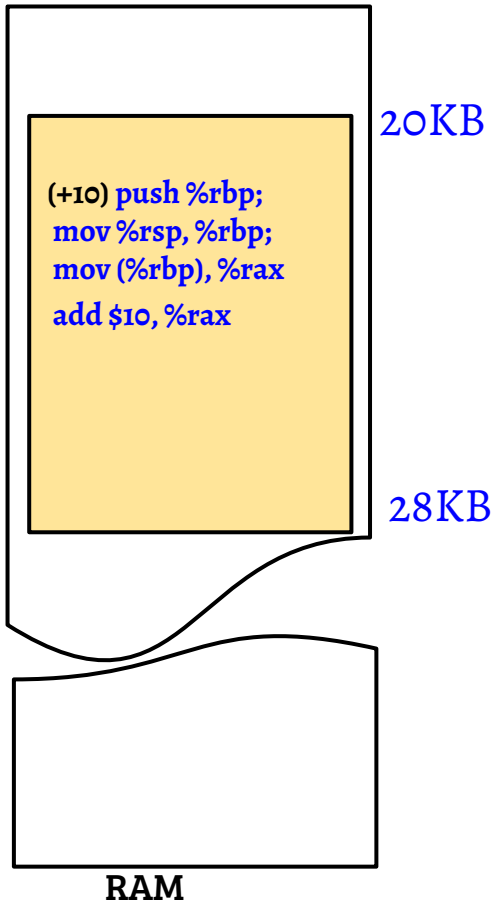


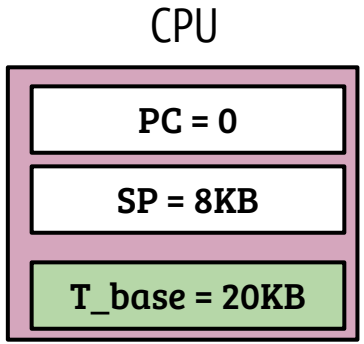
- When the process returns to user space, the registers are loaded with virtual addresses
- Code is loaded into physical memory (@20KB)
- At the start of "func" execution
  - Instruction fetch address is 10
  - SP will be around 8KB

Process state after exec( )



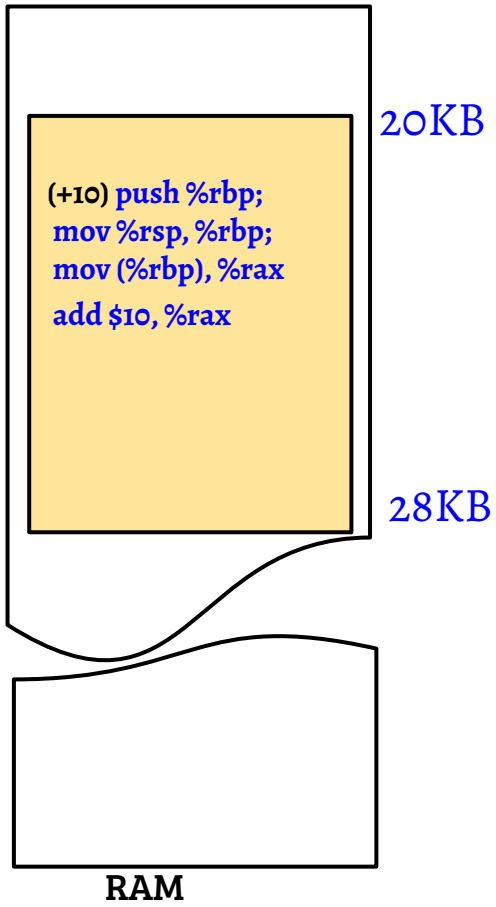
Translation

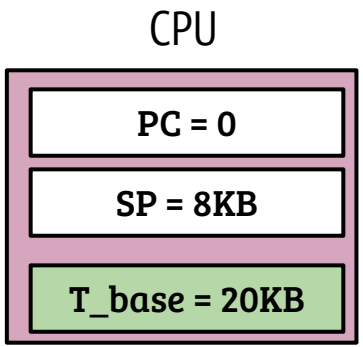


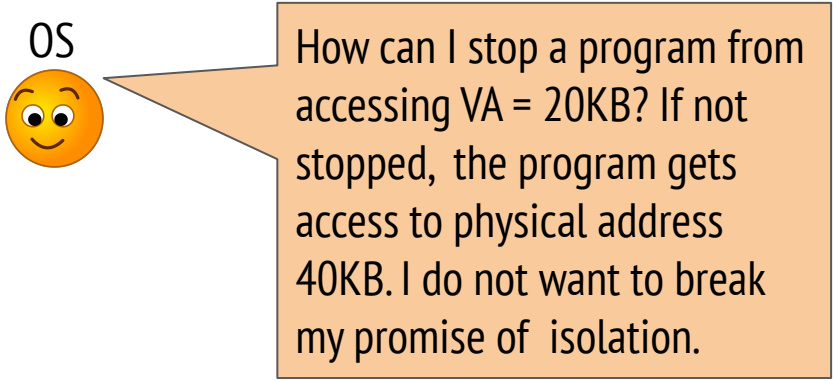


- In this case, base register value should be 20KB
- InsFetch (vaddr = 10) ⇒ InsFetch (paddr = 20KB +10)
- How "push %rbp" works?
- Assuming RSP = 8KB, "push %rbp" results in a memory store at address (8KB - 8)
  - CPU translates the address to (28KB - 8)

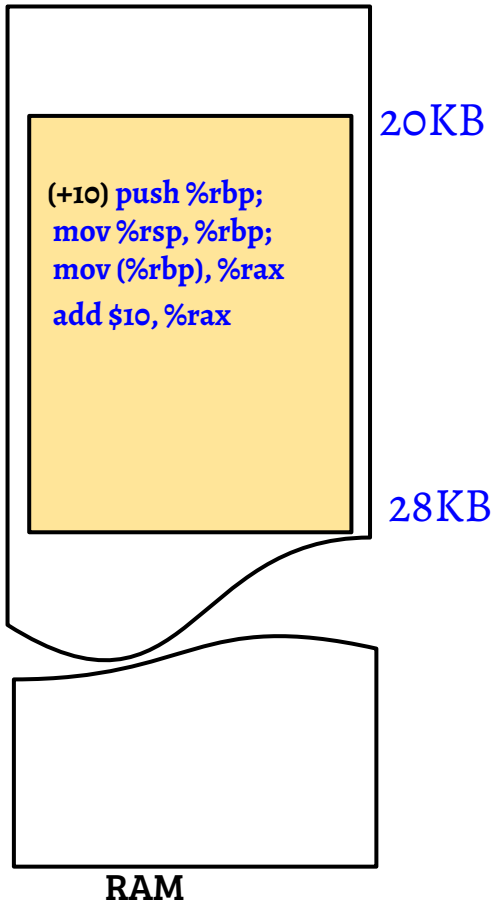
Isolation: How to stop illegal access?

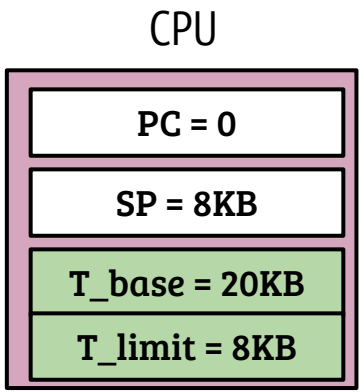






Isolation: How to stop illegal accesses?

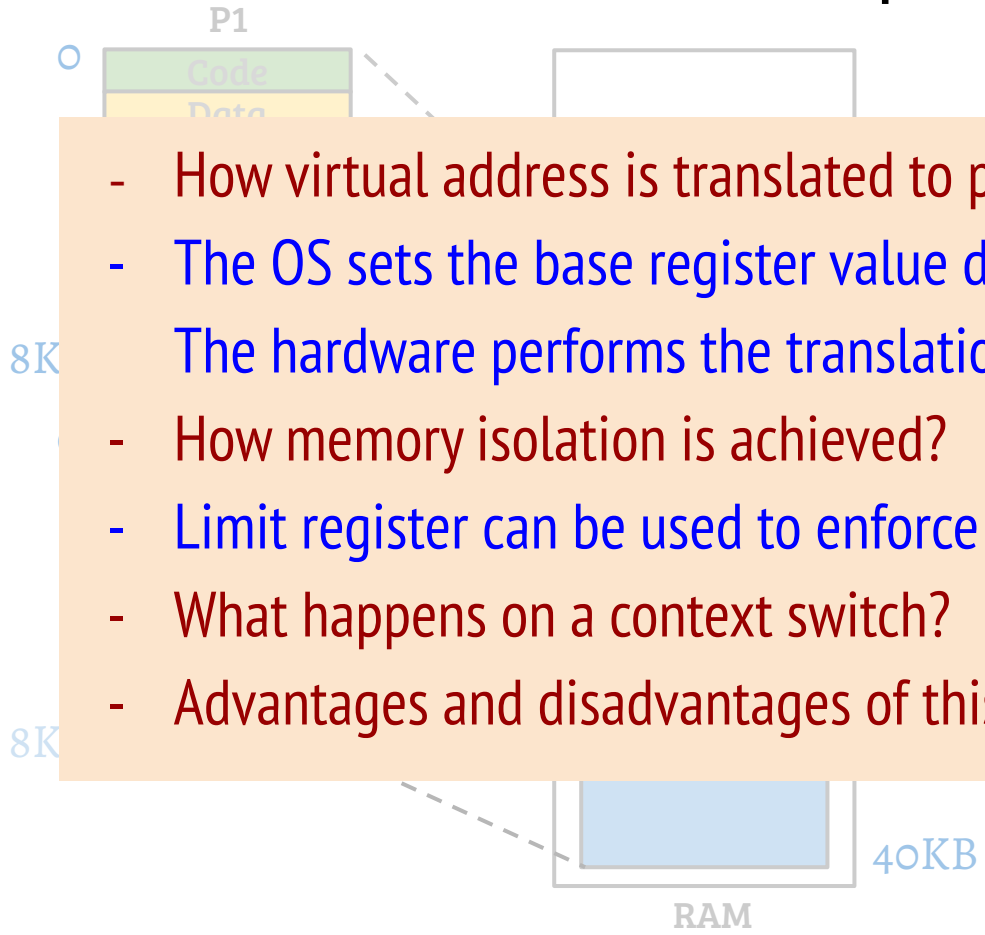




Once a cry baby always a cry baby! I also provide a limit register to enforce the limit during translation. Before you ask, these registers can only be changed from privileged mode

- The hardware raises a fault if some program violates the limit.
- The OS fault handler may kill the process
- (WE-1) T_base and T_limit values across processes

Translation at address space granularity



- How virtual address is translated to physical address?
- The OS sets the base register value depending on the physical location. The hardware performs the translation using the base value.
- How memory isolation is achieved?
- Limit register can be used to enforce memory isolation
- What happens on a context switch?
- Advantages and disadvantages of this scheme

Context switch and translation information

- The base and limit register values can be saved in the outgoing process PCB during context switch
- Loaded from PCB to the CPU when a process is scheduled
- (WE-2) User-to-OS context switching

Translation at address space granularity

- How virtual address is translated to physical address?
- The OS sets the base register value depending on the physical location.
  The hardware performs the translation using the base value.
- How memory isolation is achieved?
- Limit register can be used to enforce memory isolation
- What happens on a context switch?
- Save and restore limit and base registers
- Advantages and disadvantages of this scheme

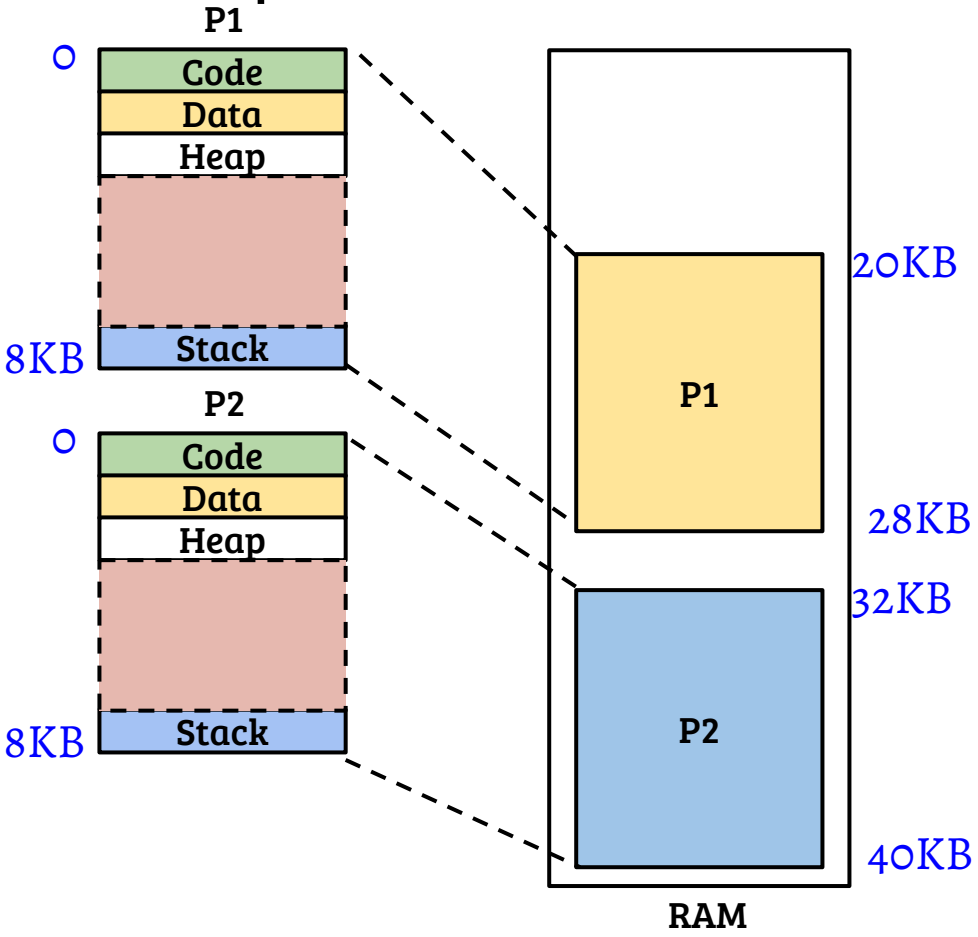
Translation at address space granularity: Issues

- Physical memory must be greater than address space size
  - Unrealistic, against the philosophy of address space abstraction
  - Small address space size ⇒ Unhappy user
- Memory inefficient
  - Physical memory size is same as address space size irrespective of actual usage ⇒ Memory wastage
  - Degree of multiprogramming is very less

CS330: Operating Systems

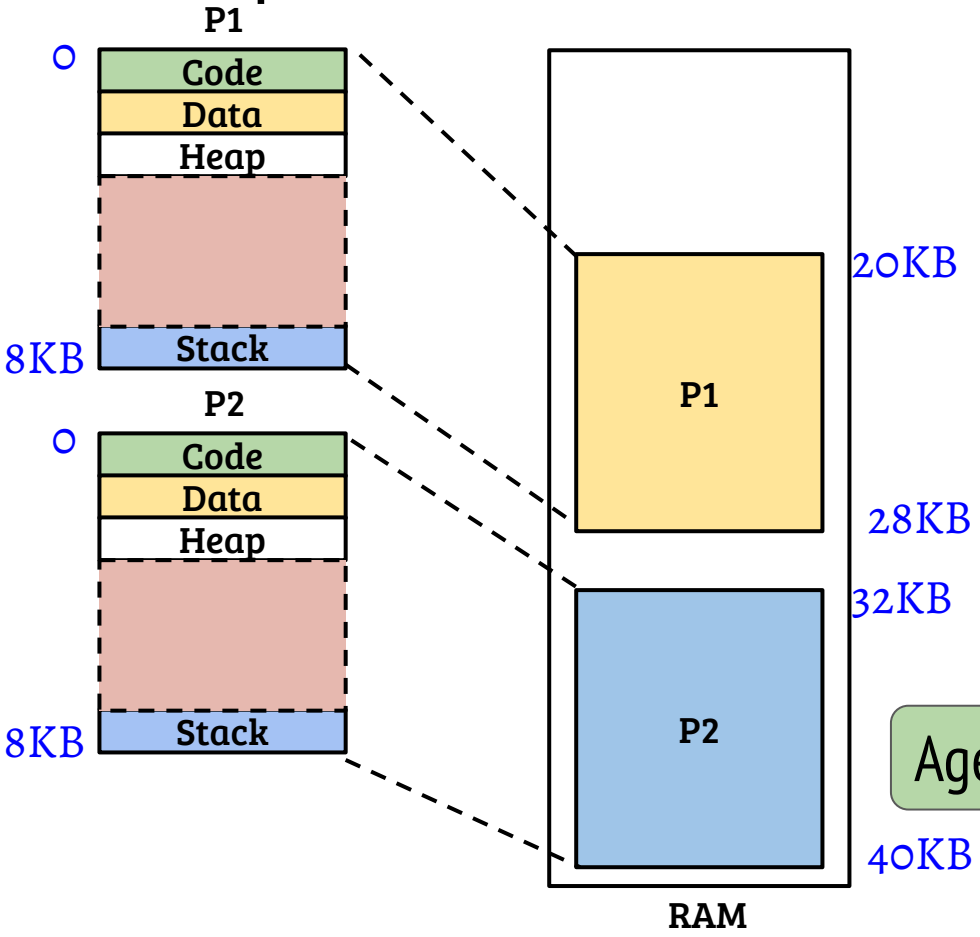
Virtual memory: Segmentation

Recap: Translation at address space granularity



- Physical memory of same size as the address space size is allocated to each process
- Issues: Memory inefficient, inflexible

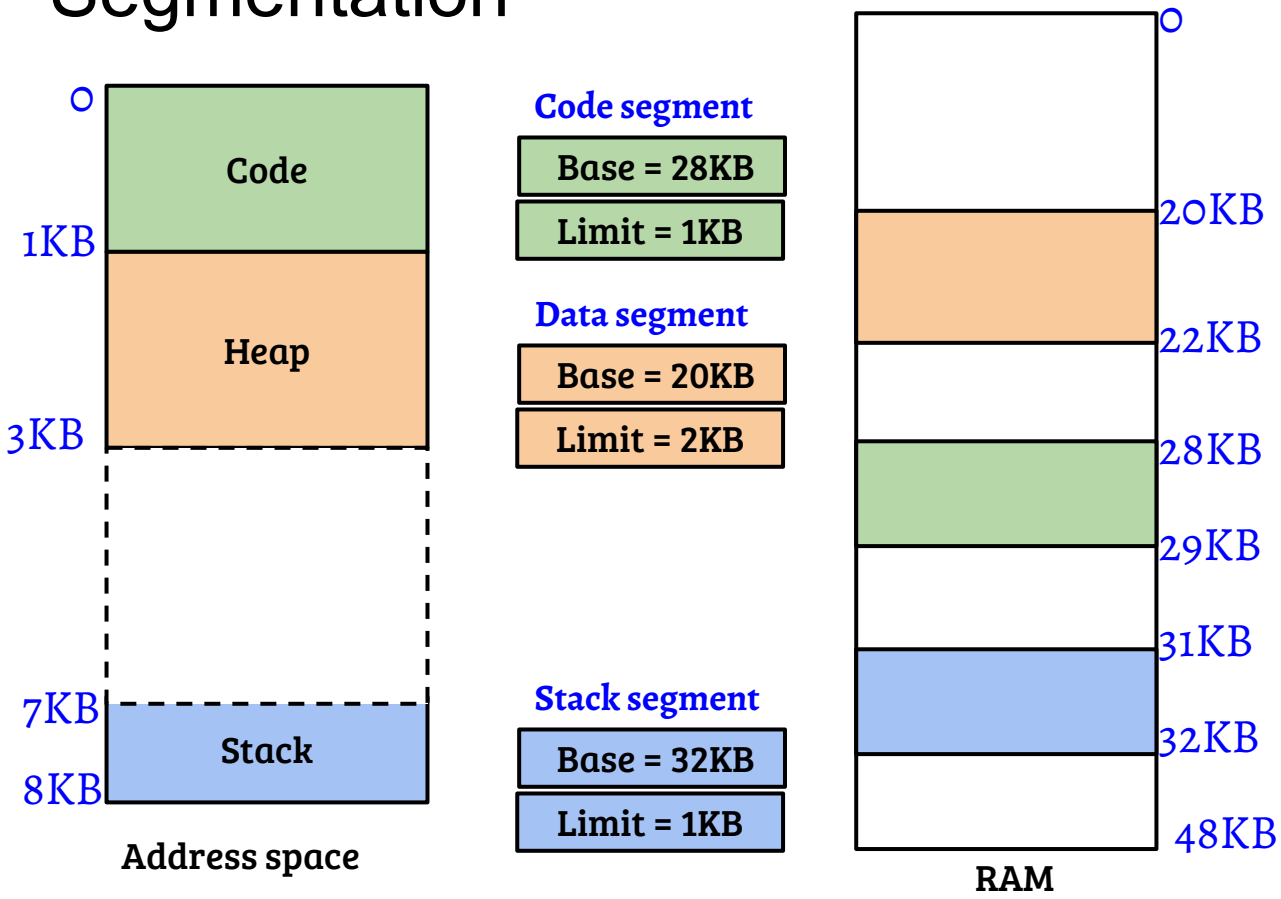
Recap: Translation at address space granularity



- Physical memory of same size as the address space size is allocated to each process
- Issues: Memory inefficient, inflexible

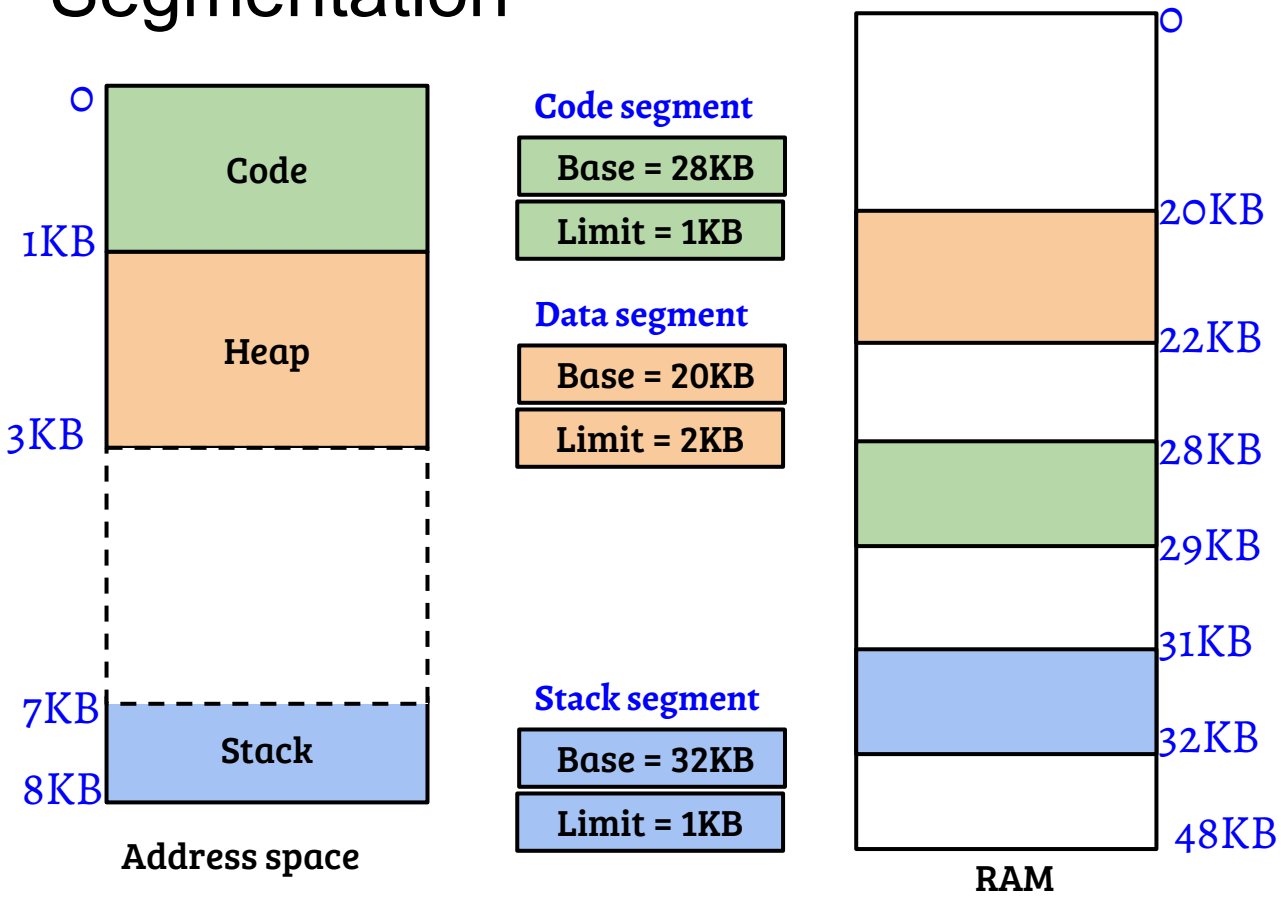
Agenda: Translation at segment granularity

Segmentation



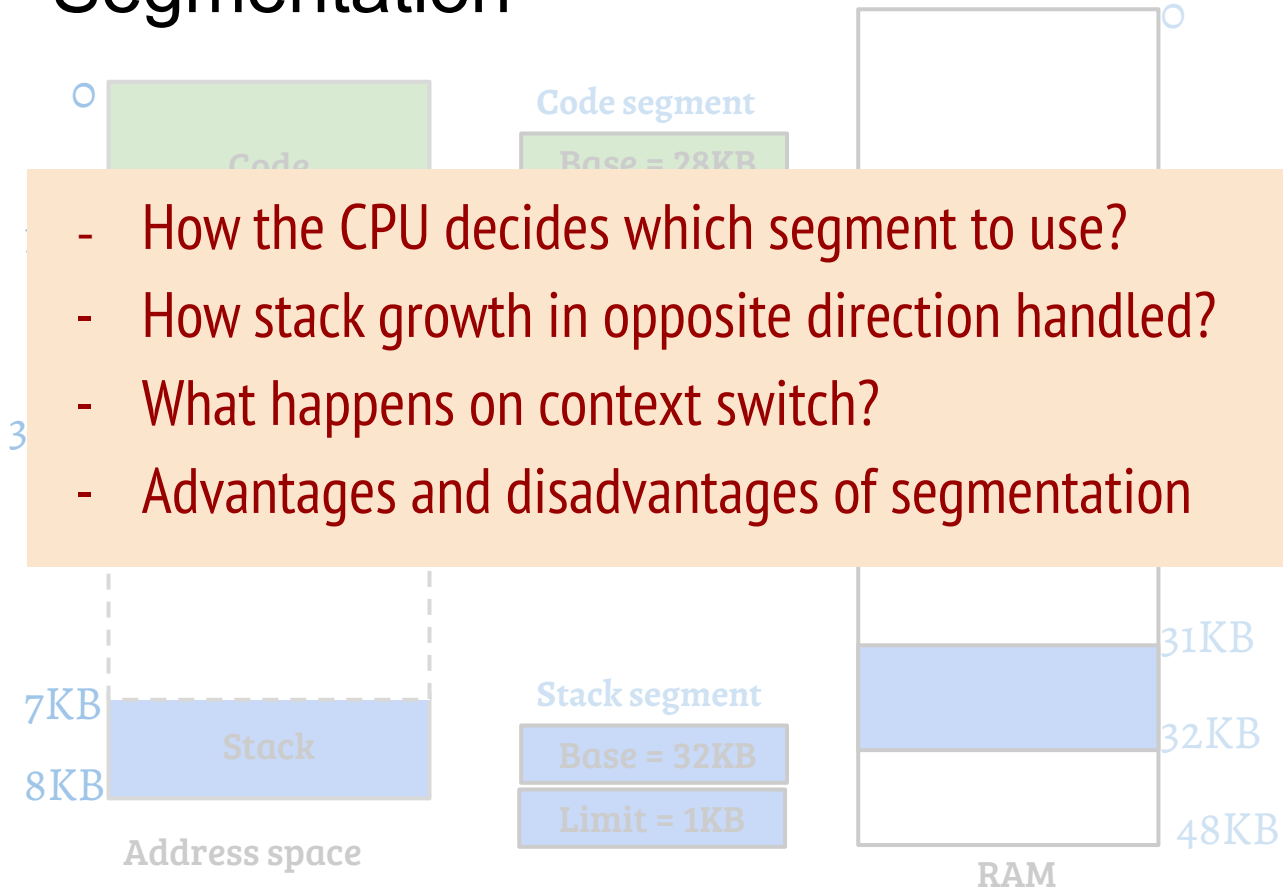
- Extension of the basic scheme with more base-limit register pairs

Segmentation



- Example
  - Code address
  - Data address

Segmentation



- How the CPU decides which segment to use?
- How stack growth in opposite direction handled?
- What happens on context switch?
- Advantages and disadvantages of segmentation

Segmentation: Explicit addressing

- Part of the address is used to explicitly specify segments
- In our example,
  - virtual address space = 8KB, address length = 13 bits and there are three segments
  - Two MSB bits used to specify the segment: “00” for code, “01” for data and “11” for stack
  - The hardware selects the segment register based on the value of two MSB bits and rest of the bits are used as the offset
  - Max. size of each segment = 2KB

Issues with explicit addressing

- Inflexible
  - Data and stack can not be sized dynamically
- Wastage of virtual address space
  - In our example, 2KB virtual address is unusable
- Note: Physical allocation is still done in an on-demand basis

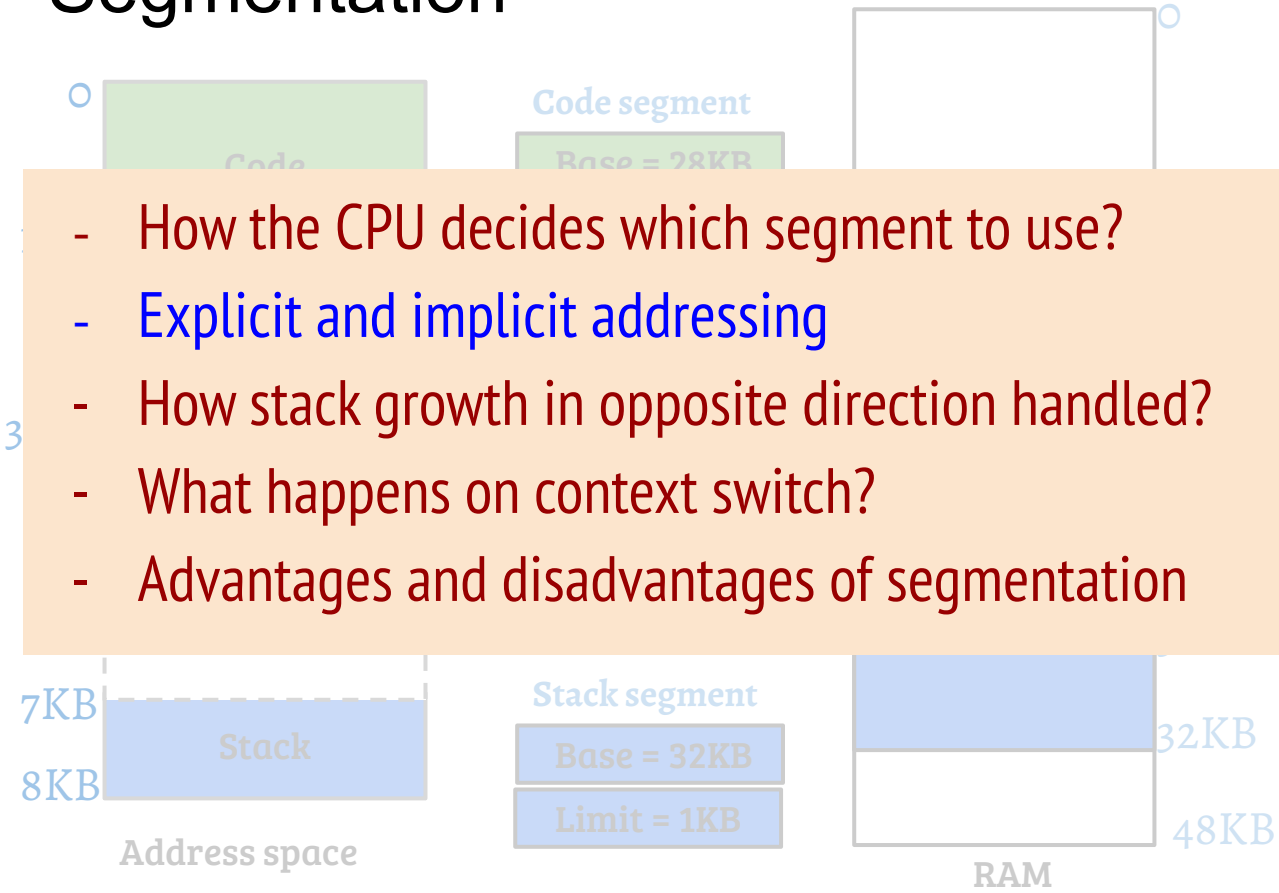
Segmentation: Implicit addressing

- The hardware selects the segment register based on the operation
- Code segment for instruction access
  - Fetch address, jump target, call address

Segmentation: Implicit addressing

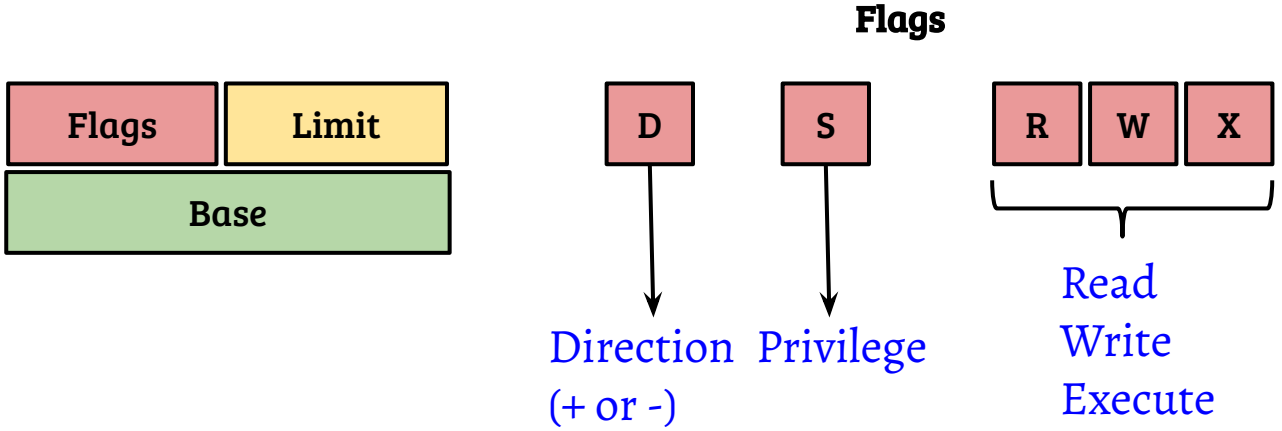
- The hardware selects the segment register based on the operation
- Code segment for instruction access
  - Fetch address, jump target, call address
- Stack segment for stack operations
  - Arguments for push and pop, indirect addressing with SP, BP
- Data segment for other addresses

Segmentation



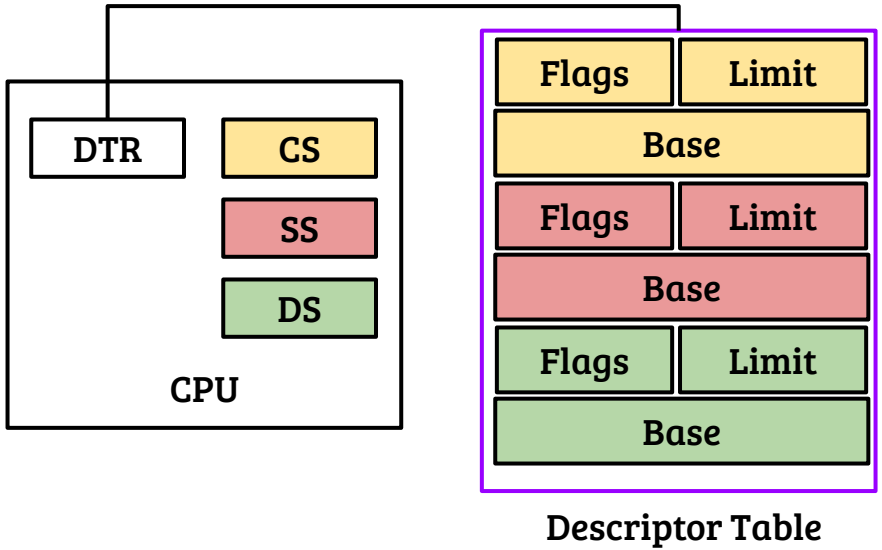
- How the CPU decides which segment to use?
- Explicit and implicit addressing
- How stack growth in opposite direction handled?
- What happens on context switch?
- Advantages and disadvantages of segmentation

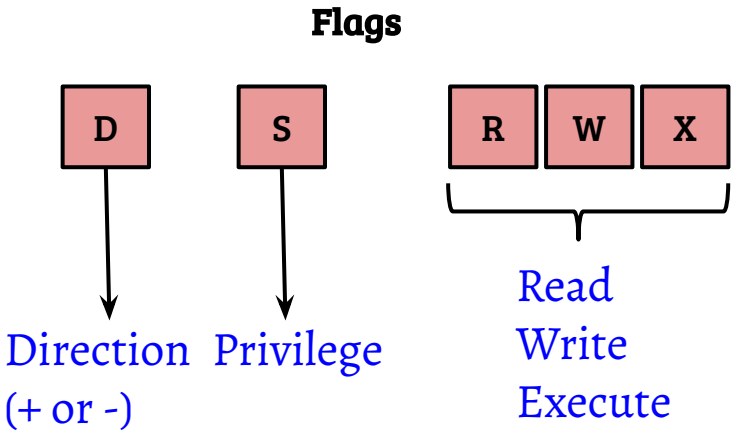
Segmentation (protection and direction)



- For stack, direction is -ve, used by hardware to calculate physical address
- “S” bit can be used to specify privilege, specifically useful in code segment
- R, W and X can be used to enforce isolation and sharing

Segmentation in reality

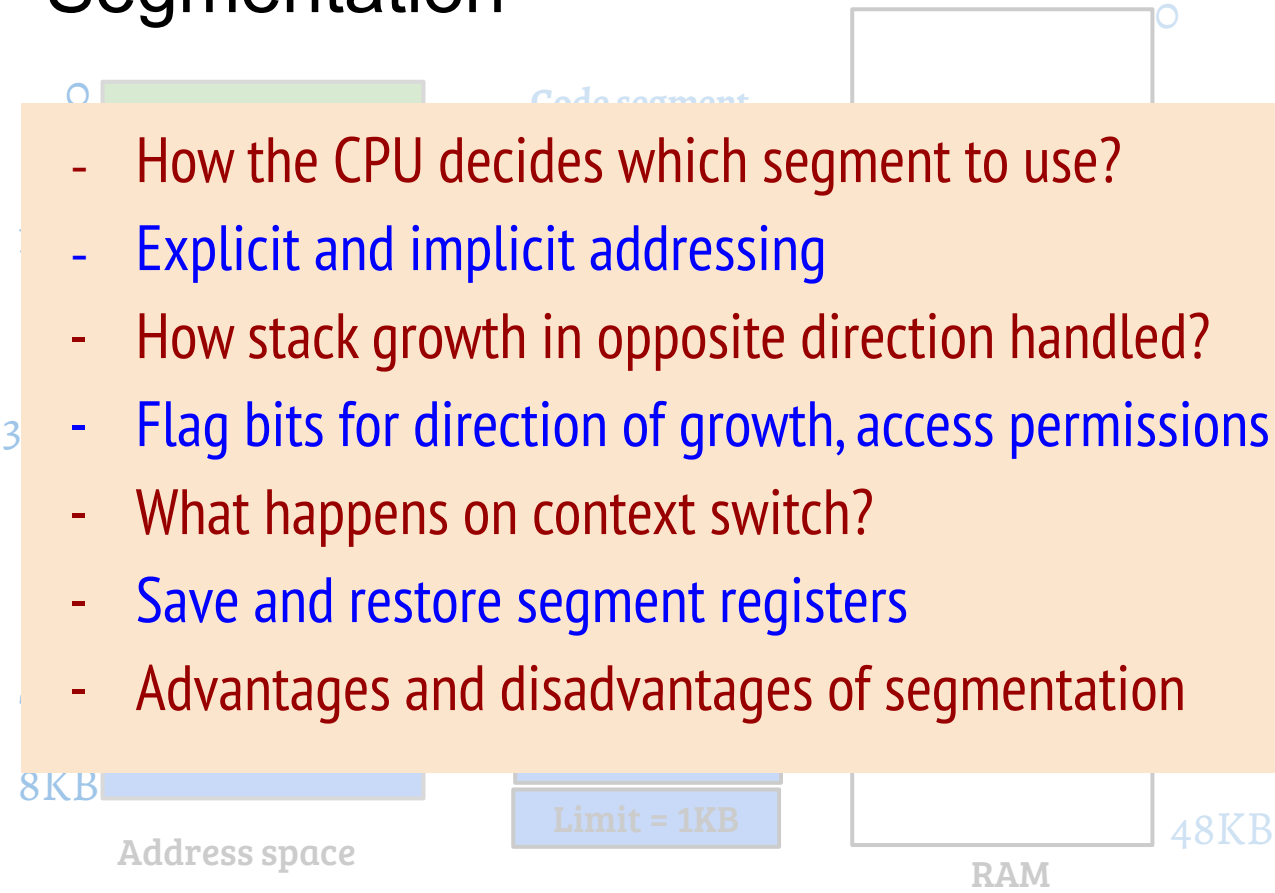




- Descriptor table register (DTR) is used to access the descriptor table
- # of descriptors depends on architecture
- Separate descriptors used for user and kernel mode

Segmentation

- How the CPU decides which segment to use?
- Explicit and implicit addressing
- How stack growth in opposite direction handled?
- Flag bits for direction of growth, access permissions
- What happens on context switch?
- Save and restore segment registers
- Advantages and disadvantages of segmentation



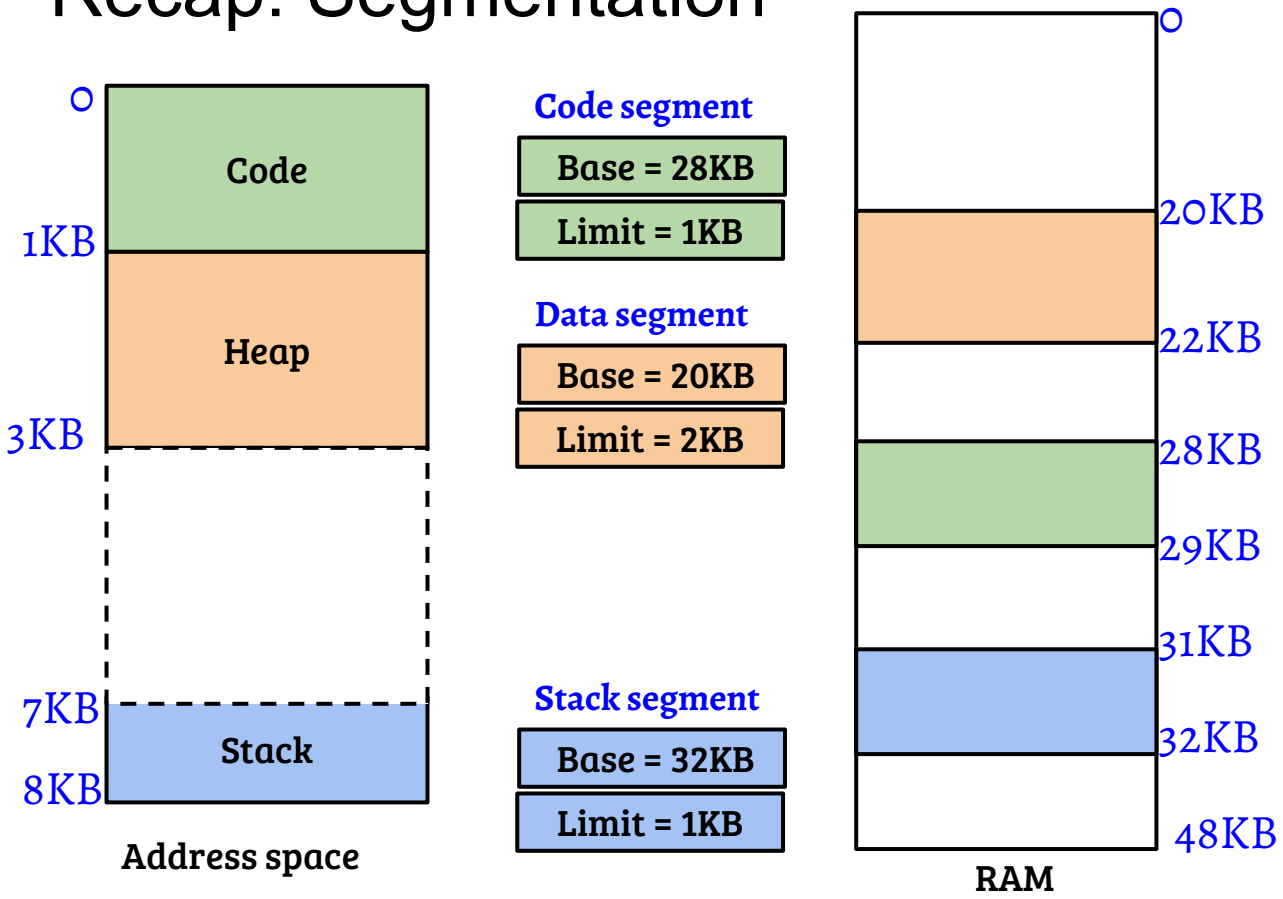
Advantages and disadvantages of segmentation

- Advantages
  - Easy and efficient address translation
  - Save memory wastage for unused addresses
- Disadvantages
  - External fragmentation
  - Can not support discontiguous sparse mapping

CS330: Operating Systems

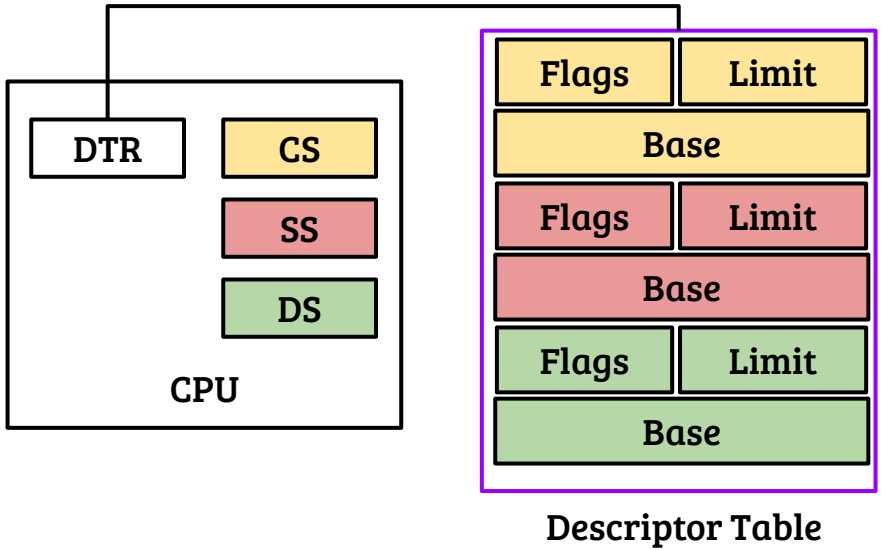
Virtual memory: Paging

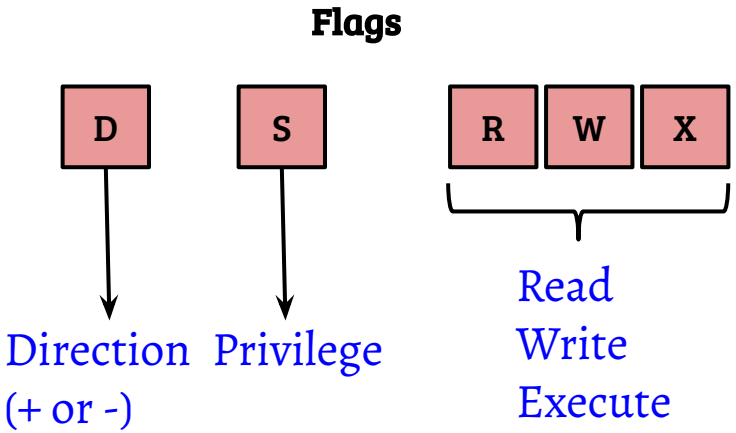
Recap: Segmentation



- Extension of the scheme for translation ar address space granularity
- Base-limit register pairs per segment

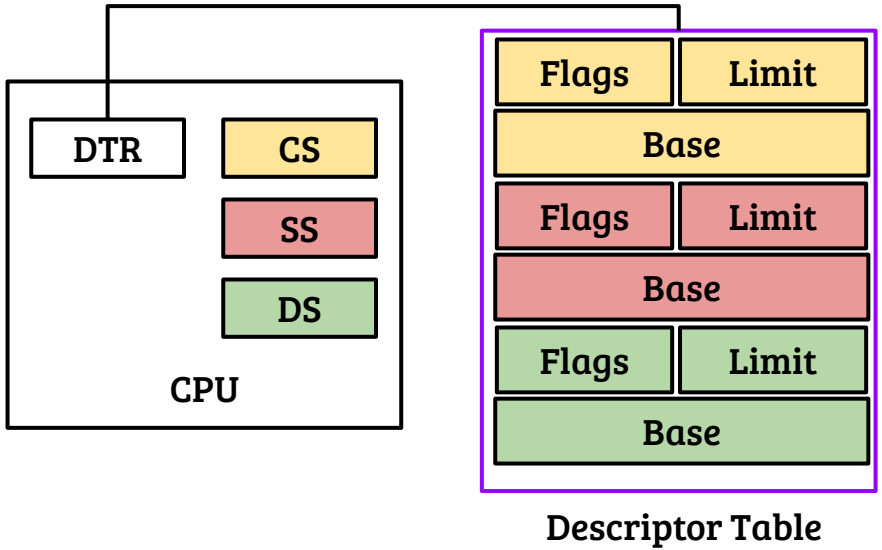
Recap: Segmentation in reality

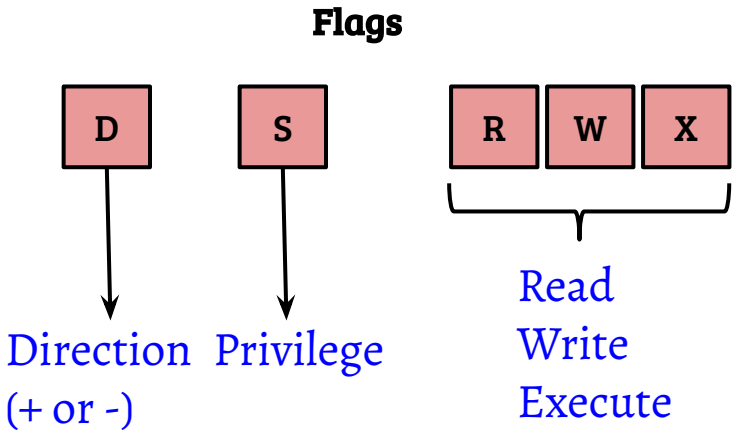




- Descriptor table register (DTR) is used to access the descriptor table
- # of descriptors depends on architecture
- Separate descriptors used for user and kernel mode

Recap: Segmentation in reality



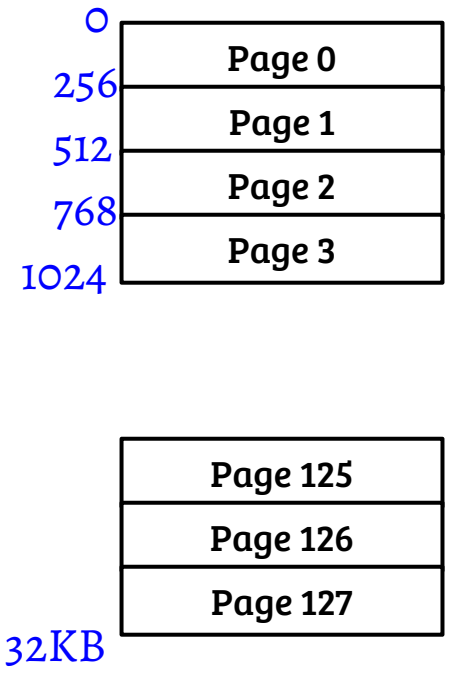


- Qn1: Can the OS address space be organized as split-mode addressing?
- Qn2: When OS uses a separate address space, how to access user addresses?

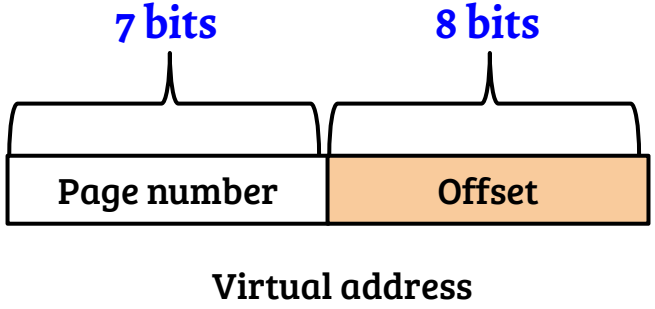
Paging

- Paging addresses the following issues with segmentation
  - External fragmentation caused due to variable sized segments
  - No support for discontinuous/sparse mapping
- The idea of paging
  - Partition the address space into fixed sized blocks (call it pages)
  - Physical memory partitioned in a similar way (call it page frames)
  - OS creates a mapping between *page* to *page frame*
  - H/W uses the mapping to translate VA to PA

Paging example (pages)

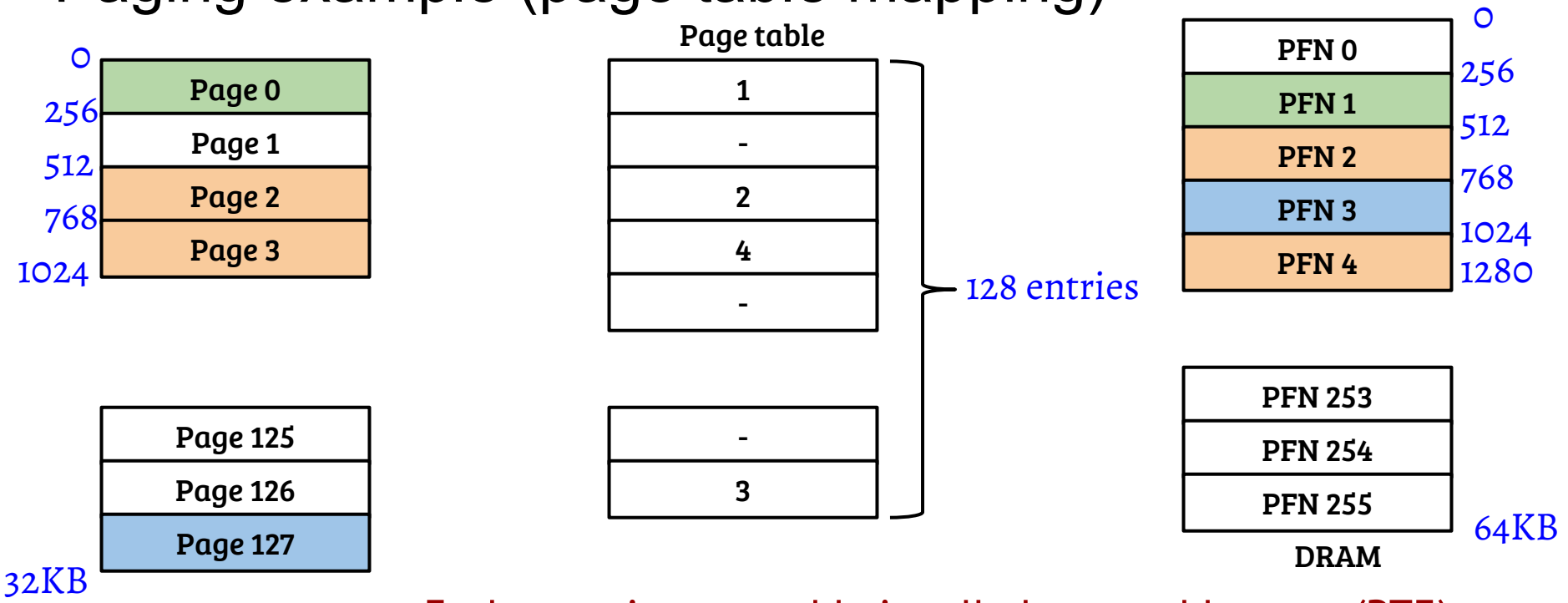


- Virtual address size = 32KB, Page size = 256 bytes
- Address length = 15 bits {0x0 - 0x7FF}
- # of pages = 128



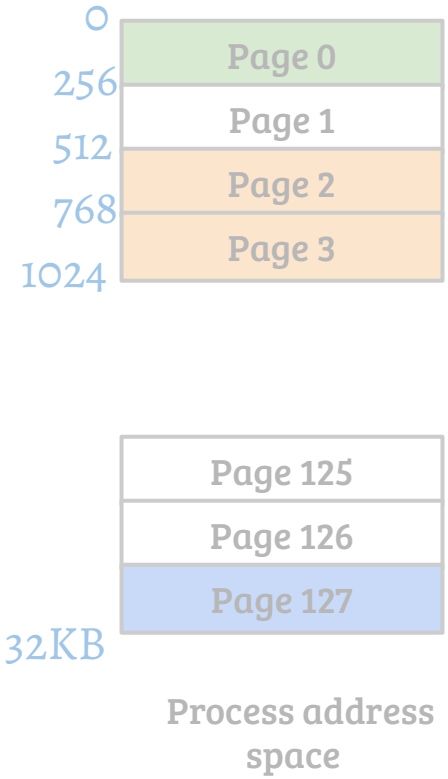
- Example: For Virtual address *0x0510*, Page number = 5, offset = 16

Paging example (page table mapping)

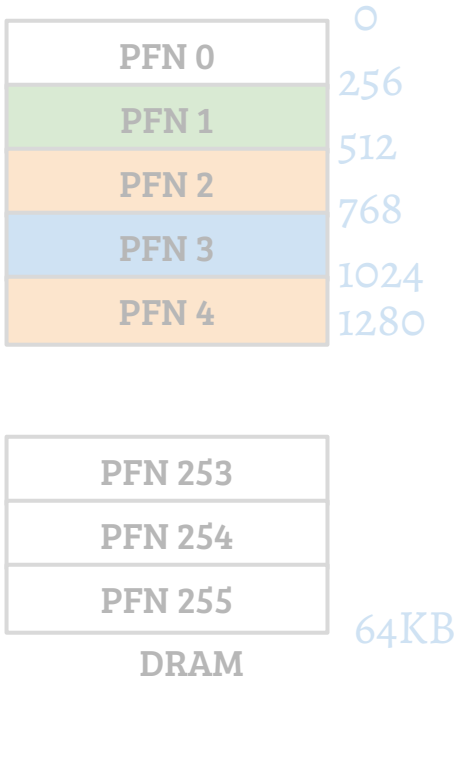


- Each entry in page table is called page table entry (PTE)
- Example mapping: page 0 ⇒ PFN 1, page 2 ⇒ PFN 2 and so on

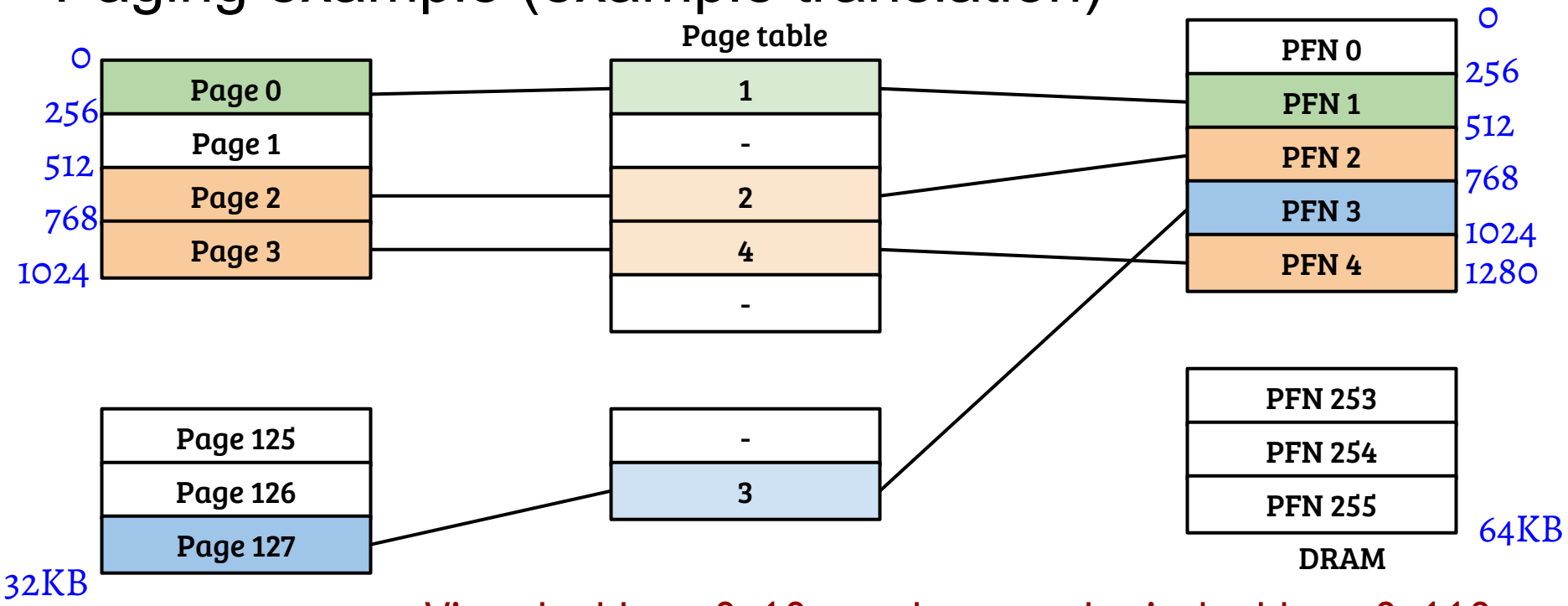
Paging example (page table walk)




```
PTW (vaddr V, PTable P)
// Input: Virtual address, Page table
// Returns physical address
{
  Entry = P[V >> 8];
  if (Entry.present)
      return (Entry.PFN << 8) + (V & 0xFF);
   Raise PageFault;
}
```

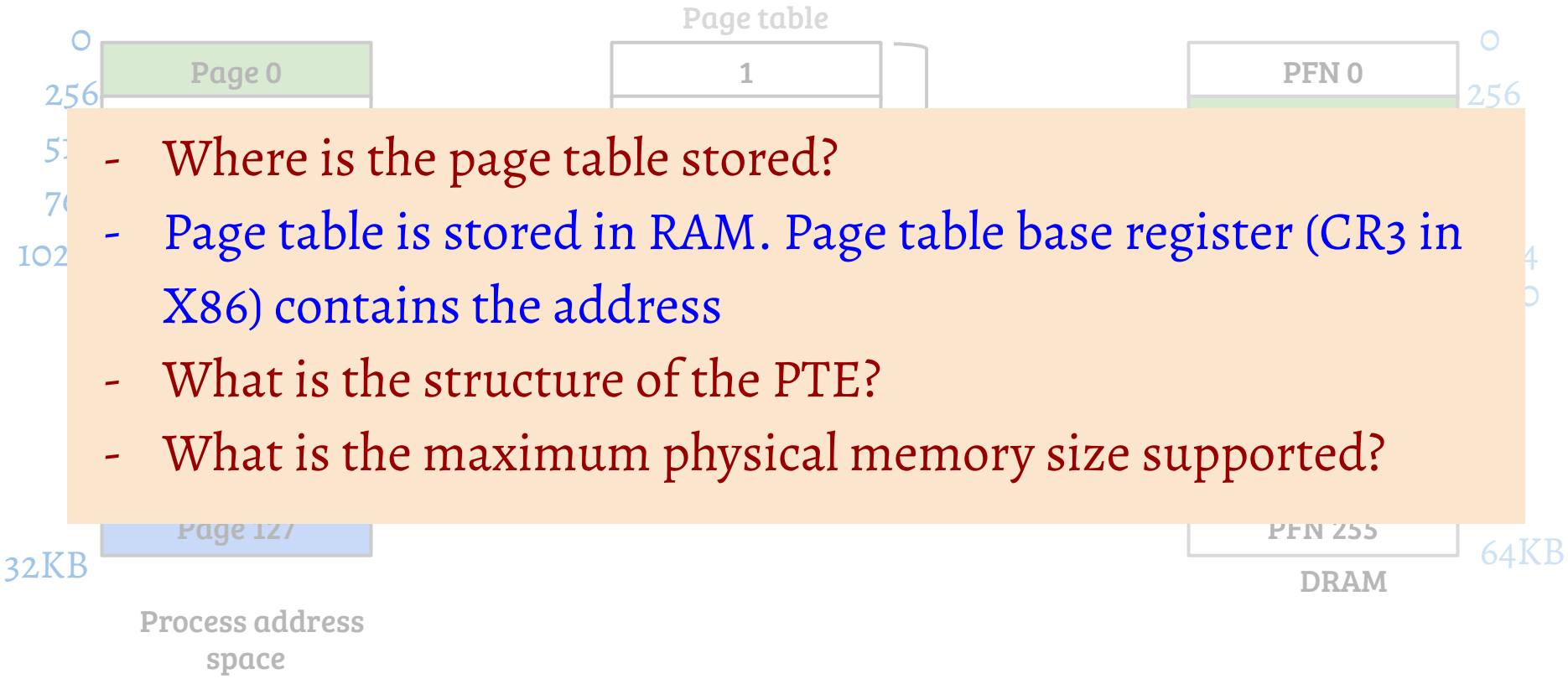


Paging example (example translation)



- Virtual address 0x10 translates to physical address 0x110
- Virtual address 0x7FF0 translates to physical address 0x3F0

Paging example (page table walk)



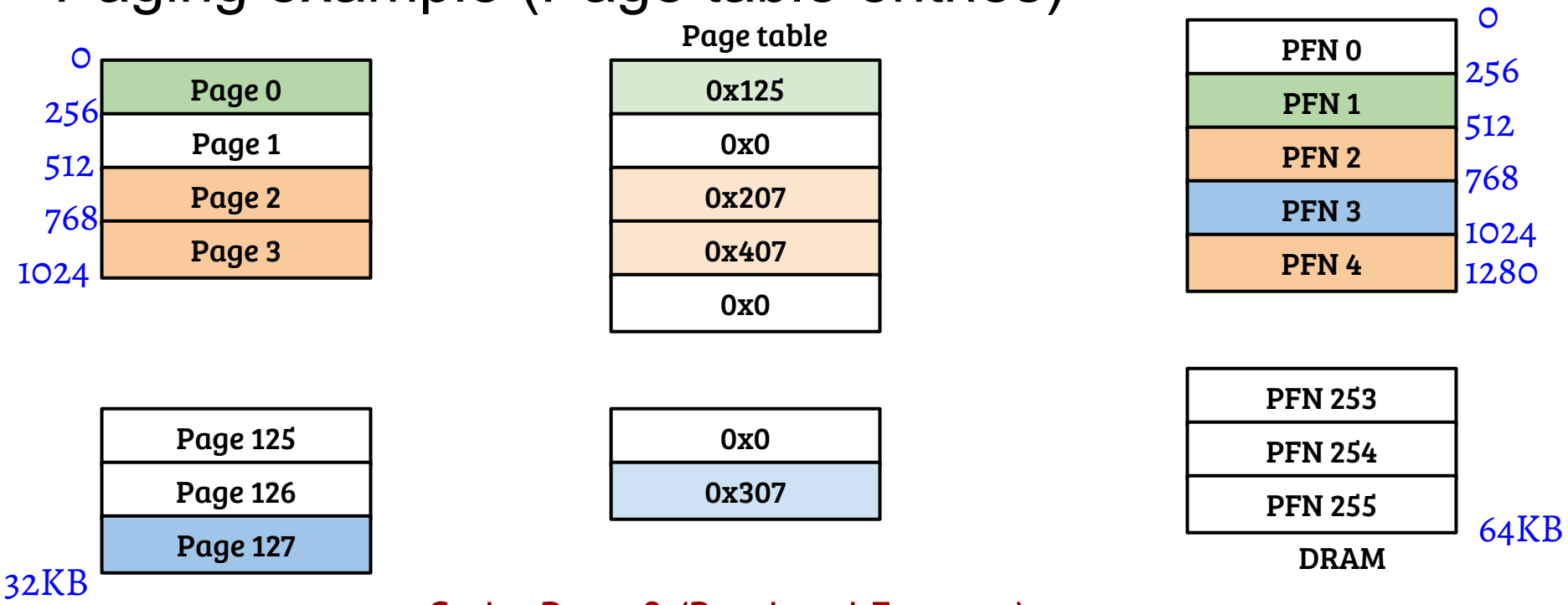
- Where is the page table stored?
- Page table is stored in RAM. Page table base register (CR3 in X86) contains the address
- What is the structure of the PTE?
- What is the maximum physical memory size supported?

Paging example (structure of an example PTE)

- PFN occupies a significant portion of PTE entry (8 bits in this example)

P	Present bit, 1 ⇒ entry is valid
W	Write bit, 1 ⇒ Write allowed
S	Privilege bit, 0 ⇒ only kernel mode access is allowed
A	Accessed bit, 1 ⇒ Address accessed (set by H/W during walk)
D	Dirty bit, 1 ⇒ Address written (set by H/W during walk)
X	Execute bit, 1 ⇒ Instruction fetch allowed for this page
	Reserved/unused bits

Paging example (Page table entries)



- Code: Page 0 (Read and Execute)
- Data: Page 2 and Page 3 (Read and Write)
- Stack: Page 127 (Read and Write)

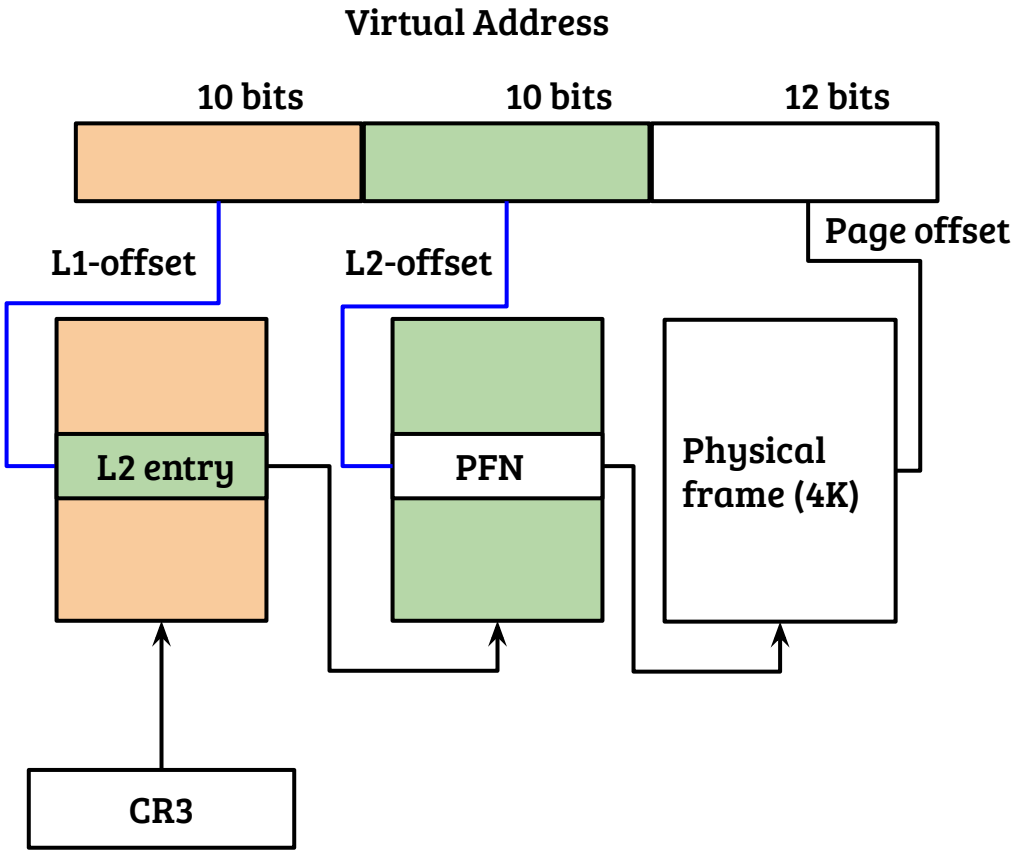
Paging example (page table walk)

- Where is the page table stored?
- Page table is stored in RAM. Page table base register (CR3 in X86) contains the address
- What is the structure of the PTE?
- Apart from the PFN, it contains access permissions and flags
- What is the maximum physical memory size supported?
- For this example, 8-bits can be used to specify 256 page frames. Maximum RAM size = 256 * 256 = 64KB

Paging: one level of page table may not be feasible!

- Consider a 32-bit address space (=4GB)
- What should be the page size for this system?
- Large page size results in *internal fragmentation*
- Assuming page size = 4KB, How many entries are required in a one-level paging system? (2^{20} entries)
- Not possible to hold 2^{20} entries in a single page
- Therefore, multi-level page tables are used in modern systems

Two-level page tables (32-bit virtual address)



- Two-level page table
- Level-1 page table contains entries pointing to Level-2 page table structures
- Level-2 entry contains PFN along with flags

CS330: Operating Systems

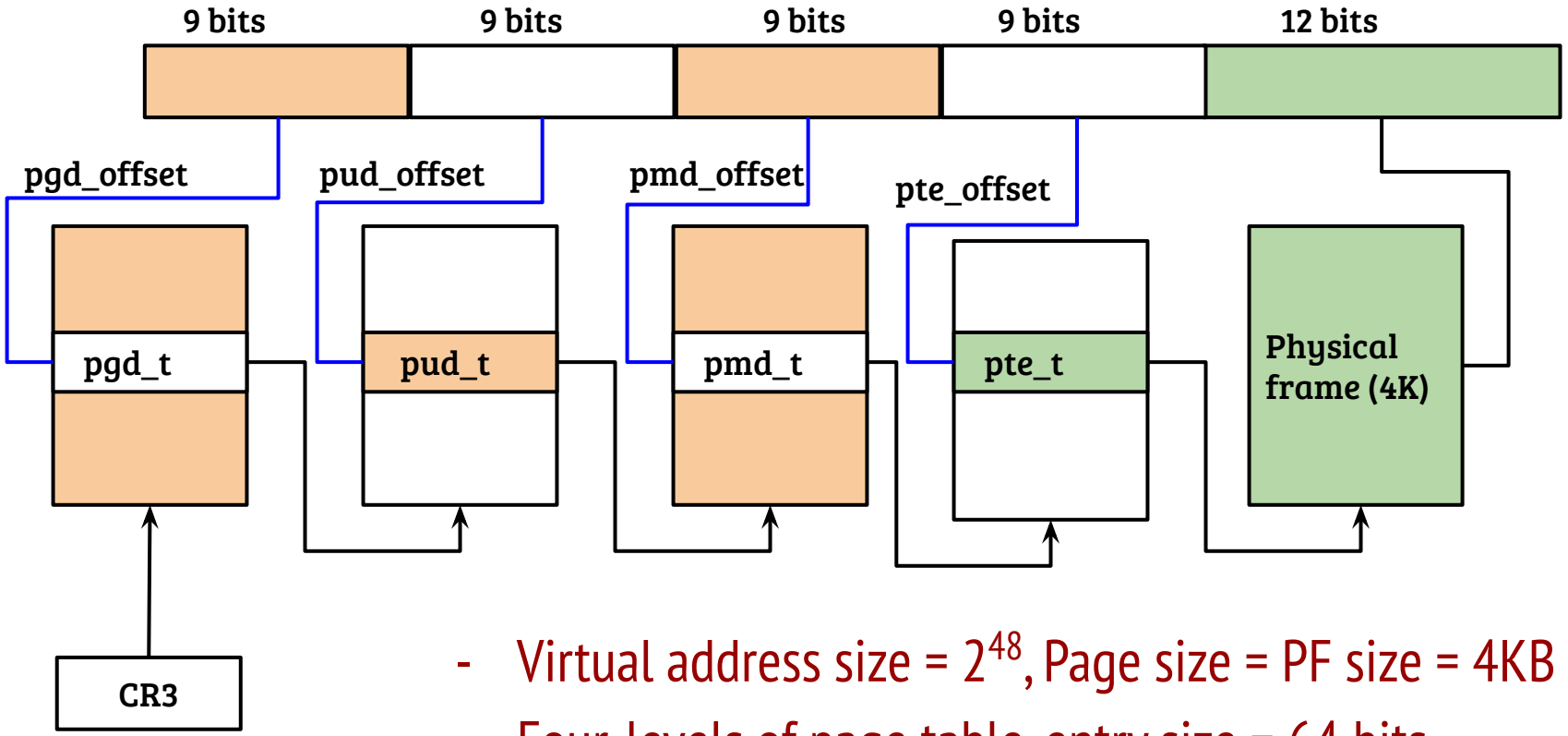
Virtual memory: Multilevel paging and TLB

Recap: Paging

- The idea of paging
  - Partition the address space into fixed sized blocks (call it pages)
  - Physical memory partitioned in a similar way (call it page frames)
  - OS creates a mapping between *page* to *page frame* , H/W uses the mapping to translate VA to PA
- With increased address space size, single level page table entry is not feasible, because
  - Increasing page size (= frame size) increases internal fragmentation
  - Small pages may not be suitable to hold all mapping entries

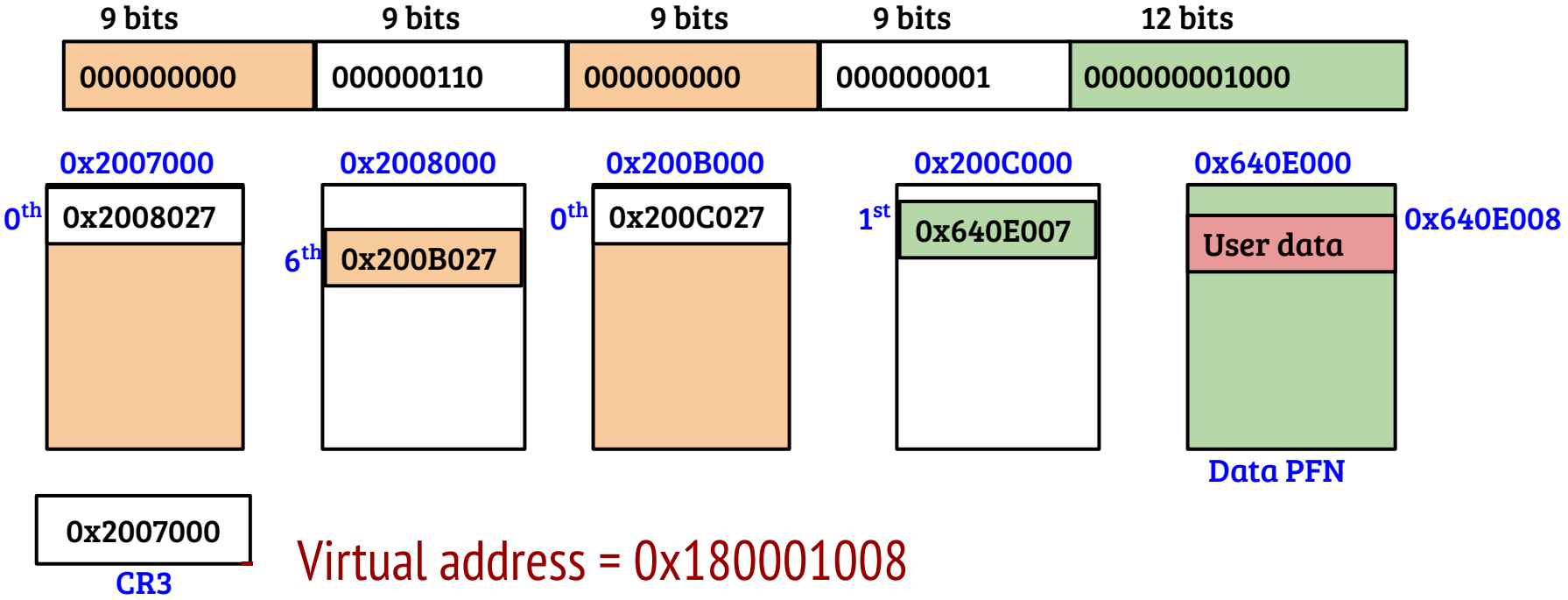
Agenda: Multi-level pages tables and their efficiency implications

4-level page tables: 48-bit VA (Intel x86_64)



- Virtual address size = 2^{48}, Page size = PF size = 4KB
- Four-levels of page table, entry size = 64 bits

4-level page tables: example translation



Virtual address = 0x180001008

- Hardware translation by repeated access of page table stored in physical memory
- Page table entry: 12 bits LSB is used for access flags

Paging: translation efficiency

```
sum = 0;
for(ctr=0; ctr<10; ++ctr)
      sum += ctr;
```

```
0x20100:  mov $0, %rax;
0x20102:  mov %rax, -8(%rbp);     // sum=0
0x20104:  mov $0, %rcx;           // ctr=0
0x20106:  cmp $10, %rcx;          // ctr < 10
0x20109:  jge  0x2011f;           // jump if >=
0x2010f:  add %rcx, %rax;
0x20111:  mov %rax, -8(%rbp);  // sum += ctr
0x20113:  inc %rcx                // ++ctr
0x20115:  jmp 0x20106             // loop
0x2011f:  ..............
```

- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging: translation efficiency

```
0x20100:  mov $0, %rax;
0x20102:  mov %rax, (%rbp):     // sum=0
```

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
  - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
  - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated? Assume stack address range 0x7FFF000 - 0x8000000
- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging: translation efficiency

0x20100: mov $0, %rax;

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
  - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
  - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated? Assume stack address range 0x7FFF000 - 0x8000000
- One code page (0x20) and one stack page (0x7FFF). Caching these translations, will save a lot of memory accesses.

Paging with TLB: translation efficiency

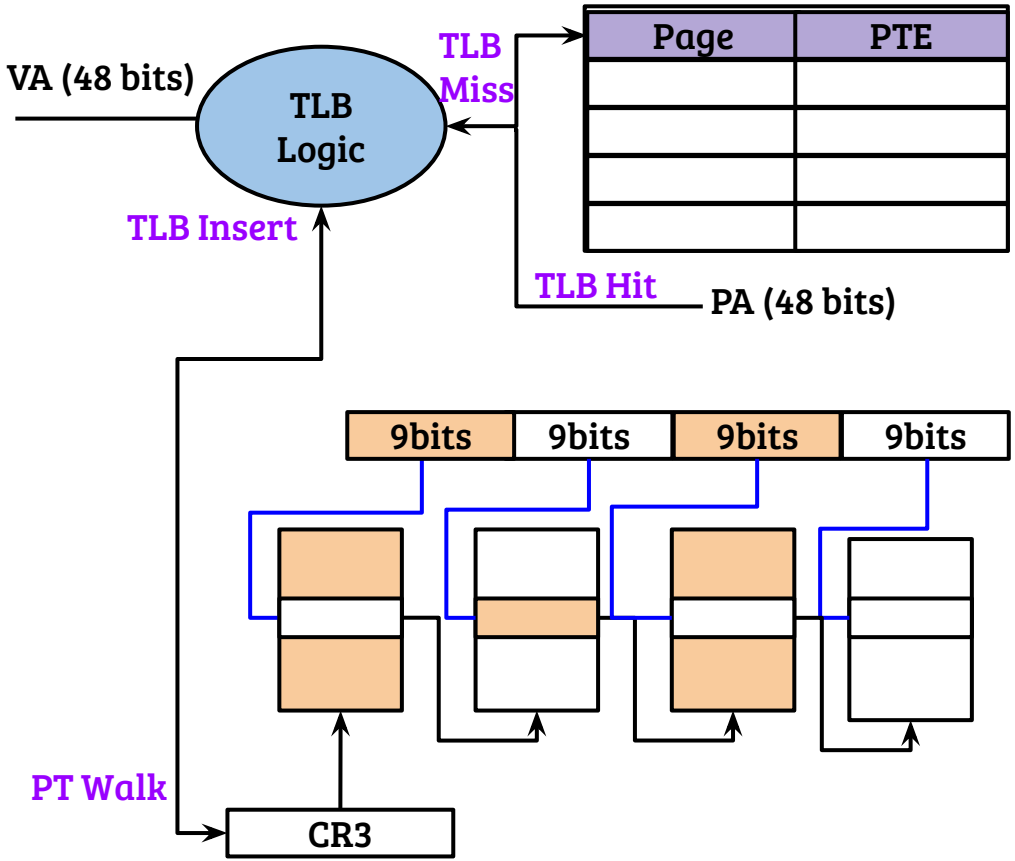
TLB

Page	PTE
0x20	0x750
0x7FFF	0x890

```
Translate(V){
        PageAddress P = V >> 12;
        TLBEntry entry = lookup(P);
        if (entry.valid) return entry.pte;
        entry = PageTableWalk(V);
        MakeEntry(entry);
        return entry.pte;
}
```

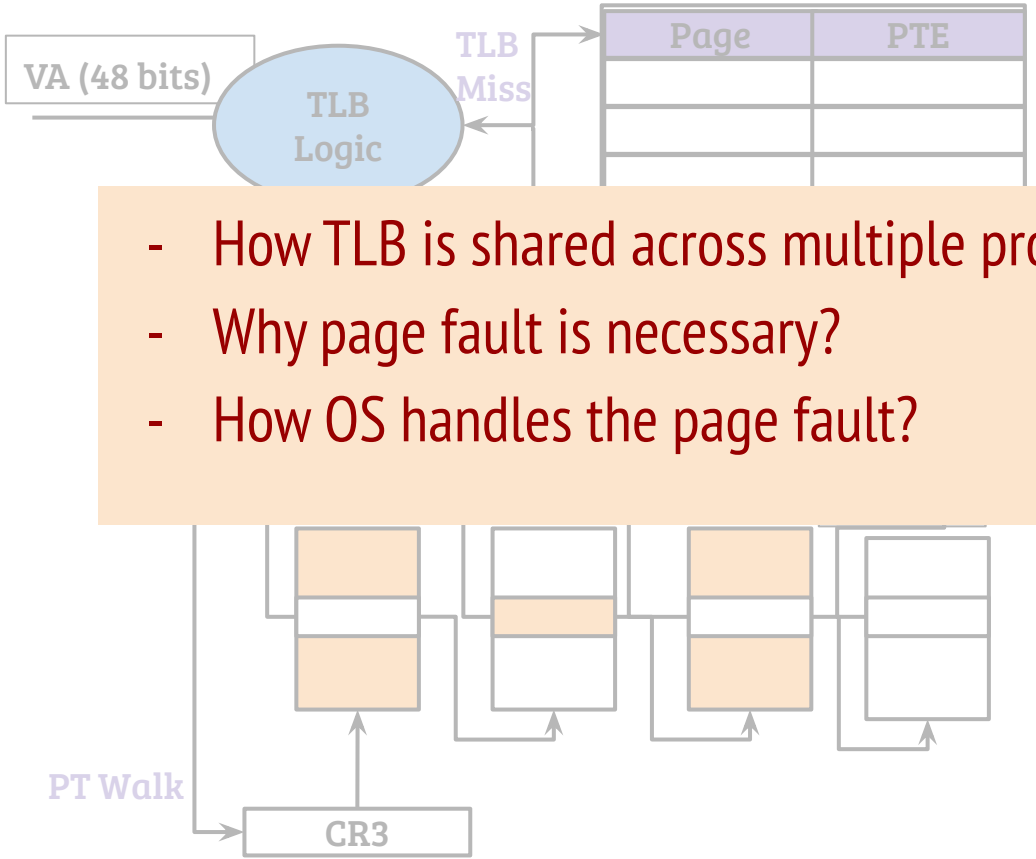
- TLB is a hardware cache to store *Page* to *PFN* mapping (with access flags)
- After first miss for instruction fetch address, all others result in a TLB hit
- Similarly, considering the stack virtual address range as 0x7FFF000 - 0x8000000, one entry in TLB avoids page table walk after first miss

Address translation (TLB + PTW)



- TLB in the path of address translation
- Separate TLBs for instruction and data, multi-level TLBs
- In X86, OS can not make entries into the TLB directly, it can flush entries

Address translation (TLB + PTW)



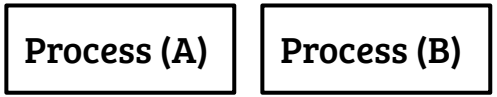
- TLB in the path of address

- How TLB is shared across multiple processes?
- Why page fault is necessary?
- How OS handles the page fault?

d

into the TLB directly, it can flush entries

TLB: Sharing across applications

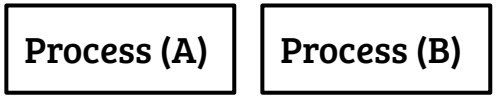


Page	PTE
0x100	0x200007
0x101	0x205007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Process B may be using the same addresses used by A. Result: Wrong translation

TLB: Sharing across applications

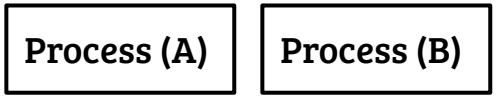


Page	PTE
~~0x100~~	~~0x200007~~
~~0x101~~	~~0x205007~~

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Correctness ensured. Performance is an issue (with frequent context switching)

TLB: Sharing across applications

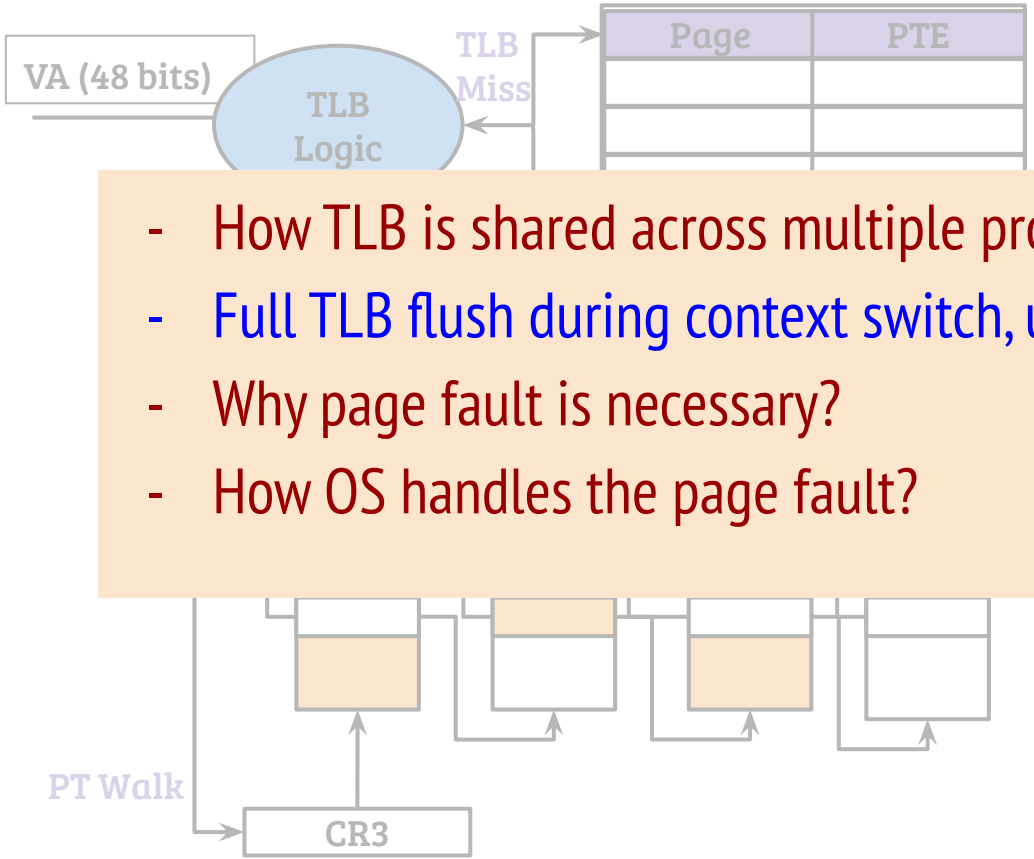


ASID	Page	PTE
A	0x100	ox200007
A	0x101	ox205007
B	0x100	0x301007
B	0x101	0x302007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Address space identified (ASID) along with each TLB entry to identify the process

Address translation (TLB + PTW)



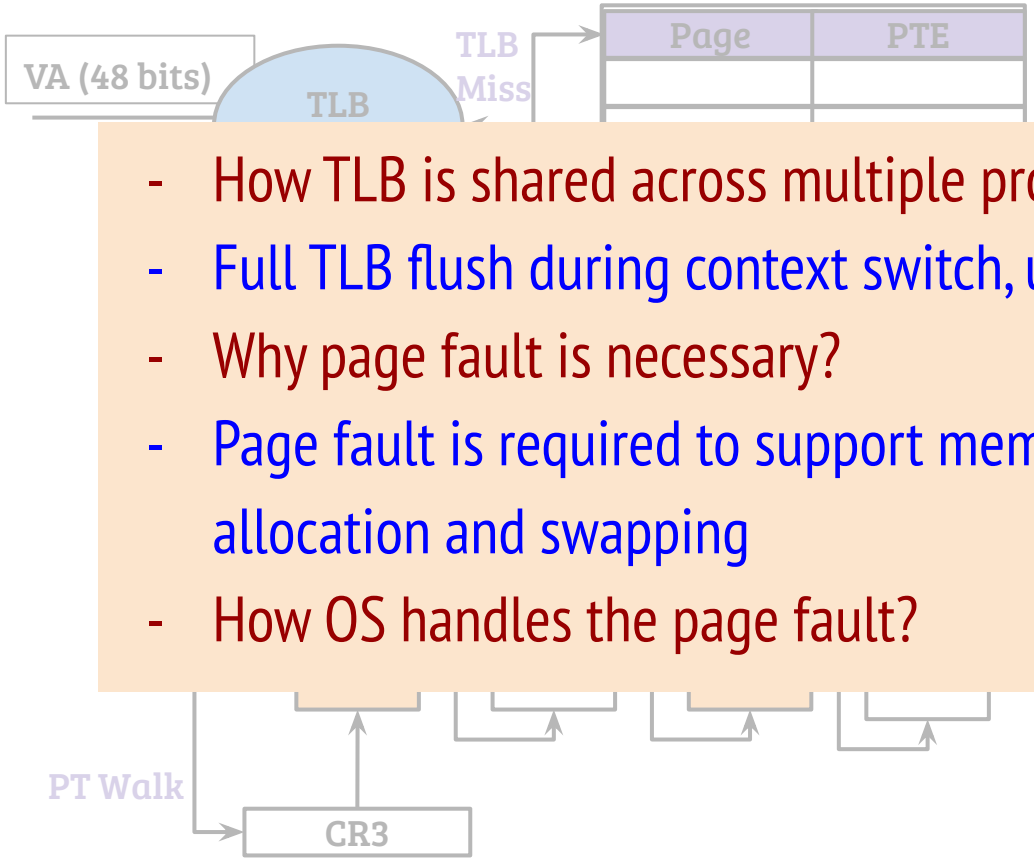
- TLB in the path of address

- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- How OS handles the page fault?

d

entries

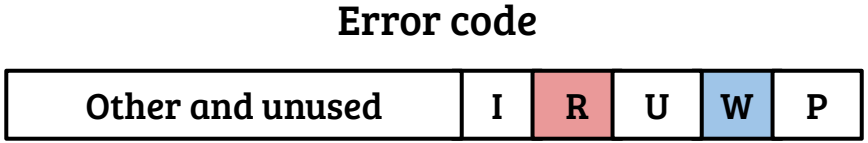
Address translation (TLB + PTW)



- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?

Page fault handling in X86: Hardware

```
If( !pte.valid ||
    (access == write && !pte.write) ||
    (cpl != 0 && pte.priv == 0)){
        CR2 = Address;
        errorCode = pte.valid
                    | access << 1
                    | cpl << 2;
        Raise pageFault;
} // Simplified
```



P — Present bit, 1 ⇒ fault is due to protection

W — Write bit, 1 ⇒ Access is write

U — Privilege bit, 1 ⇒ Access is from user mode

R — Reserved bit, 1 ⇒ Reserved bit violation

I — Fetch bit, 1 ⇒ Access is Instruction Fetch

- Error code is pushed into the kernel stack by the hardware (X86)

Page fault handling in X86: OS fault handler

```
HandlePageFault( u64 address, u64 error_code)
{
    If ( AddressExists(current → mm_state, address) &&
        AccessPermitted(current → mm_state, error_code) {
            PFN = allocate_pfn( );
            install_pte(address, PFN);
            return;
        }
    RaiseSignal(SIGSEGV);
}
```

Address translation (TLB + PTW)

- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?
- The hardware invokes the page fault handler by placing the error code and virtual address. The OS handles the page fault either fixing it or raising a SEGFAULT.

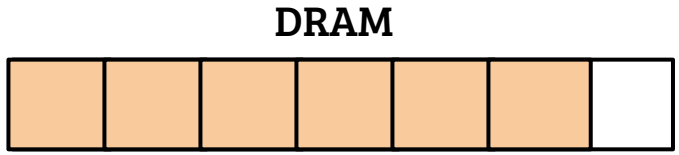
CS330: Operating Systems

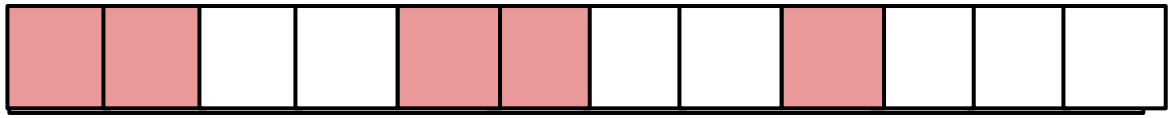
Virtual memory: Page fault and Swapping

Recap: Address translation (TLB + PTW)

- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?
- The hardware invokes the page fault handler by placing the error code and virtual address. The OS handles the page fault either fixing it or raising a SEGFAULT.

Swapping (swap-out)



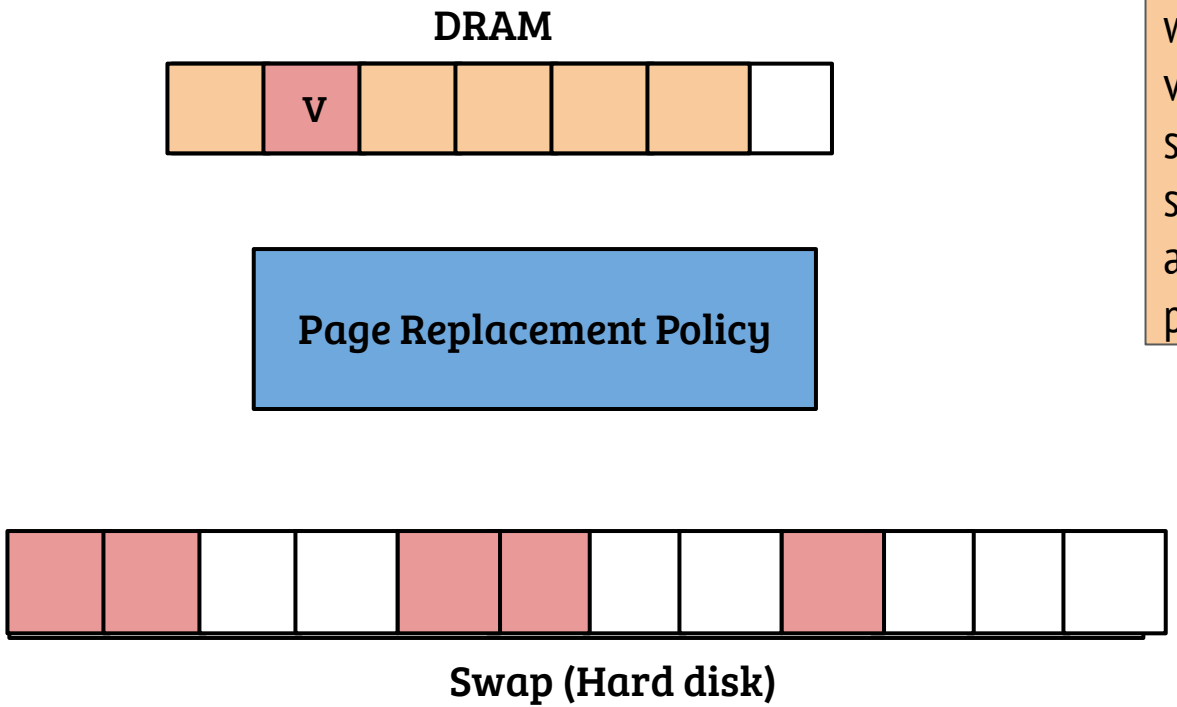


Number of free PFNs are very few in the system. I can not break my promise made to the applications. Let me swap-out some memory. But which one to swap-out?

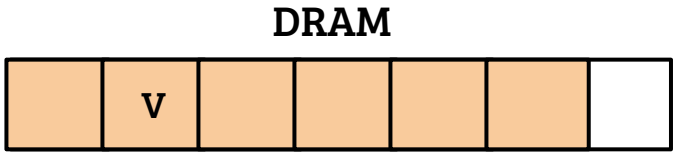
OS

AllocatePFN( )

Swapping (swap-out)

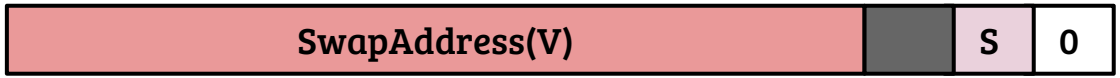


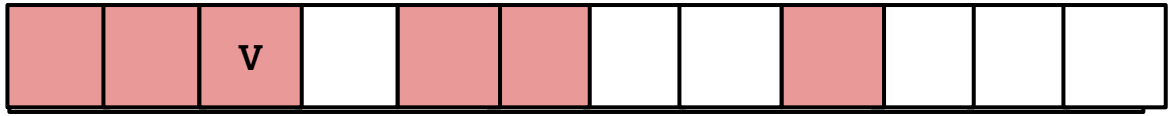
Swapping (swap-out)



PTE mapping the victim PFN (before swap)

PTE mapping the victim PFN (after swap)



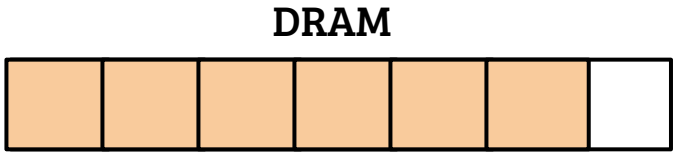


Swap (Hard disk)

Update the present-bit to 0 in the PTE such that any access to the page through the virtual address will result in a page fault. Also maintain the swap address in the PTE.

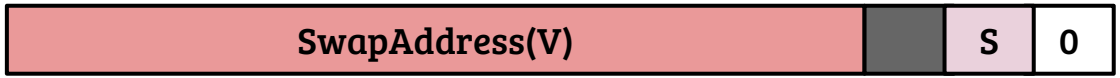
AllocatePFN( )

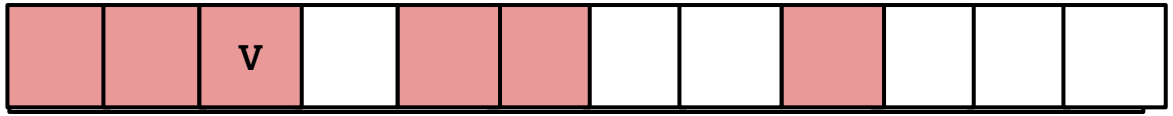
Swapping (swap-out)



PTE mapping the victim PFN (before swap)

PTE mapping the victim PFN (after swap)





Swap (Hard disk)

Content of the PFN is now in the swap device. In future, any translation using the PTE will result in a page fault. The page fault handler would copy it back from the swap device.

AllocatePFN( )

Page fault: Swap-in procedure (simplified)

```
HandlePageFault( u64 address, u64 error_code)
{
    If ( AddressExists(current → mm_state, address) &&
        AccessPermitted(current → mm_state, error_code) {
            PFN = allocate_pfn( );
            If ( is_swapped_pte(address) )        // Check if the PTE is swapped out
               swapin(getPTE(address), PFN);  // Copy the swap block  to PFN
            install_pte(address, PFN);            // and update the PTE
            return;
        }
    RaiseSignal(SIGSEGV);
}
```

Page replacement

- Objective: minimize number of page faults (due to swapping)
- We can model this problem with three parameters
  - A given sequence of access to virtual pages
  - # of memory pages (Frames)
  - Page replacement policy
- Metrics to measure the effectiveness: # of page faults, page fault rate, average memory access time

Belady's optimal algorithm (MIN)

- Strategy: Replace the page that will be referenced after the longest time
- Example:

  #of frames = 3

  Reference sequence (in temporal order)

  1, 3, 1, 5, 4, 1, 2, 5, 2, 2, 5, 3
- #of page faults = 6 (3 cold-start misses result in page faults, no swapping)
- Belady's MIN is proven to be optimal, but impractical as it requires knowledge of future access

First In First Out (FIFO)

- Strategy: Replace the page that is in memory for the longest time
- Example:

  #of frames = 3

  Reference sequence (in temporal order)

  1, 3, 1, 5, 4, 1, 2, 5, 2, 2, 5, 3
- #of page faults = 8 (3 cold-start misses)
- FIFO suffers from an anomaly known as Belady's anomaly
  - With increased #of frames, #of page fault may also increase!
  - Example access sequence: 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4
  - #of page faults with 3 frames < #of page faults with 4 frames

Least recently used (LRU)

- Strategy: Replace the page that is not referenced for the longest time
- Example:

  #of frames = 3

  Reference sequence (in temporal order)

  1, 3, 1, 5, 4, 1, 2, 5, 2, 2, 5, 3
- #of page faults = 7 (3 cold-start)
- LRU shown to be useful for workloads with access locality
- Implementation of LRU using a single accessed-bit may not be practical, can be approximated using CLOCK (homework)
- Stack property or inclusion property of eviction algorithms

CS330: Operating Systems

Threads

What is a thread?

- Threads are (almost!) independent execution entities of a single process
- Threads of a single process can be scheduled different CPUs in a concurrent manner. Therefore,
  - Each thread has a different register state and stack
  - At a given point of time, PC of different threads can be different
- How threads are different from processes?
  - Threads of a single process share the address space
  - Context switch between two threads of a process does not require switching the address space

Leverage multi-core systems

- Threads share the address space
  - Global variables can be accessed from thread functions
  - Dynamically allocated memory can be passed as thread arguments

Leverage multi-core systems

- Threads share the address space
  - Global variables can be accessed from thread functions
  - Dynamically allocated memory can be passed as thread arguments
- Example parallel computation models
  - Data parallel processing: Data is partitioned into disjoint sets and assigned to different threads
  - Task parallel processing: Each thread performs a different computation on the same data

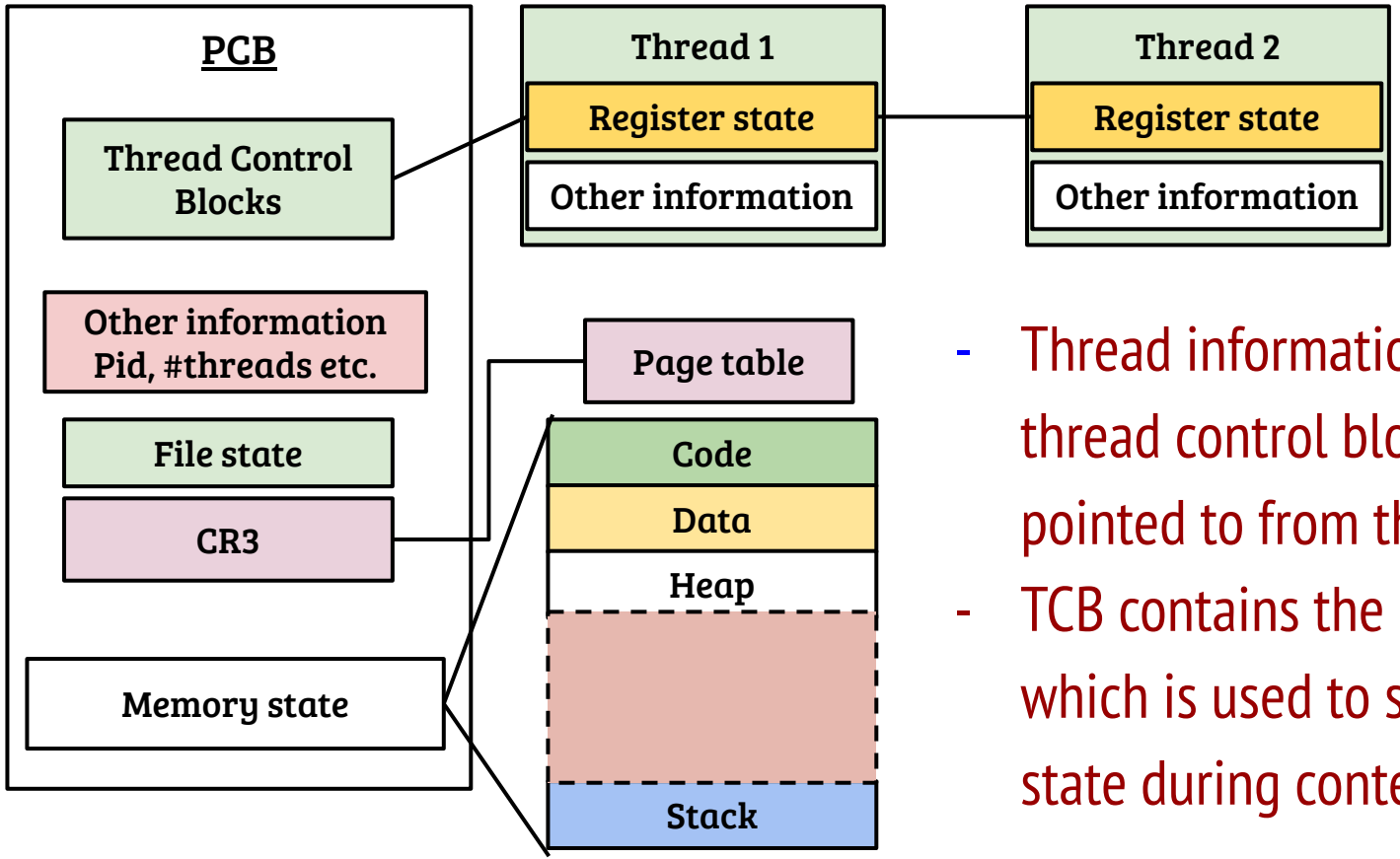
Example: Finding MAX

- Given *N* elements and a function *f*, we are required to find the element *e* such that *f(e)* is maximum
- If the computation time for function *f* is significant, we can employ multithreading with K threads using the following strategy
- Partition N elements into K non-overlapping sets and assign each thread to compute the MAX within its own set
- When all threads complete, we find out the global maximum

Multi-threaded processes

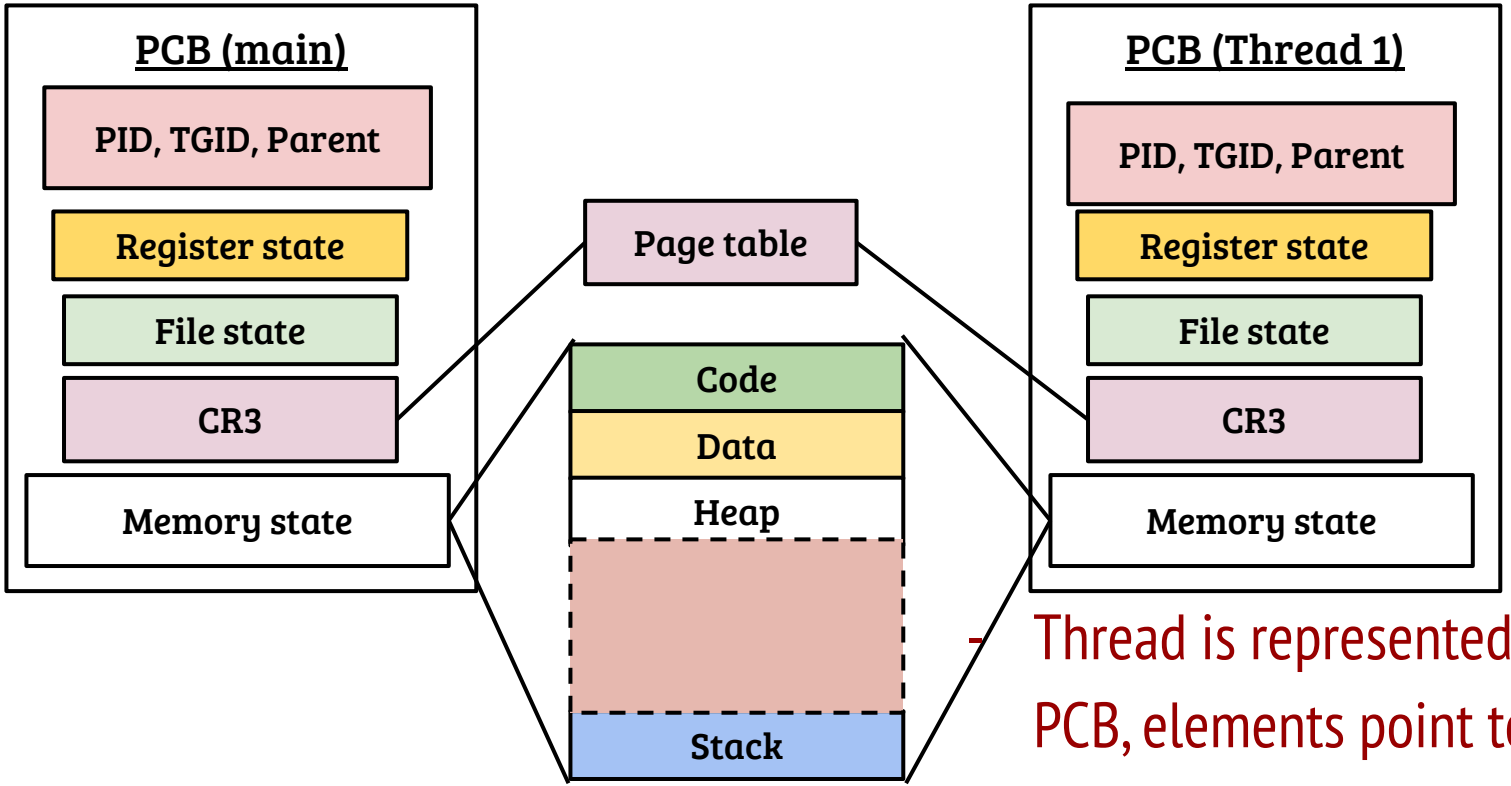
- Why multithreading is useful?
- Efficient execution on multicore systems, overlapping I/O and processing
- How does OS maintain thread related information?
- How stacks for multiple threads are managed?
- What is POSIX thread API? How is it used?

PCB of a multithreaded process



- Thread information is stored in thread control blocks (TCB) which is pointed to from the PCB
- TCB contains the register state which is used to save/restore CPU state during context switch

PCB of a multithreaded process (Linux)

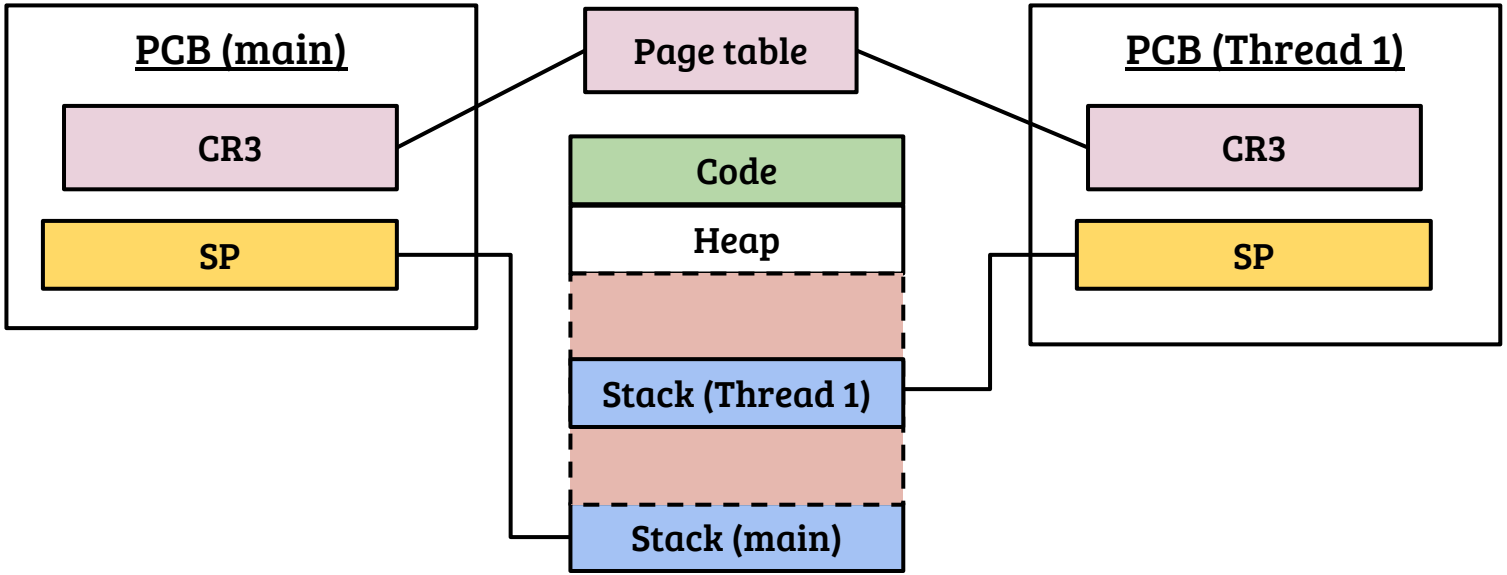


- Thread is represented by a separate PCB, elements point to the structure containing subsystem level info.

Multi-threaded processes

- Why multithreading is useful?
- Efficient execution on multicore systems, overlapping I/O and processing
- How does OS maintain thread related information?
- Maintain thread information using separate PCB or using TCB
- How stacks for multiple threads are managed?
- What is POSIX thread API? How is it used?

Stack for multi-threaded processes



- Stack for threads dynamically allocated from the address space using mmap( ) system call and passed to the OS during thread creation

Multi-threaded processes

Threads are (almost!) independent execution entities of a single process

- Why multithreading is useful?
- Efficient execution on multicore systems, overlapping I/O and processing
- How does OS maintain thread related information?
- Maintain thread information using separate PCB or using TCB
- How stacks for multiple threads are managed?
- Stacks for threads are allocated using memory allocation APIs
- What is POSIX thread API? How is it used?

switching the address space

Posix thread API (pthread_create)

```
int pthread_create( pthead_t *tid, pthread_attr_t *attr,
                    void * (*thfunc) (void*), void *arg);
```

- Creates a thread with "tid" as its handle and the thread starts executing the function pointed to by the "thfunc" argument
- A single argument (of type void *) can be passed to the thread
- Thread attribute can be used to control the thread behavior e.g., stack size, stack address etc. Passing NULL sets the defaults
- Returns 0 on success.
- Thread termination: return from thfunc, pthread_exit( ) or pthread_cancel( )
- In Linux, pthread_create and fork implemented using clone( ) system call

Posix thread API (pthread_join)

int pthread_join(pthead_t tid, void **retval)

- This call waits for the thread with handle “tid” to finish
- The return value of the thread is captured using the “retval” argument
  - The thread must allocate the return value which is freed after the process joins
- Invoking pthread_join for an already finished thread returns immediately

CS330: Operating Systems

Shared address space and concurrency

Recap: Threads

- Threads share the address space
  - Low context switch overheads
  - Global variables can be accessed from thread functions
  - Dynamically allocated memory can be passed as thread arguments
- Sharing data is convenient to design parallel computation
- Pthread API for multi-threaded programming

Threads sharing the address space

- Threads share the address space
  - Global variables can be accessed from thread functions

- Everything seems to be fine, what is the issue?
- How does OS fit into this discussion?

  - Data parallel processing: Data is partitioned into disjoint sets and assigned to different threads
  - Task parallel processing: Each thread performs a different computation on the same data

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
         counter++;
}
```

- If this function is executed by two threads, what will be the value of *counter* when two threads complete?
- Non-deterministic output
- Why?

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
         counter++;
}
```

counter++ in assembly

```
mov (counter), R1
Add 1, R1
Mov R1, (counter)
```

Even on a single processor system, scheduling of threads between the above instructions can be problematic!

Sharing can be problematic!

```
T1: mov (counter), R1    // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}
T2: mov (counter), R1    // R1 = 0
T2: Add 1, R1            // R1 = 1
T2 mov R1, (counter)     // counter = 1
{switch-out, T1 scheduled, R1 = 1}
```

- T2 executes all the instructions for one iteration of the loop, saves 1 to counter (in memory) and then, scheduled out
- T1 is switched-in, R1 value (=1) loaded from the PCB

Sharing can be problematic!

```
T1: mov (counter), R1    // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}
T2: mov (counter), R1    // R1 = 0
T2: Add 1, R1            // R1 = 1
T2 mov R1, (counter)     // counter = 1
{switch-out, T1 scheduled, R1 = 1}
T1: mov R1, (counter)    // counter = 1!
```

- T1 stores one into counter
- Value of counter should have been two
- What if "counter++" is compiled into a single instruction, e.g., "inc (counter)" ?
- Does not solve the issue on multi-processor systems!

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
         counter++;
}
```

- If this function is executed by two threads, what will be the value of *counter* when two threads complete?
- Non-deterministic output
- Why?
- Accessing shared variable in a concurrent manner results in incorrect output

Definitions

- Atomic operation: An operation is atomic if it is *uninterruptible* and *indivisible*
- Critical section: A section of code accessing one or more shared resource(s), mostly shared memory location(s)
- Mutual exclusion: Technique to allow exactly one execution entity to execute the critical section
- Lock: A mechanism used to orchestrate entry into critical section
- Race condition: Occurs when multiple threads are allowed to enter the critical section

Threads sharing the address space

- Threads share the address space
  - Global variables can be accessed from thread functions
- Everything seems to be fine, what is the issue?
- Correctness of program impacted because of concurrent access to the shared data causes race condition
- How does OS fit into this discussion?

assigned to different threads

  - Task parallel processing: Each thread performs a different computation on the same data

Critical sections in OS

- OS maintains shared information which can be accessed from different OS mode execution (e.g., system call handlers, interrupt handlers etc.)
- Example (1): Same page table entry being updated concurrently because of swapping (triggered because of low memory) and change of protection flags (because of mprotect( ) system call)
- Example (2): The queue of network packets being updated concurrently to deliver the packets to a process and receive incoming packets from the network device

Strategy to handle race conditions in OS

Contexts executing critical sections	Uniprocessor systems	Multiprocessor systems
System calls	Disable preemption	Locking
System calls, Interrupt handler	Disable interrupts	Locking + Interrupt disabling (local CPU)
Multiple interrupt handlers	Disable interrupts	Locking + Interrupt disabling (local CPU)

Threads sharing the address space

- Threads share the address space
- Everything seems to be fine, what is the issue?
- Correctness of program impacted because of concurrent access to the shared data causes race condition
- How does OS fit into this discussion?
- Concurrency issues in OS is challenging as finding the race condition itself is non-trivial

on the same data

Locking in pthread: pthread mutex

```
pthread_mutex _t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
void *thfunc(void *)
{
  int ctr = 0;
  for(ctr=0; ctr<100000; ++ctr){
    pthread_mutex_lock(&lock);      // One thread acquires lock, others wait
    counter++;                      // Critical section
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```

Design issues of locks

```
pthread_mutex _t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
    pthread_mutex_lock(&lock);     // One thread acquires lock, others wait
    counter++;                     // Critical section
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```

- Efficiency of lock and unlock operations
- Lock acquisition delay vs. wasted CPU cycles
- Fairness of the locking scheme

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?
- Hardware assisted implementations
  - Use hardware synchronization primitives like atomic operations
- Software locks are implemented without assuming any hardware support
  - Not used in practice because of high overheads

Design issues of locks

```
pthread_mutex_t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
```

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Fairness of the locking scheme

```
    counter++;                          // Critical section
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```

Lock: busy-wait (spinlock) vs. Waiting

T1	T2
lock(L) //Acquired	
Critical section	lock(L) //Lock is busy. Reschedule or Spin?
unlock(L)	Critical section unlock(L)

- With busy waiting, context switch overheads saved, wasted CPU cycles due to spinning
- Busy waiting is prefered when critical section is small and the context executing the critical section is not rescheduled (e.g., due to I/O wait)

Design issues of locks

```
pthread_mutex_t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
```

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Use waiting locks and spinlocks depending on the requirement
- Fairness of the locking scheme

```
    pthread_mutex_unlock(&lock); // Release the lock
  }
}
```

Fairness

- Given *N* threads contending for the lock, number of unsuccessful attempts for lock acquisition for all contending threads should be same
- Bounded wait property
  - Given *N* threads contending for the lock, there should be an upper bound on the number of attempts made by a given context to acquire the lock

Design issues of locks

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Use waiting locks and spinlocks depending on the requirement
- Fairness of the locking scheme
- Contending threads should not starve for the lock indefinitely

CS330: Operating Systems

Locks

Recap: Synchronization and locking

- Locking is necessary when multiple contexts access shared resources
- Example: Multiple threads, multiple OS execution contexts
- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Use waiting locks and spinlocks depending on the requirement
- Fairness of the locking scheme
- Contending threads should not starve for the lock

Agenda: Spinlocks, Semaphore and mutex (waiting locks)

Spinlock: Buggy attempt

```
1.  lock_t *L; // Initial value = 0
2.  lock(L)
3.  {
4.    while(*L);
5.    *L = 1;
6.  }
7.  unlock(L)
8.  {
9.    *L = 0;
10. }
```

- Does this implementation work?
- No, it does not ensure *mutual exclusion*
- Why?
  - Single core: Context switch between line #4 and line #5
  - Multicore: Two cores exiting the while loop by reading lock = 0
- Core issue: Compare and swap has to happen atomically!

Spinlock using atomic exchange

```
1. lock_t *L; // Initial value = 0
2. lock(L)
3. {
4.   while(atomic_xchg(*L, 1));
5. }
6. unlock(L)
7. {
8.   *lock = 0;
9. }
```

- Atomic exchange: exchange the value of memory and register atomically
- atomic_xchg (int *PTR, int val) returns the value at PTR before exchange
- Ensures mutual exclusion if “val” is stored on a register
- No fairness guarantees

Spinlock using XCHG on X86

```
lock(lock_t *L )
{
   asm volatile(
   “mov $1, %%rax;”
   “loop: xchg %%rax, (%%rdi);”
   “cmp $0, %%rax;”
    “jne loop;”
     : : : “memory” );
}
unlock(int *L ) { *L = 0;}
```

- XCHG *R*, *M* ⇒ Exchange value of register *R* and value at memory address *M*
- *RDI* register contains the lock argument
- Exercise: Visualize a context switch between any two instructions and analyse the correctness

Spinlock using compare and swap

```
1.  lock_t *L; // Initial value = 0
2.  lock(L)
3.  {
4.    while( CAS(*L, 0, 1) );
5.  }
6.  unlock(L)
7.  {
8.    *lock = 0;
9.  }
```

- Atomic compare and swap: perform the condition check and swap atomically
- CAS (int **PTR*, int *cmpval*, int *newval*) sets the value of *PTR* to *newval* if *cmpval* is equal to value at *PTR* . Returns 0 on successful exchange
- No fairness guarantees!

CAS on X86: cmpxchg

cmpxchg source[Reg] destination [Mem/Reg]
Implicit registers : rax and flags

```
1.  if rax == [destination]
2.  then
3.      flags[ZF] = 1
4.      [destination] = source
5.  else
6.      flags[ZF] = 0
7.      rax = [destination]
```

- "cmpxchg" is not atomic in X86, should be used with a "lock" prefix

Spinlock using CMPXCHG on X86

```
lock(lock_t *L )
{
asm volatile(
    “mov $1, %%rcx;”
    “loop: xor %%rax, %%rax;”
    “lock cmpxchg %%rcx, (%%rdi);”
    “jnz loop;”
    : : : “rcx”, “rax”, “memory”);
}
unlock(lock_t *L ) { *L = 0;}
```

- Value of RAX (=0) is compared against value at address in register RDI and exchanged with RCX (=1), if they are equal
- Exercise: Visualize a context switch between any two instructions and analyse the correctness

Load Linked (LL) and Store conditional (SC)

- LoadLinked (R, M)
  - Like a normal load, it loads R with value of M
  - Additionally, the hardware keeps track of future stores to M
- StoreConditional (R, M)
  - Stores the value of R to M if no stores happened to M after the execution of LL instruction (after execution, R = 1)
  - Otherwise, store is not performed (after execution R=0)
- Supported in RISC architectures like mips, risc-v etc.

Spinlock using LL and LC

```
lock_t *L;    //initial value = 0
lock( lock_t *L)
{
 while(LoadLinked(L) ||
       !StoreConditional(L, 1));
}
unlock( lock_t *L) { *L = 0;}
```

```
lock:   LL   R1,  (R2);  //R2 = lock address
        BNEQZ  R1,  lock;
        ADDUI   R1,   R0, #1;    //R1 = 1
        SC R1, (R2)
        BEQZ R1, lock
```

- Efficient as the hardware avoids memory traffic for unsuccessful lock acquire attempts
- Context switch between LL and SC results in SC to fail

Spinlocks: reducing wasted cycles

- Spinning for locks can introduce significant CPU overheads and increase energy consumption
- How to reduce spinning in spinlocks?
- Strategy: Back-off after every failure, exponential back-off used mostly

```
lock( lock_t *L) {
        u64 backoff = 0;
        while(LoadLinked(L) || !StoreConditional(L, 1)){
                if(backoff  < 63)   ++backoff;
                    pause(1 << backoff);   // Hint to processor
}
```

Fairness in spinlocks

- Spinlock implementations discussed so far are not fair,
  - no bounded waiting
- To ensure fairness, some notion of ordering is required
- What if the threads are granted the lock in the order of their arrival to the lock contention loop?
  - A single lock variable may not be sufficient
  - Example solution: Ticket spinlocks

Atomic fetch and add (xadd on X86)

xadd R, M

TmpReg T = R + [M]

R = [M]

[M] = T

- Example: M = 100; RAX = 200
- After executing “lock xadd %RAX, M”, value of RAX = 100, M = 300
- Require lock prefix to be atomic

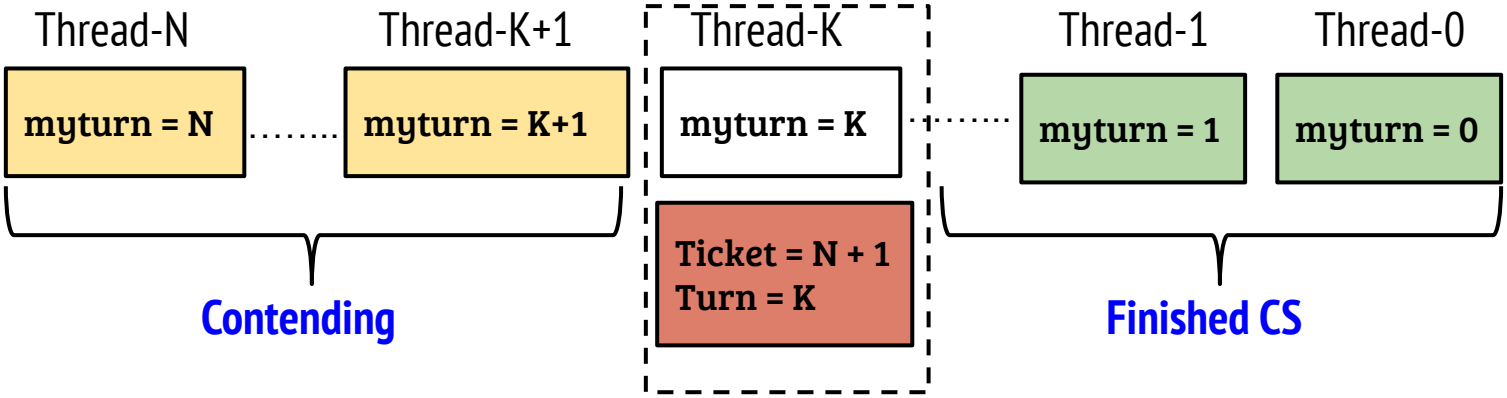
Ticket spinlocks (OSTEP Fig. 28.7)

```
struct lock_t{
        long ticket;
        long turn;
};
void init_lock (struct lock_t *L){
   L → ticket = 0;  L → turn = 0;
}
void unlock(struct lock_t *L){
      L → turn++;
}
```

```
void lock(struct lock_t *L){
  long myturn = xadd(&L → ticket, 1);
  while(myturn != L → turn)
        pause(myturn - L → turn);
}
```

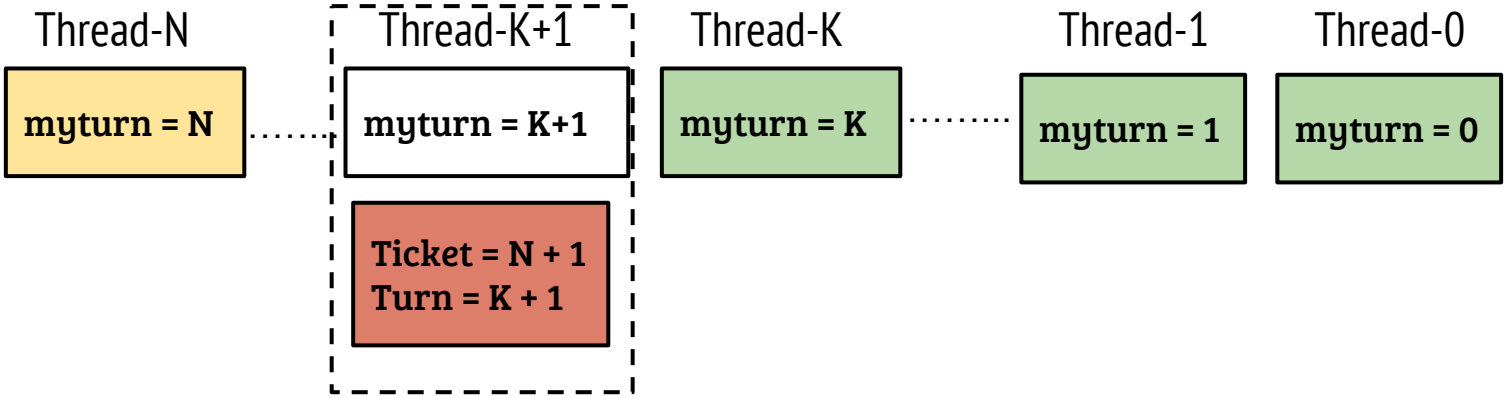
- Example: Order of arrival: T1 T2 T3
- T1 (in CS) : myturn = 0, L = {1, 0}
- T2: myturn = 1, L = {2, 0}
- T3: myturn = 2, L = {3,0}
- T1 unlocks, L = {3, 1}. T2 enters CS

Ticket spinlock



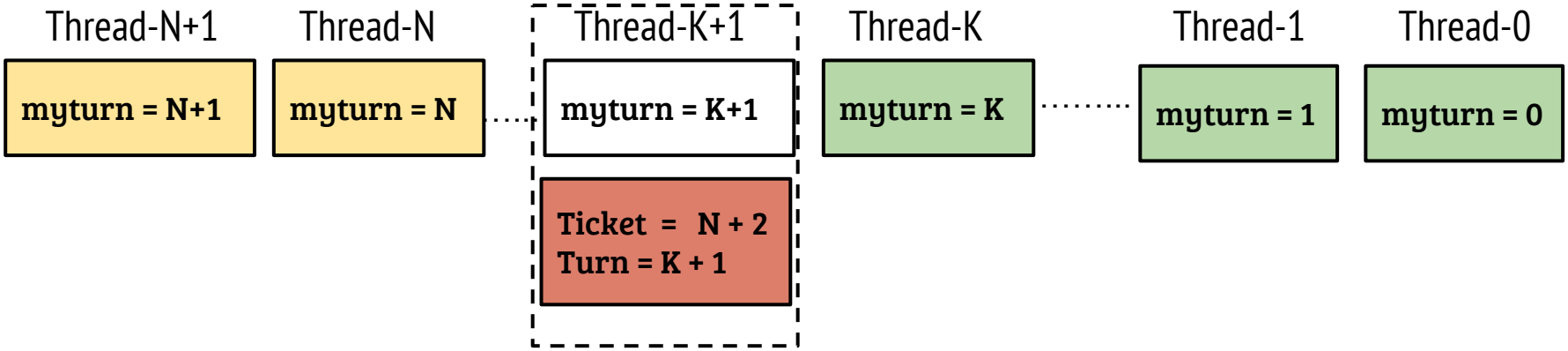
- Local variable “myturn” is equivalent to the order of arrival
- If a thread is in CS ⇒ Local Turn must be same as “Turn”
- Threads waiting = Ticket - Turn -1

Ticket spinlock



- Value of turn incremented on lock release
- Thread which arrived just after the current thread enters the CS
- When a new thread arrives, it gets the lock after the other threads ahead of the new thread acquire and release the lock

Ticket spinlock



- Ticket spinlock guarantees bounded waiting
- If *N* threads are contending for the lock and execution of the CS consumes *T* cycles, then bound = N * T (assuming negligible context switch overhead)

Ticket spinlock (with yield)

```
void lock(struct lock_t *L){
  long myturn = xadd(&L → ticket, 1);
  while(myturn != L → turn)
        sched_yield( );
}
```

- Why spin if the thread's turn is yet to come?
- Yield the CPU and allow the thread with ticket (or other non contending threads)
- Further optimization
  - Allow the thread with "myturn" value one more than "L⟶ turn" to continue spinning

Reader-writer locks

- Allows *multiple readers* or *a single writer* to enter the CS
- Example: Insert, delete and lookup operations on a search tree

```
struct BST{
        struct node *root;
        rwlock_t *lock;
};
```

```
struct node{
        item_t item;
        struct node *left;
        struct node*right;
};
```

```
void insert(BST *t, item_t item);
void lookup(BST *t, item_t item);
```

- If multiple threads call lookup( ), they may traverse the tree in parallel

Implementation of read-write locks (writers)

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}

void write_lock(rwlock_t *rL)
{
    lock(&rL → write_lock);
}
```

```
init_lock(rwlock_t *rL)
{
   init_lock(&rL → read_lock);
   init_lock(&rL → write_lock);
   rL → num_readers = 0;
}

   void write_unlock(rwlock_t *rL)
   {
       unlock(&rL → write_lock);
   }
```

- Write lock behavior is same as the typical lock, only one thread allowed to acquire the lock

Implementation of read-write locks (readers)

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
void read_lock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers++;
   if(rL → num_readers == 1)
       lock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

- The first reader acquires the write lock prevents writers to acquire lock
- The last reader releases the write lock to allow writers

```
void read_unlock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers--;
   if(rL → num_readers == 0)
       unlock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

Software lock: Buggy #1

```
int flag[2] = {0,0};
void lock (int id)    /*id = 0 or 1 */
{
    while(flag[id ∧ 1]));  // ∧ → XOR
    flag[id] = 1;
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Solution for two threads, T_0 and T_1 with id 0 and 1, respectively
- We have seen that this solution does not work, Why?
- Both threads can acquire the lock as "while condition check" and "setting the flag" is non-atomic

Software lock: Buggy #2

```
int flag[2] = {0,0};
void lock (int id)    /*id = 0 or 1 */
{
    flag[id] = 1;
    while(flag[id ∧ 1]));  // ∧ → XOR
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Does this solution work?
- No, as this can lead to a deadlock (flag[0] = flag[1] = 1) In other words the "progress" requirement is not met
- Progress: If no one has acquired the lock and there are contending threads, one of the threads must acquire the lock within a finite time

Software lock: Buggy #3

```
int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   while(turn == id ∧ 1));
}
void unlock (int id)
{
    turn = id ∧ 1;
}
```

- Assuming T_0 invokes lock( ) first, does the solution provide mutual exclusion?
- Yes it does, but there is another issue with this solution - two threads must request the lock in an alternate manner
- Progress requirement is not met
  - Argument: one of the threads stuck in an infinite loop (in non-CS code)

Peterson's solution

```
int flag[2] = {0,0};  int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   flag[id] = 1;
   turn  = id ^1;
   while(flag[id ^ 1]) && turn == (id ^1));
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Homework: Prove that mutual exclusion is guaranteed
- What about fairness?
- The lock is fair because if two threads are contending, they acquire the lock in an alternate manner
- Extending the solution to N threads is possible

CS330: Operating Systems

Semaphore, Classical problems

Semaphores

- Mutual exclusion techniques allows exactly one thread to access the critical section which can be restrictive
- Consider a scenario when a finite array of size N is accessed from a set of producer and consumer threads. In this case,
  - At most N concurrent producers are allowed if array is empty
  - At most N concurrent consumers are allowed if array is full
  - If we use mutual exclusion techniques, only one producer or consumer is allowed at any point of time

Operations on semaphore

```
struct semaphore{
                    int value;
                    spinlock_t *lock;
                    queue *waitQ;
                }sem_t;

// Operations

sem_init(sem_t *sem, int init_value);
sem_wait(sem_t *sem);
sem_post(sem_t *sem);
```

- Semaphores can be initialized by passing an initial value
- *sem_wait* waits (if required) till the value becomes +ve and returns after decrementing the value
- *sem_post* increments the value and wakes up a waiting context
- Other notations: P-V, down-up, wait-signal

Unix semaphores

```
#include <semaphore.h>

main( ){
    sem_t  s;
    int K = 5;
    sem_init(&s, 0, K);
    sem_wait(&s);
    sem_post(&s);
}
```

- Can be used to in a multi-threaded process or across multiple processes
- If second argument is 0, the semaphore can be used from multiple threads
- Semaphores initialized with value = 1 (third argument) is called a binary semaphore and can be used to implement *blocking(waiting) locks*
- Initialize: sem_init(s, 0, 1)
  lock:sem_wait(s), unlock: sem_post(s)

Semaphore usage example: wait for child

```
child( ){
      ...
     sem_post(s);
     exit(0);
}
int main (void ){
          sem_init(s, 0);
         if(fork( ) == 0)
               child( );
         sem_wait(s);
}
```

- Assume that the semaphore is accessible from multiple processes, value initialized to zero
- If parent is scheduled after the child creation, it waits till child finishes
- If child is scheduled and exits before parent, parent does not wait for the semaphore

Semaphore usage example: ordering

```
A=0; B=0;

Thread-0 {
    A = 1;
    printf("B = %d\n", B);
}

Thread-1 {
        B=1;
        printf("A = %d\n", A);
}
```

- What are the possible outputs?
- (A = 1, B= 1), (A = 1, B = 0), (A = 0, B=1)
- How to guarantee A = 1, B= 1?

Semaphore usage example: ordering

```
sem_init(s1, 0);
A=0; B=0;
Thread - 0 {
    A = 1;
    sem_wait(s1);
    printf("B = %d\n", B);
}
Thread - 1 {
        B=1;
        sem_post(s1);
        printf("A = %d\n", A);
}
```

- What are the possible outputs?
- (A = 1, B = 1), (A=0, B=1)
- How to guarantee A = 1, B= 1?

Ordering with two semaphores

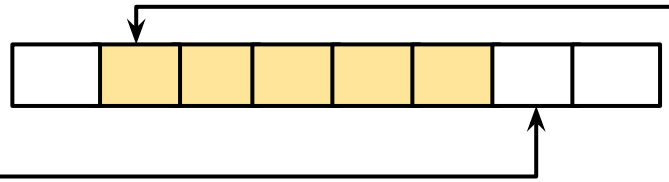
```
sem_init(s1, 0);
sem_init(s2, 0)
A=0; B=0;

Thread - 0
{
    A = 1;
    sem_post(s1);
    sem_wait(s2);
    printf("%d\n", B);
}
```

```
Thread - 1
{
    B=1;
    sem_wait(s1);
    sem_post(s2);
    printf("%d\n", A);
}
```

- Waiting for each other guarantees desired output

Producer-consumer problem

```
DoProducerWork( ){
 while(1){
   item_t item = prod_p( );
   produce(item);
 }
}
```



```
DoConsumerWork( ){
 while(1){
   item_t item = consume( );
   cons_p(item);
 }
}
```

- A buffer of size N, one or more producers and consumers
- Producer produces an element into the buffer (after processing)
- Consumer extracts an element from the buffer and processes it
- Example: A multithreaded web server, network protocol layers etc.
- How to solve this problem using semaphores?

Buggy #1

```
item_t A[n], pctr=0, cctr = 0;
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used);
    item_t item = A[cctr];;
    cctr = (cctr + 1) % n;
    sem_post(&empty);
    return item;
}
```

- This solution does not work. What is the issue?
- The counters (pctr and cctr) are not protected, can cause race conditions

Buggy #2

```
item_t A[n], pctr=0, cctr = 0; lock_t *L = init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    lock(L);  sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);  unlock(L);
}
```

```
item_t consume( ) {
    lock(L);  sem_wait(&used);
    item_t  item = A[cctr];;
    cctr = (cctr + 1) % n;
    sem_post(&empty); unlock(L);
    return item;
}
```

- What is the problem?
- Consider empty = 0 and producer has taken lock before the consumer. This results in a deadlock, consumer waits for L and producer for empty

A working solution

```
item_t A[n], pctr=0, cctr = 0;  lock_t *L = init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty); lock(L);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    unlock(L); sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used); lock(L)
    item_t item = A[cctr];
    cctr = (cctr + 1) % n;
    unlock(L); sem_post(&empty);
    return item;
}
```

- The solution is deadlock free and ensures correct synchronization, but very much serialized (inside produce and consume)
- What if we use separate locks for producer and consumer?

Solution with separate mutexes

```
item_t A[n], pctr=0, cctr = 0;  lock_t *P = init_lock( ), *C=init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty); lock(P);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    unlock(P); sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used); lock(C)
    item_t item = A[cctr];
    cctr = (cctr + 1) % n;
    unlock(C); sem_post(&empty);
    return item;
}
```

- Does this solution work?
- Homework: Assume that item is a large object and copy of item takes long time. How can we perform the copy operation without holding the lock?

CS330: Operating Systems

Concurrency bugs

Common issues in concurrent programs

- Atomicity issues
- Failure of ordering assumption
- Deadlocks

Concurrency bugs - atomicity issues

```
char *ptr;   // Allocated before use
void T1( )
{
    ...
    strcpy(ptr, “hello world!”);
   …
}
```

```
void T2( )
{
    ...
     if(some_condition){
          free(ptr); ptr = NULL;
     }
    ...
}
```

- This code is buggy. What is the issue?
- T2 can free the pointer before T1 uses it.
- How to fix it?

Concurrency bugs - atomicity issues

```
char *ptr;  // Allocated before use
void T1( )
{
    ...
    if(ptr)  strcpy(ptr, “hello world!”);
   …
}
```

```
void T2( )
{
    ...
     if(some_condition){
          free(ptr); ptr = NULL;
     }
    ...
}
```

- Does the above fix (checking ptr in T1) work?
- Not really. Consider the following order of execution:
- T1: “if(ptr)” T2: “free(ptr)” T1: “strcpy” Result: Segfault

Concurrency bugs - ordering issues

```
1.  bool pending;
2.  void T1( )
3.  {
4.      pending = true;
5.      do_large_processing( );
6.      while (pending);
7.  }
```

```
1.  void T2( )
2.  {
3.      do_some_processing( );
4.      pending = false;
5.      some_other_processing( );
6.  }
```

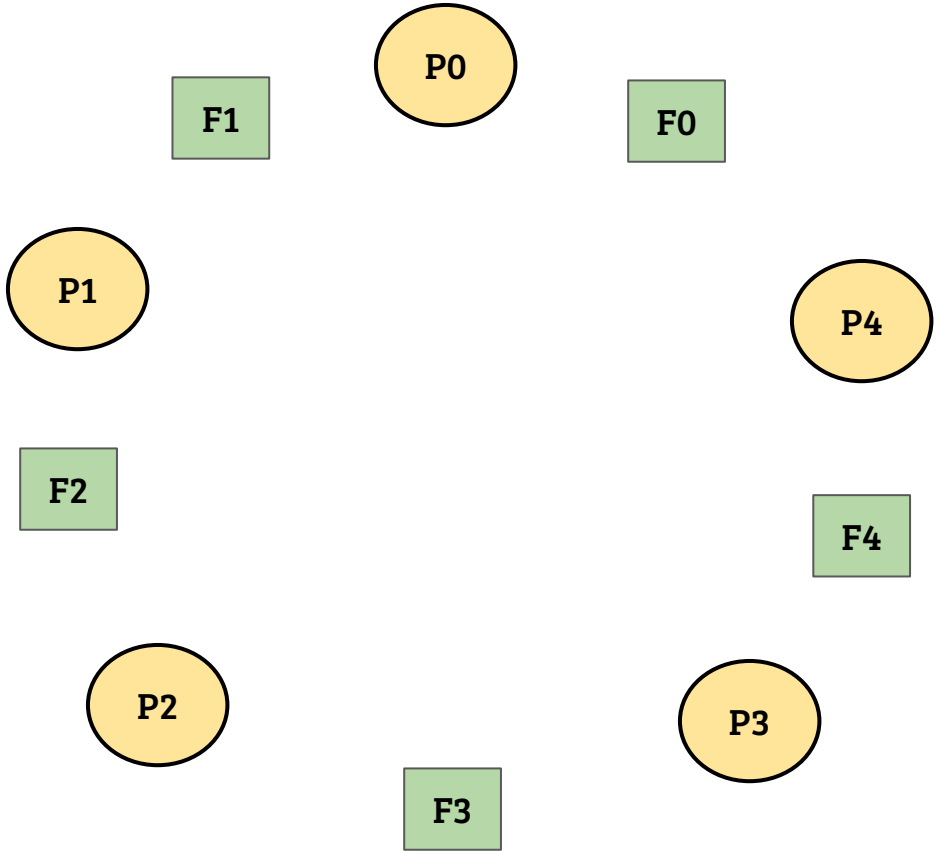
- This code works with the assumption that line#4 of T2 is executed after line#4 of T1
- If this ordering is violated, T1 is stuck in the while loop

Concurrency bugs - deadlocks

```
struct acc_t{
        lock_t *L;
        id_t  acc_no;
        long balance;
}
void txn_transfer( acc_t *src,
                    acc_t *dst,  long amount)
{
   lock(src → L);  lock(dst → L);
   check_and_transfer(src, dst, amount);
   unlock(dst → L); unlock(src → L);
}
```

- Consider a simple transfer transaction in a bank
- Where is the deadlock?
- T1: txn_transfer(iitk, cse, 10000)
  - lock (iitk), lock (cse)
- T2: txn_transfer(cse, iitk, 5000)
  - lock (cse), lock(iitk)

Dining philosophers




```
atomic_t forks[5];
Philosopher( int id)
{
   while (1) {
      think( );
      acquire(forks[id]);
      acquire(forks[(id+1) % 5]);
      eat( );
      release( forks[(id+1) % 5]);
      release(forks[id]);
   }
}
```

Conditions for deadlock

- Mutual exclusion: exclusive control of resources (e.g, thread holding lock)
- Hold-and-wait: hold one resource and wait for other
- No resource preemption: Resources can not be forcibly removed from threads holding them
- Circular wait: A cycle of threads requesting locks held by others. Specifically, a cycle in the directed graph G (V, E) where V is the set of processes and (v1, v2) $\in$ E if v1 is waiting for a lock held by v2

All of the above conditions should be satisfied for a deadlock to occur

Solutions for deadlocks

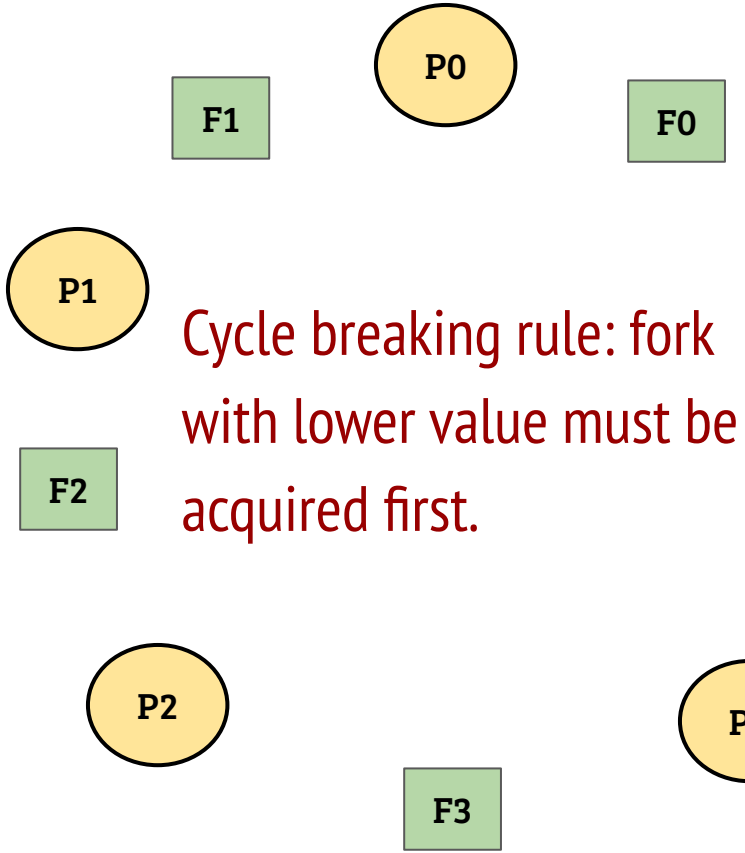
- Remove mutual exclusion: lock free data structures
- Either acquire all resources or no resource
  - trylock(lock) APIs can be used (e.g., pthread_mutex_trylock( ))
- Careful scheduling: Avoid scheduling threads such that no deadlock occur
- Most commonly used technique is to avoid circular wait. This can be achieved by ordering the resources and acquiring them in a particular order from all the threads.

Concurrency bugs - avoiding deadlocks

```
struct acc_t{
        lock_t *L;
        id_t  acc_no;
        long balance;
}
void txn_transfer( acc_t *src,
                   acc_t *dst,  long amount)
{
    lock(src → L);  lock(dst → L);
    check_and_transfer(src, amount);
    unlock(dst → L); unlock(src → L);
}
```

- Deadlock in a simple transfer transaction in a bank
- While acquiring locks, first acquire the lock for the account with lower “acc_no” value
- Account number comparison performed before acquiring the lock

Dining philosophers: breaking the deadlock



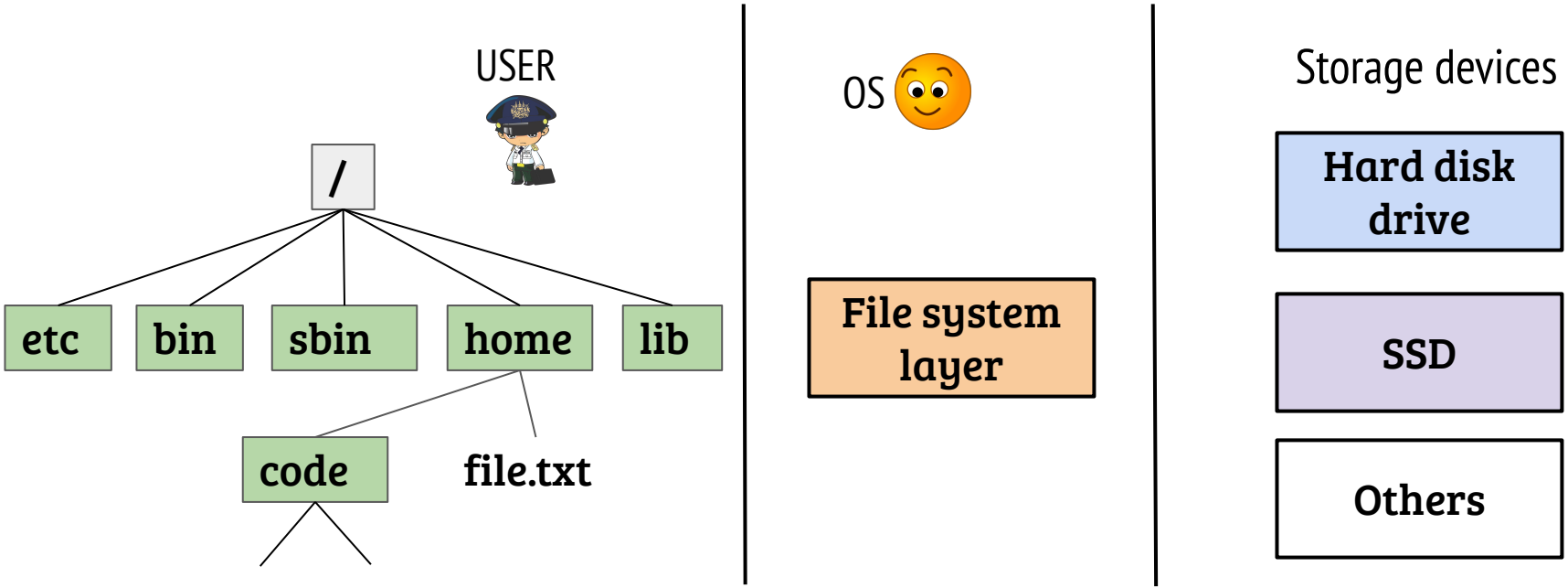
Cycle breaking rule: fork with lower value must be acquired first.

```
atomic_t forks[5];
Philosopher( int id)
{
   while (1) {
      if(id == 4){
         acquire(0);
         acquire(4);
      }else{
        acquire(forks[id]);
        acquire(forks[id+1]);
      }
      ...
}
```

CS330: Operating Systems

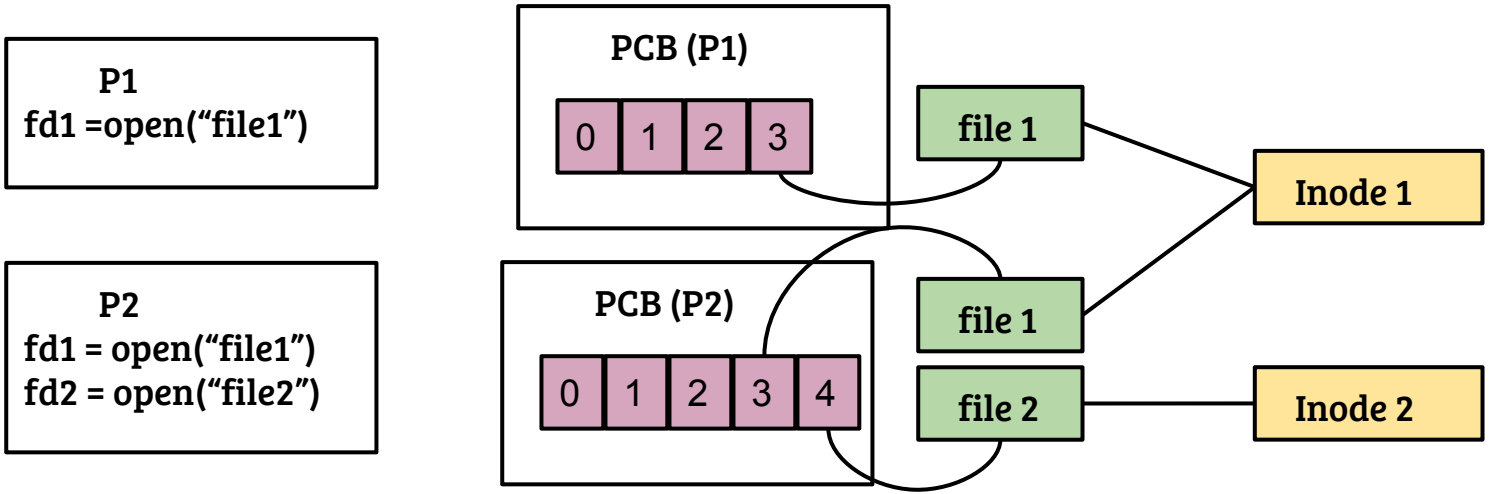
Filesystem

Recap: file system



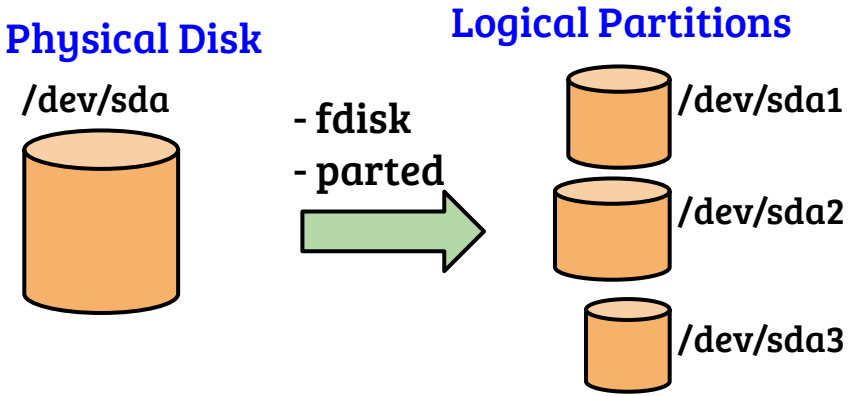
- File system is an important OS subsystem
  - Provides abstractions like files and directories
  - Hides the complexity of underlying storage devices

Recap: Process view of file



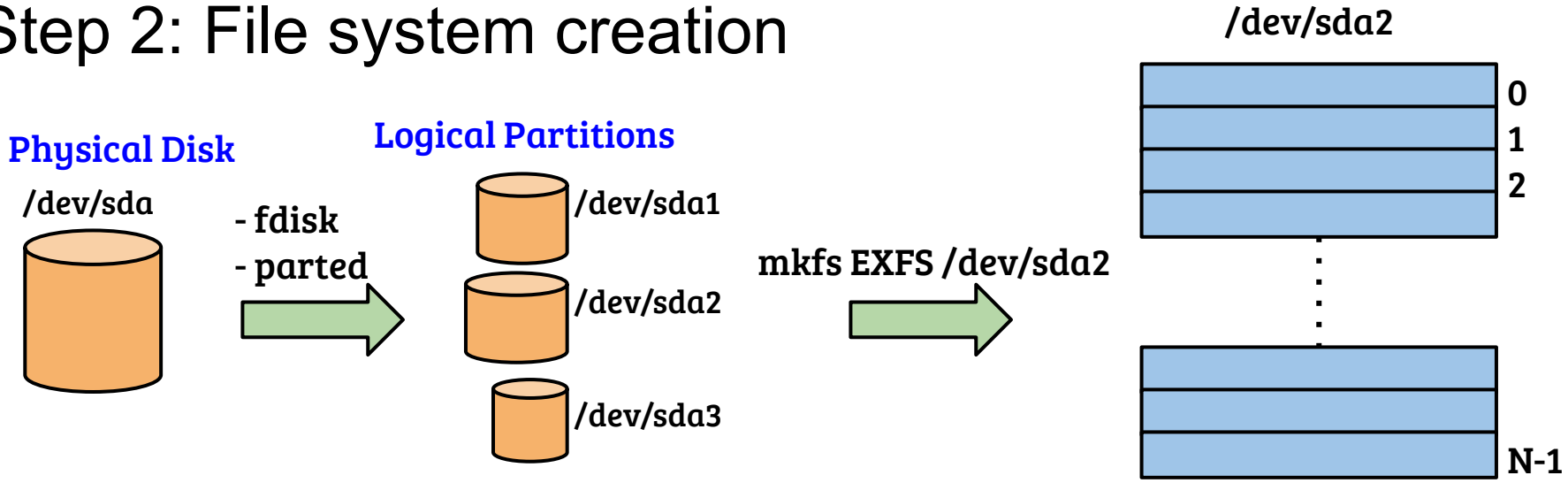
- Per-process file descriptor table with pointer to a "file" object
- file object → inode (in-memory) is many-to-one
- How is the inode maintained in a persistent manner? How to access data at different offsets of a file? How directory structure is maintained?

Step-1: Disk device partitioning



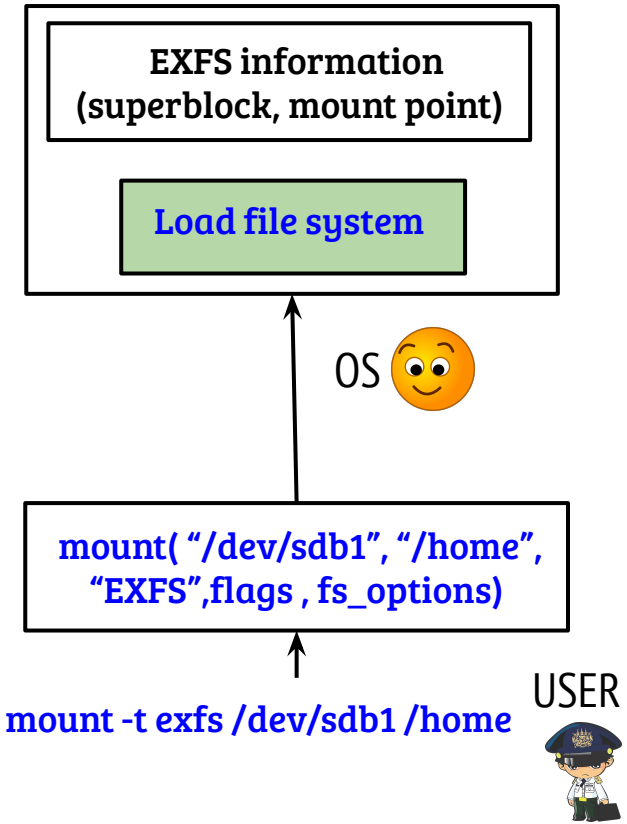
- Partition information is stored in the boot sector of the disk
- Creation of partition is the first step
  - It does not create a file system
- A file system is created on a partition to manage the physical device and present the logical view
- All file systems provide utilities to initialize file system on the partition (e.g., MKFS)

Step 2: File system creation



- MKFS creates initial structures in the logical partition
  - Creates the entry point to the filesystem (known as the super block)
  - At this point the file system is ready to be mounted

Step 3: File system mounting



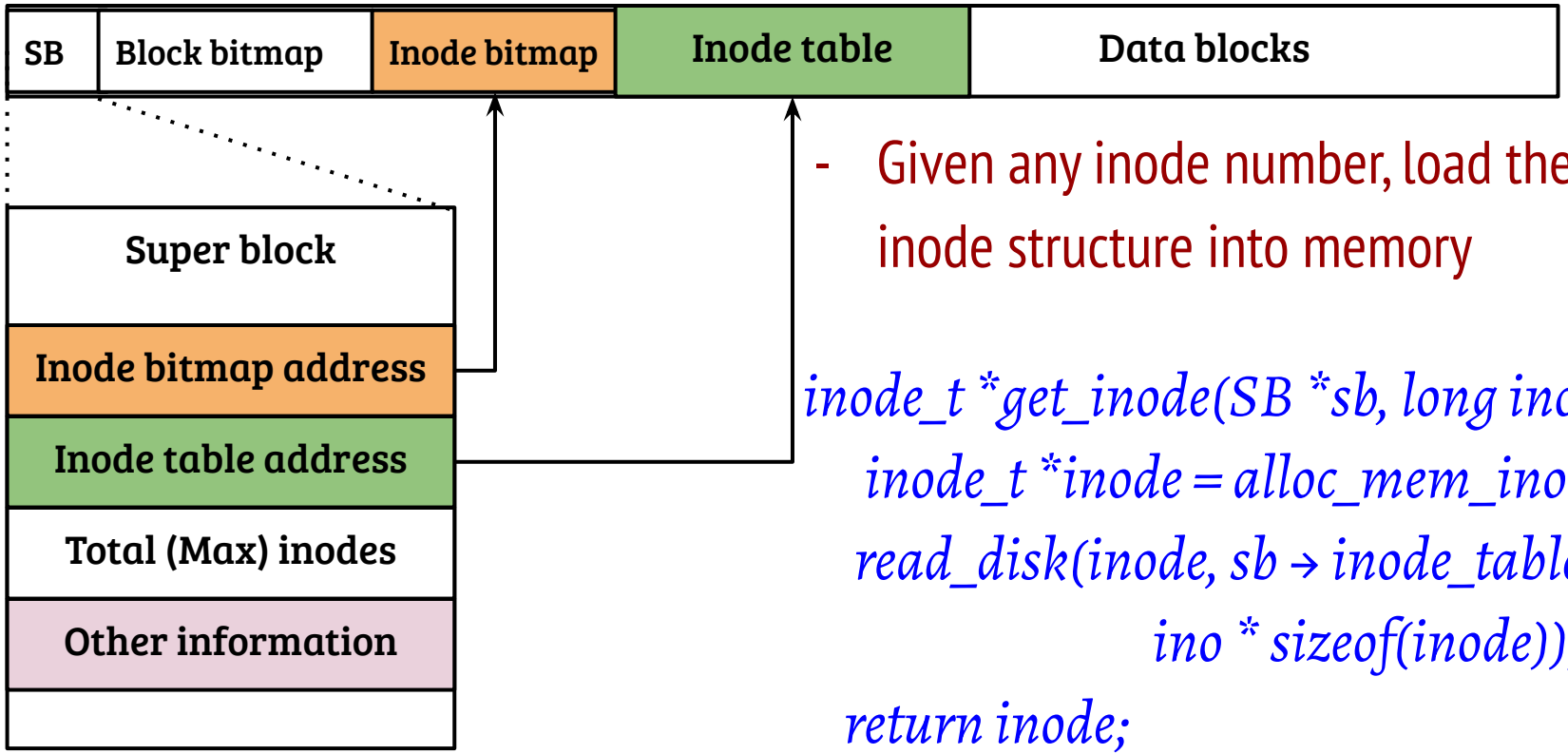
- *mount( )* associates a superblock with the file system mount point
- Example: The OS will use the superblock associated with the mount point "/home" to reach any file/dir under "/home"
- Superblock is a copy of the on-disk superblock along with other information

Structure of an example superblock

```
struct superblock{
    u16 block_size;
    u64 num_blocks;
    u64 last_mount_time;
    u64 root_inode_num;
    u64 max_inodes;
    disk_off_t inode_table;
    disk_off_t blk_usage_bitmap;
     …
};
```

- Superblock contains information regarding the device and the file system organization in the disk
- Pointers to different metadata related to the file system are also maintained by the superblock
  - Ex: List of free blocks is required before adding data to a new file/directory

File system organization



- Given any inode number, load the inode structure into memory

```
inode_t *get_inode(SB *sb, long ino){
    inode_t *inode = alloc_mem_inode( );
    read_disk(inode, sb → inode_table +
                        ino * sizeof(inode));
    return inode;
}
```

File system organization

- File system is mounted, the inode number for root of the file system (i.e., the mount point) is known, root inode can be accessed.
- How to search/lookup files/directories under root inode?
- Specifically,
  - How to locate the content in disk?
  - How to keep track of size, permissions etc.?

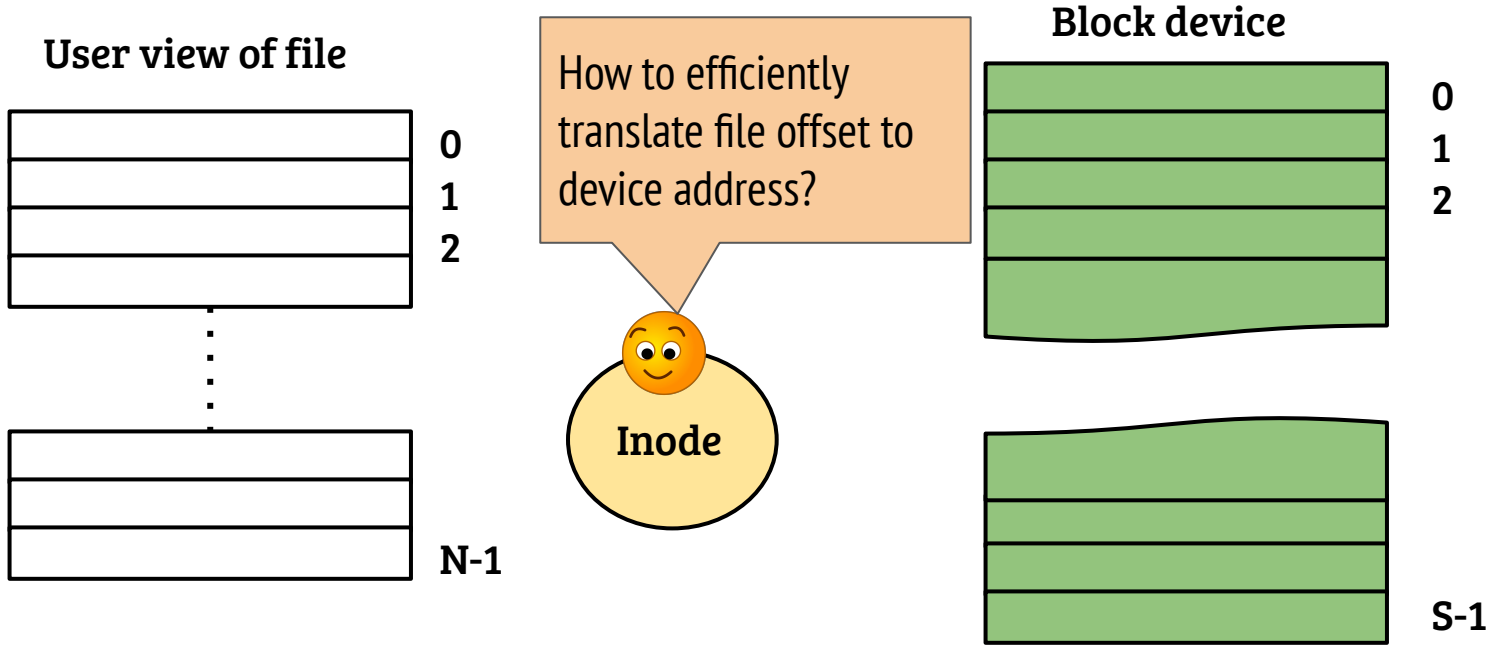
return inode;

}

Inode

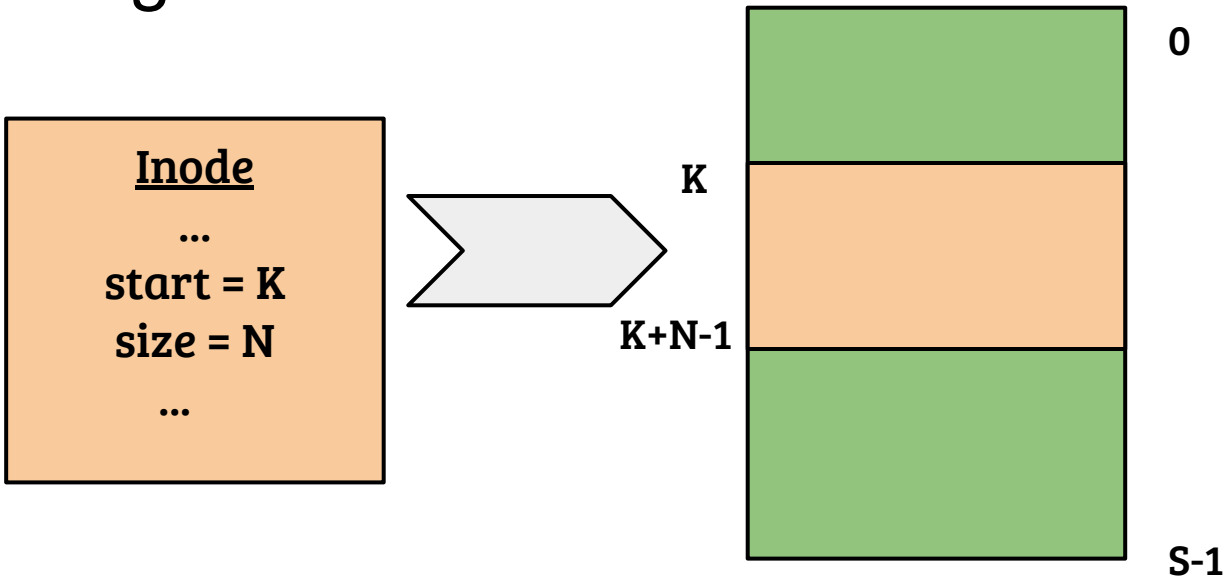
- A on-disk structure containing information regarding files/directories in the unix systems
  - Represented by a unique number in the file system (e.g., in Linux, “ls -i filename” can be used to print the inode)
  - Contains access permissions, access time, file size etc.
  - *Most importantly, inode contains information regarding the file data location on the device*
- Directory inodes also contain information regarding its content, albeit the content is structured (for searching files)

Problem: file offset to disk address mapping



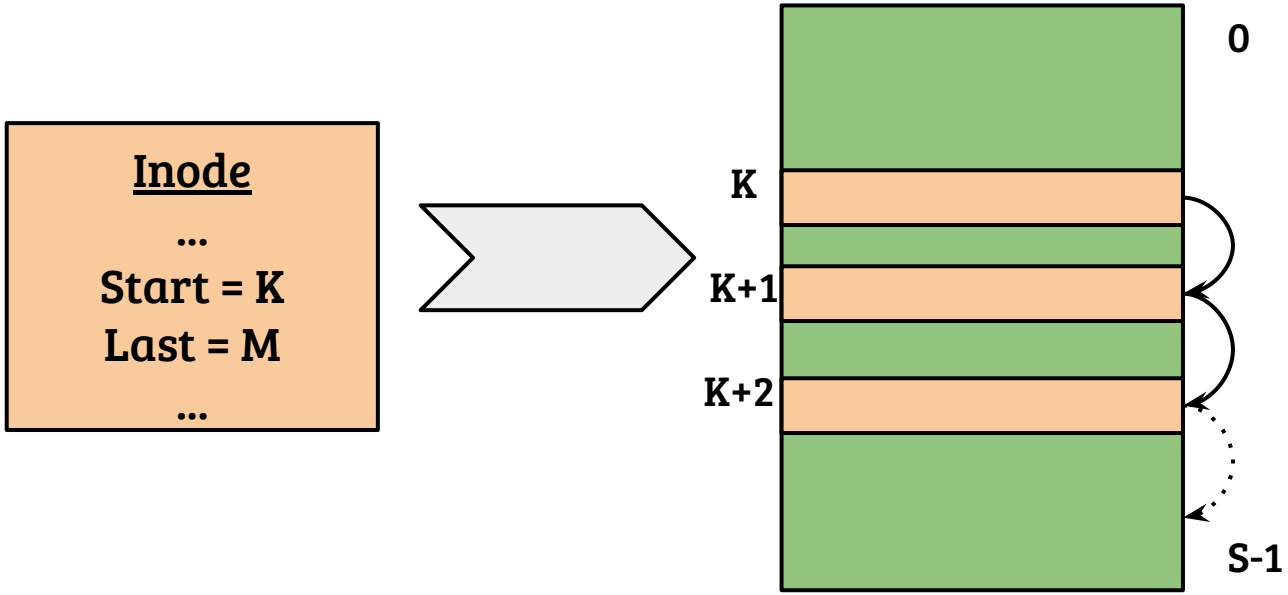
- File size can range from few bytes to gigabytes
- Can be accessed in a sequential or random manner
- How to design the mapping structure?

Contiguous allocation



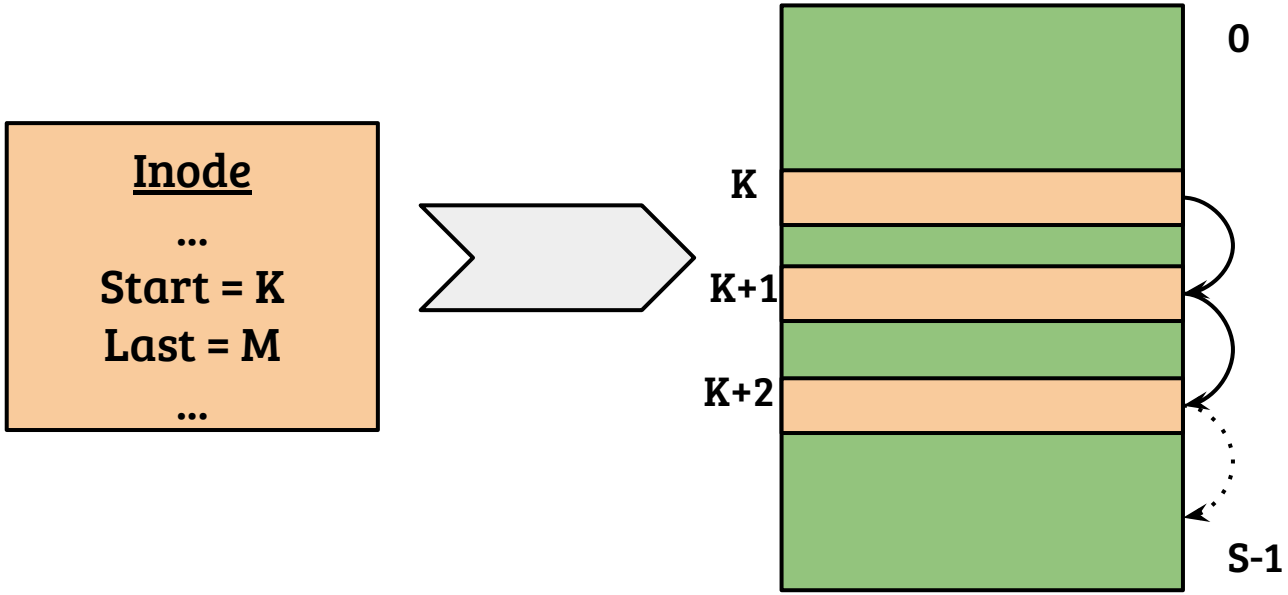
- Works nicely for both sequential and random access
- Append operation is difficult, How to expand files? Require relocation!
- External fragmentation is a concern

Linked allocation



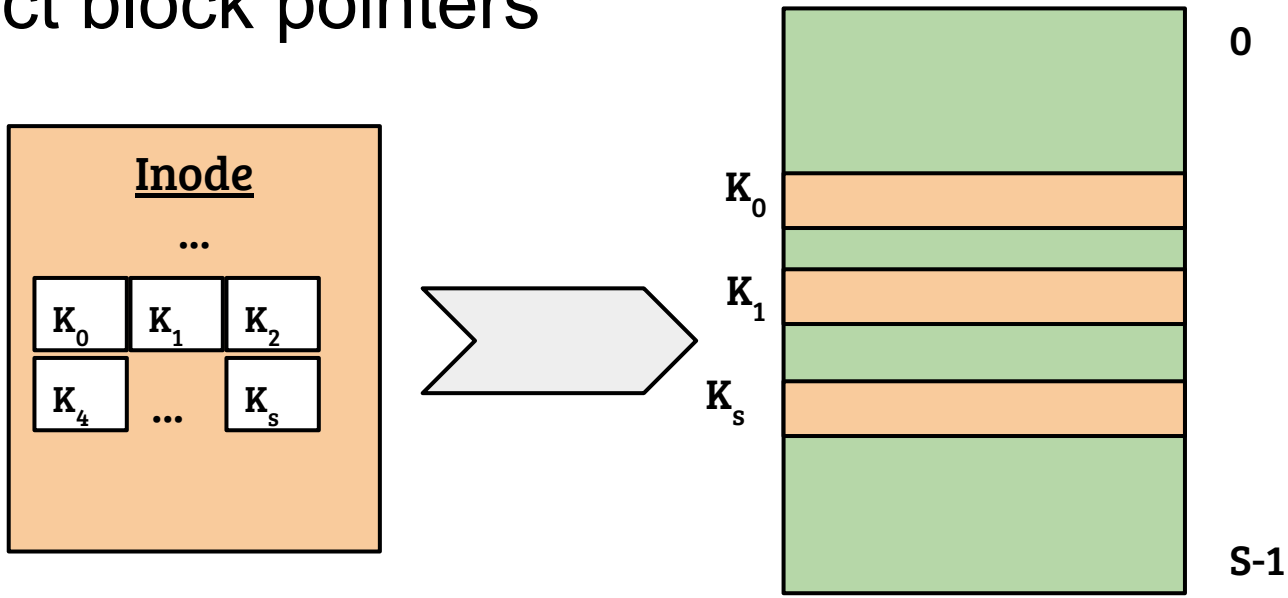
- Every block contains pointer to next block
- Advantage: flexible, easy to grow and shrink, Disadvantage: random access
- Why maintain last block not size?

Linked allocation



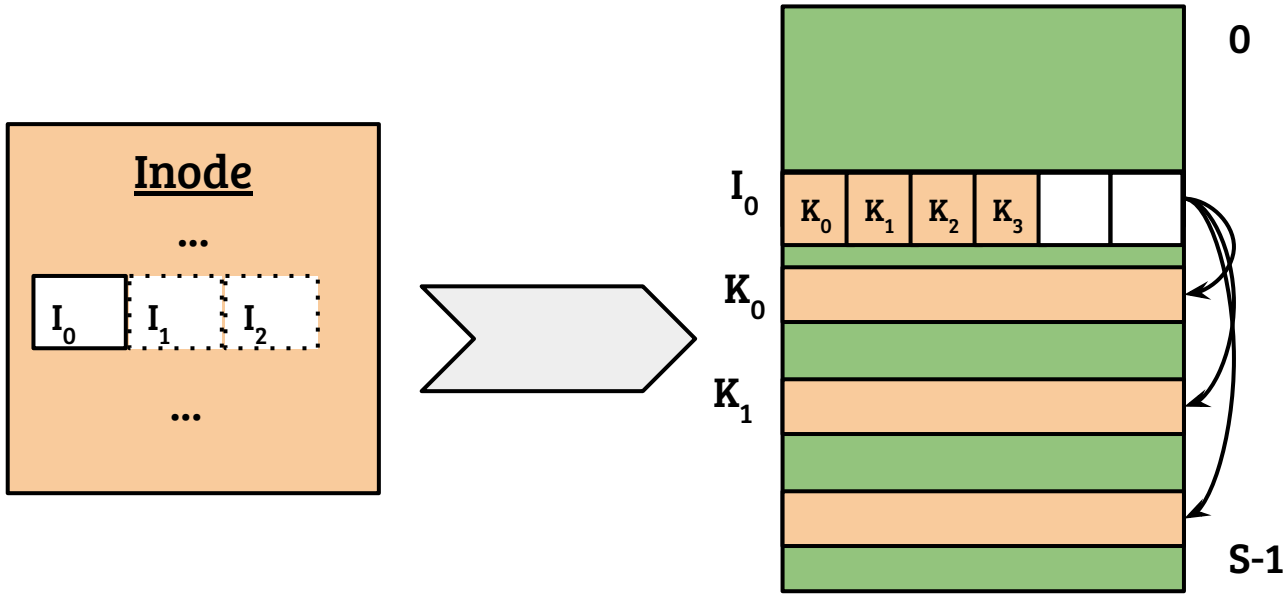
- Every block contains pointer to next block
- Advantage: flexible, easy to grow and shrink, Disadvantage: random access
- Why maintain last block not size? Efficient append operation!

Direct block pointers



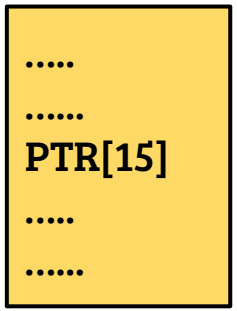
- Inode contains direct pointers to the block
- Flexible: growth, shrink, random access is good
- Can not support files of larger size!

Indirect block pointers



- Inode contains the pointers to a block containing pointers to data blocks
- Advantages: flexible, random access is good
- Disadvantages: Indirect block access overheads (even for small files)

Hybrid block pointers: Ext2 file system



Direct pointers {PTR [0] to PTR [11]}

File block address (0 -11)

Single indirect {PTR [12]}

File block address (12 -1035)

Double indirect {PTR [13]}

File block address (1036 to 1049611)

Triple indirect {PTR [14]}

File block address (?? to ??)

File system organization

- File system is mounted, the inode number for root of the file system (mount point) is known, root inode can be accessed. However,
- How to search/lookup files/directories under root inode?
- Specifically,
  - How to locate the content in disk?
  - Index structures in inode are used to map file offset to disk location
  - How to keep track of size, permissions etc.?
  - Inode is used to maintain these information

Organizing the directory content

Fixed size directory entry

```
struct dir_entry{
        inode_t  inode_num;
        char name[FNAME_MAX];
};
```

- Fixed size directory entry is a simple way to organize directory content
- Advantages: avoid fragmentation, rename
- Disadvantages: space wastage

Organizing the directory content

Fixed size directory entry

```
struct dir_entry{
        inode_t  inode_num;
        char name[FNAME_MAX];
};
```

Variable size directory entry

```
struct dir_entry{
        inode_t  inode_num;
        u8 entry_len;
        char name[name_len];
};
```

- Variable sized directory entries contain length explicitly
- Advantages: less space wastage (compact)
- Disadvantages: inefficient rename, require compaction

File system organization

- File system is mounted, the inode number for root of the file system (mount point) is known, root inode can be accessed. However,
- How to search/lookup files/directories under root inode?
- Read the content of the root inode and search the next level dir/file
- Specifically,
  - How to locate the content in disk?
  - Index structures in inode are used to map file offset to disk location
  - How to keep track of size, permissions etc.?
  - Inode is used to maintain these information

CS330: Operating Systems

Filesystem: caching and consistency

Recap: file system organization

- File systems maintain several meta-data structures like super blocks, inodes, directory entries to provide a file system abstractions like files, directories
- How to search/lookup files/directories in a given path?
- Read the content of the root inode and search the next level dir using the name and find out its inode number
- Read the inode to check permissions and repeat the process
- Inode contains the index structures to deduce the disk block address given an logical offset

}

File system and caching

- Accessing data and metadata from disk impacts performance
- Many file operations require multiple block access
- Examples:
  - Opening a file

```
fd = open("/home/user/test.c", O_RDWR);
```

  - Normal shell operations

```
/home/user$ ls
```

File system and caching

- Accessing data and metadata from disk impacts performance
- Many file operations require multiple block access
- Executables, configuration files, library etc. are accessed frequently
- Many directories containing executables, configuration files are also accessed very frequently. Metadata blocks storing inodes, indirect block pointers are also accessed frequently

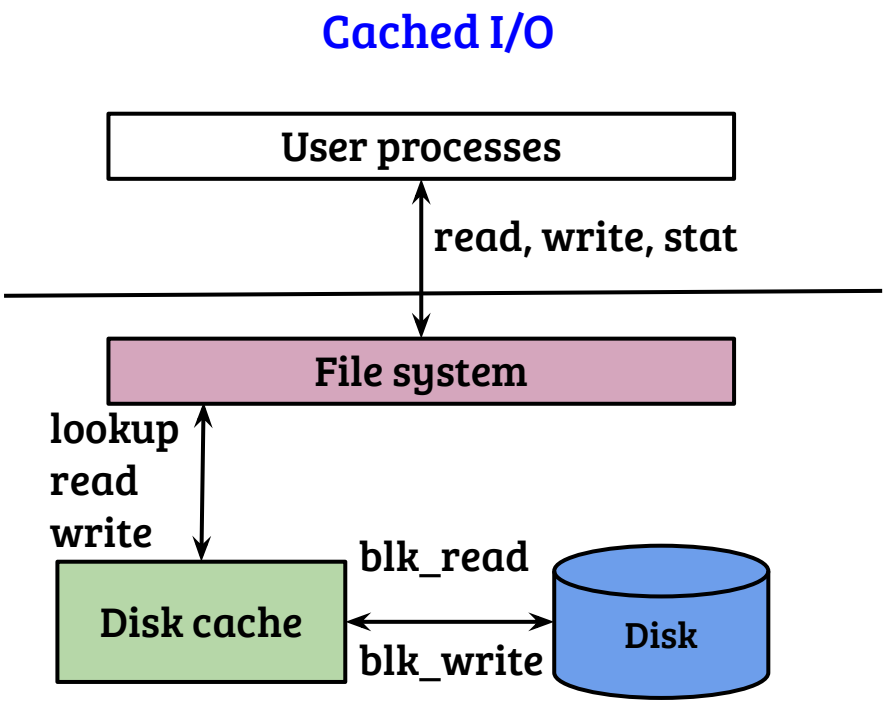
/home/user$ ls

File system and caching

- Accessing data and metadata from disk impacts performance
- Can we store frequently accessed disk data in memory?
  - What is the storage and lookup mechanism? Are the data and metadata caching mechanisms same?
  - Are there any complications because of caching?
  - How the cache managed? What should be the eviction policy?

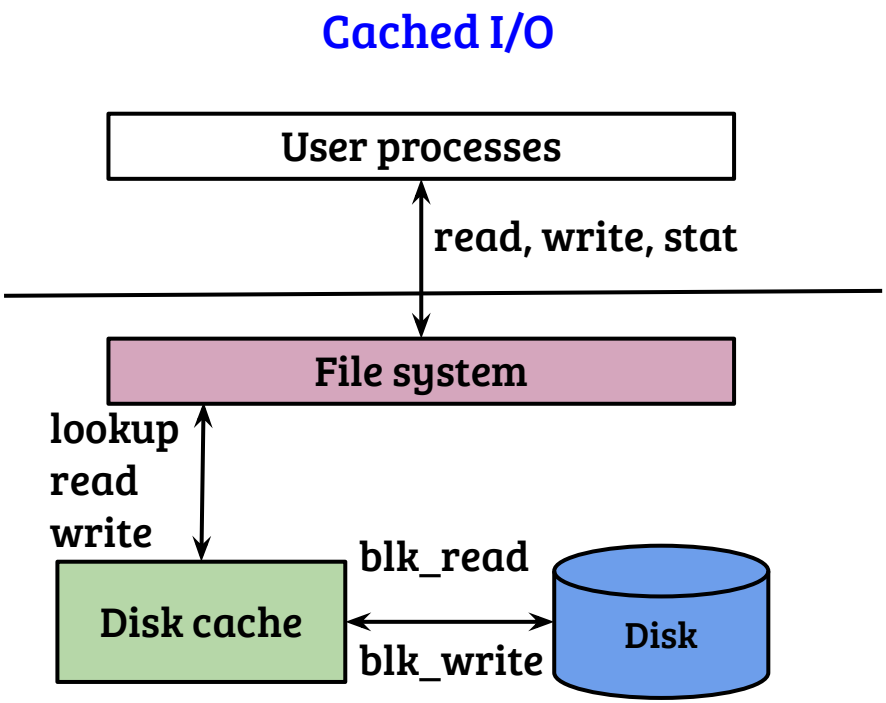
/home/user$ ls

Block layer caching



- Lookup memory cache using the block number as the key
- How does the scheme work for data and metadata?
- For data caching, file offset to block address mapping is required before using the cache
- Works fine for metadata as they are addressed using block numbers

File layer caching (Linux page cache)



- Store and lookup memory cache using {inode number, file offset} as the key
- For data, index translation is not required for file access
- Metadata may not have a file association, should be handled differently (using a special inode may be!)

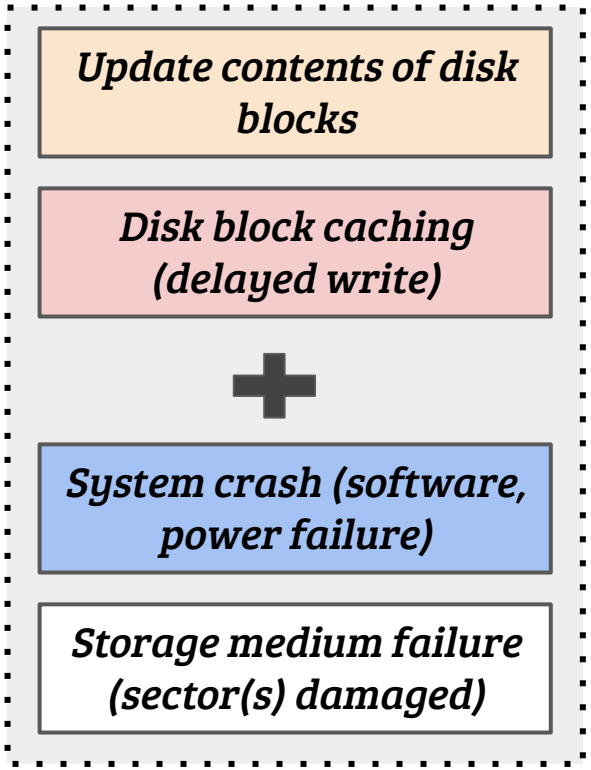
File system and caching

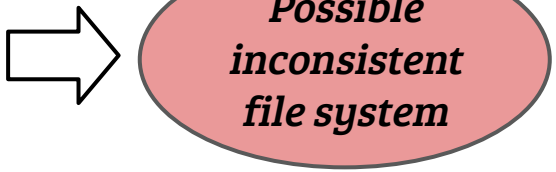
- Accessing data and metadata from disk impacts performance
- Can we store frequently accessed disk data in memory?
  - What is the storage and lookup mechanism? Are the data and metadata caching mechanisms same?
  - File layer caching is desirable as it avoids index accesses on hit, special mechanism required for metadata.
  - Are there any complications because of caching?
  - How the cache managed? What should be the eviction policy?

Caching and consistency

- Caching may result in inconsistency, but what type of consistency?
- System call level guarantees
  - Example-1: If a write( ) system call is successful, data must be written
  - Example-2: If a file creation is successful then, file is created.
  - Difficult to achieve with asynchronous I/O
- Consistency w.r.t. file system invariants
  - Example-1: If a block is pointed to by an inode data pointers then, corresponding block bitmap must be set
  - Example-2: Directory entry contains an inode, inode must be valid
  - Possible, require special techniques

File system inconsistency: root causes

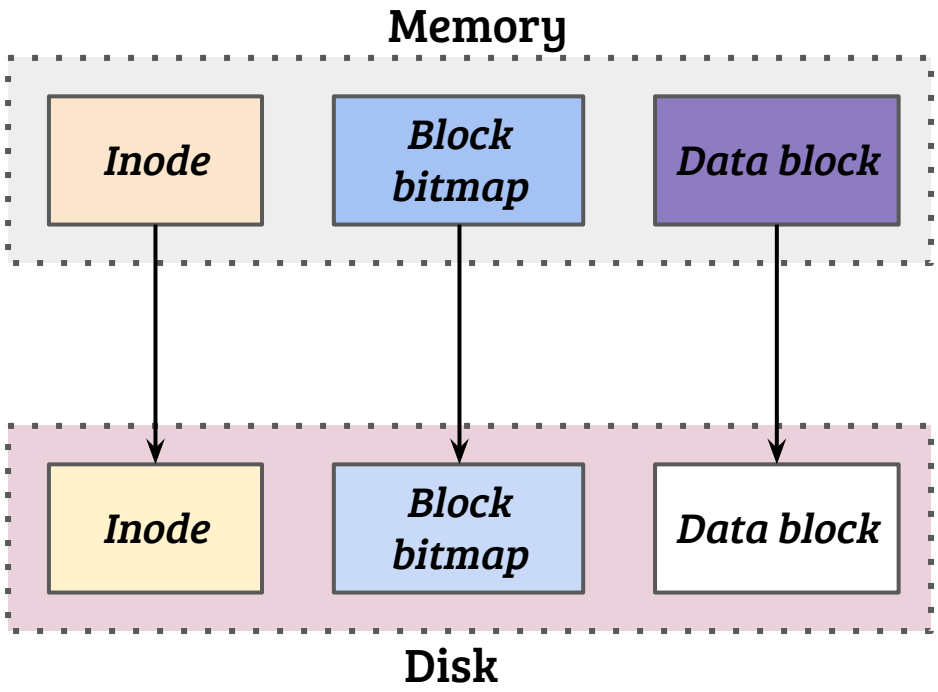




- No consistency issues if user operation translates to read-only operations on the disk blocks
- Device level atomicity guarantees matter!

Example: Append to a file

- Steps: (i) seek to the end of file, (ii) allocate a new block, (iii) write user data
- Inode modifications: size and block pointers
- Block bitmap update: set used block bit for the newly allocated block(s)
- Data update: data block content is updated



Three write operations reqd. to complete the operation, what if some of them are incomplete?

Failure scenarios and implications

Written	Yet to be written	Implications
Data block	Inode, Block bitmap	File system is consistent (Lost data)
Inode	Block bitmap, Data block	File system is inconsistent (correctness issues)
Block bitmap	Inode, Data block	File system is inconsistent (space leakage)

- All failure scenarios may not result in consistency issues!

Failure scenarios and implications

Written	Yet to be written	Implications
Data block, Block bitmap	Inode	File system is inconsistent (space leakage)
Inode, Data block	Block bitmap	File system is inconsistent (correctness issues)
Inode, Block bitmap	Data block	File system is consistent (Incorrect data)

- Careful ordering of operations may reduce the risk of inconsistency
- But, how to ensure correctness?

File system consistency with *fsck*

- Strategy: Do not worry about consistency, recover after abrupt failures
- During FS mount, check if it had been cleanly unmounted when it was last used, How to know?
  - Maintain the last unmount information on superblock
- If the FS was not cleanly unmounted, perform sanity checks at different levels: *superblock*, *block bitmap*, *inode*, *directory content*
- Sanity checks and verifying invariants across metadata. Examples,
  - Block bitmap vs. Inode block pointers
  - Used inodes vs. directory content