Operating Systems

Virtual Memory Management

Hongliang Liang, BUPT

Sep. 2023

Paging: A Review

Main benefit of paging: It provides a process with a **contiguous virtual** address space, but allows the process to access **non-contiguous physical** memory

Hardware requirements —

- MMU hardware must support page tables
- Entire set of virtual-to-physical address mappings needs to be stored in a page table in memory

Paging: Basic Elements

- CPU accesses virtual address
- MMU needs to access the page table to map the virtual address to a physical address
- MMU uses the Translation Lookaside Buffer (TLB) to cache page table entries
- The page table is accessed on a **TLB miss**, the entry is added to the TLB, and the instruction is retried
- MMU uses the TLB to enforce page-level protection

Virtual Memory: Problem and Basic Idea

Problem: What if the total memory requirement from all address spaces does not fit into the available physical memory?

Virtual Memory: Problem and Basic Idea

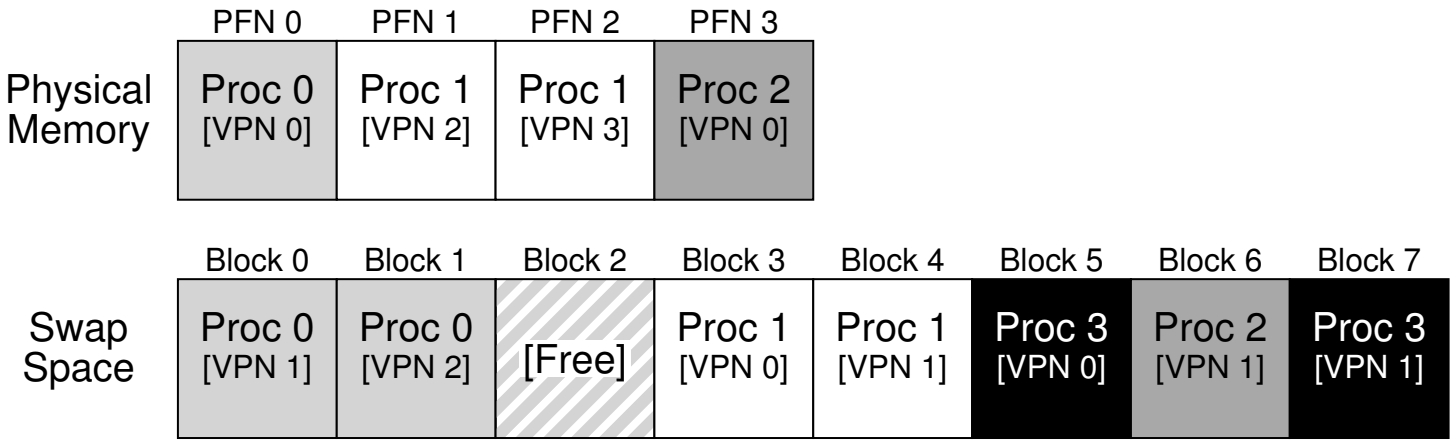
Problem: What if the total memory requirement from all address spaces does not fit into the available physical memory?

Basic idea: Use the disk as the swap space

Benefit: Allows running programs larger than physical memory

Called **virtual memory** since the OS kernel uses the disk as physical memory **transparently**, and processes do not have to know about it!

Swap space: an example



Some pages may be on the disk, but not in the swap space – pages in the code segment are in the executable file.

The Present Bit

Is the page currently **present** in the main memory, or is it swapped out to the disk (or part of the executable file on the disk)?

If it is not present, what should the hardware do during address translation?

The Present Bit

Is the page currently **present** in the main memory, or is it swapped out to the disk (or part of the executable file on the disk)?

If it is not present, what should the hardware do during address translation?

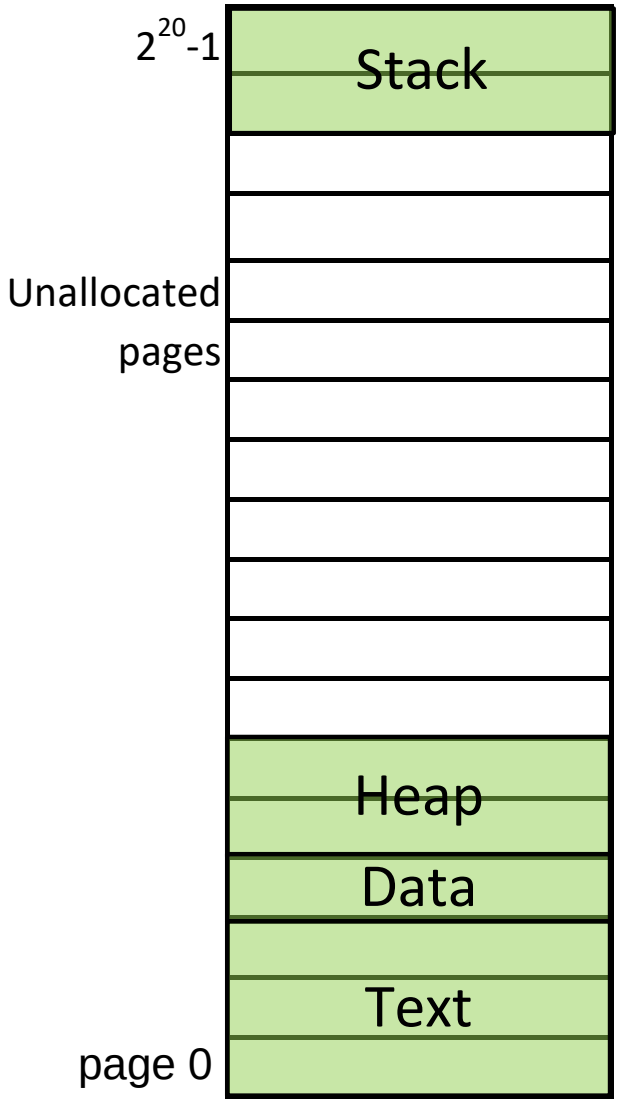
Page Fault

It turns out that the Page Fault handler is one of the most versatile handlers in the OS kernel!

- When the present bit is not set, Page Fault occurs since the page is not in the main memory
- When the valid bit is not set, Page Fault also occurs (sometimes called Invalid Page Fault), and it is up to the Page Fault handler to raise a segmentation fault!

When memory is allocated in the addr space

When the stack or heap grows, a page may be requested, the OS should allocate a new page and maps it to a frame, and execution continues



Growth in the heap and stack

Heap

- Library routines (`malloc()`, `new()` etc.) can service small requests from a pool of free memory already allocated within the virtual address space
- When these routines run out of space, program needs to make a memory allocation system call
- OS allocates a frame, inserts a new valid entry into the page table for the heap segment
- Example system call in Linux: `brk()`

Growth in the heap and stack

Heap

- Library routines (`malloc()`, `new()` etc.) can service small requests from a pool of free memory already allocated within the virtual address space
- When these routines run out of space, program needs to make a memory allocation system call
- OS allocates a frame, inserts a new valid entry into the page table for the heap segment
- Example system call in Linux: `brk()`

Stack

- Grows on demand without needing a system call!
- How does this happen?
- Take advantage of **page faults**

Page Faults

The page table entries for pages outside the mapped regions are marked **invalid**

If a program accesses an address in an invalid page, the processor generates a **page fault** exception, and traps into the kernel

- Invokes the **page fault handler in OS**
- Provides the handler with the **faulting address**, i.e., the virtual address that caused the fault

The Page Fault Handler

If faulting address is **close** to the stack, then

- OS allocates a frame, inserts an entry into the page table for the stack
- OS then restarts the instruction causing the fault
- After the page fault is handled, the process accesses the extended stack

Otherwise, i.e., faulting address is **far** from the stack

- OS issues a **segmentation fault** exception to the process!

Page Miss — Not present in the main memory

The page is not present in the main memory, but still valid (mapped) — it is now on the disk

- The **present** bit is 0, but the **valid** bit is 1

If a free physical frame is available, allocate it

- Figure out the page number from the faulting address
- Find out the disk address corresponding to the page
- Read the needed page into this frame, update the page table. The page could be stored in the executable or swap space
- **Restart** the faulting instruction in the interrupted process

What if a free frame is not available?

Select a frame based on a **page replacement** algorithm

- **Evict** the page mapped to this frame
- Clear the present bit in the page table entry containing that frame. The next access to this page will cause a Page Fault
- If the page table entry is dirty, write the page to the swap space

A free physical frame is now available, allocate it

Summary so far: Virtual Memory with Paging

What does the hardware do?

Protection Fault

Page Fault

What does the hardware do?

```
1 VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2 (Success, TlbEntry) = TLB_Lookup(VPN)
3 if (Success == True) // TLB Hit
4   if (CanAccess(TlbEntry.ProtectBits) == True)
5     Offset = VirtualAddress & OFFSET_MASK
6     PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7     Register = AccessMemory(PhysAddr)
8   else
9     RaiseException(PROTECTION_FAULT)
10 else // TLB Miss
11   PTEAddr = PTBR + (VPN * sizeof(PTE))
12   PTE = AccessMemory(PTEAddr)
13   if (PTE.Valid == False)
14     RaiseException(SEGMENTATION_FAULT)
15   else
16     if (CanAccess(PTE.ProtectBits) == False)
17       RaiseException(PROTECTION_FAULT)
18     else if (PTE.Present == True)
19       // assuming hardware-managed TLB
20       TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21       RetryInstruction()
22     else if (PTE.Present == False)
23       RaiseException(PAGE_FAULT)
```

What does the hardware do?

A process executing on CPU generates a virtual address

TLB does a lookup using the page number of the address

Page number matches, TLB gets page table entry for virtual address. Otherwise, a **TLB miss**

TLB checks if the protection bits allows operation. Otherwise, generate a **protection fault**

TLB combines the frame number from the page table entry with the offset to generate a physical address

MMU reads the physical address from cache or memory, returns the value to the CPU

On a TLB Miss

Page table entry for page is not in TLB or invalid in TLB

Hardware option (x86):

- MMU loads page table entry from the page table (located in memory) into TLB
- OS not involved
- OS has already setup the page table so hardware can access it

Software option (SPARC, MIPS):

- Generate TLB miss fault and trap to OS
- OS looks up page table, loads page table entry in TLB, then returns from trap

On a Protection Fault

Trap to OS kernel

Page table entry has protection bits inconsistent with operation – Read/write/execute operation not permitted on the page

Normally, OS sends **segmentation fault**, kills thread

OS may also use this fault for **copy-on-write and memory-mapped files**

On a Page Fault

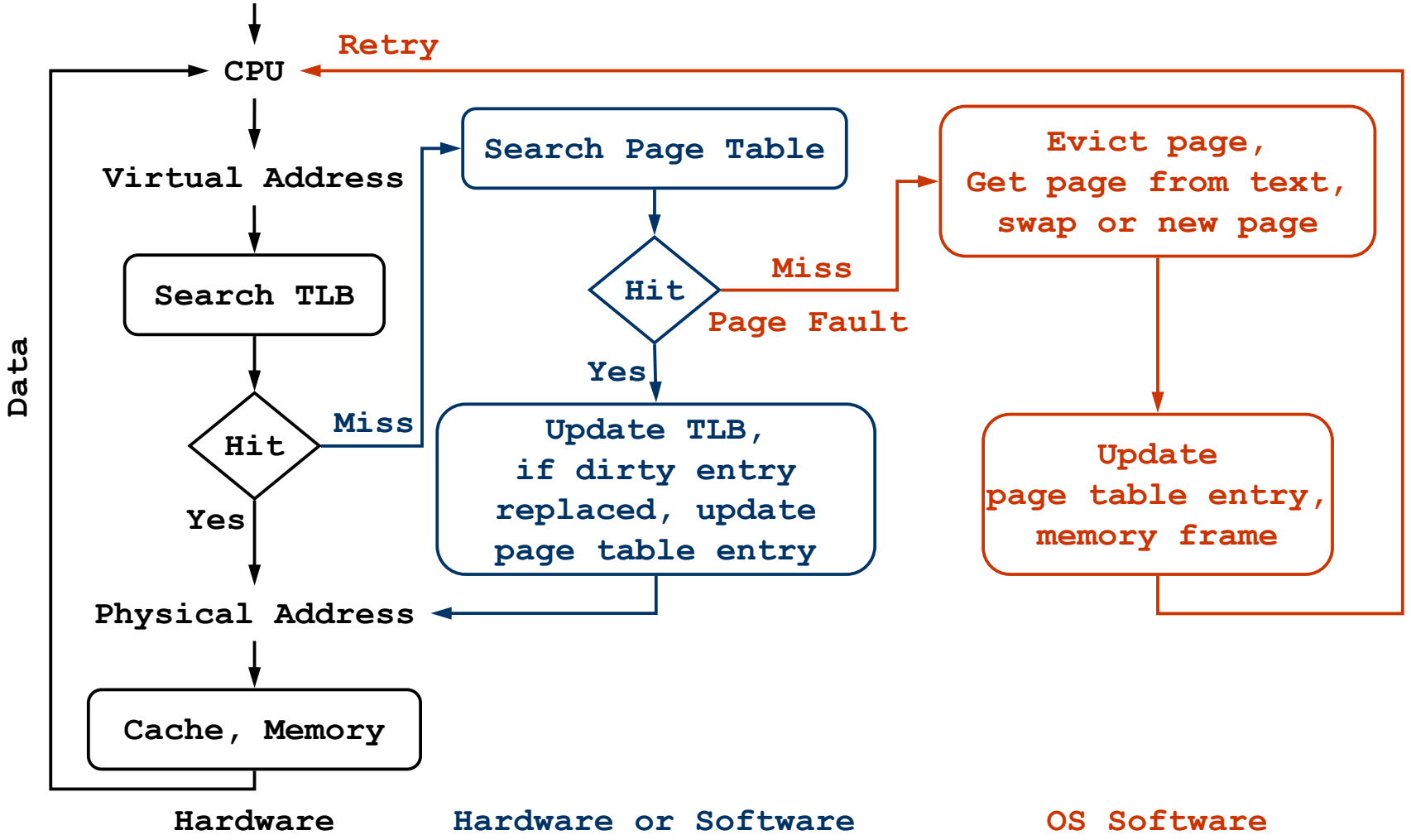
Page table entry: the valid or present bit is 0

If page not mapped into any address space region, OS sends **segmentation fault** or grows stack

Otherwise, the page is mapped, but not in physical memory

- OS allocates a free frame that is available – May have to evict pages mapped to this frame
- **Reads data from disk**, possibly from the swap space, into the frame
- Maps page table entry in page table to frame, makes it **present**

Virtual Memory Hierarchy



Paging Performance

Let p be the probability of a page fault ($0 \leq p \leq 1$), *mat* be the memory-access time, *pfst* be the page fault service time. The effective access time (EAT) is then $EAT = (1 - p) \times mat + p \times pfst$

Paging Performance

Let p be the probability of a page fault ($0 \leq p \leq 1$), *mat* be the memory-access time, *pfst* be the page fault service time. The effective access time (EAT) is then $EAT = (1 - p) \times mat + p \times pfst$

When *pfst*= 8 ms, *mat*= 200 ns and p= 0.001,

```
EAT = (1 - p) * 200 + p * 8,000,000
= 200 + 7,999,800 * p (in ns) = 8.2us
```

The computer would be slowed down by a factor of 40 ($(8200 - 200) \div 200$) because of demand paging!

Paging Performance

Let p be the probability of a page fault ($0 \leq p \leq 1$), *mat* be the memory-access time, *pfst* be the page fault service time. The effective access time (EAT) is then $EAT = (1 - p) \times mat + p \times pfst$

When *pfst*= 8 ms, *mat*= 200 ns and p= 0.001,

```
EAT = (1 - p) * 200 + p * 8,000,000
= 200 + 7,999,800 * p (in ns) = 8.2us
```

The computer would be slowed down by a factor of 40 ($(8200 - 200) \div 200$) because of demand paging!

If we want performance degradation $\leq 10\%$, we need to keep the probability of page faults at the following level:

```
220 > 200 + 7,999,800 * p,
20 > 7,999,800 * p, p < 0.0000025.
```

That is, we can allow fewer than one memory access out of 399,990 to page-fault.

Paging Performance

Paging works best if there are plenty of free frames

If all frames are full of dirty pages, then two disk operations are needed for **each page fault**

We can use a **paging/swap daemon** to improve paging performance

Paging Daemon

A good idea is to **periodically** write out **dirty** pages to speed up the delay handling page faults

A **paging daemon** kernel thread wakes up periodically and counts the number of free frames

If too few are available

- Select a frame based on a page replacement algorithm
- Write it to the swap space and mark the frame as clean
- If this frame is needed later then it is still in memory
- If an empty frame is needed later, this frame can be evicted

Prefetching

Demand paging

- Pages should only be brought into memory if the running process demands them
- Page faults can only be processed one page at a time

Instead, suppose we can **predict** future page usage based on the current fault

We can now **prefetch** other pages

Managing the Swap Area

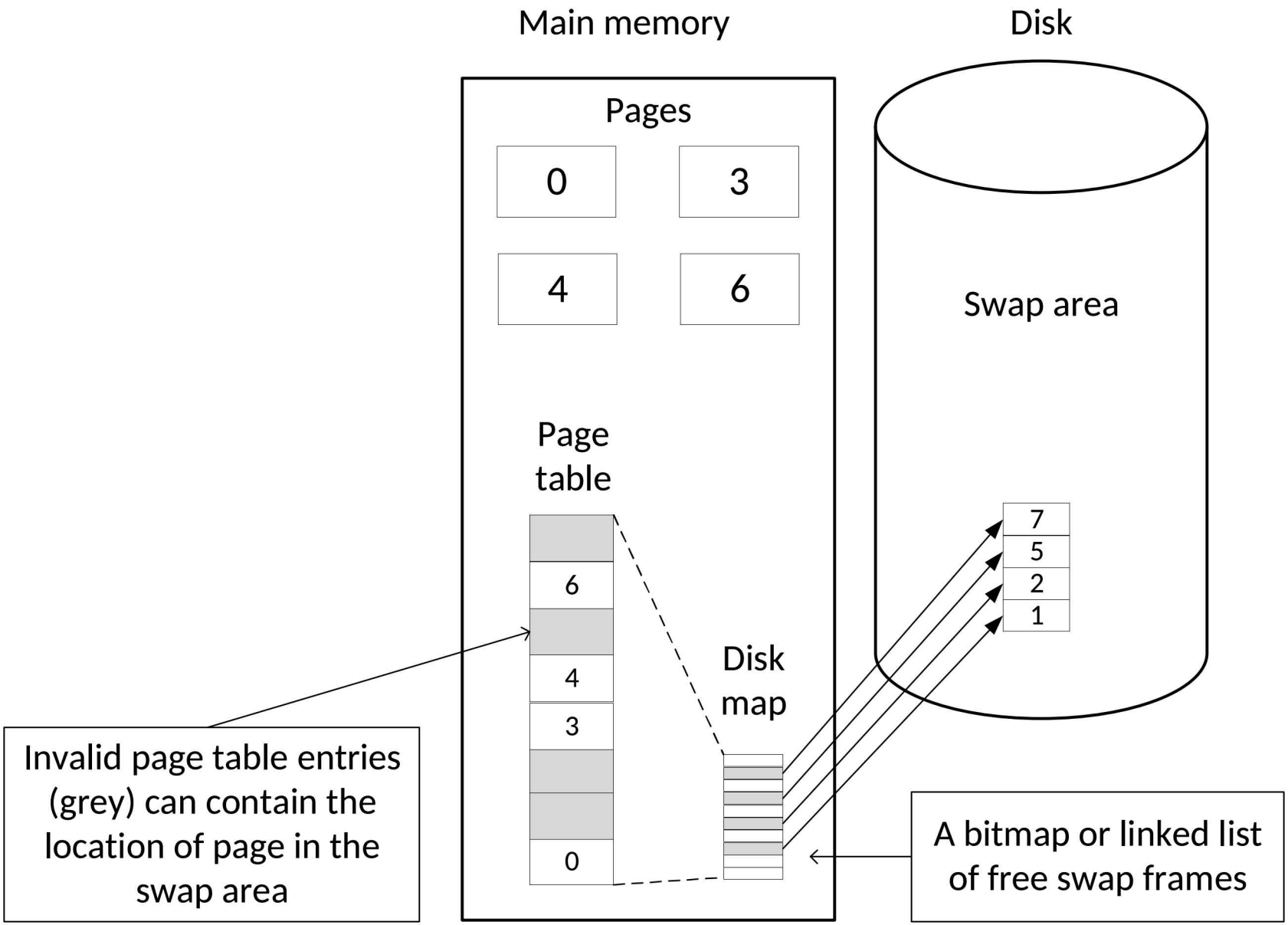
Where should dirty pages be placed in the swap area?

Use technique similar to managing memory
- Use a bitmap or linked list of "swap frames"

When evicting a page to the swap area

- Find a free swap frame
- Write the page to this location on the disk
- Record the location of the swap frame: we can use the invalid page table entry – On page fault, handler uses the page table entry to locate the frame in swap area
- Next time the page is swapped out, it may be written elsewhere

Example


Main memory
Disk
Pages
0
3
4
6
Swap area
Page table
6
4
3
0
Disk map
7
5
2
1
Invalid page table entries (grey) can contain the location of page in the swap area
A bitmap or linked list of free swap frames

Motivation for Sharing Pages

Typically, processes run in their own address space because it provides protection

However, sometimes processes need to share memory
For example, web server parent and children processes may wish to only share some data (e.g., data cache)

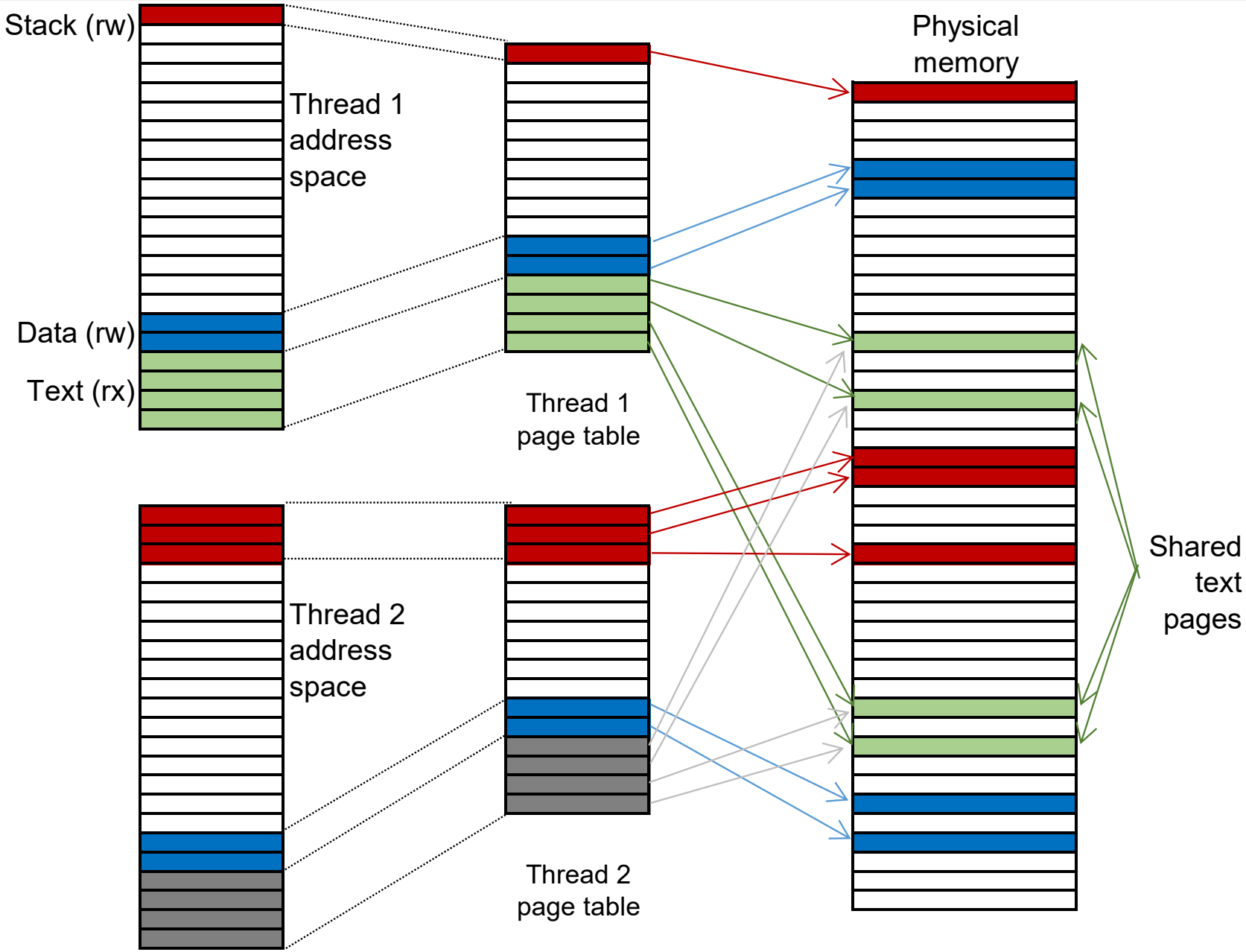
If several users run the same program, why not keep one copy of common pages?

Page Sharing

Two (or more) processes can share memory by mapping a page to the same physical frame in their page tables

- Page table entries can have different protection
  e.g., one process has read only, another write enabled
- Shared memory can be mapped at the same or different virtual addresses in each address space
  –Different addresses: flexible, but shared memory pointers are invalid
- Must update all page table entries when the frame is evicted
  –Requires a one-to-many frame map

Page Sharing



Copy-On-Write Page Sharing

Consider the `Fork()` system call

- `Fork()` creates a new address space for a child process
- It copies pages in the parent's virtual address space to the child's address space
- Copying every page in an address space is **expensive**
- However, processes can't notice the difference between copying and sharing unless pages are **modified**

With **copy-on-write**, pages are shared until they are modified

Copy-On-Write Page Sharing

Initialize new page table for the child on Fork()
However, the page table **shares** parent's frames

Mark all pages in parent and child as **read-only temporarily**. Share all pages until protection fault

On a write protection fault, if page is protected for copy-on-write, then

- Copy the page and mark both copies as **writable**
- Resume execution as if no fault occurred

Need to **update** all copy-on-write page table entries when a shared physical frame is **evicted**

The mmap() System Call for Sharing Memory

- The `mmap()` system call asks the kernel to **map a region in a file to physical memory**
- Allows **sharing memory across address spaces**
  - Very useful when loading shared libraries in multiple address spaces
- The region of the file can be loaded with the private or shared mode
  - **Private**: Not shared with other address spaces — copy-on-write
  - **Shared**: shared with other address spaces — write changes to the file
- Allows accessing a file using a memory interface
  - Processes read and write files using load/store rather than read/write!
  - These files are called **memory-mapped files**

Memory-Mapped Files

OS maps file **contiguously** within an address space

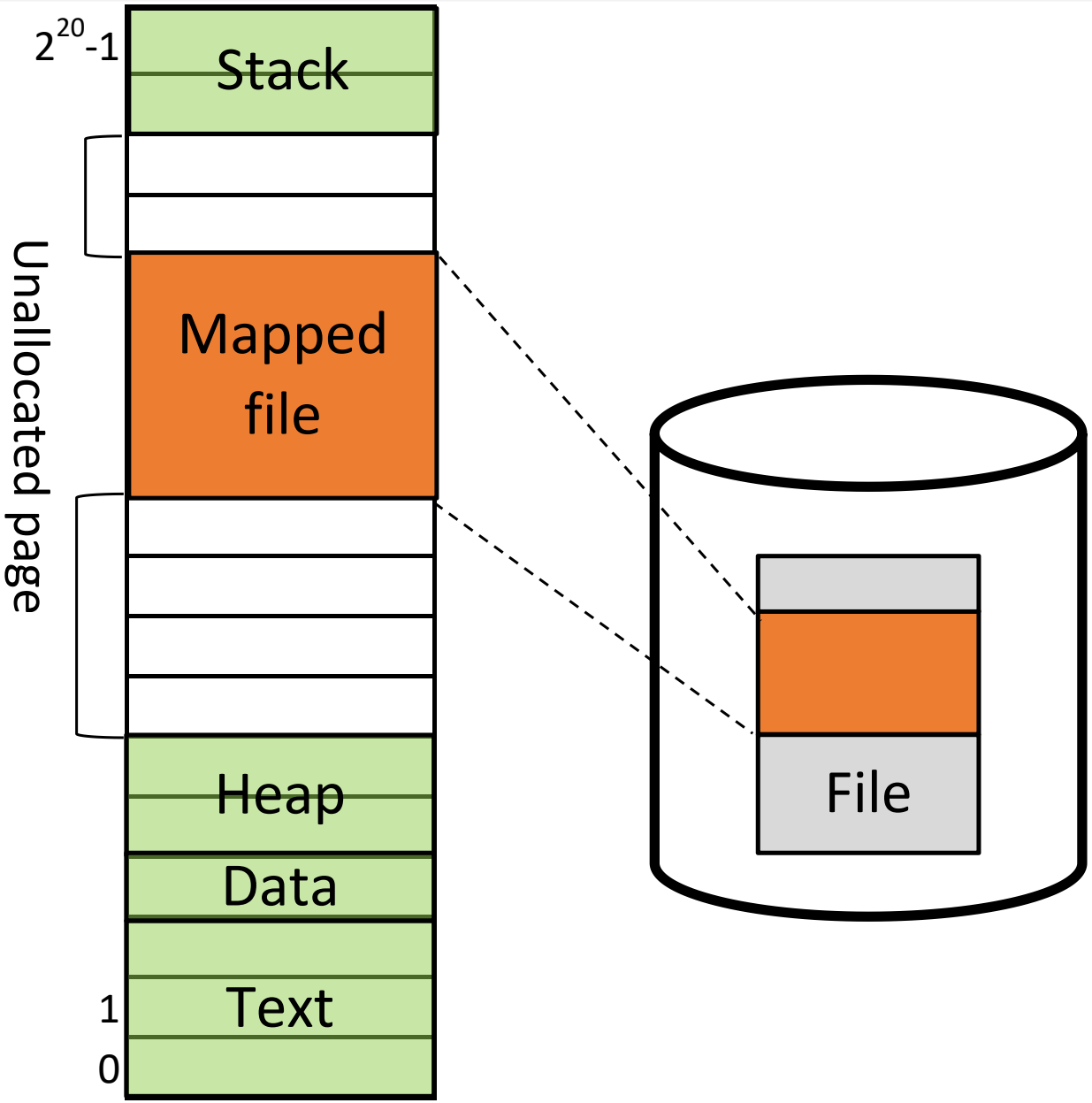
- Say, virtual address `file_base` stores start of file
- Then, `file_base` + N refers to offset N in file

Allows loading file data in memory on demand (**demand paging**)

- OS reads a frame from file when pages that are not present but valid are accessed
- OS writes a frame to file when a dirty page is evicted
- Essentially, file is used for the backing store, instead of the swap area

When different address spaces map the same file, they can **share file data via memory**

Any region of the file can be mapped



The Kernel and Address Spaces

How does the kernel access itself in memory?

The Kernel and Address Spaces

How does the kernel access itself in memory?

1. **Paging is turned off in kernel mode** (as in BLITZ): OS can access physical memory directly, including page tables
   - Loses all benefits of paging: cannot be non-contiguous

The Kernel and Address Spaces

How does the kernel access itself in memory?

1. **Paging is turned off in kernel mode** (as in BLITZ): OS can access physical memory directly, including page tables
   - Loses all benefits of paging: cannot be non-contiguous
2. The kernel has its own address space, uses its own page table
   - It includes the page tables of all user address spaces
   - Harder for the kernel to read data structures in the user program

The Kernel and Address Spaces

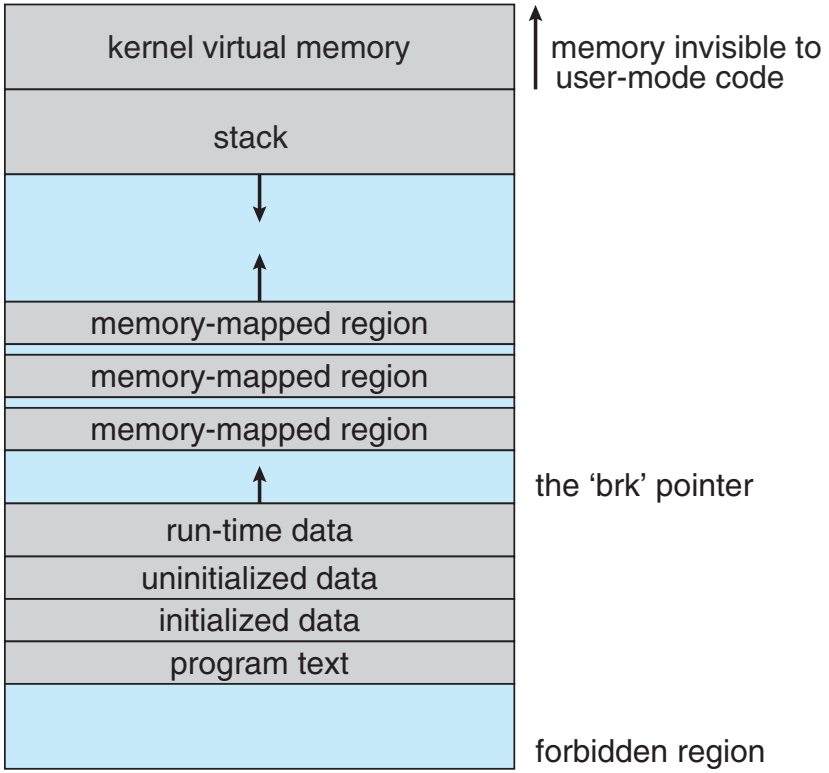
How does the kernel access itself in memory?

1. **Paging is turned off in kernel mode** (as in BLITZ): OS can access physical memory directly, including page tables
   - Loses all benefits of paging: cannot be non-contiguous
2. The kernel has its own address space, uses its own page table
   - It includes the page tables of all user address spaces
   - Harder for the kernel to read data structures in the user program
3. OS maps the kernel to part of the address space of each thread
   - No need to change the page table base register when switching to kernel mode
   - Easy for the kernel to read data structures in the user program

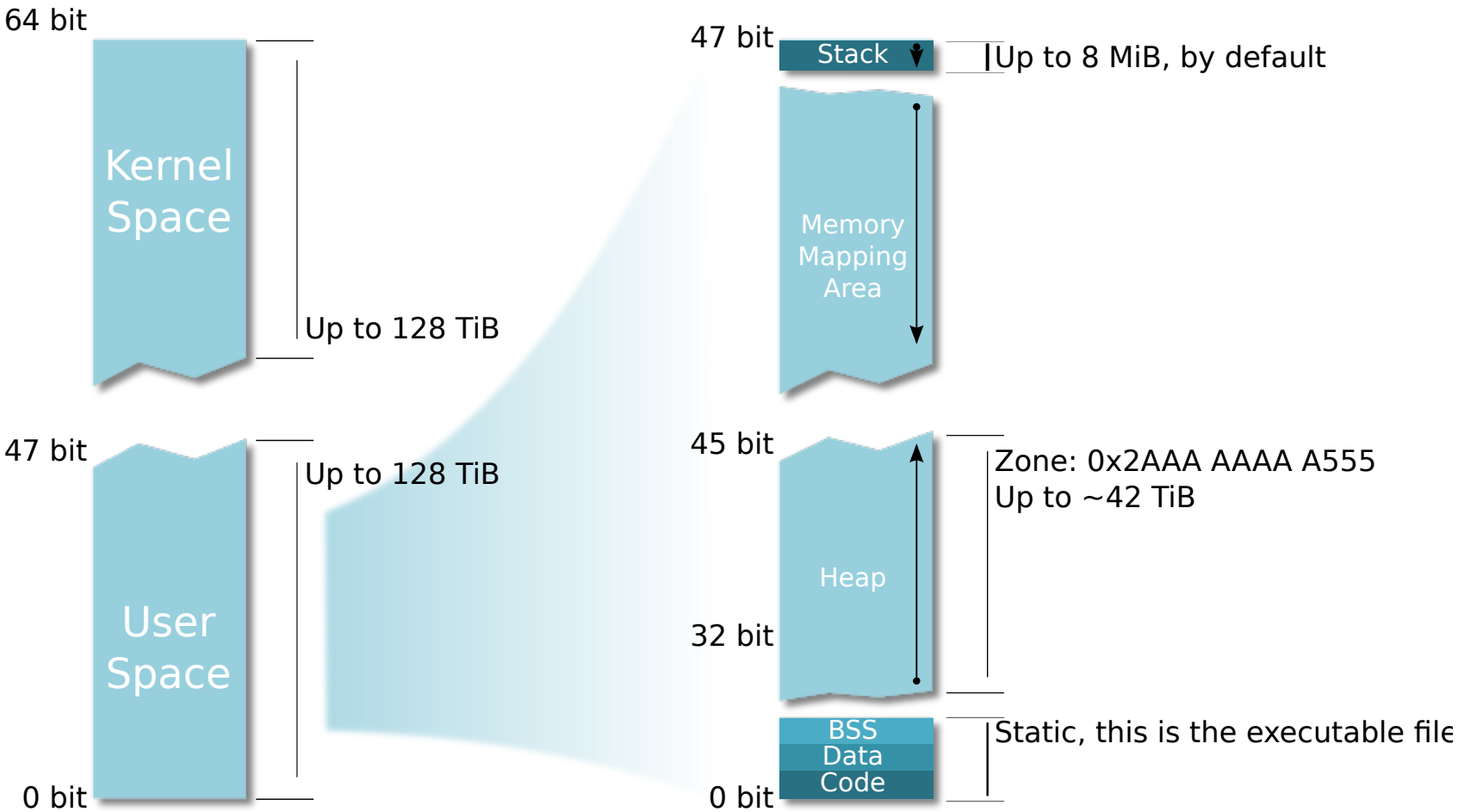
Memory layout in Linux (32 bits)

A 32-bit processor can address a maximum of 4GB of memory. Linux kernels split the 4GB address space between user processes (the first 3GB) and the kernel (final 1GB starting at 0xc0000000).

Sharing the address space gives a number of performance benefits; in particular, the hardware's address translation buffer (TLB) can be shared between the kernel and user space.



Memory layout in Linux (64 bits)



So Kernel + User Spaces add for 256 TiB which is a tiny part of the 16 777 216 TiB addressable over 64 bit!