CS 1217

Lecture 13 – Virtual Address Spaces, Address Translations

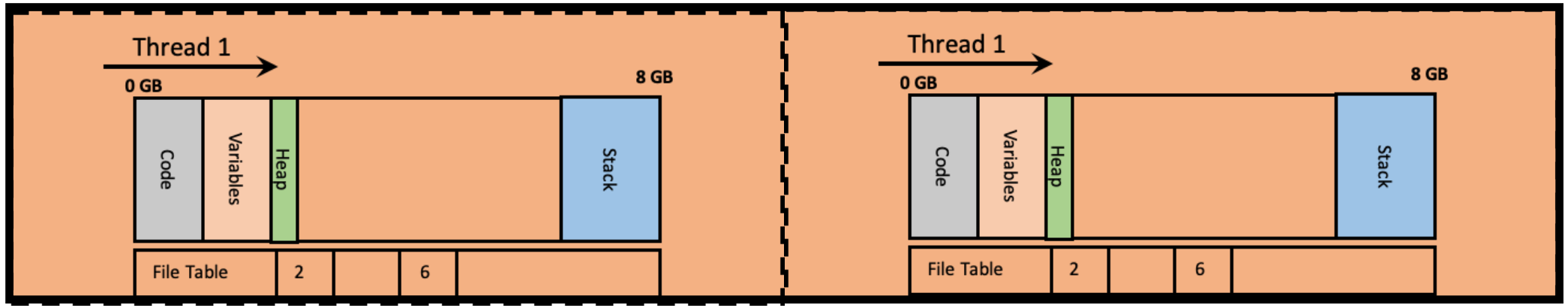
Recap

- Resource sharing: Time multiplexing vs Space multiplexing
- Fragmentation
  - Internal
  - External

Goals: Memory Management

- What have we done so far?
  - Give processes (the illusion of) contiguous address spaces
    - Why? Convention allows processes to find information at predefined locations
    - I always put my code and static variables at 0x10000."
    - "My heap always starts at 0x20000000 and grows **up**."
    - "My stack always starts at 0xFFFFFFFF and grows **down**."
  - Coarse grained management of memory capacity

Coarse Grained Management



0 GB **Main Memory** **16 GB**

“Virtual” Address Spaces

- **Address translation**: 0x10000 to Process 1 is not the same as 0x10000 to Process 2 is not the same as…

- **Protection**: address spaces are intended to provide a *private* view of memory to each process.

- **Memory management**: together one or several processes may have **more address space** allocated than physical memory on the machine.

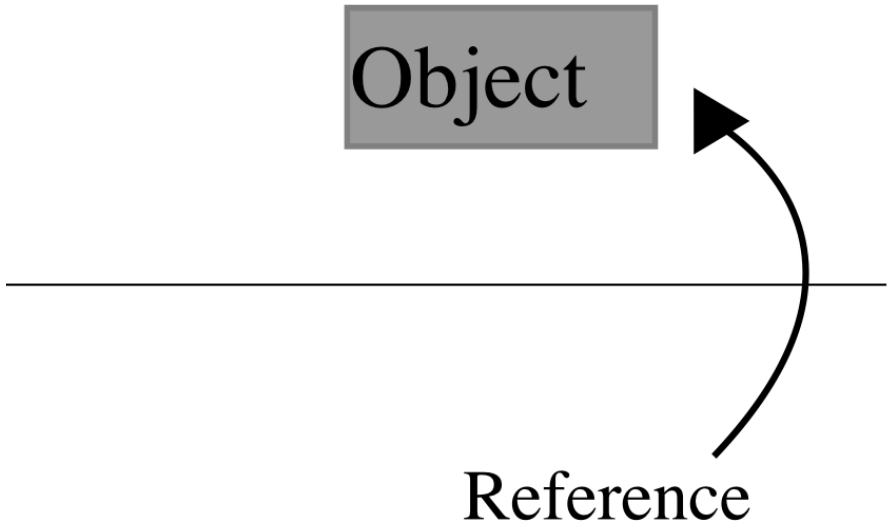
Virtual Addresses vs. Physical Addresses

- Data accessed via the memory interface as using **virtual addresses**:
  - A **physical address** points to memory
  - A **virtual address** points to something that *acts like* memory
- Virtual addresses have much richer **semantics** than physical addresses

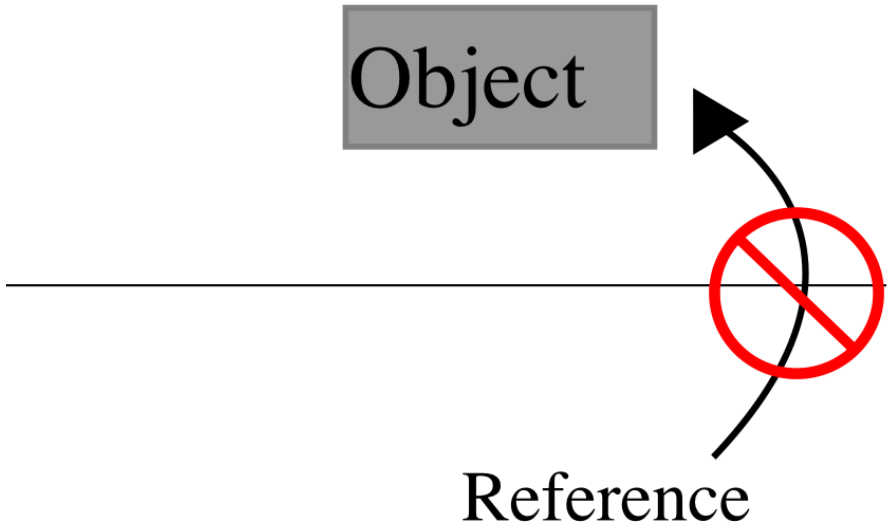
Indirection is Useful

- Indirection = Translation
- Every memory access by a process requires a level of indirection
  - Translating Virtual Address to Physical Address
- Nutshell: Every address that a process accesses is (essentially) a reference
- This provides the kernel a lot of control over how to manage memory
- Why: References can be **revoked, shared, moved,** and **altered**.

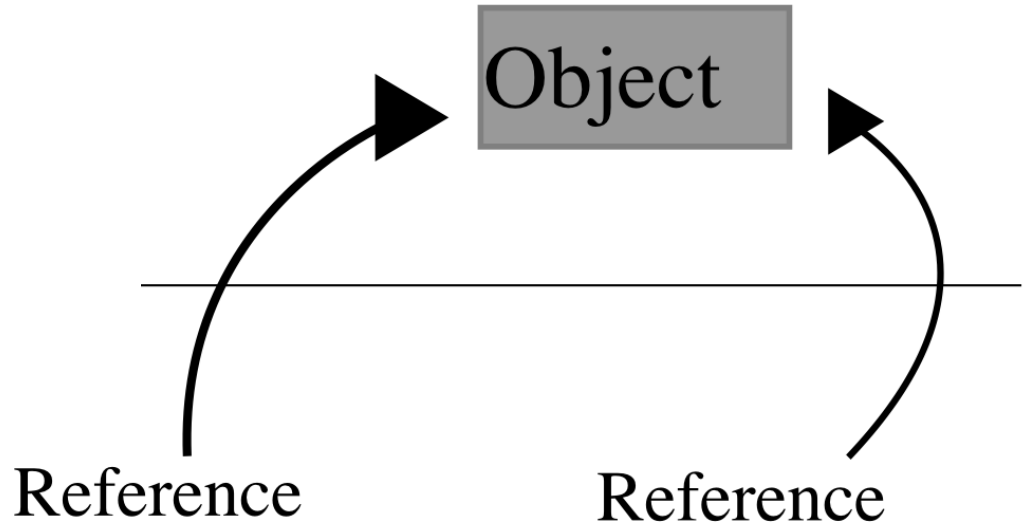
Indirection is Useful



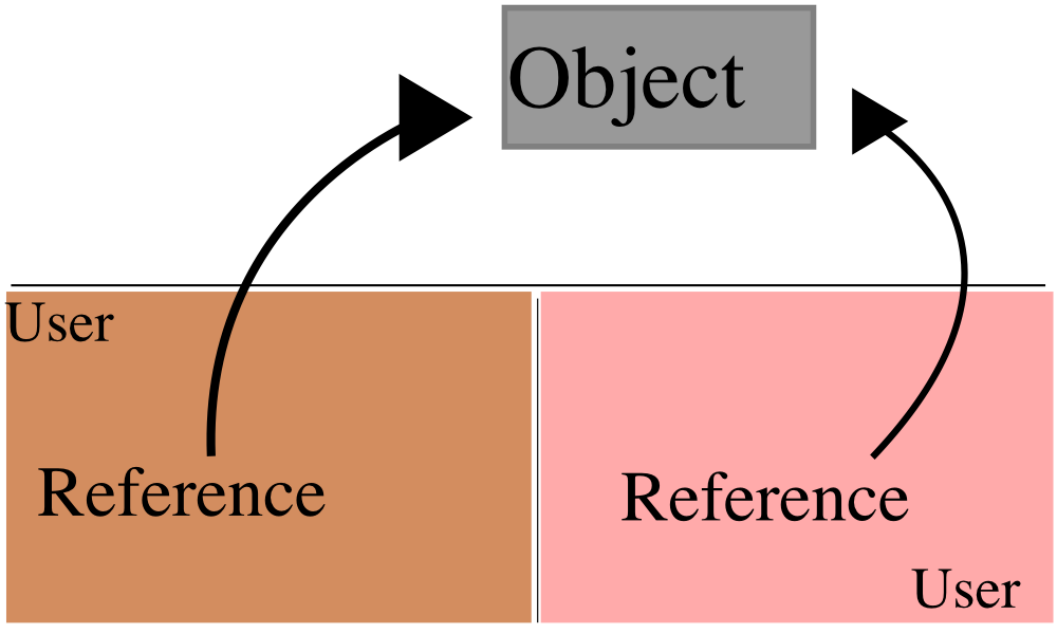
Indirection is Useful



Indirection is Useful



Indirection is Useful



What All Can Virtual Addresses Be?

- Virtual Address → Physical Address
  - In memory data
- Virtual Address → Disk, Block, Offset
  - **Data on disk**, but…the kernel may be caching it in *memory*
- Virtual Address → IP Address, Physical Address
  - in memory on **another machine**
- Virtual Address → Device, Port
  - a port on a **hardware device**

Permissions and Protection

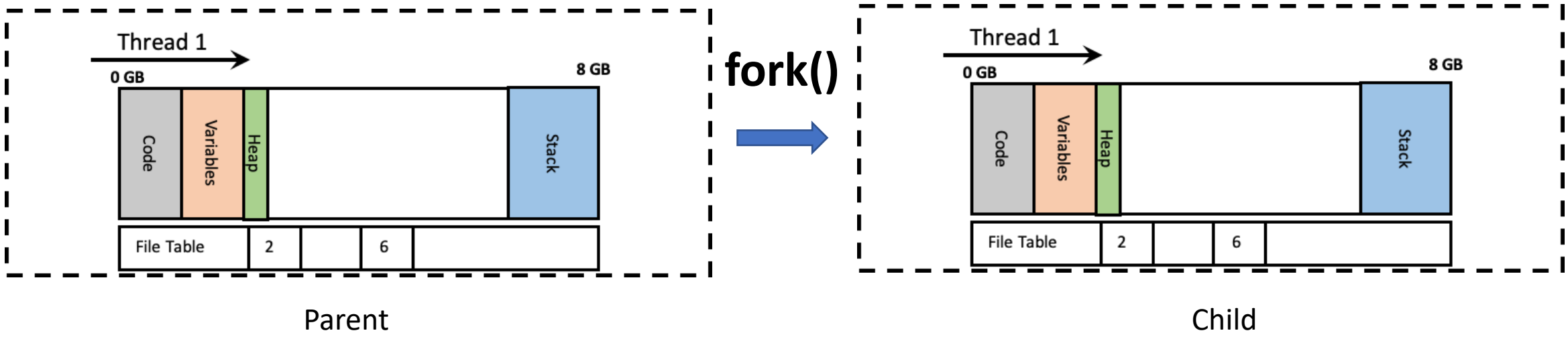
- A range of virtual addresses might only be used by the kernel, in kernel mode
- Virtual addresses may be assigned **read**, **write** or **execute** permissions

  - **read/write**: a process can load/store to this address.
  - **execute**: a process can load and execute instructions from this address

Address Mapping : pmap

```
cs304@cs304-devel:~$ pmap 2002
2002:   bash
0000560445ea5000   1040K r-x-- bash
00005604461a8000     16K r---- bash
00005604461ac000     36K rw--- bash
00005604461b5000     40K rw---   [ anon ]
00005604480ff000   1604K rw---   [ anon ]
00007fb4b8a15000   1948K r-x-- libc-2.27.so
00007fb4b8bfc000   2048K ----- libc-2.27.so
00007fb4b8dfc000     16K r---- libc-2.27.so
00007fb4b8e00000      8K rw--- libc-2.27.so
00007fb4b945d000      4K rw---   [ anon ]
00007fff083df000    132K rw---   [ stack ]
```

Creating Virtual Addresses: fork()



- What happens to virtual addresses?
- fork() **copies** the address space of the calling process

Creating Virtual Addresses: fork()

```c
int i = 2, ret;

ret = fork();

    if (ret != 0) {
      printf("Parent Addr: 0X%x\n", &i);
      i = 4;
      printf("Parent Value: %d\n", i);
    }
    else {
      printf("Child Addr: 0X%x\n", &i);
      i = 3;
      printf("Child Value: %d\n", i);
    }
```

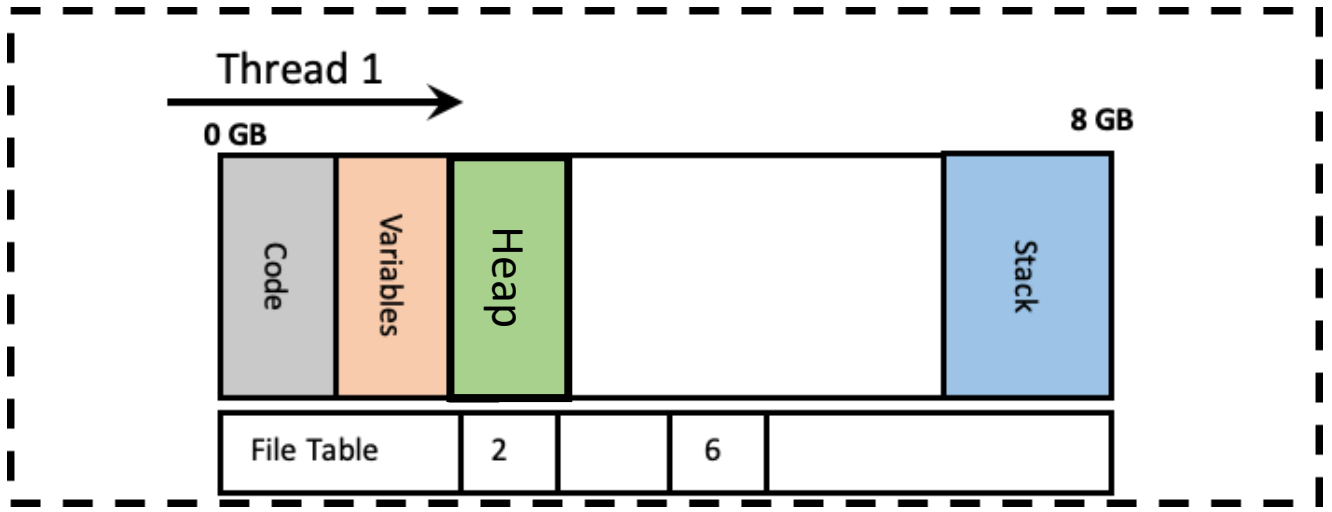
- The child has the same virtual addresses as the parent but they point to **different** memory locations

Creating Virtual Addresses: exec()

- How does exec() know what the address space of the new process is going to look like?
  - The ELF file has a blueprint
- exec() creates and initializes **virtual addresses** that (mainly) point to **memory**
  - **code**, usually marked *read-only* and executable
  - **data**, marked *read-write*, but not executable
  - **heap**, an area used for *dynamic allocations*, marked read-write
  - **stack** space for the *first* thread

Creating Virtual Addresses: sbrk()

- Programmers use malloc() for dynamic memory allocation in C
- malloc() in turn uses sbrk(), a system call
- sbrk() asks the kernel to move the **break point**, or the point at which the process heap ends



Creating Virtual Addresses: mmap()

- mmap() is a system call that creates virtual addresses that map to a portion of a **file**

- Advantages?

Policy vs. Mechanism

- Virtual Address Spaces
  - Policy

- Virtual Address Translation
  - Mechanism