Virtual Memory

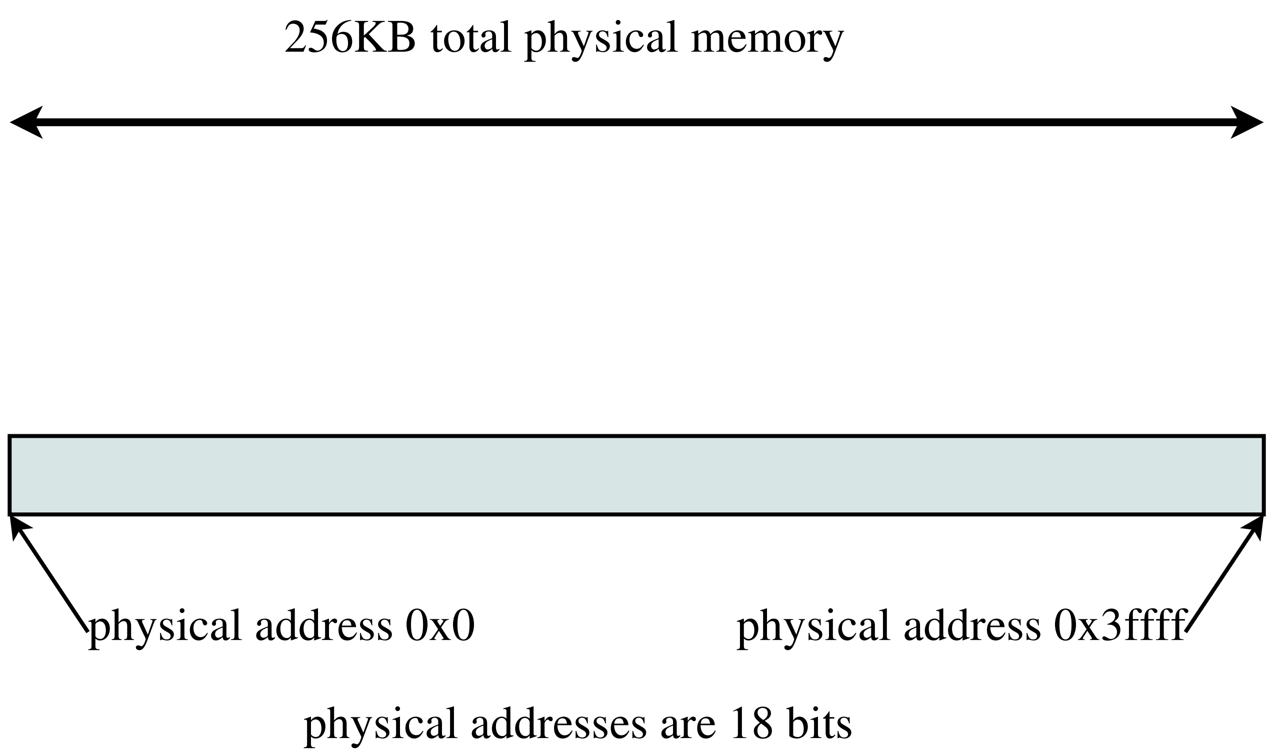
key concepts

virtual memory, physical memory, address translation, MMU, TLB, relocation, paging, segmentation, executable file, swapping, page fault, locality, page replacement

reading

Three Easy Pieces: Chapters 12-24

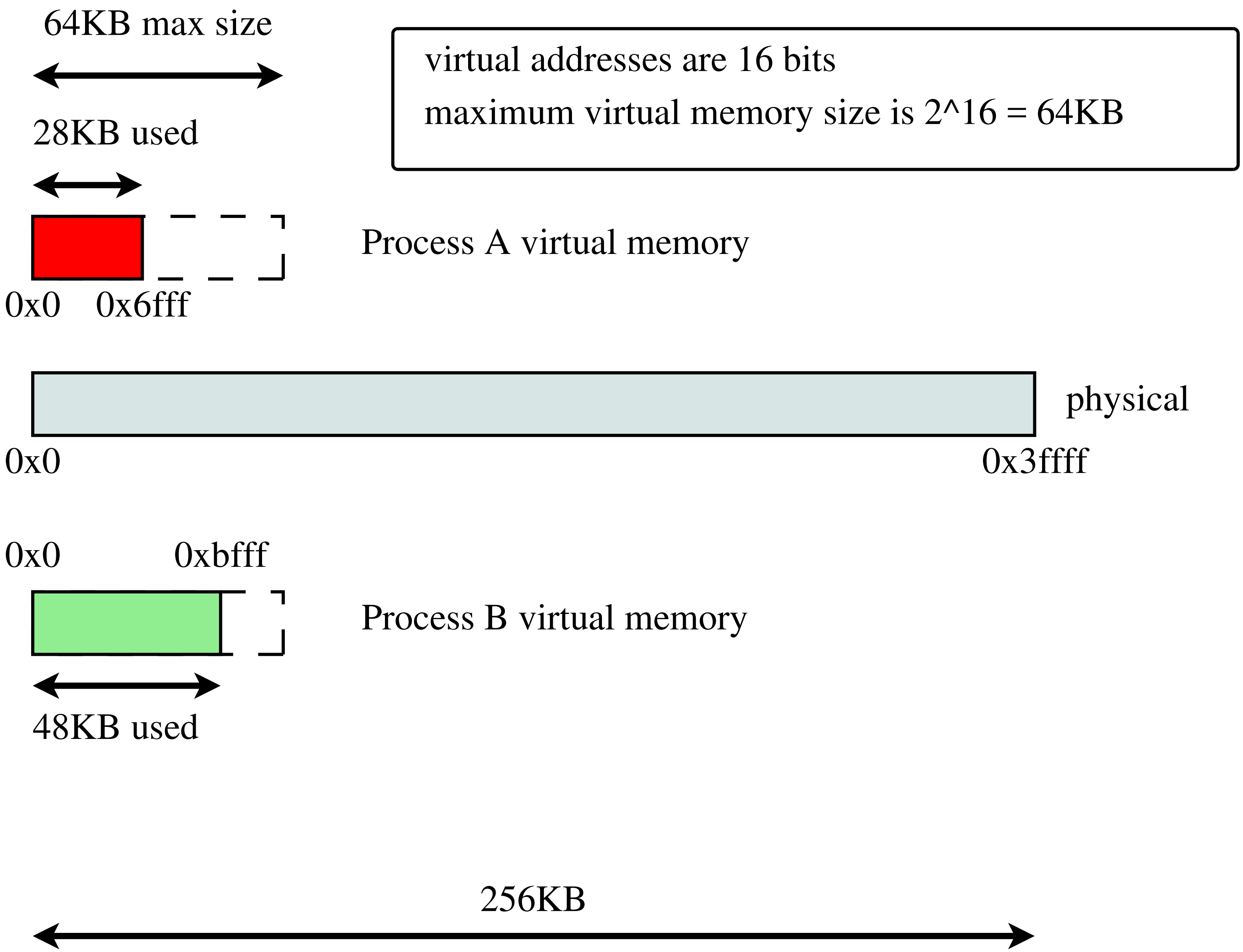
Physical Memory



Physical Addresses

- If physical addresses have P *bits*, the maximum amount of addressable physical memory is 2^P *bytes* (assuming a byte-addresseable machine).
  - Sys/161 MIPS processor uses 32 bit physical addresses ($P = 32$) $\Rightarrow$ maximum physical memory size of 2^{32} bytes, or 4GB.
  - Larger values of P are common on modern processors, e.g., $P = 48$, which allows 256 TB of physical memory to be addressed.
  - The small example on the previous slide uses $P = 18$
- The actual amount of physical memory on a machine may be less than the maximum amount that can be addressed.

Virtual Memory



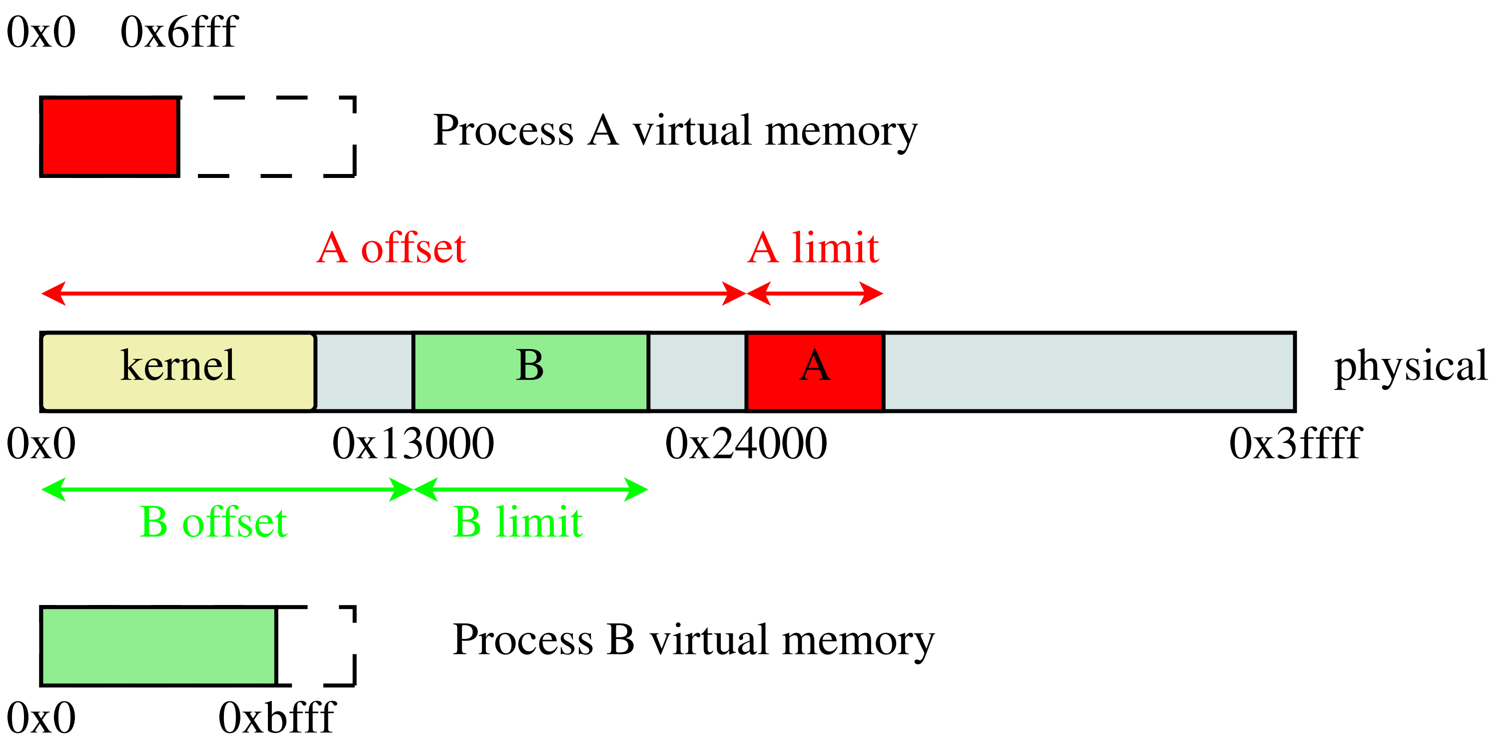
Virtual Addresses

- The kernel provides a separate, private *virtual* memory for each process.
- The virtual memory of a process holds the code, data, and stack for the program that is running in that process.
- If virtual addresses are V bits, the *maximum* size of a virtual memory is 2^V bytes.
  - For the MIPS, $V = 32$.
  - In our example slides, $V = 16$.
- Running applications see only virtual addresses, e.g.,
  - program counter and stack pointer hold *virtual addresses* of the next instruction and the stack
  - pointers to variables are *virtual addresses*
  - jumps/branches refer to *virtual addresses*
- Each process is isolated in its virtual memory, and cannot access other process' virtual memories.

Address Translation

- Each virtual memory is mapped to a different part of physical memory.
- Since virtual memory is not real, when an process tries to access (load or store) a virtual address, the virtual address is *translated* (mapped) to its corresponding physical address, and the load or store is performed in physical memory.
- Address translation is performed in hardware, using information provided by the kernel.

Dynamic Relocation



Address Translation for Dynamic Relocation

- CPU includes a *memory management unit (MMU)*, with a a *relocation register* and a *limit register*.
  - relocation register holds the physical offset (R) for the running process' virtual memory
  - limit register holds the size L of the running process' virtual memory
- To translate a virtual address v to a physical address p:

  if $v \geq L$ then generate exception
  else
  $p \leftarrow v + R$

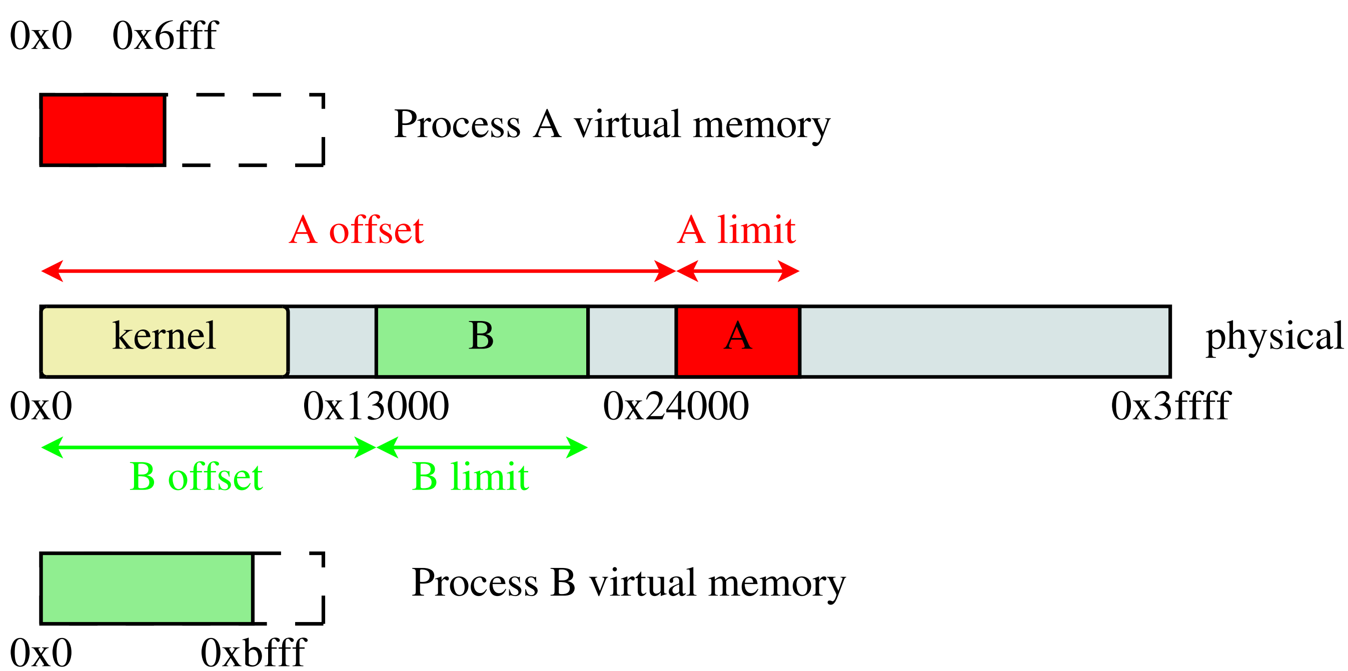
- Translation is done in hardware by the MMU
- The kernel maintains a separate R and L for each process, and changes the values in the MMU registers when there is a context switch between processes

Properties of Dynamic Relocation

- Each virtual address space corresponds to a *contiguous range of physical addresses*
- The kernel is responsible for deciding *where* each virtual address space should map in physical memory
  - The OS must track which parts of physical memory are in use, and which parts are free
  - Since different address spaces may have different sizes, the OS must allocate/deallocate variable-sized chunks of physical memory
  - This creates the potential for *fragmentation* of physical memory

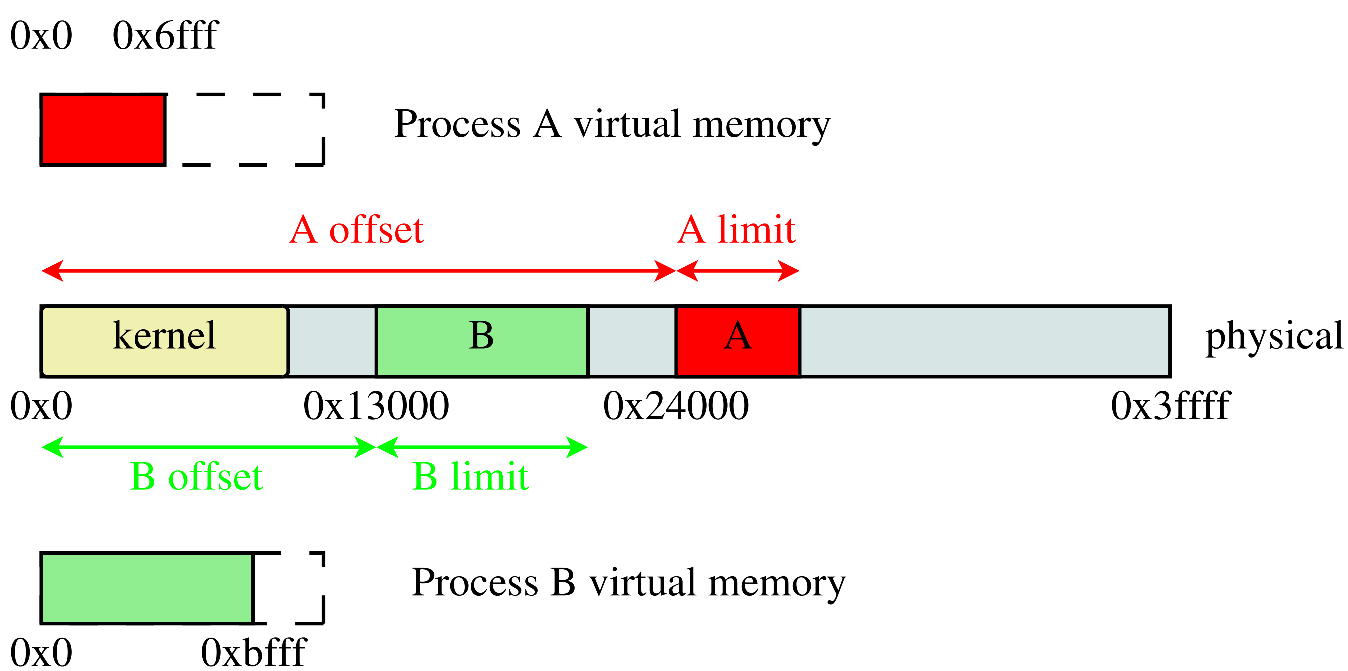
Dynamic Relocation Example: Process A

```
Limit register:       0x0000 7000
Relocation register:  0x0002 4000

v =  0x102c       p = ?
v =  0x8800       p = ?
v =  0x0000       p = ?
```

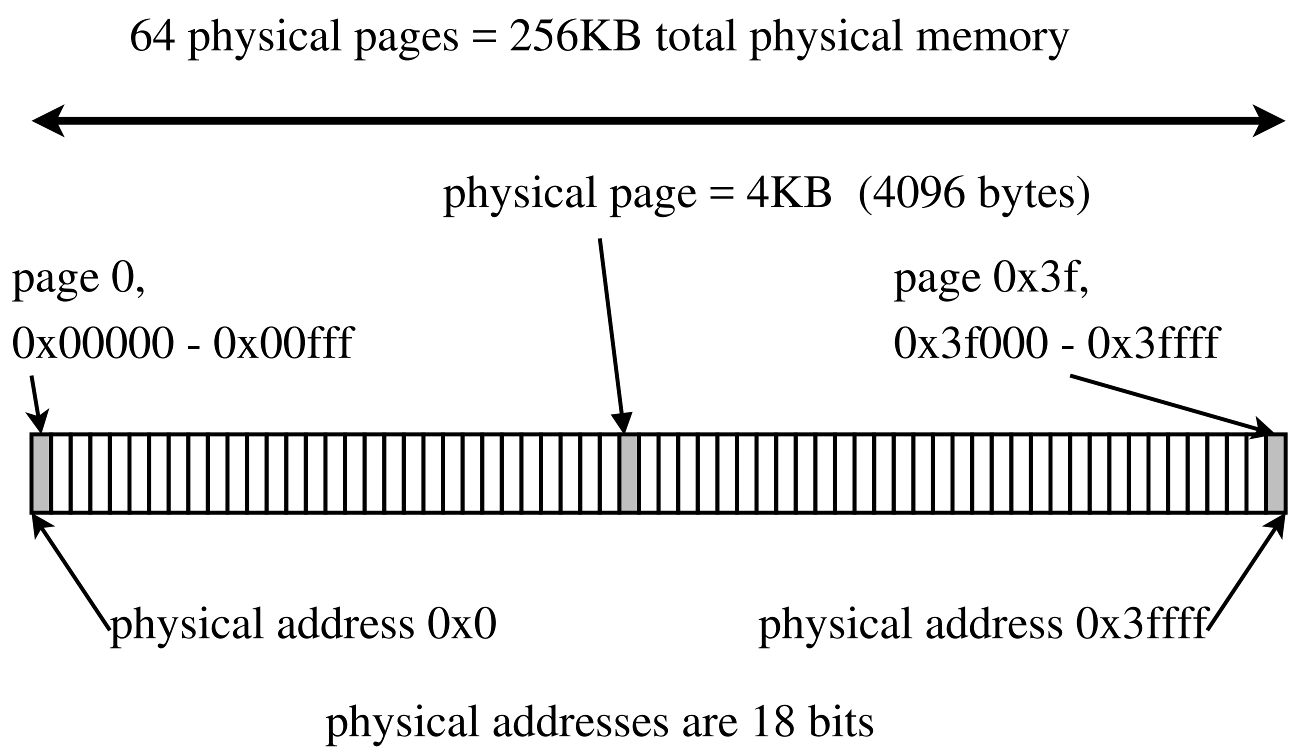


Dynamic Relocation Example: Process B

```
Limit register:        0x0000 c000
Relocation register:   0x0001 3000

v =  0x102c        p = ?
v =  0x8800        p = ?
v =  0x0000        p = ?
```

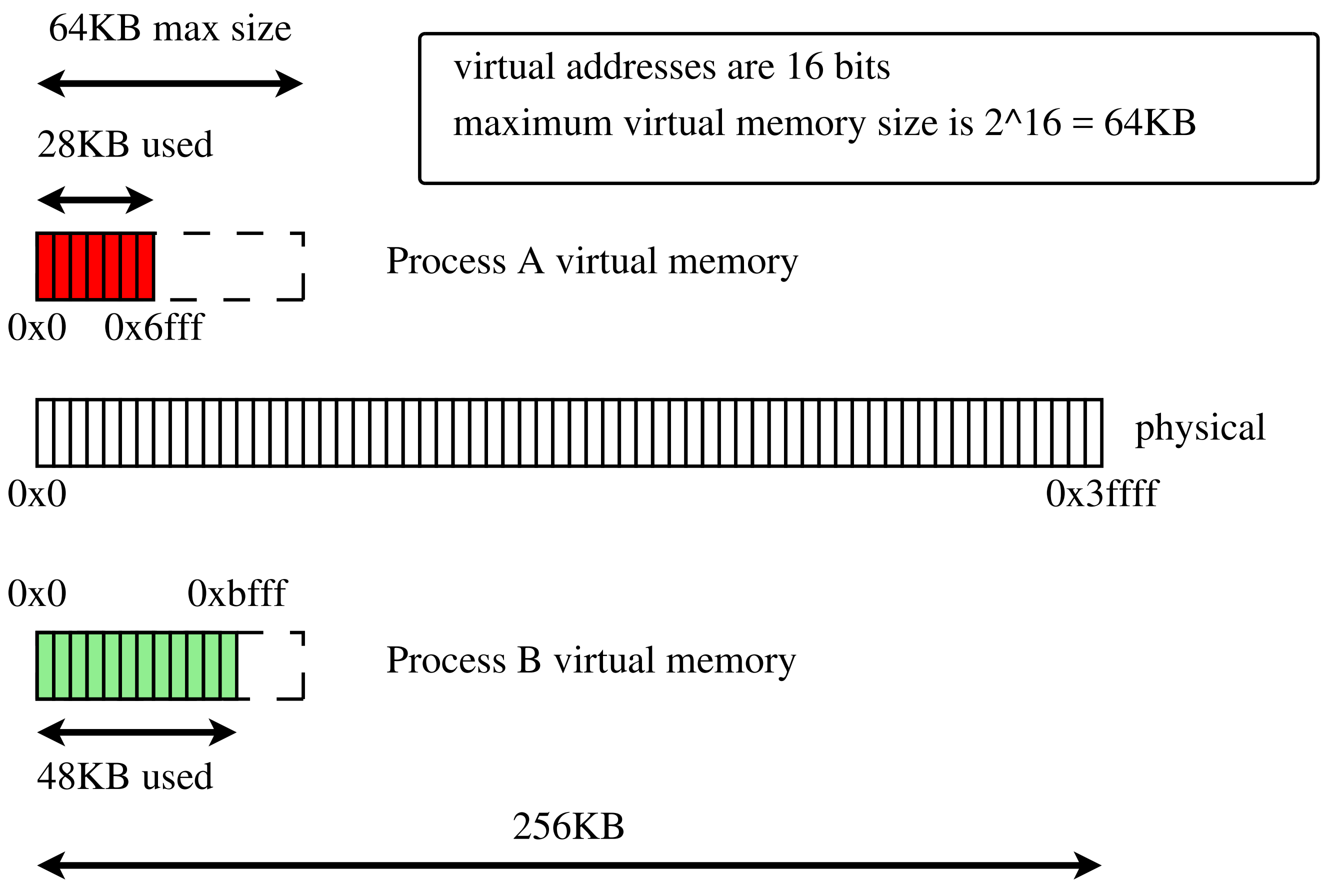


Paging: Physical Memory



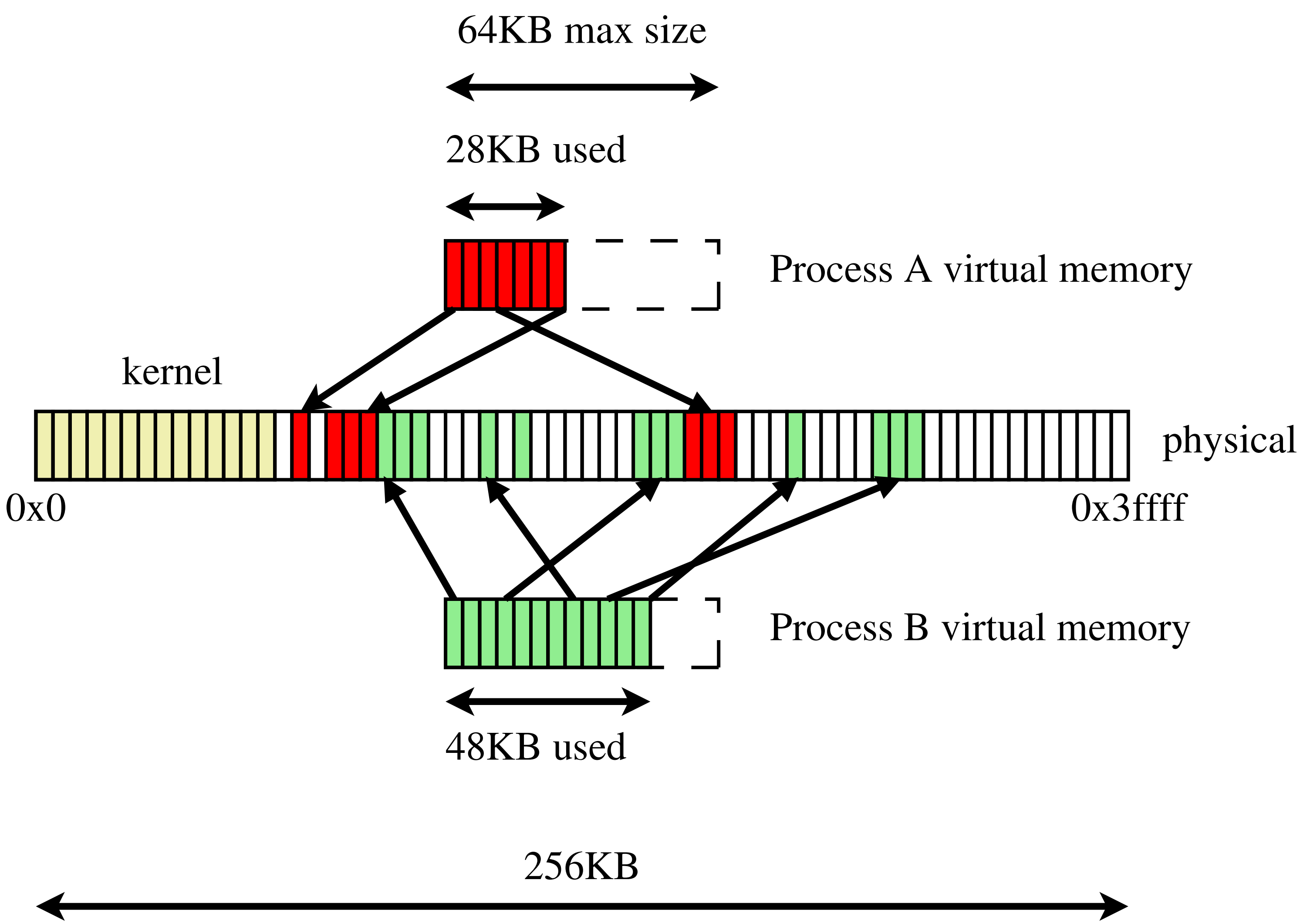
Physical memory is divided into fixed-size chunks called *frames* or *physical pages*. In this example, the frame size is 2^{12} bytes (4KB).

Paging: Virtual Memory



Virtual memories are divided into fixed-size chunks called *pages*. Page size is equal to frame size: 4KB in this example.

Paging: Address Translation



Each page maps to a different frame. Any page can map to any frame.

Page Tables

Process A Page Table

Page	Frame	Valid?
0x0	0x0f	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1
0x4	0x11	1
0x5	0x12	1
0x6	0x13	1
0x7	0x00	0
0x8	0x00	0
...	...	...
0xe	0x00	0
0xf	0x00	0

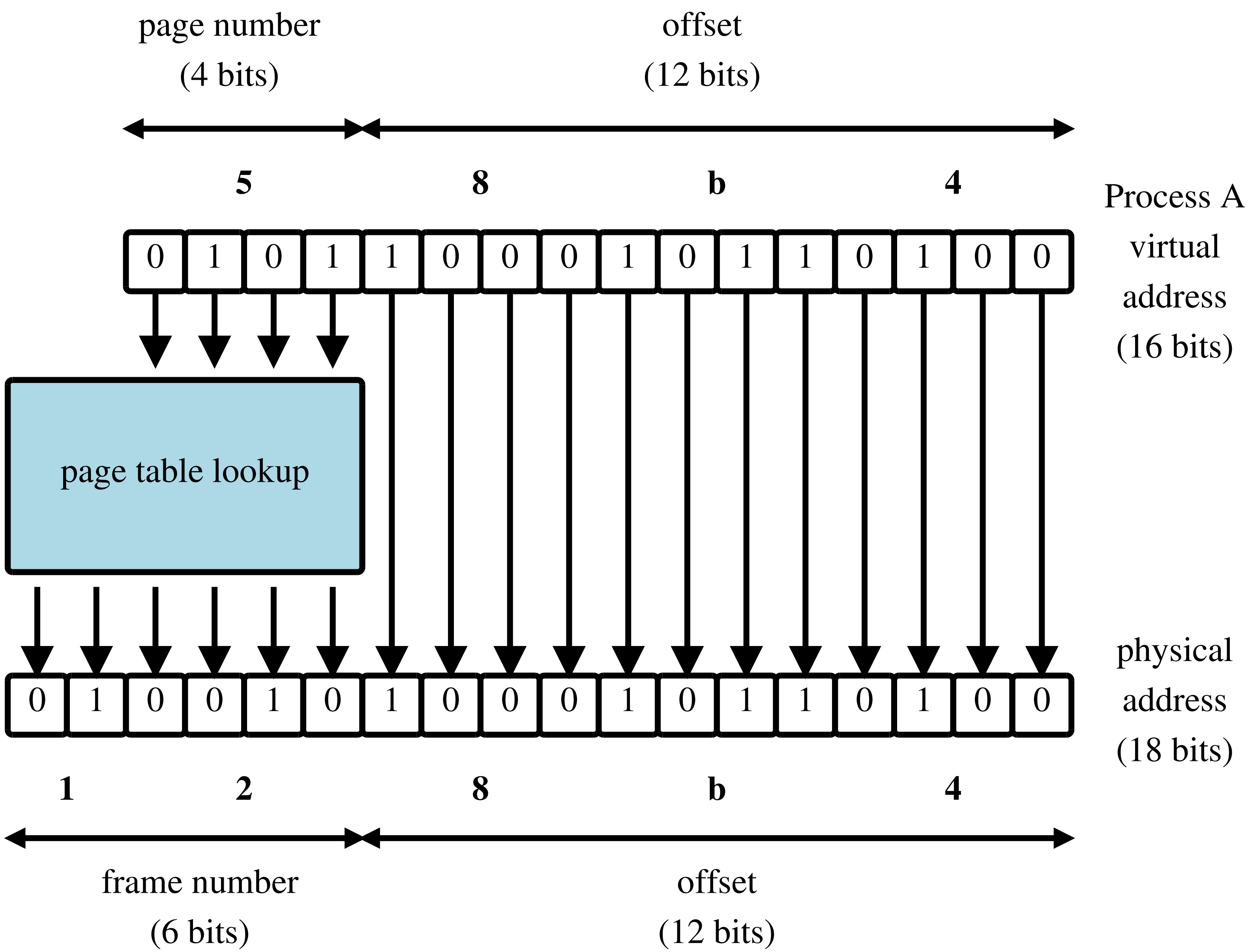
Process B Page Table

Page	Frame	Valid?
0x0	0x14	1
0x1	0x15	1
0x2	0x16	1
0x3	0x23	1
...	...	...
0x9	0x32	1
0xa	0x33	1
0xb	0x2c	1
0xc	0x00	0
0xd	0x00	0
0xe	0x00	0
0xf	0x00	0

Address Translation in the MMU, Using Paging

- The MMU includes a *page table base register* which points to the page table for the current process
- How the MMU translates a virtual address:
  1. determines the *page number* and *offset* of the virtual address
     - page number is the virtual address divided by the page size
     - offset is the virtual address modulo the page size
  2. looks up the page's entry (PTE) in the current process page table, using the page number
  3. if the PTE is not valid, raise an exception
  4. otherwise, combine page's frame number from the PTE with the offset to determine the physical address
     - physical address is (frame number * frame size) + offset

Address Translation Illustrated



Address Translation Examples (Process A)

Process A Page Table

Page	Frame	Valid?
0x0	0x0f	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1
0x4	0x11	1
0x5	0x12	1
0x6	0x13	1
0x7	0x00	0
0x8	0x00	0
...	...	...
0xe	0x00	0
0xf	0x00	0

```
v = 0x102c  p = ?

v = 0x9800  p = ?

v = 0x0024  p = ?
```

Address Translation Examples (Process B)

```
v = 0x102c  p = ?

v = 0x9800  p = ?

v = 0x0024  p = ?
```

Process B Page Table

Page	Frame	Valid?
0x0	0x14	1
0x1	0x15	1
0x2	0x16	1
0x3	0x23	1
...	...	...
0x9	0x32	1
0xa	0x33	1
0xb	0x2c	1
0xc	0x00	0
0xd	0x00	0
0xe	0x00	0
0xf	0x00	0

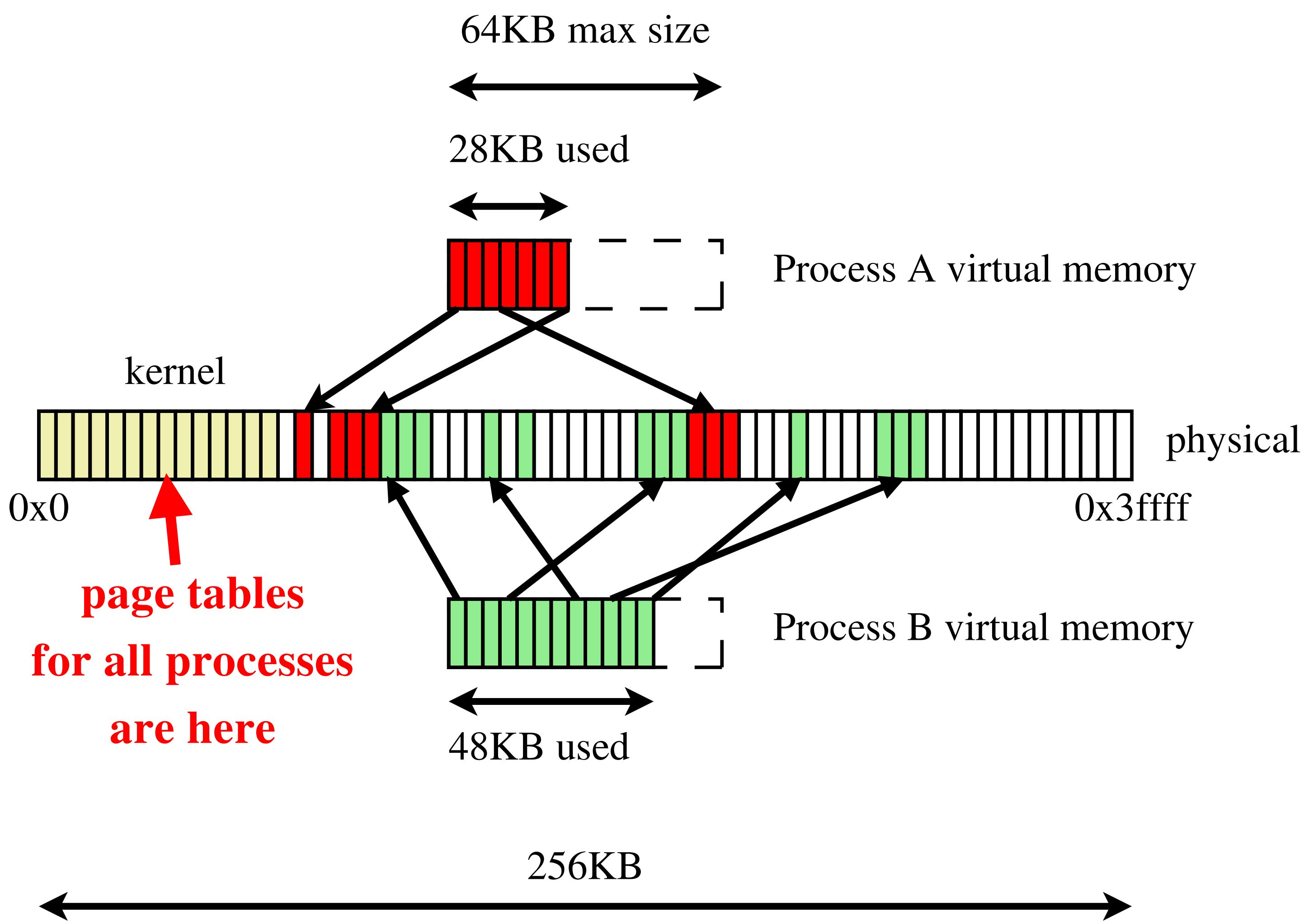
Other Information Found in PTEs

- PTEs may contain other fields, in addition to the frame number and valid bit
- Example 1: write protection bit
  - can be set by the kernel to indicate that a page is read-only
  - if a write operation (e.g., MIPS `lw`) uses a virtual address on a read-only page, the MMU will raise an exception when it translates the virtual address
- Example 2: bits to track page usage
  - reference (use) bit: has the process used this page recently?
  - dirty bit: have contents of this page been changed?
  - these bits are set by the MMU, and read by the kernel (more on this later!)

Page Tables: How Big?

- A page table has one PTE for each page in the virtual memory
  - page table size = (number of pages)*(size of PTE)
  - number of pages = (virtual memory size)/(page size)
- The page table a 64KB virtual memory, with 4KB pages, is 64 bytes, assuming 32 *bits* for each PTE
- Page tables for larger virtual memories are larger (more on this later)

Page Tables: Where?



Page tables are kernel data structures.

Summary: Roles of the Kernel and the MMU

- Kernel:
  - Manage MMU registers on address space switches (context switch from thread in one process to thread in a different process)
  - Create and manage page tables
  - Manage (allocate/deallocate) physical memory
  - Handle exceptions raised by the MMU
- MMU (hardware):
  - Translate virtual addresses to physical addresses
  - Check for and raise exceptions when necessary

TLBs

- Execution of each machine instruction may involve one, two or more memory operations
  - one to fetch instruction
  - one or more for instruction operands
- Address translation through a page table adds one extra memory operation (for page table entry lookup) for each memory operation performed during instruction execution.
- This can be slow!
- Solution: include a Translation Lookaside Buffer (TLB) in the MMU
  - TLB is a small, fast, dedicated cache of address translations, in the MMU
  - Each TLB entry stores a (page# $\rightarrow$ frame#) mapping

TLB Use

- What the MMU does to translate a virtual address on page `p`:

```
if there is an entry (p,f) in the TLB then
   return f      /*  TLB hit! */
else
   find p's frame number (f) from the page table
   add (p,f) to the TLB, evicting another entry if full
   return f      /*  TLB miss */
```

- If the MMU cannot distinguish TLB entries from different address spaces, then the kernel must clear or invalidate the TLB on each context switch from one process to another.

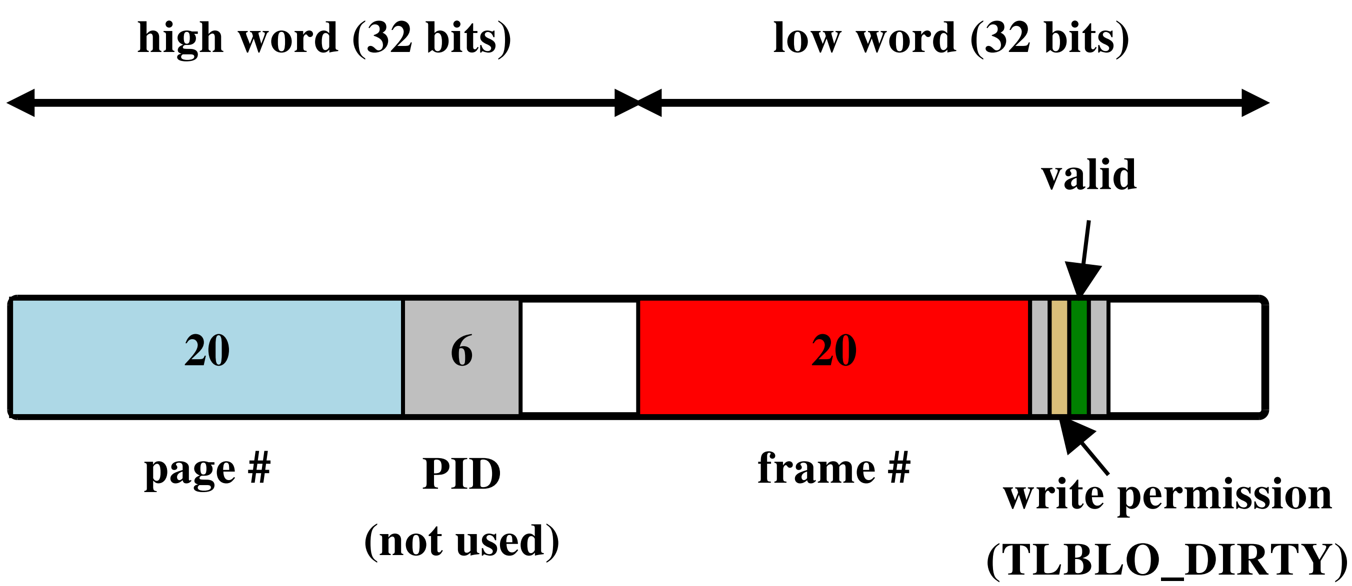
Software-Managed TLBs

- The TLB described on the previous slide is a *hardware-managed TLB*
  - the MMU handles TLB misses, including page table lookup and replacement of TLB entries
  - MMU must understand the kernel's page table format
- The MIPS has a *software-managed TLB*, which translates a virtual address on page `p` like this:

```
if there is an entry (p,f) in the TLB then
   return f     /*  TLB hit! */
else
   raise exception  /* TLB miss */
```

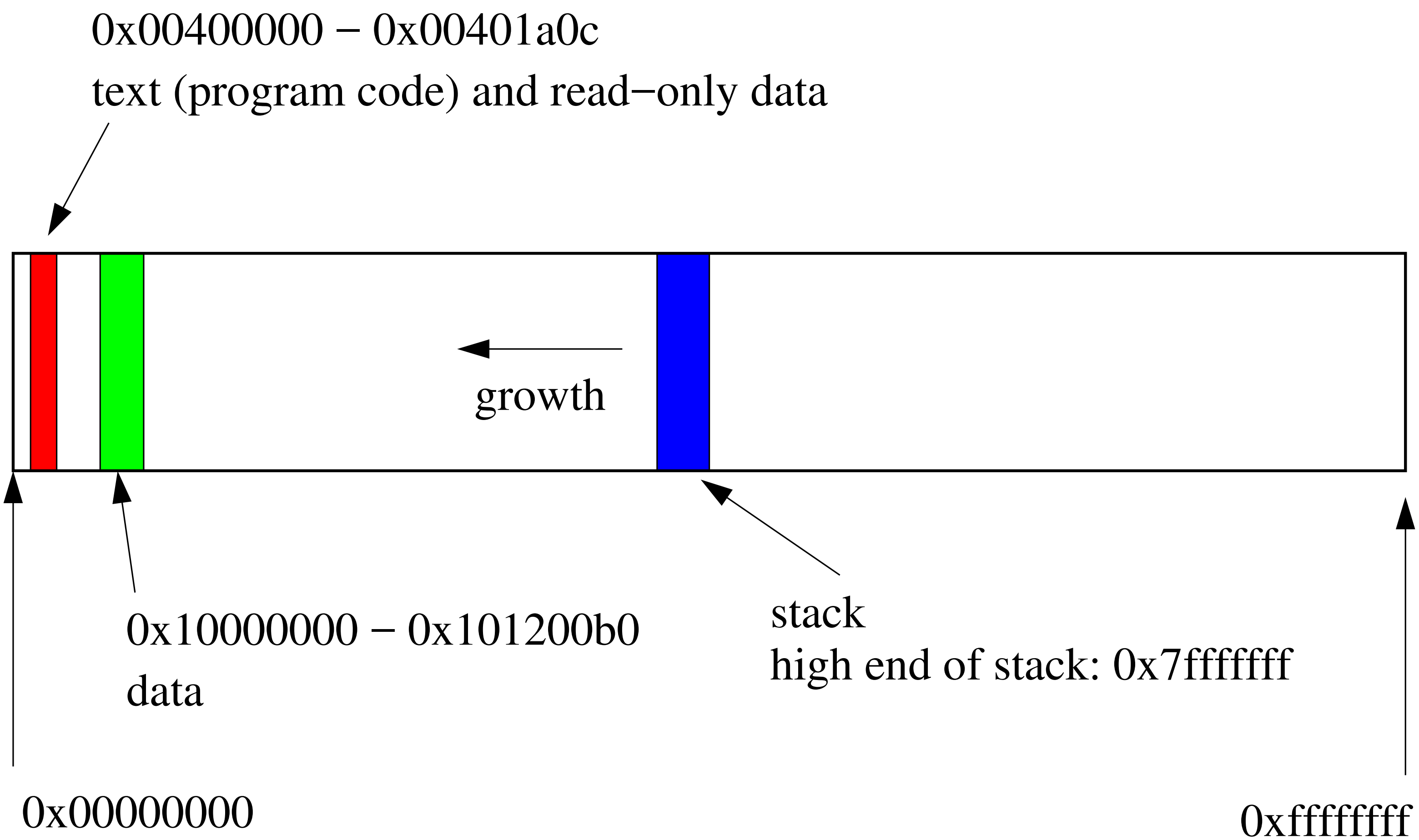
- In case of a TLB miss, the kernel must
  1. determine the frame number for `p`
  2. add `(p,f)` to the TLB, evicting another entry if necessary
- After the miss is handled, the instruction that caused the exception is re-tried

The MIPS R3000 TLB



The MIPS TLB has room for 64 entries. Each entry is 64 bits (8 bytes) long, as shown. See `kern/arch/mips/include/tlb.h`

A More Realistic Virtual Memory



This diagram illustrates the layout of the virtual address space for the OS/161 test application `user/testbin/sort`

Large, Sparse Virtual Memories

- Virtual memory may be large
  - MIPS: $V = 32$, max virtual memory size is 2^{32} bytes (4 GB)
  - x86-64: $V = 48$, max virtual memory size is 2^{48} bytes (256 TB)
- Much of the virtual memory may be unused.
  - `testbin/sort` needs only about 1.2MB of the full 4GB virtual memory it runs in.
- Applicaion may use *discontiguous* segments of the virtual memory
  - In the `testbin/sort` address space, the data and stack segments are widely spaced. (Why?)

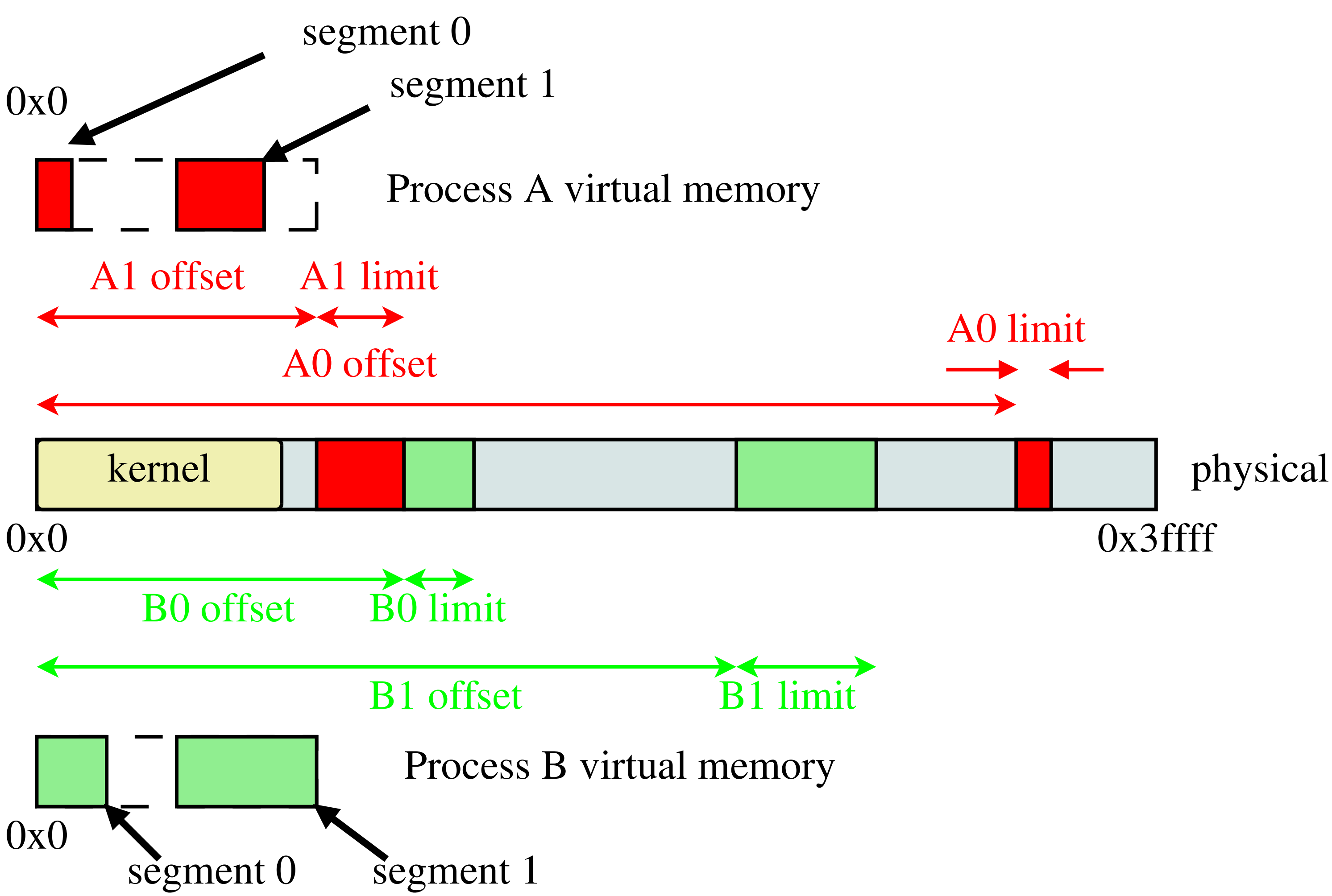
Limitations of Simple Address Translation Approaches

- A kernel that used simple dynamic relocation would have to allocate 2GB of contiguous physical memory for `testbin/sort`'s virtual memory
  - even though `sort` only uses about 1.2MB
- A kernel that used simple paging would require a page table with 2^{20} PTEs (assuming page size is 4 KB) to map `tesbin/sort`'s address space.
  - this page table is actually larger than the virtual memory that `sort` needs to use!
  - most of the PTEs are marked as invalid
  - this page table has to be contiguous in kernel memory

Segmentation

- Instead of mapping the entire virtual memory to physical, we can provide a separate mapping for each *segment* of the virtual memory that the application actually uses.
- Instead of a single offset and limit for the entire address space, the kernel maintains an offset and limit for each segment.
- With segmentation, a virtual address can be thought of as having two parts: (segment ID, offset within segment)
- with K bits for the segment ID, we can have up to:
  - 2^K segments
  - 2^{V-K} bytes per segment
- The kernel decides where each segment is placed in physical memory.
  - Fragmentation of physical memory is possible

Segmented Address Space Diagram



Translating Segmented Virtual Addresses

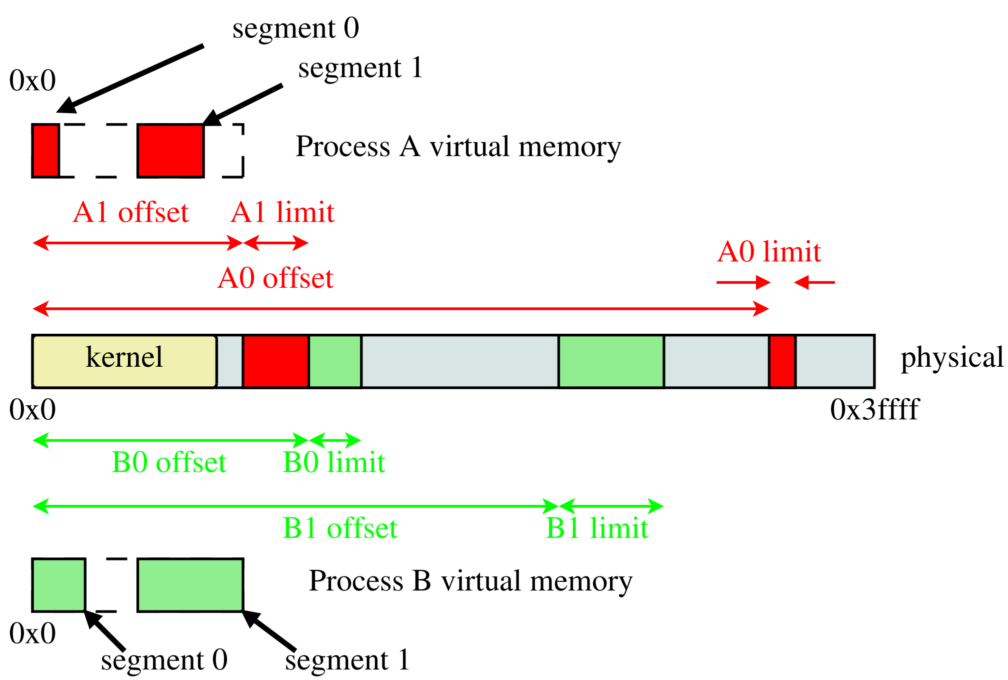
- Many different approaches for translating segmented virtual addresses
- Approach 1: MMU has a relocation register and a limit register for each segment
  - let R_i be the relocation offset and L_i be the limit offset for the ith segment
  - To translate virtual address v to a physical address p:

```
split p into segment number (s) and
    address within segment (a)
if a ≥ L_s then generate exception
else
  p ← a + R_i
```

  - As for dynamic relocation, the kernel maintains a separate set of relocation offsets and limits for each process, and changes the values in the MMU's registers when there is a context switch between processes.

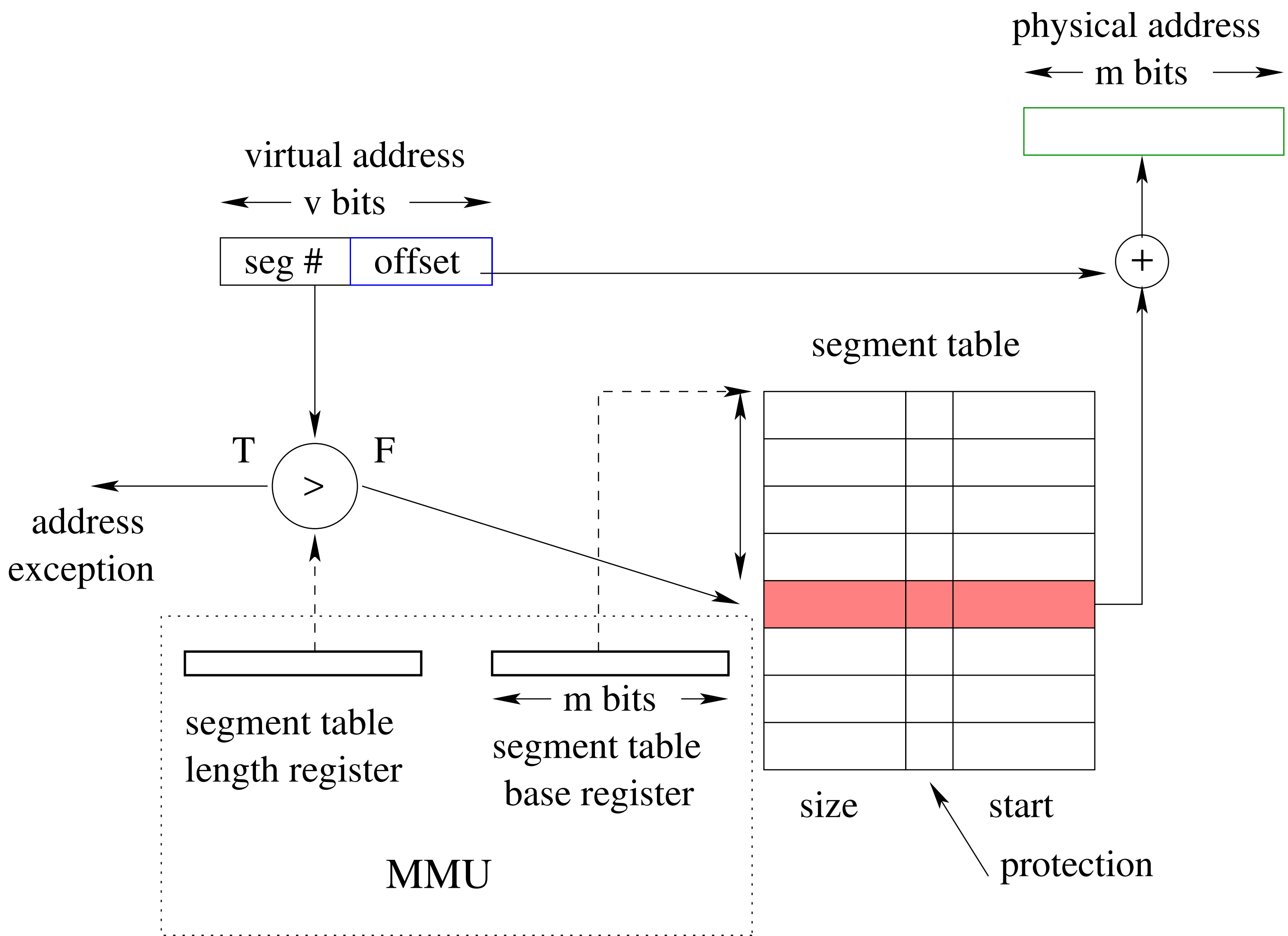
Segmented Address Translation Example (Process A)

```
Limit Register 0: 0x2000   Relocation Register 0: 0x38000
Limit Register 1: 0x5000   Relocation Register 1: 0x10000

v =  0x1240   segment=?   offset=?   p = ?
v =  0xa0a0   segment=?   offset=?   p = ?
v =  0x66ac   segment=?   offset=?   p = ?
v =  0xe880   segment=?   offset=?   p = ?
```



Translating Segmented Virtual Addresses

- Approach 2: Maintain a segment table



Two-Level Paging

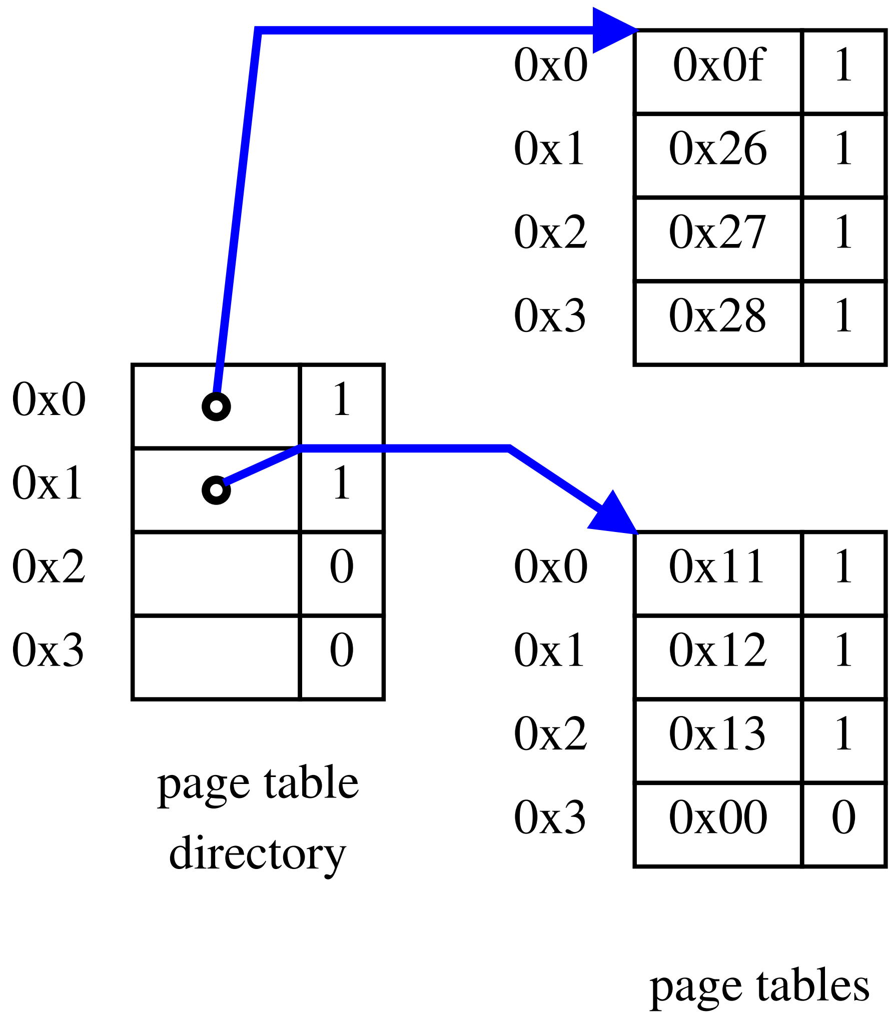
- Instead of having a single page table to map an entire virtual memory, we can split the page table into smaller page tables, and add page table directory.
  - instead of one large, contiguous table, we have multiple smaller tables
  - if all PTEs in a smaller table are invalid, we can avoid creating that table entirely
- each virtual address has three parts:
  - level one page number: used to index the directory
  - level two page number: used to index a page table
  - offset within the page

Two-Level Paging Example (Process A)

single level paging

0x0	0x0f	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1
0x4	0x11	1
0x5	0x12	1
0x6	0x13	1
0x7	0x00	0
0x8	0x00	0
0x9	0x00	0
....		
0xe	0x00	0
0xf	0x00	0

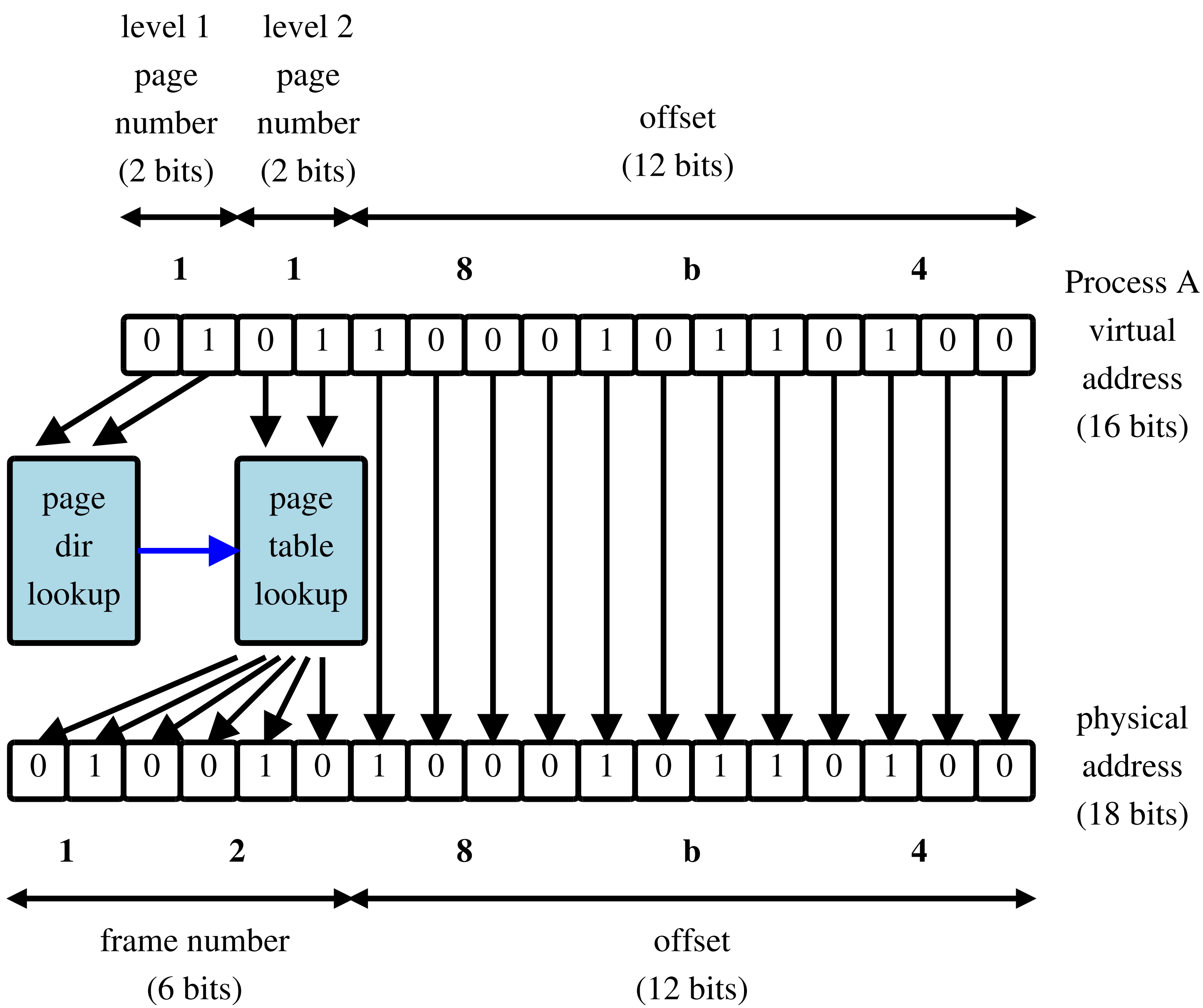
two level paging



Address Translation with Two-Level Paging

- The MMU's *page table base register* points to the page table directory for the current process.
- Each virtual address v has three parts: (p_1, p_2, o)
- How the MMU translates a virtual address:
  1. index into the page table directory using p_1 to get a pointer to a 2nd level page table
  2. if the directory entry is not valid, raise an exception
  3. index into the 2nd level page table using p_2 to find the PTE for the page being accessed
  4. if the PTE is not valid, raise an exception
  5. otherwise, combine the frame number from the PTE with o to determine the physical address (as for single-level paging)

Two-Level Address Translation Example



Limits of Two-Level Paging

- One goal of two-level paging was to keep individual page tables small.
- Suppose we have 40 bit virtual addresses ($V = 40$) and that
  - the size of a PTE is 4 bytes
  - page size is 4KB (2^{12} bytes)
  - we'd like to limit each page table's size to 4KB
- Problem: for large address spaces, we may need a large page table directory!
  - there can be up to 2^{28} pages in a virtual memory
  - a single page table can hold 2^{10} PTEs
  - we may need up to 2^{18} page tables
  - our page table directory will have to have 2^{18} entries
  - if a directory entry is 4 bytes, the directory will occupy 1MB
- this is the problem we were trying to avoid by introducing a second level

Multi-Level Paging

- We can solve the large directory problem by introducing additional levels of directories.
- Example: 4-level paging in x86-64 architecture
- Properties of Multi-Level Paging
  - Can map large virtual memories by adding more levels.
  - Individual page tables/directories can remain small.
  - Can avoid allocating page tables and directories that are not needed for programs that use a small amount of virtual memory.
  - TLB misses become more expensive as the number of levels goes up, since more directories must be accessed to find the correct PTE.

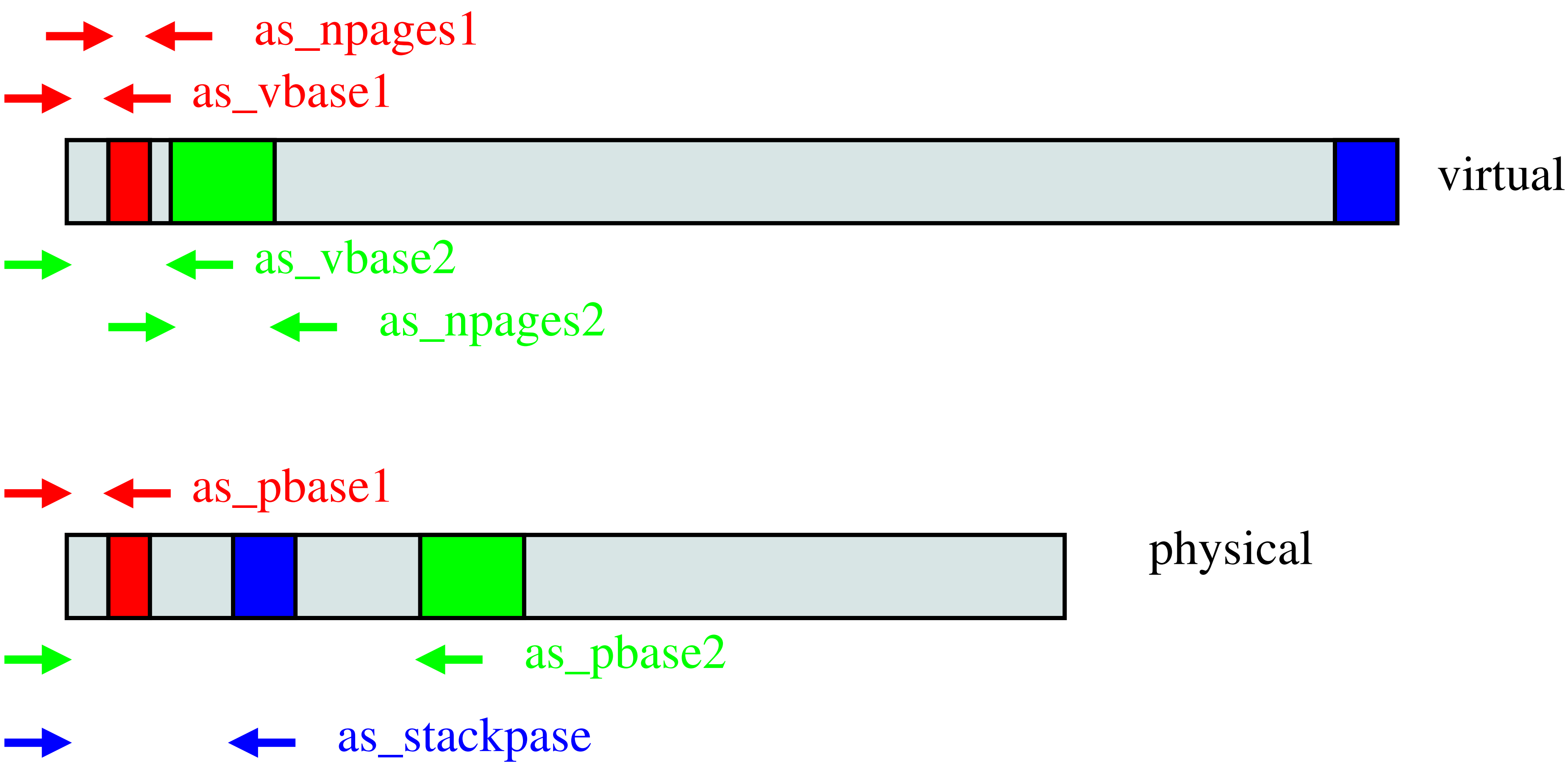
Virtual Memory in OS/161 on MIPS: **`dumbvm`**

- the MIPS uses 32-bit paged virtual and physical addresses
- the MIPS has a software-managed TLB
  - TLB raises an exception on every TLB miss
  - kernel is free to record page-to-frame mappings however it wants to
- TLB exceptions are handled by a kernel function called `vm_fault`
- `vm_fault` uses information from an `addrspace` structure to determine a page-to-frame mapping to load into the TLB
  - there is a separate `addrspace` structure for each process
  - each `addrspace` structure describes where its process's pages are stored in physical memory
  - an `addrspace` structure does the same job as a page table, but the `addrspace` structure is simpler because OS/161 places all pages of each segment *contiguously* in physical memory

The `addrspace` Structure

```
struct addrspace {
  vaddr_t as_vbase1; /* base virtual address of code segment */
  paddr_t as_pbase1; /* base physical address of code segment */
  size_t as_npages1; /* size (in pages) of code segment */
  vaddr_t as_vbase2; /* base virtual address of data segment */
  paddr_t as_pbase2; /* base physical address of data segment */
  size_t as_npages2; /* size (in pages) of data segment */
  paddr_t as_stackpbase; /* base physical address of stack */
};
```

addrspace Diagram



Address Translation: OS/161 dumbvm Example

- Note: in OS/161 the stack is 12 pages and the page size is 4 KB = 0x1000.

Variable/Field	Process 1	Process 2
as_vbase1	0x0040 0000	0x0040 0000
as_pbase1	0x0020 0000	0x0050 0000
as_npages1	0x0000 0008	0x0000 0002
as_vbase2	0x1000 0000	0x1000 0000
as_pbase2	0x0080 0000	0x00A0 0000
as_npages2	0x0000 0010	0x0000 0008
as_stackpbase	0x0010 0000	0x00B0 0000

	Process 1	Process 2
Virtual addr	0x0040 0004	0x0040 0004
Physical addr =	____________ ?	____________ ?
Virtual addr	0x1000 91A4	0x1000 91A4
Physical addr =	____________ ?	____________ ?
Virtual addr	0x7FFF 41A4	0x7FFF 41A4
Physical addr =	____________ ?	____________ ?
Virtual addr	0x7FFF 32B0	0x2000 41BC
Physical addr =	____________ ?	____________ ?

Initializing an Address Space

- When the kernel creates a process to run a particular program, it must create an address space for the process, and load the program's code and data into that address space

---

OS/161 *pre-loads* the address space before the program runs. Many other OS load pages *on demand*. (Why?)

- A program's code and data is described in an *executable file*, which is created when the program is compiled and linked
- OS/161 (and some other operating systems) expect executable files to be in ELF (**E**xecutable and **L**inking **F**ormat) format
- The OS/161 `execv` system call re-initializes the address space of a process

```
int execv(const char *program, char **args)
```

- The `program` parameter of the `execv` system call should be the name of the ELF executable file for the program that is to be loaded into the address space.

ELF Files

- ELF files contain address space segment descriptions, which are useful to the kernel when it is loading a new address space
- the ELF file identifies the (virtual) address of the program's first instruction
- the ELF file also contains lots of other information (e.g., section descriptors, symbol tables) that is useful to compilers, linkers, debuggers, loaders and other tools used to build programs

Address Space Segments in ELF Files

- The ELF file contains a header describing the segments and segment *images*.
- Each ELF segment describes a contiguous region of the virtual address space.
- The header includes an entry for each segment which describes:
  - the virtual address of the start of the segment
  - the length of the segment in the virtual address space
  - the location of the start of the segment image in the ELF file (if present)
  - the length of the segment image in the ELF file (if present)
- the image is an exact copy of the binary data that should be loaded into the specified portion of the virtual address space
- the image may be smaller than the address space segment, in which case the rest of the address space segment is expected to be zero-filled

---

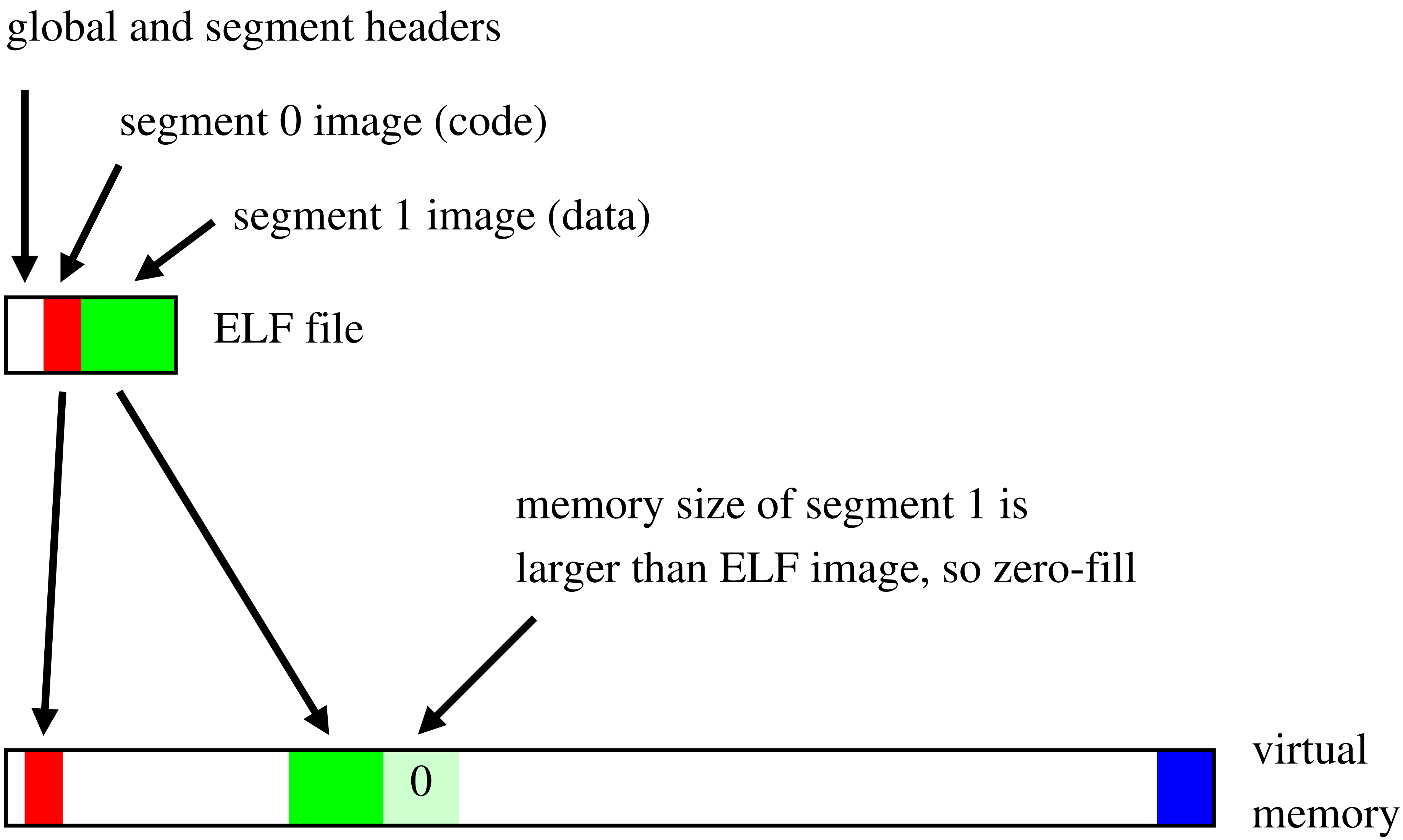
To initialize an address space, the OS/161 kernel copies segment images from the ELF file to the specifed portions of the virtual address space.

ELF Files and OS/161

- OS/161's `dumbvm` implementation assumes that an ELF file contains two segments:
  - a *text segment*, containing the program code and any read-only data
  - a *data segment*, containing any other global program data
- the ELF file does not describe the stack (why not?)
- `dumbvm` creates a *stack segment* for each process. It is 12 pages long, ending at virtual address `0x7fffffff`

Look at `kern/syscall/loadelf.c` to see how OS/161 loads segments from ELF files

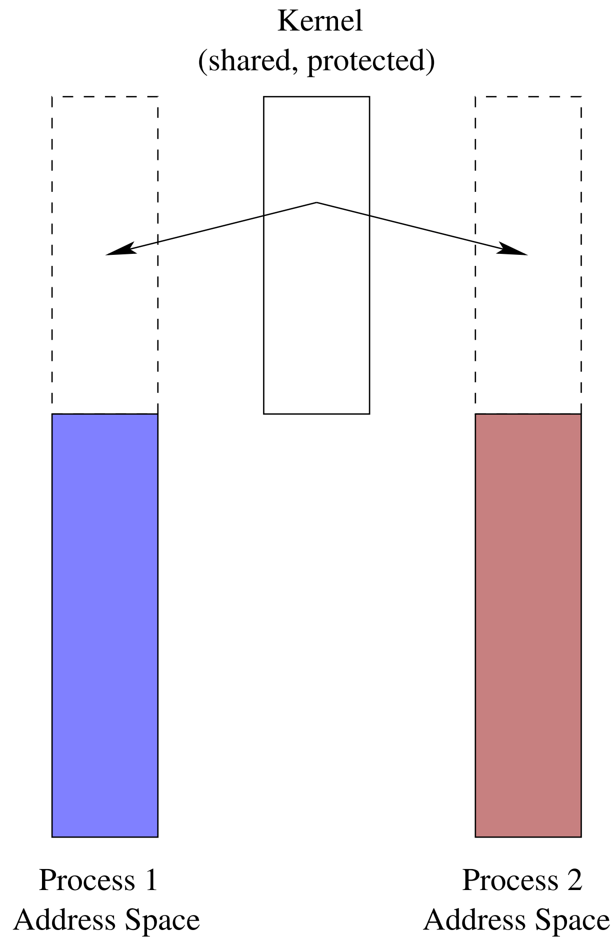
ELF File Diagram



Virtual Memory for the Kernel

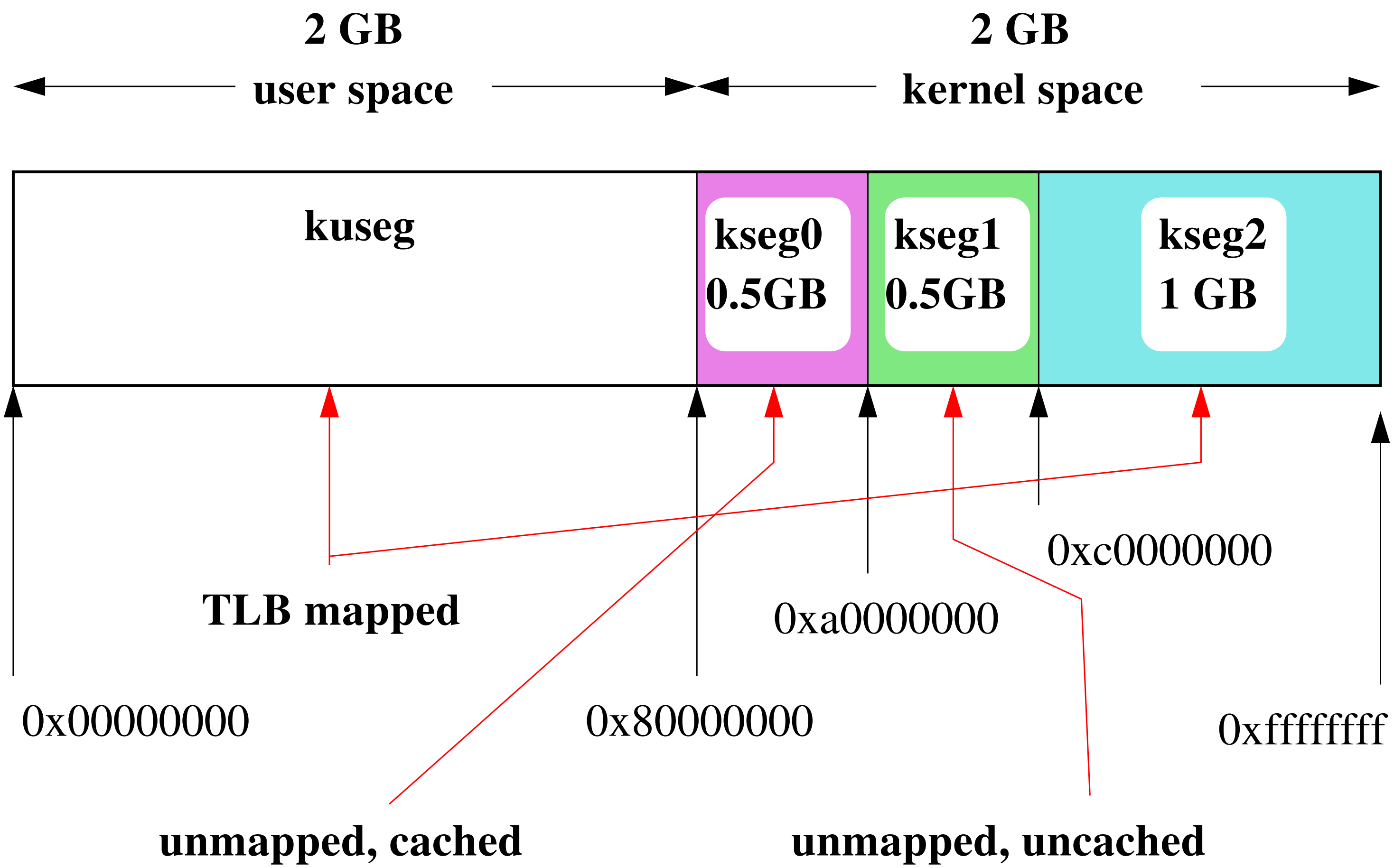
- We would like the kernel to live in virtual memory, but there are some challenges:
    1. **Bootstrapping:** Since the kernel helps to implement virtual memory, how can the kernel run in virtual memory when it is just starting?
    2. **Sharing:** Sometimes data need to be copied between the kernel and application programs? How can this happen if they are in different virtual address spaces?
- The sharing problem can be addressed by making the kernel's virtual memory *overlap* with process' virtual memories.
- Solutions to the bootstrapping problem are architecture-specific.

The Kernel in Process' Address Spaces



Attempts to access kernel code/data in user mode result in memory protection exceptions, not invalid address exceptions.

User Space and Kernel Space on the MIPS R3000



In OS/161, user programs live in kuseg, kernel code and data structures live in kseg0, devices are accessed through kseg1, and kseg2 is not used.

Exploiting Secondary Storage

Goals:

- Allow virtual address spaces that are larger than the physical address space.
- Allow greater multiprogramming levels by using less of the available (primary) memory for each process.

Method:

- Allow pages from virtual memories to be stored in secondary storage, i.e., on disks or SSDs.
- Swap pages (or segments) between secondary storage and primary memory so that they are in primary memory when they are needed.

Resident Sets and Present Bits

- When swapping is used, some pages of each virtual memory will be in memory, and others will not be in memory.
  - The set of virtual pages present in physical memory is called the *resident set* of a process.
  - A process's resident set will change over time as pages are swapped in and out of physical memory
- To track which pages are in physical memory, each PTE needs to contain an extra bit, called the *present* bit:
  - $\text{valid} = 1, \text{present} = 1$: page is valid and in memory
  - $\text{valid} = 1, \text{present} = 0$: page is valid, but not in memory
  - $\text{valid} = 0, \text{present} = x$: invalid page

Page Faults

- When a process tries to access a page that is not in memory, the problem is detected because the page's *present* bit is zero:
  - on a machine with a hardware-managed TLB, the MMU detects this when it checks the page's PTE, and generates an exception, which the kernel must handle
  - on a machine with a software-managed TLB, the kernel detects the problem when it checks the page's PTE after a TLB miss.
- This event (attempting to access a non-resident page) is called a *page fault*.
- When a page fault happens, it is the kernel's job to:
  1. Swap the page into memory from secondary storage, evicting another page from memory if necessary.
  2. Update the PTE (set the *present* bit)
  3. Return from the exception so that the application can retry the virtual memory access that caused the page fault.

Secondary Storage is Slow

- Access times for disks are measured in *milli*seconds, SSD read latencies are 10's-100's of *micro*seconds.
- Both of these are much higher than memory access times (100's of *nano*seconds)
- Suppose that secondary storage access is 1000 times slower than memory access. Then:
  - If there is one page fault every 10 memory accesses (on average), the average memory access time with swapping will be about 100 times larger than it would be without swapping.
  - If there is one page fault every 100 memory accesses (on average), the average memory access time with swapping will be about 10 times larger than it would be without swapping.
  - If there is one page fault every 1000 memory accesses (on average), the average memory access time with swapping will be about 2 times larger than it would be without swapping.

Peformance with Swapping

- To provide good performance for virtual memory accesses, the kernel should try to ensure that page faults are rare.
- Some techniques the kernel can use to improve performance:
  - limit the number of processes, so that there is enough physical memory per process
  - try to be smart about *which* pages are kept in physical memory, and which are evicted.
  - hide latencies, e.g., by *prefetching* pages before a process needs them

A Simple Replacement Policy: FIFO

- replacement policy: when the kernel needs to evict a page from physical memory, which page should it evict?
- the FIFO policy: replace the page that has been in memory the longest
- a three-frame example:

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	d	d	d	e	e	e	e	e	e
Frame 2		b	b	b	a	a	a	a	a	c	c	c
Frame 3			c	c	c	b	b	b	b	b	d	d
Fault ?	x	x	x	x	x	x	x			x	x	

Optimal Page Replacement

- There is an optimal page replacement policy for demand paging, called MIN: replace the page that will not be referenced for the longest time.

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	a	a	a	a	a	a	c	c	c
Frame 2		b	b	b	b	b	b	b	b	b	d	d
Frame 3			c	d	d	d	e	e	e	e	e	e
Fault ?	x	x	x	x			x			x	x	

- MIN requires knowledge of the future.

Locality

- Real programs do not access their virtual memories randomly. Instead, they exhibit *locality*:
  - **temporal locality:** programs are more likely to access pages that they have accessed recently than pages that they have not accessed recently.
  - **spatial locality:** programs are likely to access parts of memory that are close to parts of memory they have accessed recently.
- Locality helps the kernel keep page fault rates low.

Least Recently Used (LRU) Page Replacement

- the same three-frame example:

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	d	d	d	e	e	e	c	c	c
Frame 2		b	b	b	a	a	a	a	a	a	d	d
Frame 3			c	c	c	b	b	b	b	b	b	e
Fault ?	x	x	x	x	x	x	x			x	x	x

Measuring Memory Accesses

- The kernel is not aware which pages a program is using unless there is an exception.
- This makes it difficult for the kernel to exploit locality by implementating a replacement policy like LRU.
- The MMU can help solve this problem by tracking page accesses in hardware.
- Simple scheme: add a *use bit* (or *reference bit*) to each PTE. This bit:
  - is set by the MMU each time the page is used, i.e., each time the MMU translates a virtual address on that page
  - can be read and cleared by the kernel.
- The use bit provides a small amount of memory usage information that can be exploited by the kernel.

The Clock Replacement Algorithm

- The clock algorithm (also known as “second chance”) is one of the simplest algorithms that exploits the use bit.
- Clock is identical to FIFO, except that a page is “skipped” if its use bit is set.
- The clock algorithm can be visualized as a victim pointer that cycles through the page frames. The pointer moves whenever a replacement is necessary:

```
while use bit of victim is set
   clear use bit of victim
   victim = (victim + 1) % num_frames
choose victim for replacement
victim = (victim + 1) % num_frames
```