Virtual Memory

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

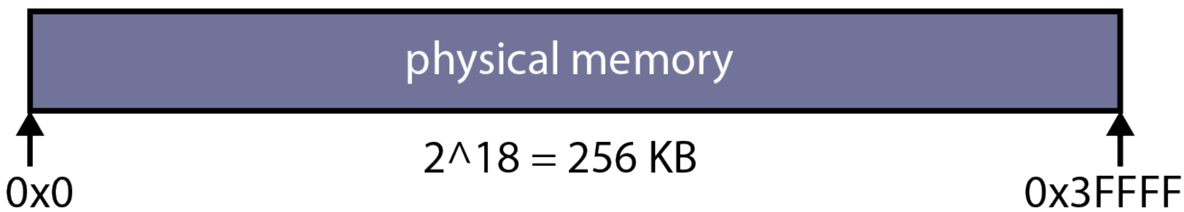
Ali Mashtizadeh and Lesley Istead

David R. Cheriton School of Computer Science
University of Waterloo

Fall 2019

Physical Memory

- If physical addresses are P **bits**, then the maximum amount of addressable physical memory is 2^P **bytes**.
  - Sys/161 MIPS CPU uses 32 bit physical addresses ($P = 32$) $\Rightarrow$ maximum physical memory size of 2^{32} bytes, or 4GB.
  - Larger values of P on modern CPUs, e.g., $P = 48 \Rightarrow$ 256 TB of physical memory to be addressed.
  - The examples in these slides use $P = 18$

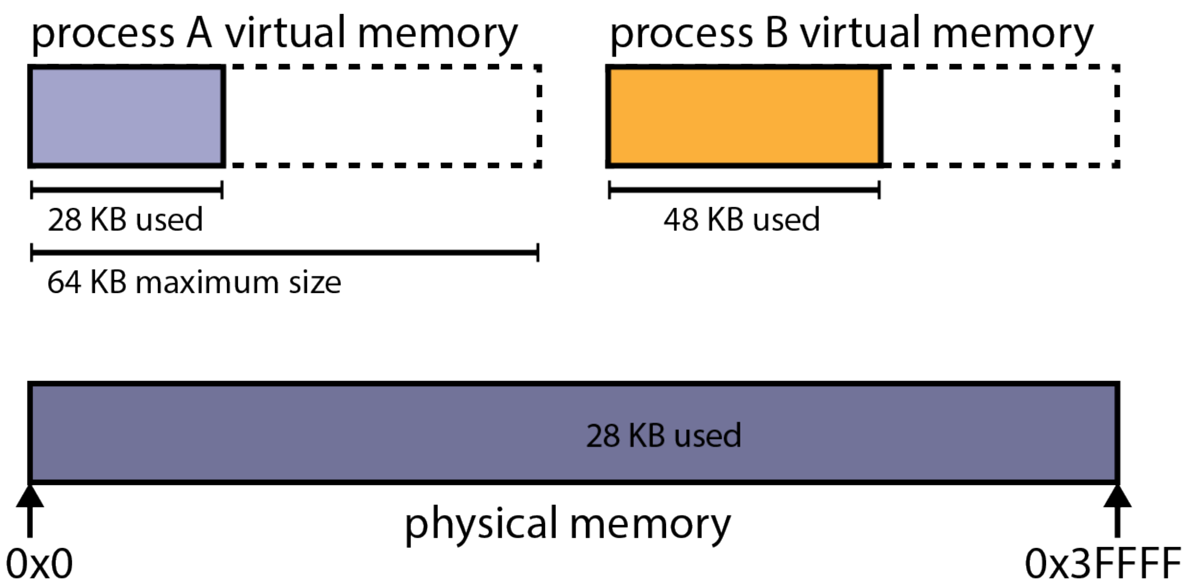


The actual amount of physical memory on a machine may be less than the maximum amount that can be addressed.

Virtual Memory

- 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.

Virtual Memory



virtual addresses are 16 bits
maximum virtual memory size is 64KB

Why virtual memory?

- isolate processes from each other; kernel
- potential to support virtual memory larger than physical memory
- the total size of all VMs can be larger than physical memory (greater support for multiprocessing)

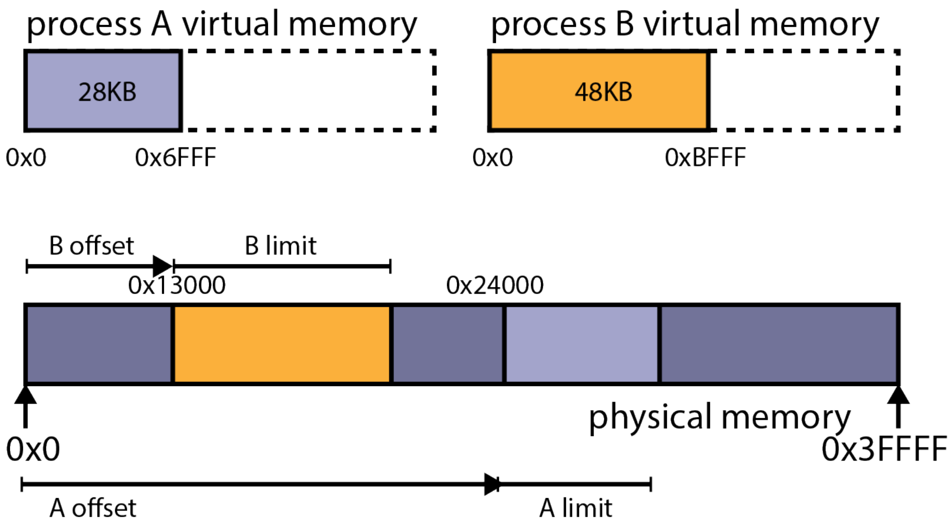
> The concept of virtual memory dates back to a doctoral thesis in 1956. Burroughs (1961) and Atlas (1962) produced the first commercial machines with virtual memory support.

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, on the **Memory Managment Unit, MMU**, using information provided by the kernel.

> Even the program counter (PC) is a virtual address. Each instruction requires at **least one** translation. Hence, the translation is done in hardware, which is faster than software.

Dynamic Relocation



offset, or, relocation (R) is the position in physical memory where the processes memory begins. **limit (L)** is the amount of memory used by the process. The MMU has a register for each.
Given virtual address v:
if $v < L$ then physical address $p = v +$ relocation
else **exception**

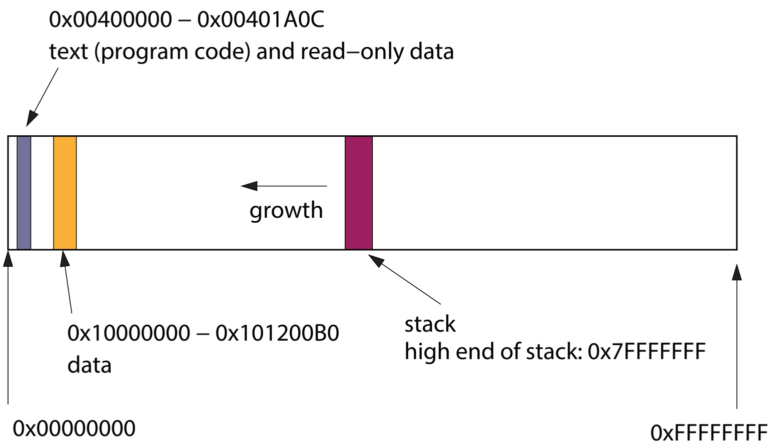
Dynamic Relocation Example

Process A	Process B
Limit Register: 0x0000 7000	Limit Register: 0x0000 C000
Relocation Register: 0x0002 4000	Relocation Register: 0x0001 3000
v = 0x102C p = ?	v = 0x102C p = ?
v = 0x8000 p = ?	v = 0x8000 p = ?
v = 0x0000 p = ?	v = 0x0000 p = ?

Recall

Addresses that cannot be translated produce **exceptions**.
Though efficient, dynamic relocation suffers from **fragmentation**.

A More Realistic Virtual Memory



This is the layout of the virtual address space for the OS/161 test application `user/testbin/sort`. Note that it requires a total of 1.2MB divided between code, data, and stack segments. However, the virtual memory is 4GB in size (32bit addressing).

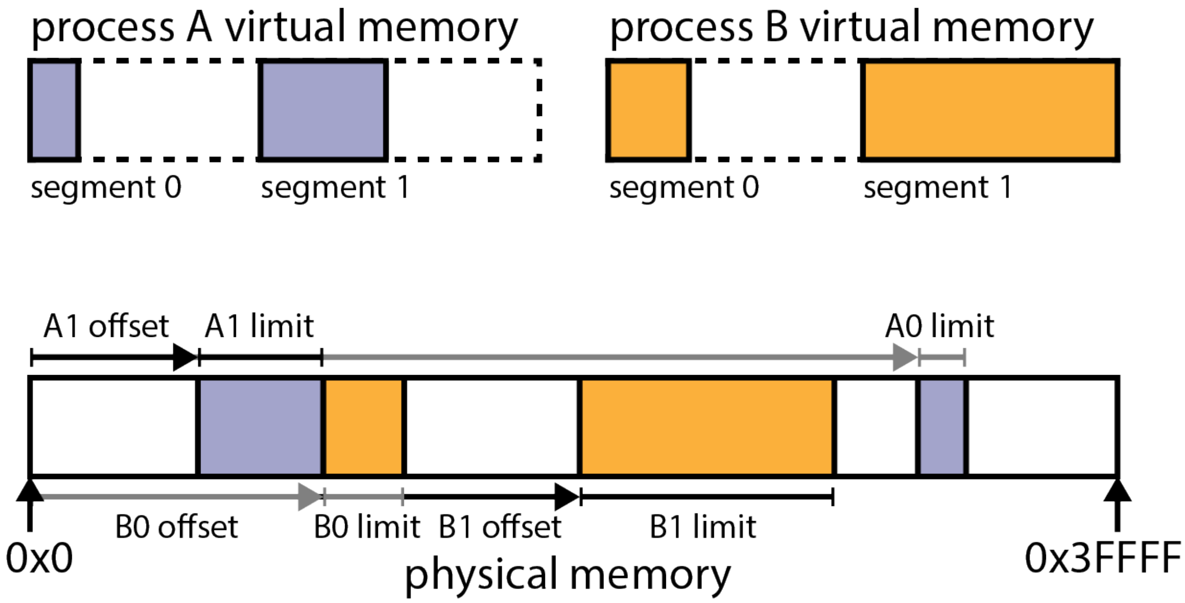
Dynamic relocation would require 2GB of space for `sort`. Why? Because dynamic relocation uses a single, contiguous block of virtual memory for its address spaces, and 2GB is the smallest, continguous chunk that would fit the entire address space for `sort`.

Segmentation

- Instead of mapping the entire virtual memory to physical, we map each **segment** of the virtual memory that the application uses separately.
- 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 still possible

If there are 4 segments, $\log(4) = 2$ bits are required to represent the segment number. The maximum size of each segment is then: $2^{V-K} = 2^{V-2}$ bytes.

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 for the ith segment
  - To translate virtual address v to a physical address p:

```
split v 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

Segment	Limit Register	Relocation Register
0	0x2000	0x38000
1	0x5000	0x10000

Process B

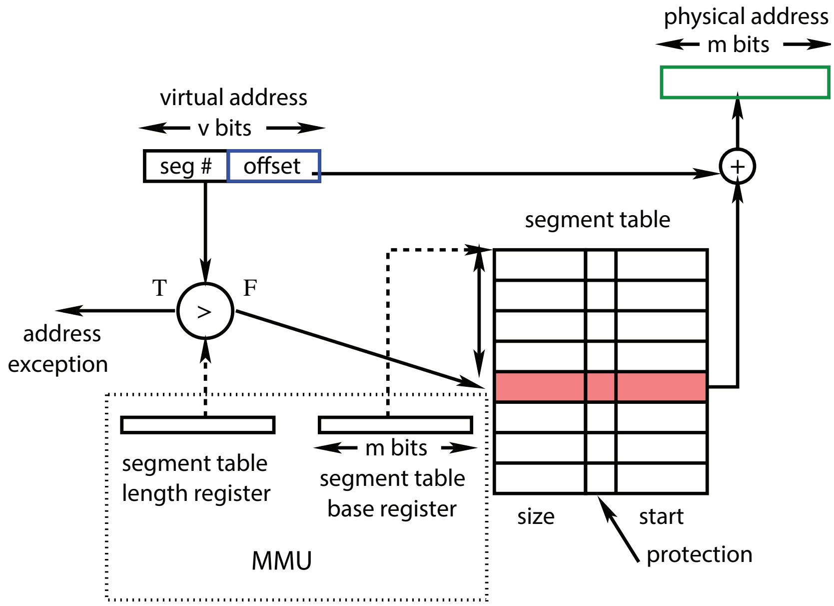
Segment	Limit Register	Relocation Register
0	0x3000	0x15000
1	0xB000	0x22000

Translate the following for process A and B:

Address	Segment	Offset	Physical Address
v = 0x1240			
v = 0xA0A0			
v = 0x66AC			
v = 0xE880			

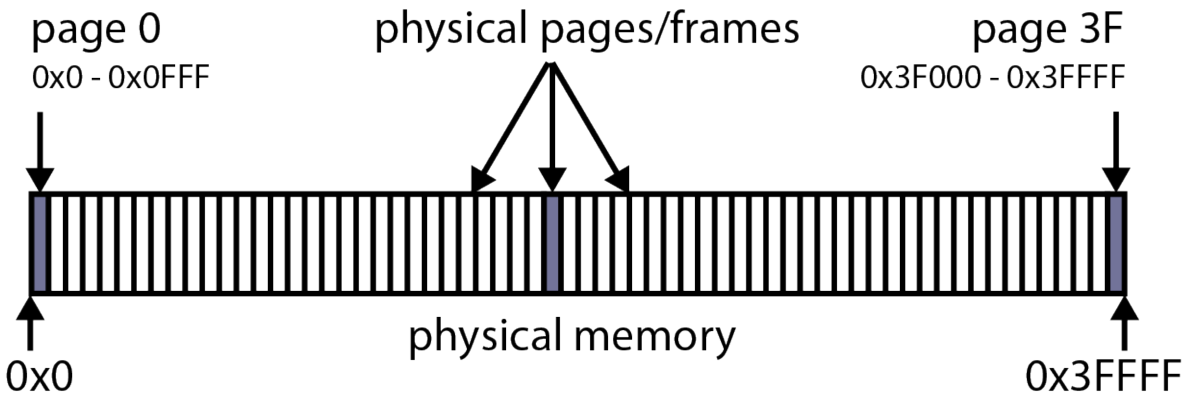
Translating Segmented Virtual Addresses

- **Approach 2:** Maintain a segment table



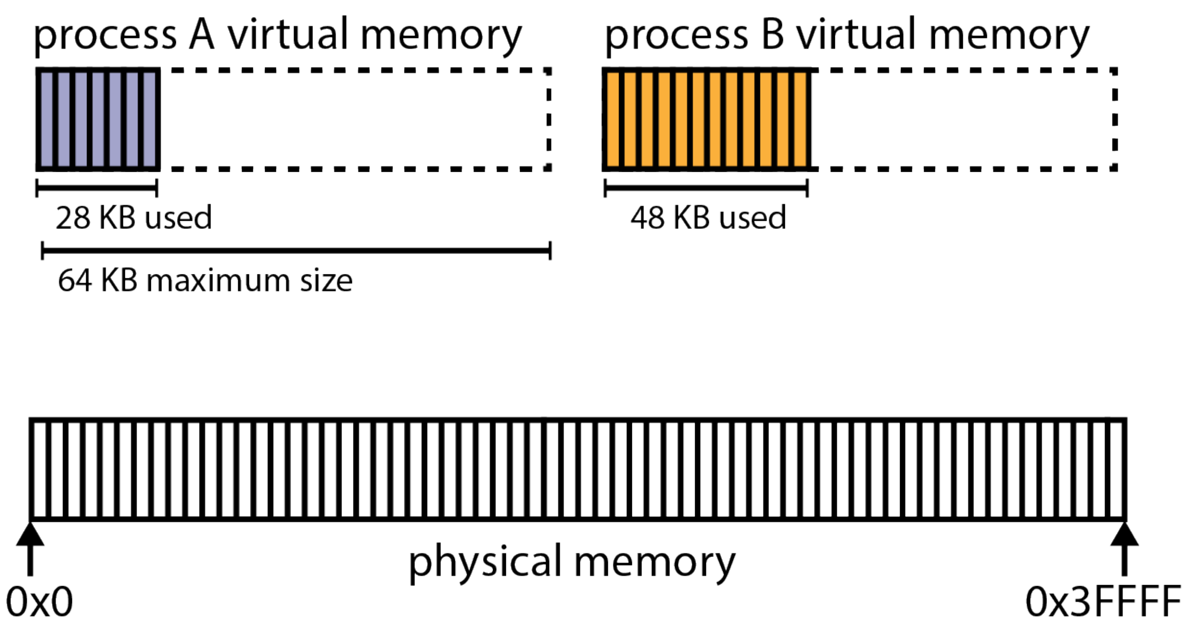
If the segment number in v is greater than the number of segments throw an **exception**. Otherwise, use the segment number to lookup the limit and relocation values from the segment table.

Paging: Physical Memory



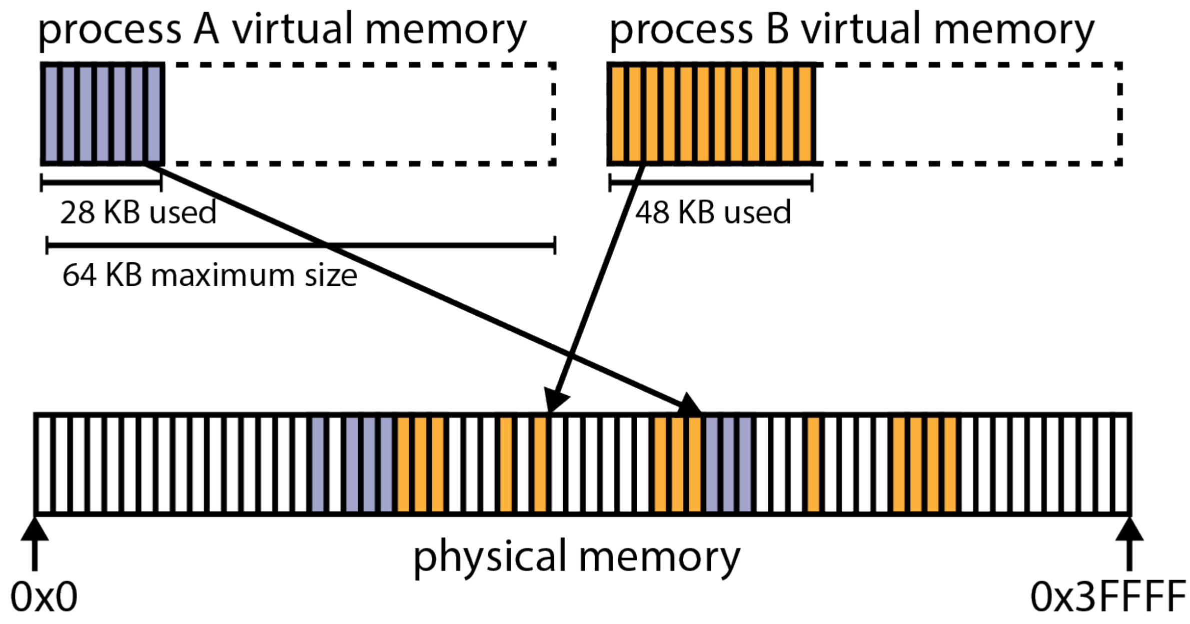
Physical memory is divided into fixed-size chunks called **physical pages**, or, **frames**. Physical addresses, in this example, are 18bits.
Physical page size, in this example, is 4KB.
There are $2^{18}/2^{12} = 2^{6} = 64$ frames of physical memory.

Paging: Virtual Memory



Virtual memory is divided into fixed-size chunks called **pages**.
Page size equals frame size.
Virtual addresses, in this example, are 16bits.
Page size, in this example, is 4KB.
There are $2^{16}/2^{12} = 2^{4} = 16$ pages in virtual memory.

Paging: Address Translation



Each page maps to a different frame. Any page can map to any frame. Pages are mapped to frames using a **page table**.

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

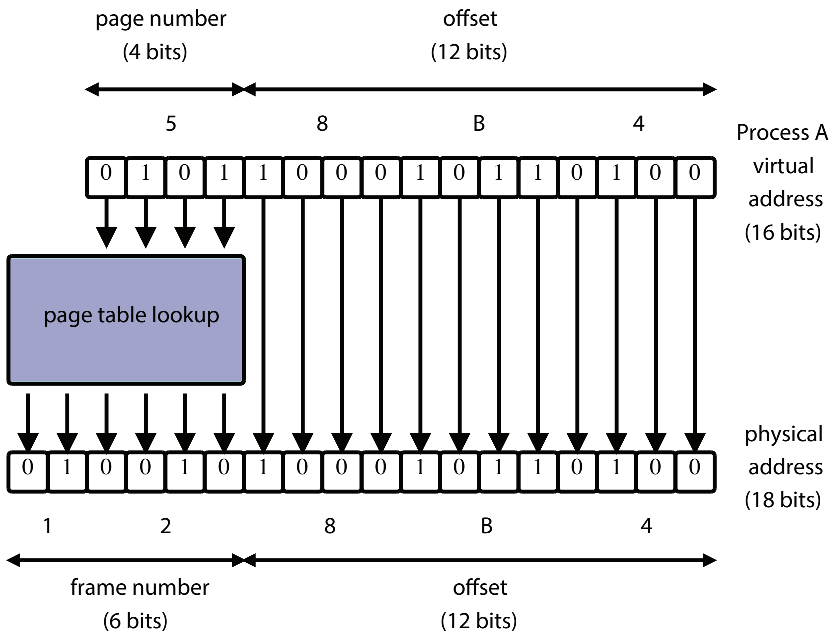
Each row in the page table is a **page table entry (PTE)**.
The table is indexed by **page number**.

The **valid** bit is used to indicate if the PTE is used or not, because not all pages of virtual memory may be used by the address space. If it is 1, the PTE maps a page in the address space to physical memory. If it is 0, the PTE does not correspond to a page in the address space. **Number of PTEs = Maximum Virtual Memory Size / Page Size**

Paging: Address Translation in the MMU

- 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**

Paging: Address Translation Illustrated



Number of Bits for Offset = log(**Page Size**)
Number of PTEs = **Maximum Virtual Memory Size / Page Size**
Number of Bits for Page Number = log(**Number of PTEs**)

Paging: Address Translation Example

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

Translate for Process A and Process B

Virtual Address	Process A	Process B
v = 0x102C	p =	p =
v = 0x9800	p =	p =
v = 0x0024	p =	p =

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)**
  - Recall: **Number of Pages = Maximum Virtual Memory Size / Page Size**
  - **Size of PTE** is typically provided
- The page table a 64KB virtual memory, with 4KB pages, is 64 bytes, assuming 32 **bits** for each PTE

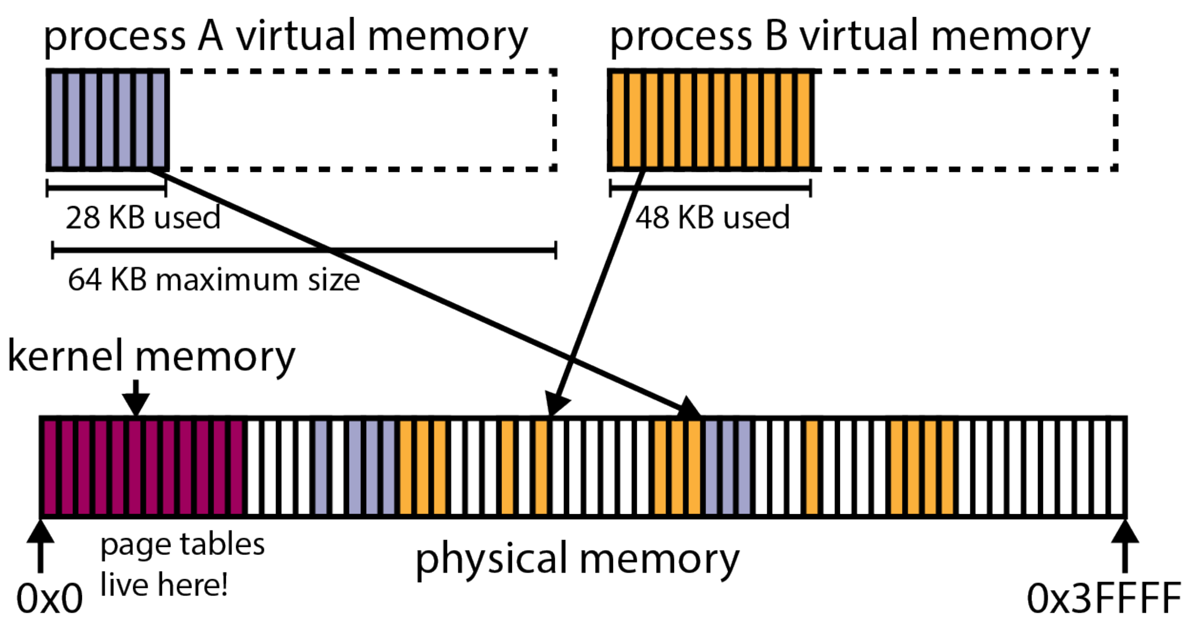
Larger Virtual Memory

If $V = 32$bits, then the maximum virtual memory size is 4GB. Assuming page size is 4KB, and PTE size is 32bits (4bytes), the page table would be 4MB.

If $V = 48$bits, then the page table would be: $2^{48}/2^{12} * 2^{2} = 2^{38}$bytes, or, 256GB.

Page tables can get very, very large.

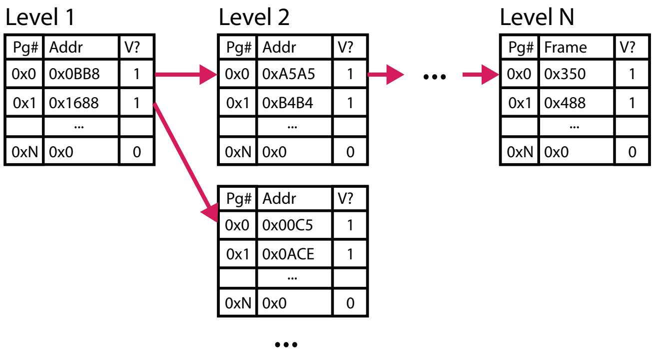
Page Tables: Where?



Page tables are kernel data structures. They live in the kernel's memory. If $P = 48$ and $V = 48$, how many page tables can fit into the kernel's memory?

Shrinking the Page Table: Multi-Level Paging

- Instead of having a single page table to map an entire virtual memory, we can **organize** it and split the page table into multiple **levels**.
  - a large, contiguous table is replaced with multiple smaller tables, each fitting onto a single page
  - **if a table contains no valid PTEs, do not create that table**



The lowest-level page table (N), contains the frame number. All higher level tables contain pointers to tables on the next level.

Two-Level Paging Example (Process A)

Single-Level Paging

Page	Frame	V?
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

Two-Level Paging

Directory

Page	Address	V?
0x0	Table 1	1
0x1	Table 2	1
0x2	NULL	0
0x3	NULL	0

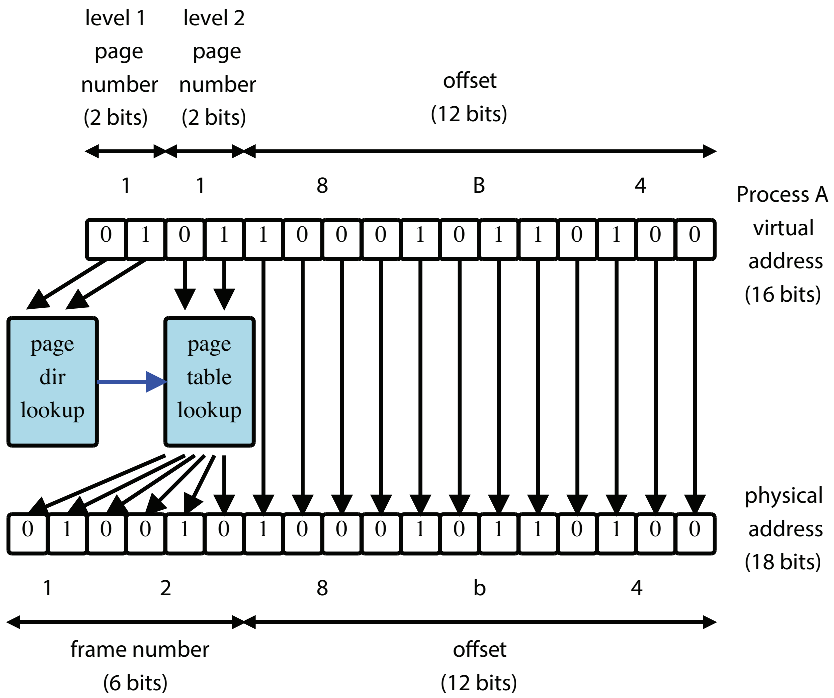
Page Tables (1 and 2)

Page	Frame	V?
0x0	0x0F	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1

Page	Frame	V?
0x0	0x11	1
0x1	0x12	1
0x2	0x13	1
0x3	NULL	0

V? → Valid?
If a PTE is not valid, it does not matter what the frame or address is.

Two-Level Paging: Address Translation



The address translation is the **same** as single-level paging: **Physical Address = Frame Number * Page Size + offset**. The lookup is different.

Multi-Level Paging: Address Translation

- The MMU's **page table base register** points to the page table directory for the current process.
- Each virtual address v has n parts: $(p_1, p_2, \ldots, p_n, o)$
- 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 a pointer to a 3rd level page table
  4. if the entry is not valid, raise an exception
  5. ...
  6. index into the n-th level page table using p_n to find a PTE for the page being accessed
  7. if the PTE is not valid, raise an exception
  8. otherwise, combine the frame number from the PTE with o to determine the physical address (as for single-level paging)

How Many Levels?

- One goal of multi-level paging is to reduce the size of individual page tables.
- Ideally, each table would fit on a single page.
- As V increases, so does the need for more levels.
  - If $V = 40$ (40 bit virtual addresses), page size is 4KB, and, PTE size is 4 bytes.
  - There are $2^{40}/2^{12} = 2^{28}$ pages in virtual memory.
  - $2^{12}/2^{2} = 2^{10}$ PTEs fit on a single page.
  - Need up to $2^{28}/2^{10} = 2^{18}$ page tables, so the directly must hold 2^{18} entries, which requires $2^{18} * 2^{2} = 2^{20}$ or 1MB of space!
- When the number of entries required exceeds a page, add more levels to map larger virtual memories.

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

- Each assembly instruction requires a minimum of one memory operation
  - one to fetch instruction, one or more for instruction operands
- Address translation through a page table adds a **minimum of 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.

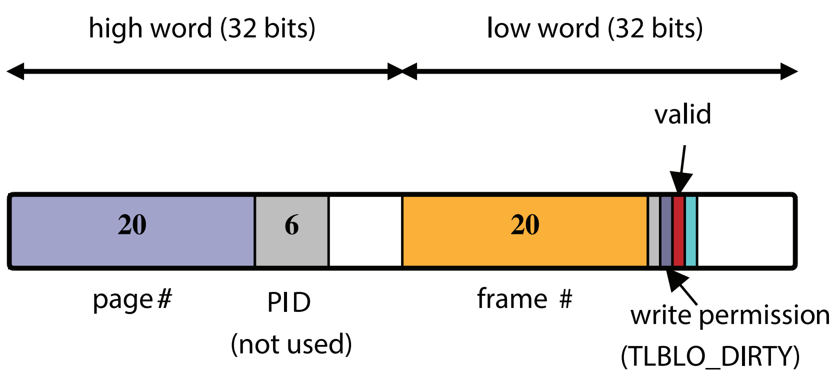
Software-Managed TLBs

- 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`

Paging - Conclusion

- paging does not introduce external fragmentation
- multi-level paging reduces the amount of memory required to store page-to-frame mappings
- TLB misses are increasingly expensive with deeper page tables
  - To translate an address causing A TLB miss for a three-level page table requires **three** memory accesses
    - one for each page table

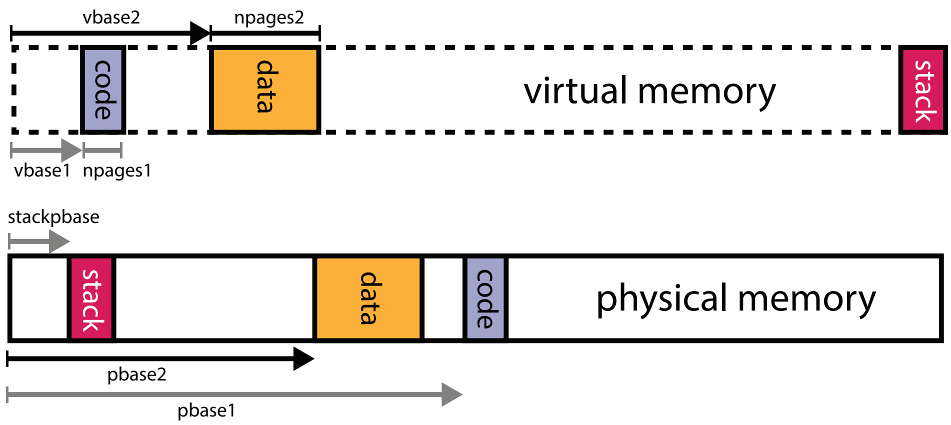
> Paging originates in the late 1950s/early 1960s.
>
> Current Intel CPUs support 4-level paging with 48bit virtual addresses. Support for 5-level paging with 57bit virtual addresses is coming and Linux already supports it.

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
  - Recall: software 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 */
};
```



dumbvm Address Translation

```
vbase1 = as->as_vbase1;
vtop1 = vbase1 + as->as_npages1 * PAGE_SIZE;
vbase2 = as->as_vbase2;
vtop2 = vbase2 + as->as_npages2 * PAGE_SIZE;
stackbase = USERSTACK - DUMBVM_STACKPAGES * PAGE_SIZE;
stacktop = USERSTACK;
if (faultaddress >= vbase1 && faultaddress < vtop1) {
        paddr = (faultaddress - vbase1) + as->as_pbase1;
}
else if (faultaddress >= vbase2 && faultaddress < vtop2) {
        paddr = (faultaddress - vbase2) + as->as_pbase2;
}
else if (faultaddress >= stackbase && faultaddress < stacktop) {
        paddr = (faultaddress - stackbase) + as->as_stackpbase;
}
else {
        return EFAULT;
}
```

USERSTACK = 0x8000 0000, **DUMBVM_STACKPAGES** = 12,
PAGE_SIZE = 4KB.

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**,
- 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
  - The ELF header describes the segment **images**:
    - the virtual address of the start of the segment
    - the length of the segment in the virtual address space
    - the location of the segment in the ELF
    - the length of the segment in the ELF
- the ELF file identifies the (virtual) address of the program's first instruction (the **entry point**)
- 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

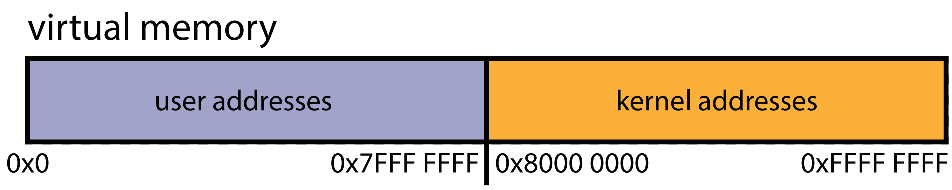
OS/161 ELF Files

- 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 images in the ELF file are an exact copy of the binary data to be stored in the address space
- **BUT** 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`

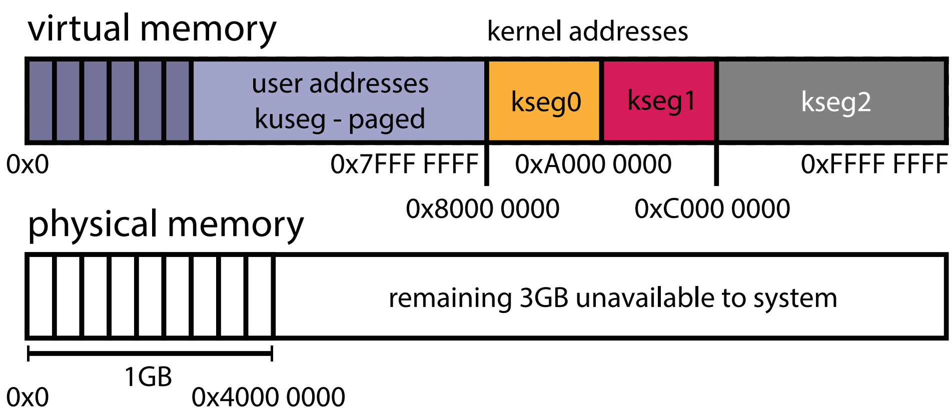
> The image in the ELF may be smaller than the segment it is loaded into in the address space, in which case the rest of the address space segment is expected to be zero-filled.
>
> Look at `kern/syscall/loadelf.c` to see how OS/161 loads segments from ELF files

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.



OS/161 Memory

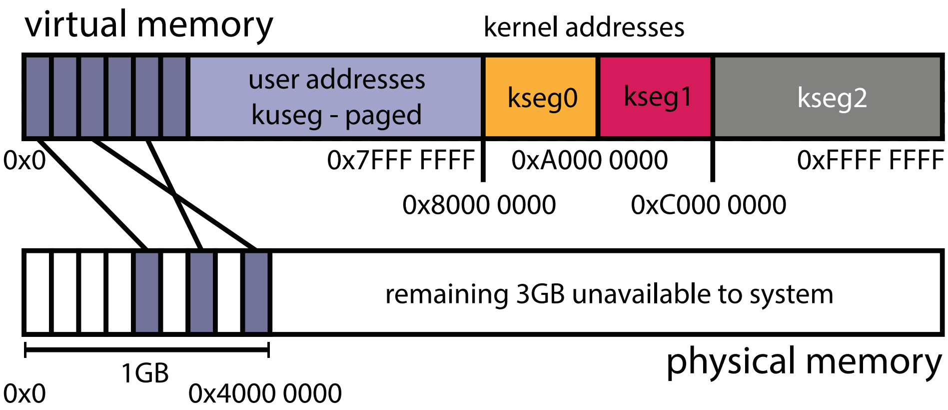


Sys/161 only supports 1GB of physical memory. The remaining 3GB are not available/usable. The kernel's virtual memory is divided into three segments:

- kseg0 - 512MB - for kernel data structures, stacks, etc.
- kseg1 - 512MB - for addressing devices
- kseg2 - 1GB - unused

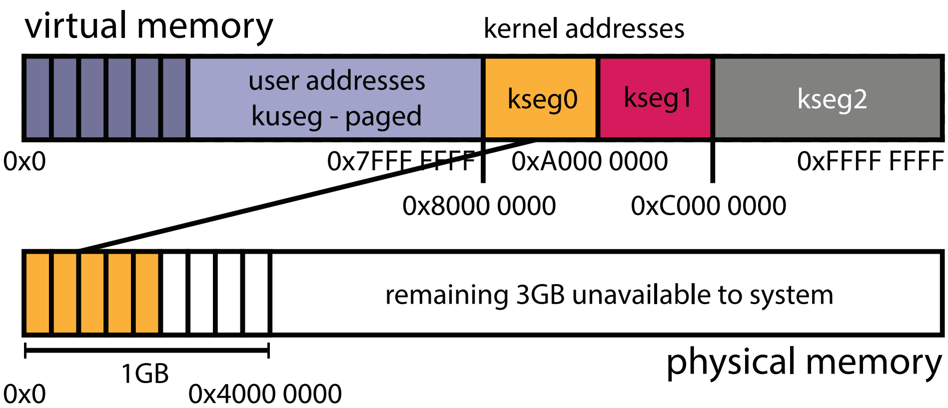
Physical memory is divided into frames. Frame use is managed by the kernel in the **coremap**.

OS/161 Memory



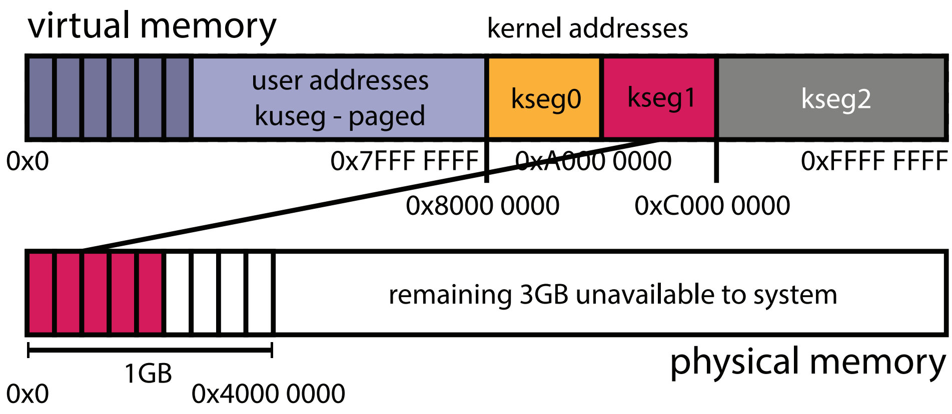
User virtual memory, kuseg, is paged. The kernel maintains the page-to-frame mappings for each process. The TLB is used to translate kuseg virtual addresses to physical ones.

OS/161 Memory



Addresses within kseg0 are for the kernel's data structures, stacks, and code. To translate kseg0 addresses to physical ones **subtract 0x8000 0000** from the virtual address.
kseg0 maps to the first 512MB of physical memory, though it may not use all of this space.
The TLB is **NOT** used.

OS/161 Memory



Addresses within kseg1 are for accessing devices. To translate kseg1 addresses to physical ones **subtract 0xA000 0000** from the virtual address.
kseg1 maps to the first 512MB of physical memory, though it does not use all of this space.
The TLB is **NOT** used.
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 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 (i.e., the TLB should not contain any entries that are not present).
- 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.

Page Faults are Slow

- Accessing secondary storage (**milliseconds**, or, **microseconds** for SSDs) can be orders of magnitude slower than RAM (**nanoseconds**)
  - Suppose that secondary storage access is 1000 times slower than memory access. Then:

Frequency of Fault	Average Memory Access Time w/ Swapping
1 in 10 memory accesses	100 times slower
1 in 100	10 times slower
1 in 1000	2 times slower

- To improve performance of virtual memory with on-demand paging, reduce the occurrence of page faults
  - 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.
- 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
```

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	