

## Virtual and Physical Addresses

- Physical addresses are provided directly by the machine.
  - one physical address space per machine
  - the size of a physical address determines the maximum amount of addressable physical memory
- Virtual addresses (or logical addresses) are addresses provided by the OS to processes.
  - one virtual address space *per process*
- Programs use virtual addresses. As a program runs, the hardware (with help from the operating system) converts each virtual address to a physical address.
- The conversion of a virtual address to a physical address is called *address translation*.

---

---

On the MIPS, virtual addresses and physical addresses are 32 bits long. This limits the size of virtual and physical address spaces.

---

---

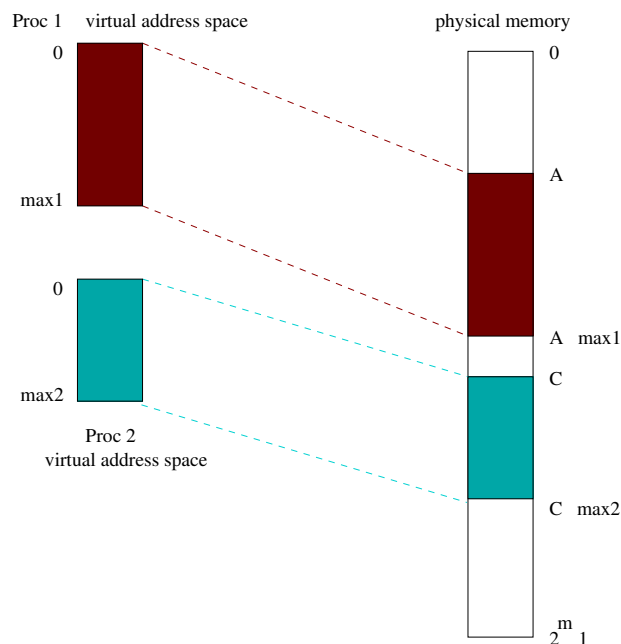
## Simple Address Translation: Dynamic Relocation

- hardware provides a *memory management unit (MMU)* which includes a *relocation register* and a *limit register* (or *bound register*).
- to translate a virtual address to a physical address, the MMU:
  - checks whether the virtual address is larger than the limit in the limit register
  - if it is, the MMU raises an *exception*
  - otherwise, the MMU adds the base address (stored in the relocation register) to the virtual address to produce the physical address
- The OS maintains a separate base address and limit for each process, and ensures that the relocation and limit registers in the MMU always contain the base address and limit of the currently-running process.
- To ensure this, the OS must normally change the values in the MMU's registers during each context switch.

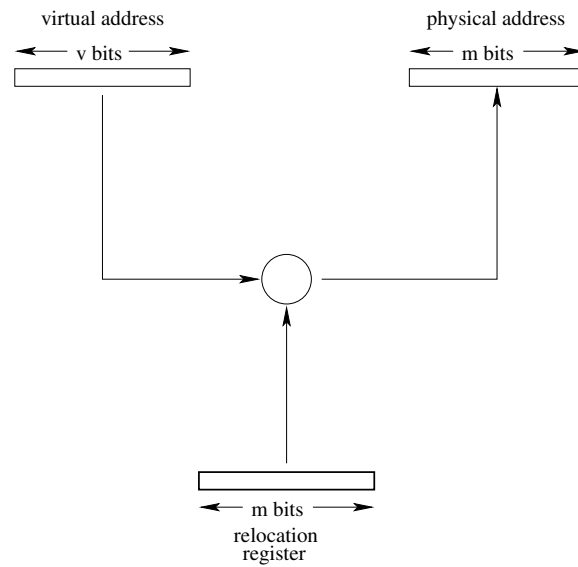
### Properties of Dynamic Relocation

- each virtual address space corresponds to a *contiguous range of physical addresses*
- the OS is responsible for deciding *where* each virtual address space should map to 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 *external fragmentation* of physical memory: wasted, unallocated space
- the MMU is responsible for performing all address translations, using base and limit information provided to it by the the OS

### Dynamic Relocation: Address Space Diagram



### Dynamic Relocation Mechanism

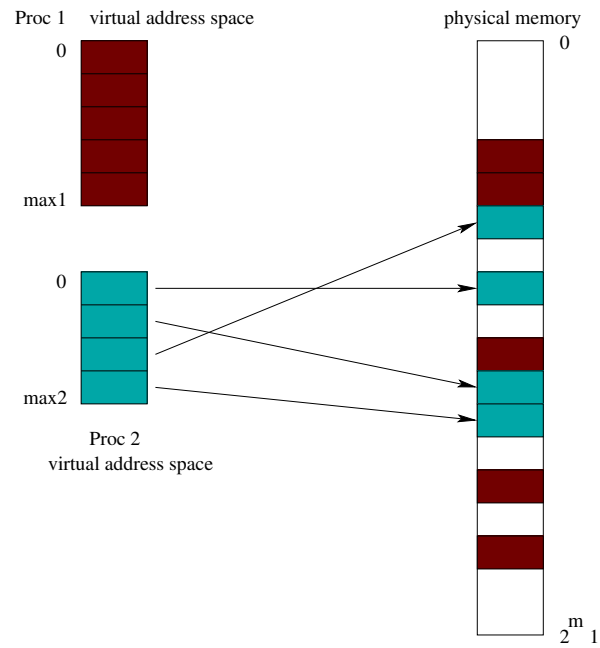


This diagram shows only the address translation, not the limit check.

### Address Translation: Paging

- Each virtual address space is divided into fixed-size chunks called *pages*
- The physical address space is divided into *frames*. Frame size matches page size.
- OS maintains a *page table* for each process. Page table specifies the frame in which each of the process's pages is located.
- At run time, MMU translates virtual addresses to physical using the page table of the running process.

### Address Space Diagram for Paging



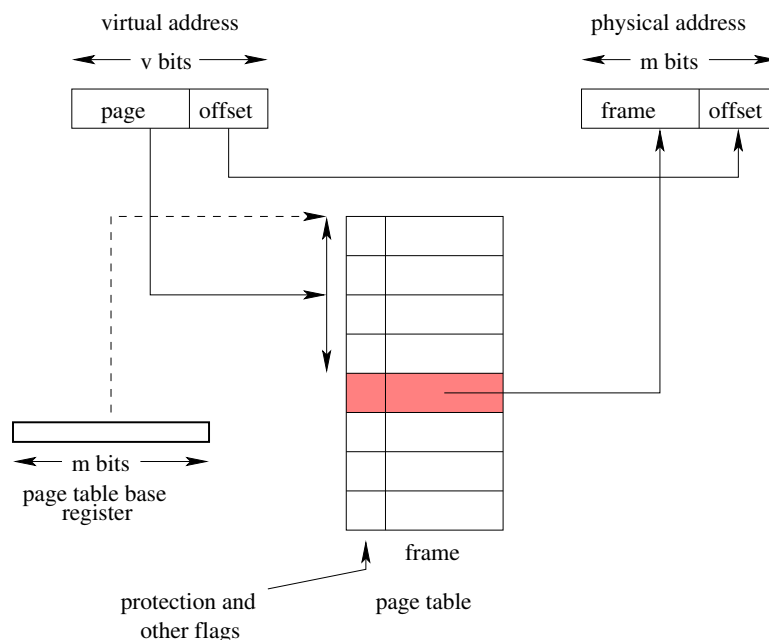
### Properties of Paging

- OS is responsible for deciding which frame will hold each page
  - simple physical memory management
  - potential for *internal fragmentation* of physical memory: wasted, allocated space
  - virtual address space need not be physically contiguous in physical space after translation.
- MMU is responsible for performing all address translations using the page table that is created and maintained by the OS.
- The OS must normally change the values in the MMU registers on each context switch, so that they refer to the page table of the currently-running process.

## How the MMU Translates Virtual Addresses

- The MMU includes a *page table base register* and a *page table length register*.
  - the base register contains the (physical) address of the first page table entry for the currently-running process
  - the length register contains the number of entries in the page table of the currently running process.
- To translate a virtual address, the MMU:
  - determines the *page number* and *offset* of the virtual address
  - checks whether the page number is larger than the value in the page table length register
  - if it is, the MMU raises an exception
  - otherwise, the MMU uses the page table to determine the *frame number* of the frame that holds the virtual page, and combines the frame number and offset to determine the physical address

## Paging Mechanism



## Page Table Entries

- the primary payload of each page table entry (PTE) is a frame number
- PTEs typically contain other information as well, such as
  - information provided by the kernel to control address translation by the MMU, such as:
    - \* valid bit: is the process permitted to use this part of the address space?
    - \* present bit: is this page mapped into physical memory (useful with page replacement, to be discussed later)
    - \* protection bits: to be discussed
  - information provided by the MMU to help the kernel manage address spaces, such as:
    - \* reference (use) bit: has the process used this page recently?
    - \* dirty bit: has the process changed the contents of this page?

## Validity and Protection

- during address translation, the MMU checks that the page being used by the process has a *valid* page table entry
  - typically, each PTE contains a *valid bit*
  - invalid PTEs indicate pages that the process is not permitted to use
- the MMU may also enforce other protection rules, for example
  - each PTE may contain a *read-only* bit that indicates whether the corresponding page is read-only, or can be modified by the process
- if a process attempts to access an invalid page, or violates a protection rule, the MMU raises an exception, which is handled by the kernel

---

---

The kernel controls which pages are valid and which are protected by setting the contents of PTEs and/or MMU registers.

---

---

### Summary: Roles of the Kernel and the MMU

- Kernel:
  - manage MMU state 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

### Speed of Address Translation

- 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
  - Simple address translation through a page table can cut instruction execution rate in half.
  - More complex translation schemes (e.g., multi-level paging) are even more expensive.
- Solution: include a Translation Lookaside Buffer (TLB) in the MMU
  - TLB is a fast, fully associative address translation cache
  - TLB hit avoids page table lookup

## TLB

- Each entry in the TLB contains a (page number, frame number) pair.
- If address translation can be accomplished using a TLB entry, access to the page table is avoided.
  - This is called a *TLB hit*.
- Otherwise, translate through the page table.
  - This is called a *TLB miss*.
- TLB lookup is much faster than a memory access. TLB is an associative memory - page numbers of all entries are checked simultaneously for a match. However, the TLB is typically small (typically hundreds, e.g. 128, or 256 entries).
- If the MMU cannot distinguish TLB entries from different address spaces, then the kernel must clear or invalidate the TLB on each context switch. (Why?)

## TLB Management

- An TLB may be *hardware-controlled* or *software-controlled*
- In a hardware-controlled TLB, when there is a TLB miss:
  - The MMU (hardware) finds the frame number by performing a page table lookup, translates the virtual address, and adds the translation (page number, frame number pair) to the TLB.
  - If the TLB is full, the MMU evicts an entry to make room for the new one.
- In a software-controlled TLB, when there is a TLB miss:
  - the MMU simply causes an exception, which triggers the kernel exception handler to run
  - the kernel must determine the correct page-to-frame mapping and load the mapping into the TLB (evicting an entry if the TLB is full), before returning from the exception
  - after the exception handler runs, the MMU retries the instruction that caused the exception.



## The MIPS R3000 TLB

- The MIPS has a software-controlled TLB that can hold 64 entries.
- Each TLB entry includes a virtual page number, a physical frame number, an address space identifier (not used by OS/161), and several flags (valid, read-only).
- OS/161 provides low-level functions for managing the TLB:
  - TLB\_Write:** modify a specified TLB entry
  - TLB\_Read:** read a specified TLB entry
  - TLB\_Probe:** look for a page number in the TLB
- If the MMU cannot translate a virtual address using the TLB it raises an exception, which must be handled by OS/161.

---



---

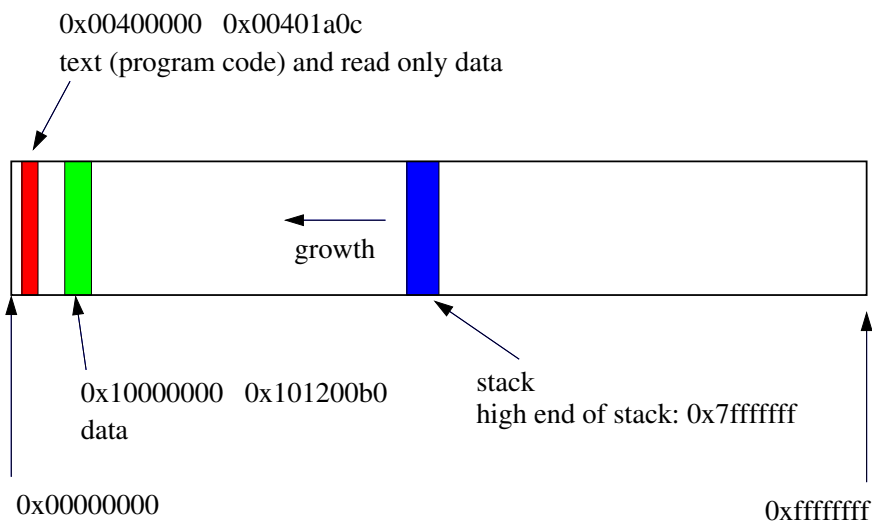
See `kern/arch/mips/include/tlb.h`

---



---

## What is in a Virtual Address Space?




---



---

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

---



---

### Address Translation In OS/161: dumbvm

- OS/161 starts with a very simple virtual memory implementation
- virtual address spaces are described by `addrspace` objects, which record the mappings from virtual to physical addresses

```
struct addrspace {
    #if OPT_DUMBVM
        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_stackbase; /* base physical address of stack */
    #else
        /* Put stuff here for your VM system */
    #endif
};
```

- Notice that each segment must be mapped contiguously into physical memory.

### Address Translation Under dumbvm

- the MIPS MMU tries to translate each virtual address using the entries in the TLB
- If there is no valid entry for the page the MMU is trying to translate, the MMU generates a TLB fault (called an *address exception*)
- The `vm_fault` function (see `kern/arch/mips/vm/dumbvm.c`) handles this exception for the OS/161 kernel. It uses information from the current process' `addrspace` to construct and load a TLB entry for the page.
- On return from exception, the MIPS retries the instruction that caused the exception. This time, it may succeed.

---



---

`vm_fault` is not very sophisticated. If the TLB fills up, OS/161 will crash!

---



---

## 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 specified 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 Sections and Segments

- In the ELF file, a program's code and data are grouped together into *sections*, based on their properties. Some sections:
  - .text:** program code
  - .rodata:** read-only global data
  - .data:** initialized global data
  - .bss:** uninitialized global data (Block Started by Symbol)
  - .sbss:** small uninitialized global data
- not all of these sections are present in every ELF file
- normally
  - the `.text` and `.rodata` sections together form the text segment
  - the `.data`, `.bss` and `.sbss` sections together form the data segment
- space for *local* program variables is allocated on the stack when the program runs

## The `user/uw-testbin/segments.c` Example Program (1 of 2)

```
#include <unistd.h>

#define N    (200)

int x = 0xdeadbeef;
int t1;
int t2;
int t3;
int array[4096];
char const *str = "Hello World\n";
const int z = 0xabcdcdba;

struct example {
    int ypos;
    int xpos;
};
```

### The user/uw-testbin/segments.c Example Program (2 of 2)

```

int
main()
{
    int count = 0;
    const int value = 1;
    t1 = N;
    t2 = 2;
    count = x + t1;
    t2 = z + t2 + value;

    reboot(RB_POWEROFF);
    return 0; /* avoid compiler warnings */
}

```

### ELF Sections for the Example Program

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	Flg
[ 0]		NULL	00000000	000000	000000	
[ 1]	.text	PROGBITS	00400000	010000	000200	AX
[ 2]	.rodata	PROGBITS	00400200	010200	000020	A
[ 3]	.reginfo	MIPS_REGINFO	00400220	010220	000018	A
[ 4]	.data	PROGBITS	10000000	020000	000010	WA
[ 5]	.sbss	NOBITS	10000010	020010	000014	WAp
[ 6]	.bss	NOBITS	10000030	020010	004000	WA

...

Flags: W (write), A (alloc), X (execute), p (processor specific)

## Size = number of bytes (e.g., .text is 0x200 = 512 bytes)

## Off = offset into the ELF file

## Addr = virtual address

---

The cs350-readelf program can be used to inspect OS/161 MIPS ELF files: `cs350-readelf -a segments`

---

## ELF Segments for the Example Program

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
REGINFO	0x010220	0x00400220	0x00400220	0x00018	0x00018	R	0x4
LOAD	0x010000	0x00400000	0x00400000	0x00238	0x00238	R E	0x10000
LOAD	0x020000	0x10000000	0x10000000	0x00010	0x04030	RW	0x10000

- segment info, like section info, can be inspected using the `cs350-readelf` program
- the REGINFO section is not used
- the first LOAD segment includes the `.text` and `.rodata` sections
- the second LOAD segment includes `.data`, `.sbss`, and `.bss`

## Contents of the Example Program's `.text` Section

Contents of section `.text`:

```

400000 3c1c1001 279c8000 2408fff8 03a8e824 <...'...$......$
...
## Decoding 3c1c1001 to determine instruction
## 0x3c1c1001 = binary 11110000011100000100000000000001
## 0011 1100 0001 1100 0001 0000 0000 0001
## instr | rs    | rt    | immediate
## 6 bits | 5 bits| 5 bits| 16 bits
## 001111 | 00000 | 11100 | 0001 0000 0000 0001
## LUI    | 0     | reg 28| 0x1001
## LUI    | unused| reg 28| 0x1001
## Load upper immediate into rt (register target)
## lui gp, 0x1001

```

---

The `cs350-objdump` program can be used to inspect OS/161 MIPS ELF file section contents: `cs350-objdump -s segments`

---

### Contents of the Example Program's `.rodata` Section

Contents of section `.rodata`:

```

400200 abcddcba 00000000 00000000 00000000 .....
400210 48656c6c 6f20576f 726c640a 00000000 Hello World.....
...
## const int z = 0xabcddcba
## If compiler doesn't prevent z from being written,
## then the hardware could.
## 0x48 = 'H' 0x65 = 'e' 0x0a = '\n' 0x00 = '\0'

```

---

The `.rodata` section contains the “Hello World” string literal and the constant integer variable `z`.

---

### Contents of the Example Program's `.data` Section

Contents of section `.data`:

```

10000000 deadbeef 00400210 00000000 00000000 .....@.....
...
## Size = 0x10 bytes = 16 bytes (padding for alignment)
## int x = deadbeef (4 bytes)
## char const *str = "Hello World\n"; (4 bytes)
## address of str = 0x10000004
## value stored in str = 0x00400210.
## NOTE: this is the address of the start
## of the string literal in the .rodata section

```

---

The `.data` section contains the initialized global variables `str` and `x`.

---



## Contents of the Example Program's `.bss` and `.sbss` Sections

```
...
10000000 D x
10000004 D str
10000010 S t3      ## S indicates sbss section
10000014 S t2
10000018 S t1
1000001c S errno
10000020 S __argv
10000030 B array   ## B indicates bss section
10004030 A _end
10008000 A _gp
```

---

---

The `t1`, `t2`, and `t3` variables are in the `.sbss` section. The `array` variable is in the `.bss` section. There are no values for these variables in the ELF file, as they are uninitialized. The `cs350-nm` program can be used to inspect symbols defined in ELF files: `cs350-nm -n <filename>`, in this case `cs350-nm -n segments`.

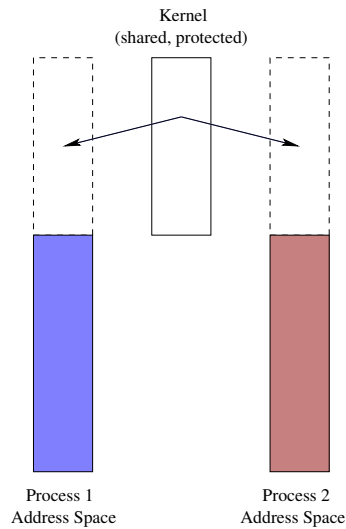
---

---

## An Address Space for the Kernel

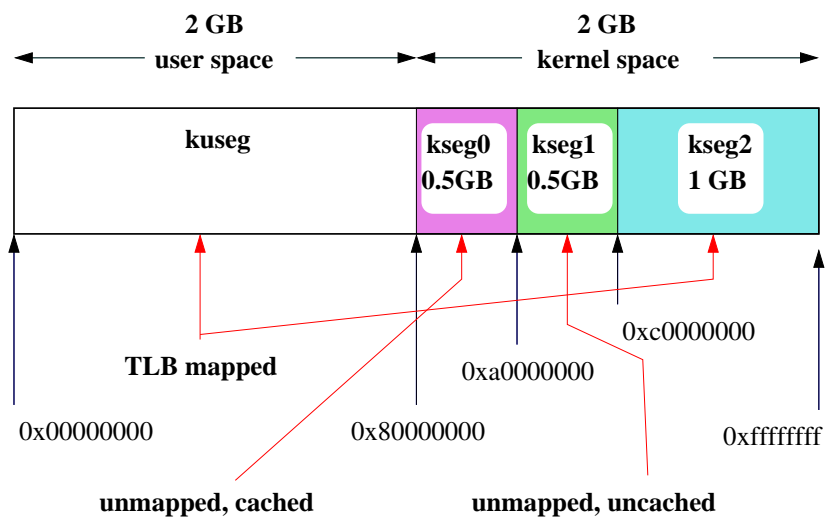
- Each process has its own address space. What about the kernel?
- Three possibilities:
  - Kernel in physical space:** disable address translation in privileged system execution mode, enable it in unprivileged mode
  - Kernel in separate virtual address space:** need a way to change address translation (e.g., switch page tables) when moving between privileged and unprivileged code
  - Kernel mapped into portion of address space of *every process*:** OS/161, Linux, and other operating systems use this approach
    - memory protection mechanism is used to isolate the kernel from applications
    - one advantage of this approach: application virtual addresses (e.g., system call parameters) are easy for the kernel to use

## The Kernel in Process' Address Spaces



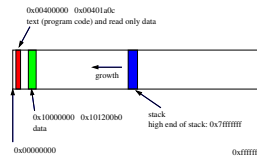
Attempts to access kernel code/data in user mode result in memory protection exceptions, not invalid address exceptions.

## Address Translation 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.

## The Problem of Sparse Address Spaces



- Consider the page table for `user/testbin/sort`, assuming a 4 Kbyte page:
  - need  $2^{19}$  page table entries (PTEs) to cover the bottom half of the virtual address space (2GB).
  - the text segment occupies 2 pages, the data segment occupies 289 pages, and OS/161 sets the initial stack size to 12 pages, so there are only 303 *valid* pages (of  $2^{19}$ ).
- If dynamic relocation is used, the kernel will need to map a 2GB address space contiguously into physical memory, even though only a tiny fraction of that address space is actually used by the program.
- If paging is used, the kernel will need to create a page table with  $2^{19}$  PTEs, almost all of which are marked as not valid.

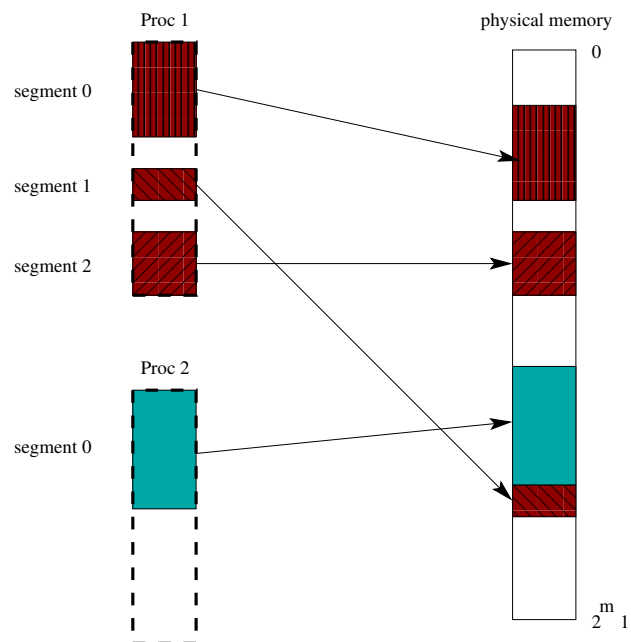
## Handling Sparse Address Spaces

- Use dynamic relocation, but provide separate base and length for each valid segment of the address space. Do not map the rest of the address space.
  - OS/161 `dumbvm` uses a simple variant of this idea, which depends on having a software-managed TLB.
  - A more general approach is *segmentation*.
- A second approach is to use *multi-level paging*
  - replace the single large linear page table with a hierarchy of smaller page tables
  - a sparse address space can be mapped by a sparse tree hierarchy
  - easier to manage several smaller page tables than one large one (remember: each page table must be contiguous in physical memory!)

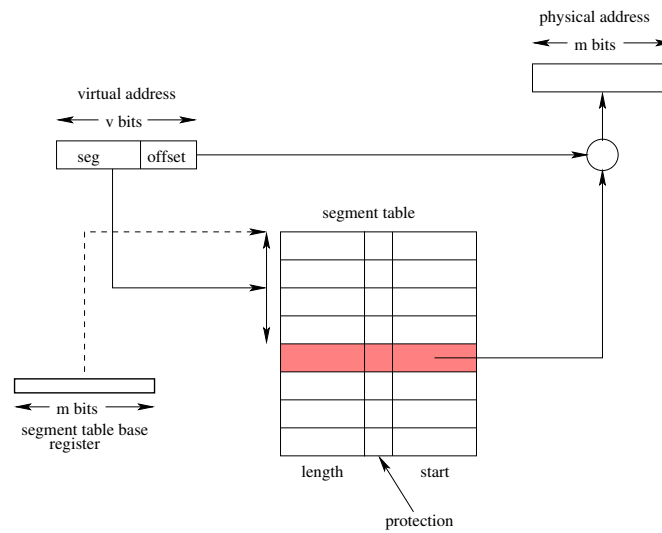
## Segmentation

- Often, programs (like `sort`) need several virtual address segments, e.g, for code, data, and stack.
- With segmentation, a virtual address can be thought of as having two parts:  
(segment ID, address within segment)
- Each segment also has a length.

## Segmented Address Space Diagram



## Mechanism for Translating Segmented Addresses



This translation mechanism requires physically contiguous allocation of segments.

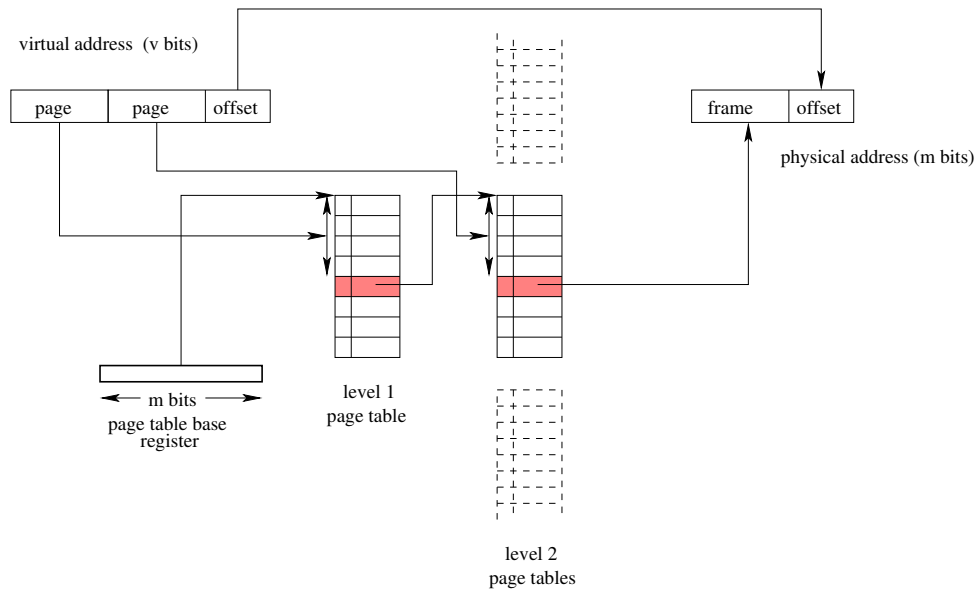
## Handling Sparse Paged Virtual Address Spaces

- Large paged virtual address spaces require large page tables.
- example:  $2^{48}$  byte virtual address space, 8 Kbyte ( $2^{13}$  byte) pages, 4 byte page table entries means

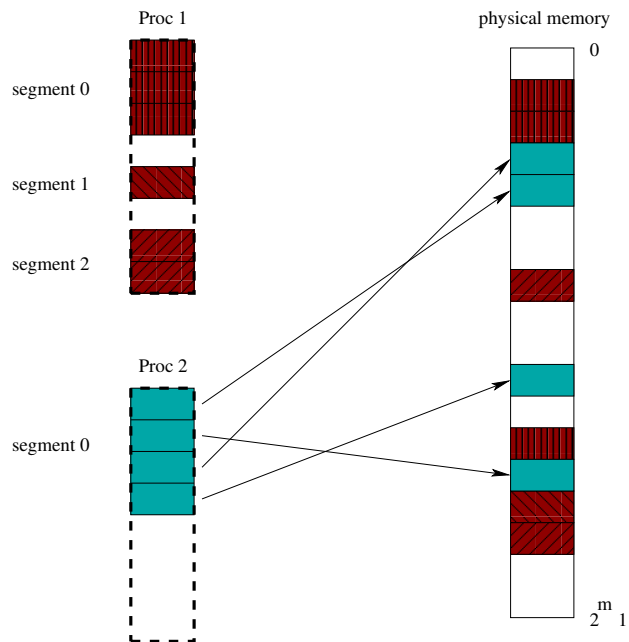
$$\frac{2^{48}}{2^{13}} \cdot 2^2 = 2^{37} \text{ bytes per page table}$$

- page tables for large address spaces may be very large, and
  - they must be in memory, and
  - they must be physically contiguous

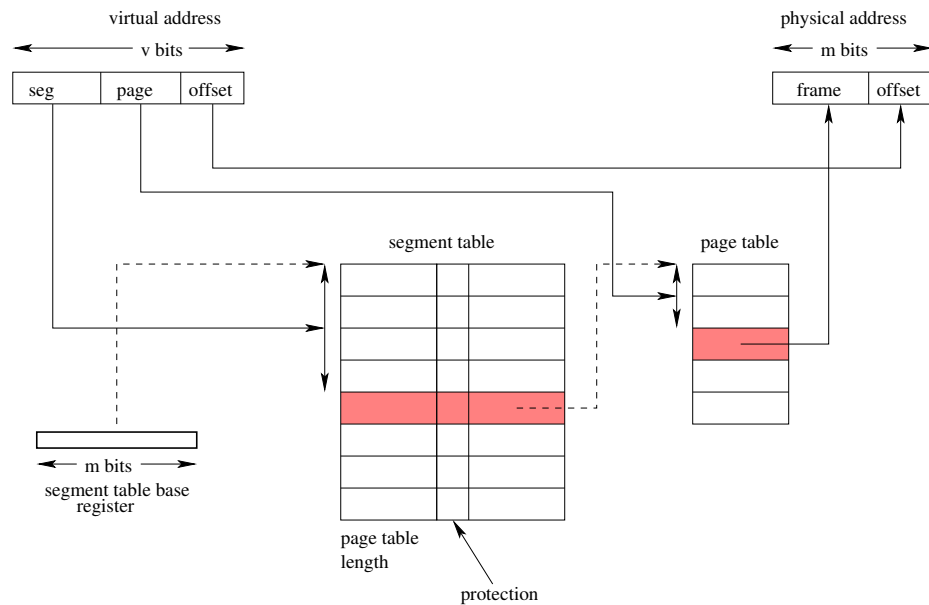
### Two-Level Paging



### Combining Segmentation and Paging



## Combining Segmentation and Paging: Translation Mechanism



## 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 (or segments) from the virtual address space to be stored in secondary storage, e.g., on disks, as well as primary memory.
- Move pages (or segments) between secondary storage and primary memory so that they are in primary memory when they are needed.

## Paging Policies

### When to Page?:

*Demand paging* brings pages into memory when they are used. Alternatively, the OS can attempt to guess which pages will be used, and *prefetch* them.

### What to Replace?:

Unless there are unused frames, one page must be replaced for each page that is loaded into memory. A *replacement policy* specifies how to determine which page to replace.

---

---

Similar issues arise if (pure) segmentation is used, only the unit of data transfer is segments rather than pages. Since segments may vary in size, segmentation also requires a *placement policy*, which specifies where, in memory, a newly-fetched segment should be placed.

---

---

## Page Faults

- When paging is used, some valid pages may be loaded into memory, and some may not be.
- To account for this, each PTE may contain a *present* bit, to indicate whether the page is or is not loaded into memory
  - $V = 1, P = 1$ : page is valid and in memory (no exception occurs)
  - $V = 1, P = 0$ : page is valid, but is not in memory (exception!)
  - $V = 0, P = x$ : invalid page (exception!)
- If  $V = 0$ , or if  $V = 1$  and  $P = 0$ , the MMU will generate an exception if a process tries to access the page. This is called a *page fault*.
- To handle a page fault, the kernel operating system must:
  - bring the missing page into memory, set  $P = 1$  in the PTE
  - while the missing page is being loaded, the faultin process is *blocked*
  - return from the exception
- the processor will then retry the instrution that caused the page fault



### Page Faults in OS/161

- things are a bit different in systems with software-managed TLBs, such as OS/161 on the MIPS processor
- MMUs with software-managed TLBs never check page tables, and thus do not interpret  $P$  bits in page table entries
- In an MMU with a software-managed TLB, either there is a valid translation for a page in the TLB, or there is not.
  - If there is not, the MMU generates an exception. It is up to the kernel to determine the reason for the exception. Is this:
    - \* an access to a valid page that is not in memory (a page fault)?
    - \* an access to a valid page that is in memory?
    - \* an access to an invalid page?
  - The kernel should ensure that a page has a translation in the TLB *only* if the page is valid and in memory. (Why?)

### A Simple Replacement Policy: FIFO

- 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.
- The OPT policy: 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	

- OPT requires knowledge of the future.

### Other Replacement Policies

- FIFO is simple, but it does not consider:
  - Frequency of Use:** how often a page has been used?
  - Recency of Use:** when was a page last used?
  - Cleanliness:** has the page been changed while it is in memory?
- The *principle of locality* suggests that usage ought to be considered in a replacement decision.
- Cleanliness may be worth considering for performance reasons.

### Locality

- Locality is a property of the page reference string. In other words, it is a property of programs themselves.
- *Temporal locality* says that pages that have been used recently are likely to be used again.
- *Spatial locality* says that pages “close” to those that have been used are likely to be used next.

---

---

In practice, page reference strings exhibit strong locality. Why?

---

---

### Least Recently Used (LRU) Page Replacement

- LRU is based on the principle of temporal locality: replace the page that has not been used for the longest time
- To implement LRU, it is necessary to track each page’s recency of use. For example: maintain a list of in-memory pages, and move a page to the front of the list when it is used.
- Although LRU and variants have many applications, true LRU is difficult to implement in virtual memory systems. (Why?)

### Least Recently Used: LRU

- 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

### The “Use” Bit

- A *use bit* (or *reference bit*) is a bit found in each page table entry that:
  - 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 operating system
- The use bit provides a small amount of efficiently-maintainable usage information that can be exploited by a page replacement algorithm.

### 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
```

### Page Cleanliness: the “Modified” Bit

- A page is *modified* (sometimes called dirty) if it has been changed since it was loaded into memory.
- A modified page is more costly to replace than a clean page. (Why?)
- The MMU identifies modified pages by setting a *modified bit* in page table entry of a page when a process *writes* to a virtual address on that page, i.e., when the page is changed.
- The operating system can clear the modified bit when it cleans the page
- The modified bit potentially has two roles:
  - Indicates which pages need to be cleaned.
  - Can be used to influence the replacement policy.

### How Much Physical Memory Does a Process Need?

- Principle of locality suggests that some portions of the process's virtual address space are more likely to be referenced than others.
- A refinement of this principle is the *working set model* of process reference behaviour.
- According to the working set model, at any given time some portion of a program's address space will be heavily used and the remainder will not be. The heavily used portion of the address space is called the *working set* of the process.
- The working set of a process may change over time.
- The *resident set* of a process is the set of pages that are located in memory.

---



---

According to the working set model, if a process's resident set includes its working set, it will rarely page fault.

---



---

### Resident Set Sizes (Example)

PID	VSZ	RSS	COMMAND
805	13940	5956	/usr/bin/gnome-session
831	2620	848	/usr/bin/ssh-agent
834	7936	5832	/usr/lib/gconf2/gconfd-2 11
838	6964	2292	gnome-smproxy
840	14720	5008	gnome-settings-daemon
848	8412	3888	sawfish
851	34980	7544	nautilus
853	19804	14208	gnome-panel
857	9656	2672	gpilotd
867	4608	1252	gnome-name-service

## Thrashing and Load Control

- What is a good multiprogramming level?
  - If too low: resources are idle
  - If too high: too few resources per process
- A system that is spending too much time paging is said to be *thrashing*. Thrashing occurs when there are too many processes competing for the available memory.
- Thrashing can be cured by load shedding, e.g.,
  - Killing processes (not nice)
  - Suspending and *swapping out* processes (nicer)

## Swapping Out Processes

- Swapping a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process. Suspending a process ensures that it is not runnable while it is swapped out.
- Which process(es) to suspend?
  - low priority processes
  - blocked processes
  - large processes (lots of space freed) or small processes (easier to reload)
- There must also be a policy for making suspended processes ready when system load has decreased.