CS 350: Operating Systems

Charles Shen

Fall 2016, University of Waterloo

Notes written from Gregor Richards's lectures.

Feel free to email feedback to me at ccshen902@gmail.com.

Contents

1 Introduction

There are three views of an operating system:

1. **Application View** (Section 1.1): what service does it provide?
2. **System View** (Section 1.2): what problems does it solve?
3. **Implementation View** (Section 1.3): how is it built?

An operating system is part cop, part facilitator.

kernel: The operating system kernel is the part of the operating system that responds to system calls, interrupts and exception.

operating system (OS): The operating system as a whole includes the kernel, and may include other related programs that provide services for application such as utility programs, command interpreters, and programming libraries.

1.1 Application View of an Operating System

The OS provides an execution environment for running programs.

- The execution environment provides a program with the processor time and memory space that it needs to run.
- The execution environment provides interfaces through which a program can use networks, storage, I/O devices, and other system hardware components.
  Interfaces provide a simplified, abstract view of hardware to application programs.
- The execution environment isolates running programs from one another and prevents undesirable interactions among them.

1.2 System View of an Operating System

The OS manages the hardware resources of a computer system.

- Resources include processors, memory, disks and other storage devices, network interfaces, I/O devices such as keyboards, mice and monitors, and so on.

- The operating system allocates resources among running programs. It controls the sharing of resources among programs.
- The OS itself also uses resources, which it must share with application programs.

1.3 Implementation View of an Operating System

The OS is a concurrent, real-time program.

- Concurrency arises naturally in an OS when it supports concurrent applications, and because it must interact directly with the hardware.
- Hardware interactions also impose timing constraints.

1.4 Operating System Abstractions

The execution environment provided by the OS includes a variety of abstract entities that can be manipulated by a running program.
Examples:

- **files and file systems**: abstract view of secondary storage
- **address spaces**: abstract view of primary memory
- **processes, threads**: abstract view of program execution
- **sockets, pipes**: abstract view of network or other message channels

2 Threads and Concurrency

Threads provide a way for programmers to express *concurrency* in a program.
A normal *sequential program* consists of a single thread of execution.
In threaded concurrent programs, there are multiple threads of executions that are all occurring at the same time.

2.1 OS/161's Thread Interface

Create a new thread:

```
int thread_fork(
  const char *name,              // name of new thread
  struct proc *proc,             // thread's process
  void (*func)                   // new thread's function
    (void *, unsigned long),
  void *datat1,                  // function's first param
  unsigned long data2            // function's second param
);
```

Terminating the calling thread:

```
void thread_exit(void);
```

Voluntarily yield execution:

```
void thread_yield(void);
```

3 Processes and System Calls

4 Assignment 2A Review

5 Virtual Memory

5.1 Physical Memory and Addresses

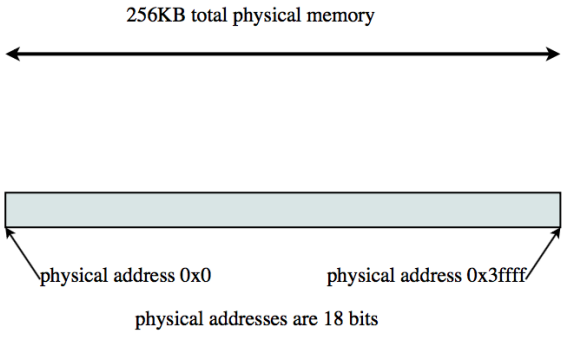


Figure 1: An example physical memory, $P = 18$

If physical addresses have P **bits**, the maximum amount of addressable physical memory is 2^P **bytes** (assuming a byte-addressable 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 256TB of physical memory to be addressed

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

5.2 Virtual Memory and 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, $V = 32$

Running applications see only virtual addresses, e.g.,

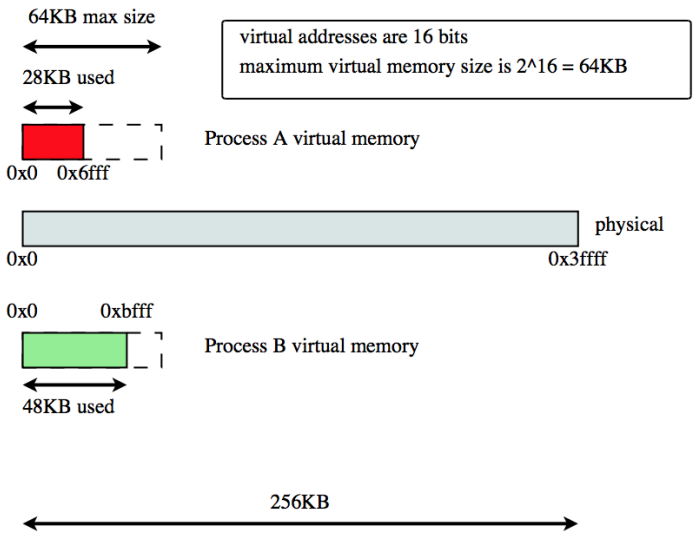


Figure 2: An example of virtual memory, $V = 16$

- 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.

5.3 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.

5.4 Address Translation for Dynamic Relocation

CPU includes a **memory management unit** (MMU), with a **relocation register** and a **limit register**.

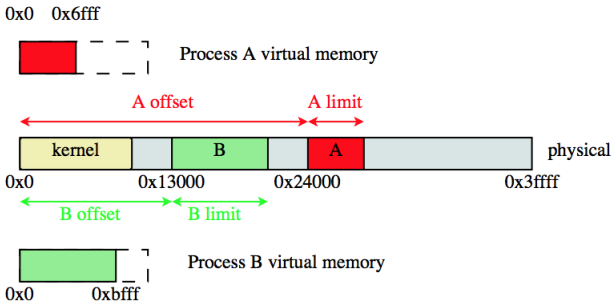


Figure 3: Example of dynamic relocation

- 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 ≥ L then generate exception
else
    p ← 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.

Example. $v =$ `0x102c` $p =$ `0x102c` + `0x24000` = `0x2502c`
$v =$ `0x8800` $p =$ `exception` since `0x8800` $\geq$ `0x7000`

5.5 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 part 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
- hence creates potential for **fragmentation** of physical memory

5.6 Paging: Physical Memory

Physical memory is divided into fixed-size chunks called **frames** or **physical pages**.

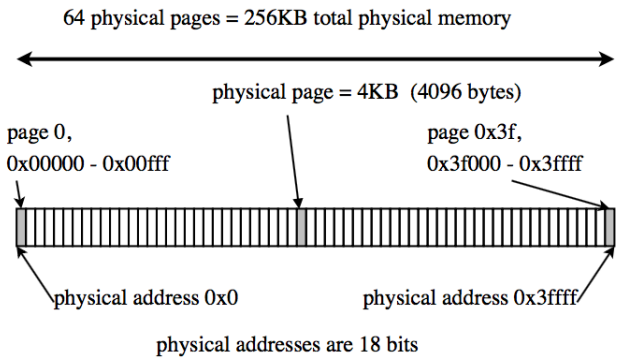


Figure 4: The frame size if 2^{12} bytes (4KB)

5.7 Paging: Virtual Memory

Virtual memories are divided into fixed-size chunks called **pages**.
Page size is equal to frame size.

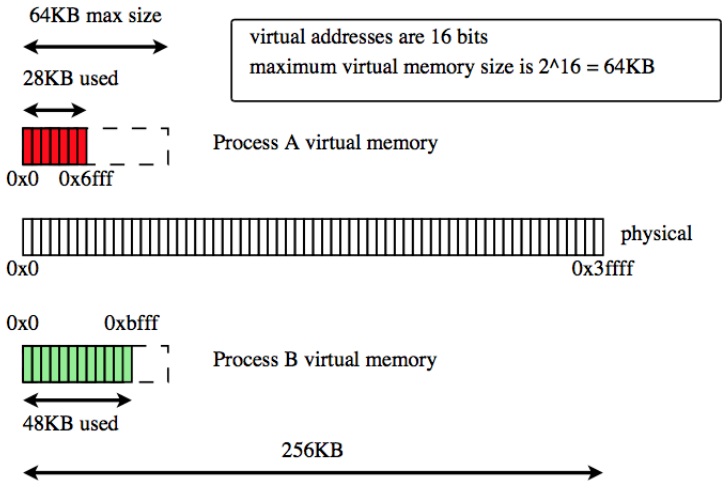
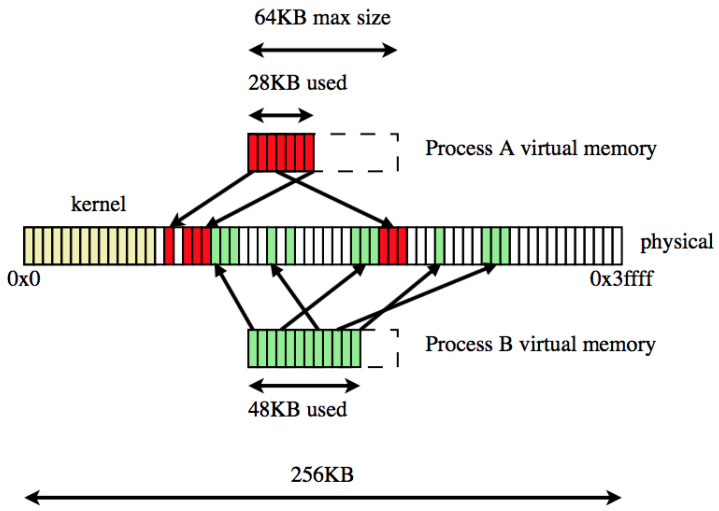


Figure 5: Page size is 4KB here

5.8 Paging: Address Translation



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

5.9 Paging: Address Translation

The MMU includes a **page table base register** which points to the page table for the current process.

How the MMU translate 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 $\cdot$ frame size) + offset

5.10 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 translate 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

5.11 Page Tables: How Big?

A page table has one PTE for each page in the virtual memory

- page table size = number of pages $\cdot$ size of PTE
- number of pages = $\frac{\text{virtual memory size}}{\text{page size}}$

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.

5.12 Page Tables: Where?

Page tables are kernel data structures, i.e. page tables for all processes are in the kernel's stack.

5.13 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

5.14 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 for address translation in the MMU
- Each TLB entry stores a (page number $\Rightarrow$ $framenumber$) mapping

5.15 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!

This 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

5.16 Software-Managed TLBs

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 of 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.

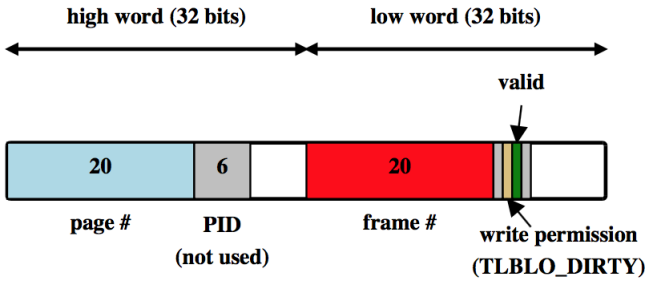


Figure 6: The MIPS R3000 TLB

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

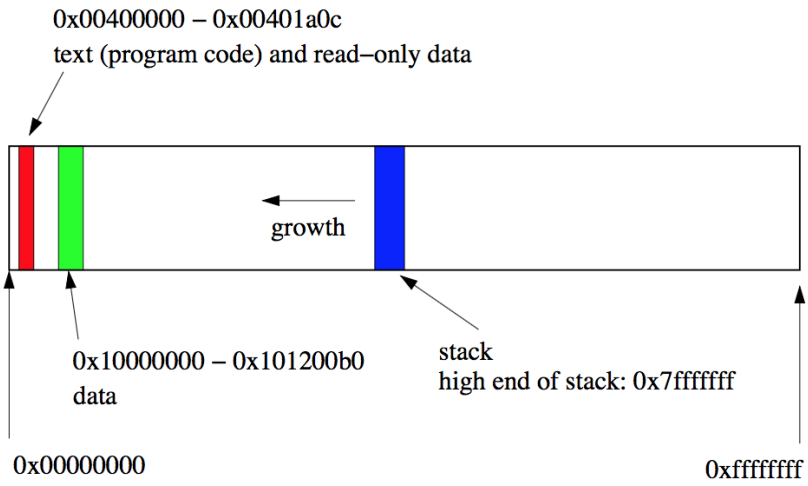


Figure 7: A more realistic virtual memory; this is a layout of the virtual address space for `user/testbin/sort` in OS/161

5.17 Large, Sparse Virtual Memories

Virtual memory may be large,

MIPS: $V = 32$, max virtual memory is 2^{32} bytes (4 GB)

x86-64: $V = 48$, max virtual memory is 2^{48} bytes (246 TB)

Much of the virtual memory may be unused (see Figure 7)!

Application may use *discontinuous segments* of the virtual memory.
One reason is that we have to give room for the stack to grow in the virtual memory!

5.18 Limitations of Simple Address Translation Approaches

A kernel that used simple dynamic relocation would have to allocate 2 GB of contiguous physical memory for `testbin/sort`'s virtual memory, even though `sort` only uses about 1.2 MB.

A kernel that used simple paging would require a page table with 2^{20} PTEs (assuming page size is 4 KB) to map `testbin/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

5.19 Segmentation

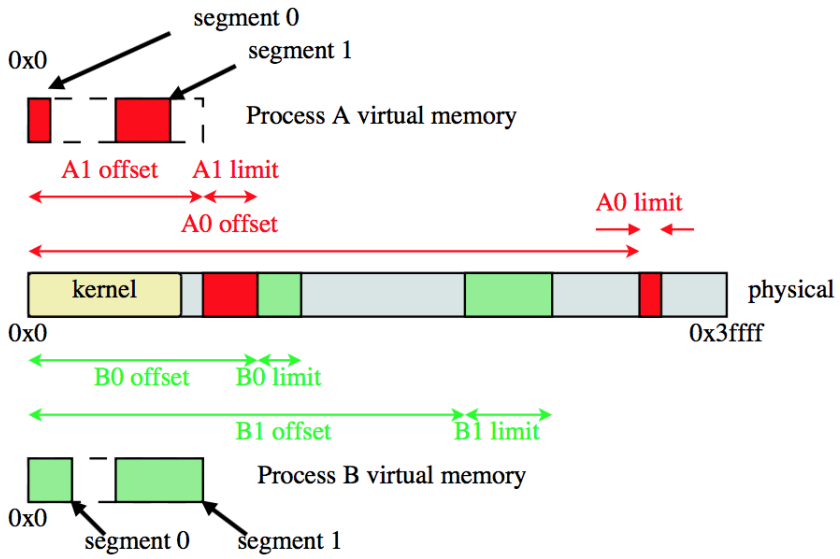


Figure 8: Example of segment address space

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,

- The MMU has multiple offset and limit registers, one pair for each segment

With segmentation, a virtual address can be thought of as having two parts: segment ID and 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!

6 Scheduling

7 Devices and Device Management

8 File Systems

9 Interprocess Communications and Networking

Indices