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

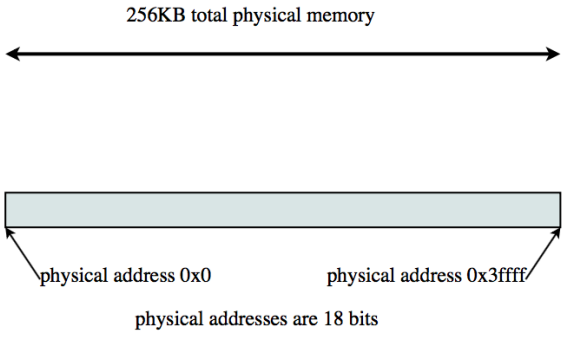


Figure 1: An example physical memory, $P = 18$

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.,

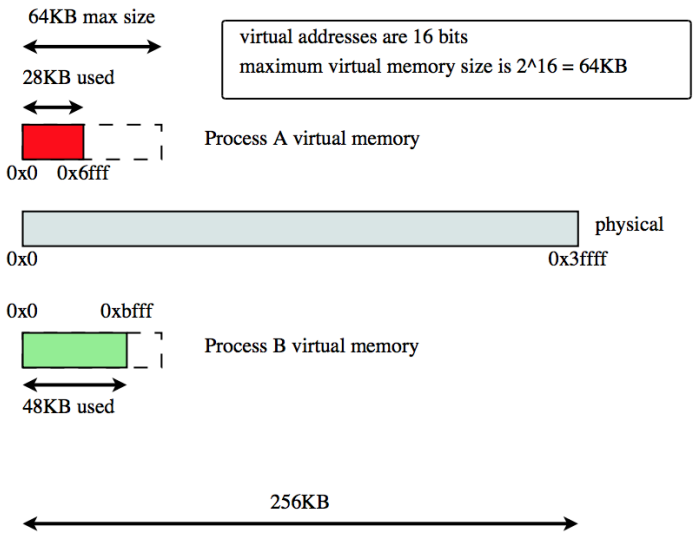


Figure 2: An example of virtual memory, $V = 16$

- 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*.

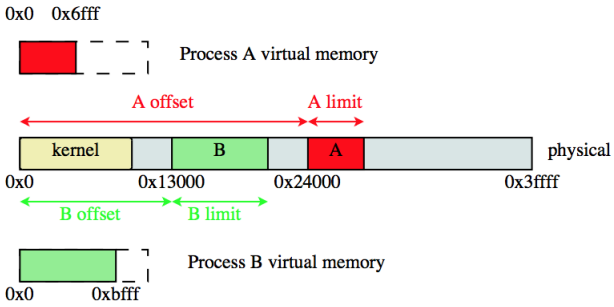


Figure 3: Example of dynamic relocation

- 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*.

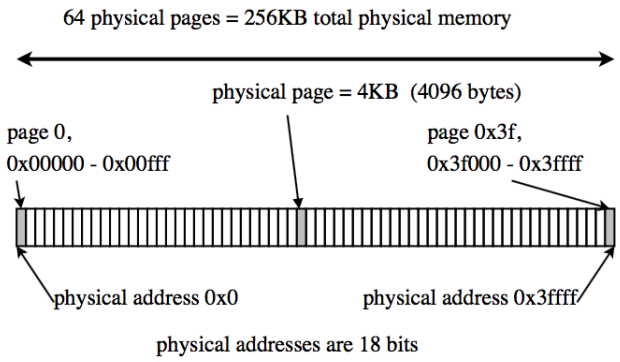


Figure 4: The frame size if 2^{12} bytes (4KB)

5.7 Paging: Virtual Memory

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

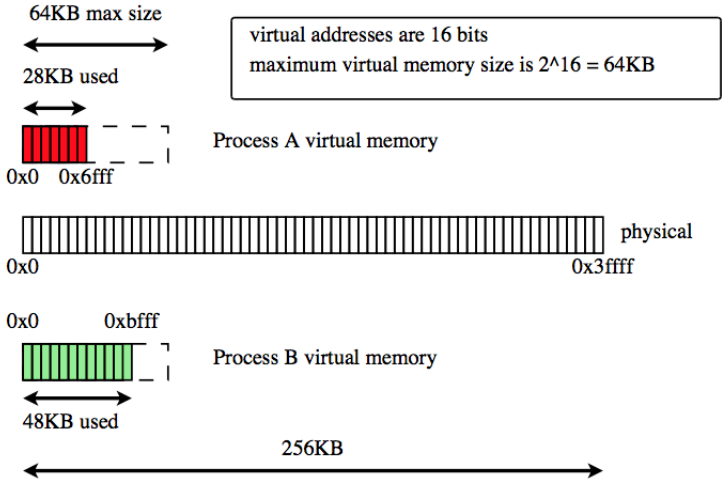
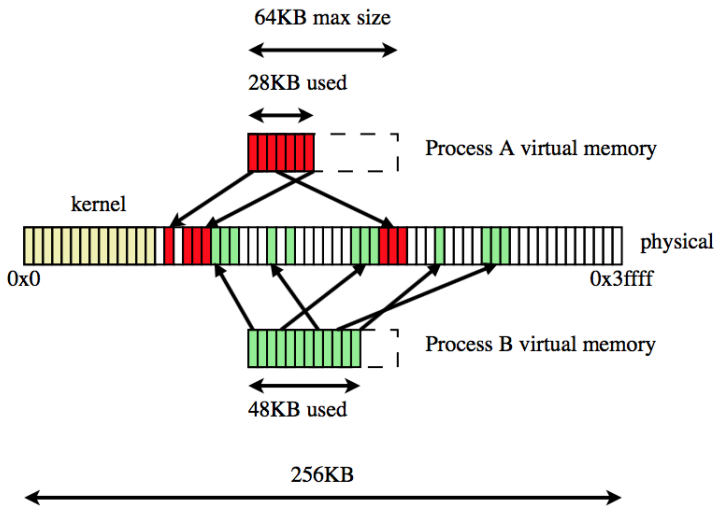


Figure 5: Page size is 4KB here

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

6 Scheduling

7 Devices and Device Management

8 File Systems

9 Interprocess Communications and Networking

Indices