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);
```

See `kern/include/thread.h`

2.2 Why Threads?

Reason 1: parallelism exposed by threads enables parallel execution if the underlying hardware supports it.
Programs can run faster!

Reason 2: parallelism exposed by threads enable better processor utilization.
If one thread has to *block*, another may be able to run.
Concurrent Program Execution (Two Threads)
Conceptually, each thread executes sequentially using its private register contents and stack.

2.3 Some Reviews

The Fetch/Execute Cycle

1. fetch instruction PC points to
2. decode and execute instruction
3. advance PC

Table 1: MIPS Registers

num	name	use	num	name	use
0	z0	always zero	24-25	t8-t9	temps (caller-save)
1	at	assembler reserved	26-27	k0-k1	kernel temps
2	v0	return val/syscall #	28	gp	global pointer
3	v1	return value	29	sp	stack pointer
4-7	a0-a3	subroutine args	30	s8/fp	frame ptr (callee-save)
8-15	t0-t7	temps (caller-save)	31	ra	return addr (for jal)
16-23	s0-s7	saved (calle-save)			

2.4 Implementing Concurrent Threads

Option 1: multiple processors, multiple cores, hardware multithreading per core

- P processors, C cores per processor, M multithreading degree per core $\Rightarrow PCM$ threads can execute simultaneously
- separate register set for each running thread, to hold its execution context

Option 2: *timesharing*

- multiple threads take turns on the same hardware
- rapidly switch from thread to thread so that all make progress

In practice, both techniques can be combined!

2.5 Timesharing and Context Switches

When timesharing, the switch from one thread to another is called a *context switch*.

What happens during a context switch:

1. decide which thread will run next (scheduling)
2. save register contents of current thread
3. load register contents of next thread

Thread context must be saved/restored carefully, since thread execution continuously changes the context!

Context Switch on the MIPS, see `kern/arch/mips/thread/switch.S`

```
switchframe_switch:
  /* a0: address of switchframe pointer of old thread. */
  /* a1: address of switchframe pointer of new thread. */
  /* Allocate stack space for saving 10 registers. 10*4 = 40 */
  addi sp, sp, -40

  sw   ra, 36(sp)  /* Save the registers */
  sw   gp, 32(sp)
  sw   s8, 28(sp)
  sw   s6, 24(sp)
  sw   s5, 20(sp)
  sw   s4, 16(sp)
  sw   s3, 12(sp)
  sw   s2, 8(sp)
  sw   s1, 4(sp)
  sw   s0, 0(sp)

  /* Store the old stack pointer in the old thread */
  sw   sp, 0(a0)

  /* Get the new stack pointer from the new thread */
  lw   sp, 0(a1)
  nop             /* delay slot for load */
```

```
lw   s0, 0(sp)  /* Now, restore the registers */
lw   s1, 4(sp)
lw   s2, 8(sp)
lw   s3, 12(sp)
lw   s4, 16(sp)
lw   s5, 20(sp)
lw   s6, 24(sp)
lw   s8, 28(sp)
lw   gp, 32(sp)
lw   ra, 36(sp)
nop             /* delay slot for load */

/* and return. */
j ra
addi sp, sp, 40 /* in delay slot */
.end switchframe_switch
```

2.6 What Causes a Context Switch?

The running thread calls `thread_yield`, running thread *voluntarily* allows other threads to run.
So we have the following stack (in growth order) after voluntary context switch:

- thread_yield() stack frame
- thread_switch() stack frame
- saved thread context (switchframe)

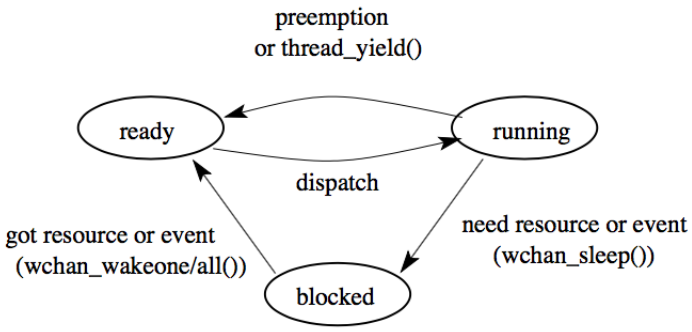
The running thread calls `thread_exit`, running thread is terminated.

The running thread *blocks*, via a call to `wchan_sleep`.

The running thread is *preempted*, running thread *involuntarily* stops running.
So we have the following stack (in growth order) after preemption:

- trap frame
- interrupt handling stack frame(s)
- thread_yield() stack frame
- thread_switch() stack frame

- saved thread context (switchframe)



running: currently executing

ready: ready to execute

blocked: waiting for something, so not ready to execute.

Figure 1: Thread States

2.7 Preemption

Without preemption, a running thread could potentially run forever, without yielding, blocking, or exiting.
Preemption means forcing a running thread to stop running, so that another thread can have a chance.
To implement preemption, the thread library must have a means of "getting control" (causing thread library code to be executed) even though the running thread has not called a thread library function.
This is normally accomplished using *interrupts*.

2.8 Review on Interrupts

An interrupt is an event that occurs during execution of a program.
Interrupts are caused by system devices (hardware), e.g., a timer, a disk controller, a network interface.

When an interrupt occurs, the hardware automatically transfer control to a fixed location in memory. At that memory location, the thread library places a procedure called an *interrupt handler*.
The interrupt handler normally:

1. create a *trap frame* to record thread context at the time of the interrupt
2. determines which device caused the interrupt and performs device-specific processing
3. restores the saved thread context from the trap frame and resumes executions of the thread

2.9 Preemptive Scheduling

A preemptive scheduler imposes a limit, called the *scheduling quantum* on how long a thread can run before being preempted.
The quantum is an *upper bound* on the amount of time that a thread can run. It may block or yield before its quantum has expired.
Periodic timer interrupts allow running time to be tracked.
If a thread has run too long, the timer interrupt handler preempts the thread by calling `thread_yield`.
The preempted thread changes state from running to ready, and it is placed on the *ready queue*.

OS/161 threads use *preemptive round-robin scheduling*.

2.10 Two-Thread Example

Thread 1 is running, thread two had previously yielded voluntarily.
Thread 1: program stack frame(s).
Thread 2: program stack frame(s), thread_yield, thread_switch, switch frame.

A time interrupt occurs. Interrupt handler runs.
Thread 1: program stack frame(s), trap frame, interrupt handler.
Thread 2: program stack frame(s), thread_yield, thread_switch, switch frame.

Interrupt handler decides THread 1 quantum has expired.
Thread 1: program stack frame(s), trap frame, interrupt handler, thread_yield.
Thread 2: program stack frame(s), thread_yield, thread_switch, switch frame.

Scheduler chooses Thread 2 to run. Context switch.
Thread 1: program stack frame(s), trap frame, interrupt handler, thread_yield, thread_switch, switch frame.
Thread 2: program stack frame(s), thread_yield, thread_switch, switch frame.

Thread 2 context is restored.
Thread 1: program stack frame(s), trap frame, interrupt handler, thread_yield, thread_switch, switch frame.
Thread 2: program stack frame(s), thread_yield.

`thread_yield` finishes, Thread 2 program resumes.
Thread 1: program stack frame(s), trap frame, interrupt handler, thread_yield, thread_switch, switch frame.
Thread 2: program stack frame(s).

Later, Thread 2 yields again. Scheduler chooses Thread 1.
Thread 1: program stack frame(s), trap frame, interrupt handler, thread_yield, thread_switch, switch frame.
Thread 2: program stack frame(s), thread_yield, thread_switch, switch frame.

Thread 1 context is restored, interrupt handler resumes.
Thread 1: program stack frame(s), trap frame, interrupt handler.
Thread 2: program stack frame(s), thread_yield, thread_switch, switch frame.

Interrupt handler restores state from trap frame and returns.
Thread 1: program stack frame(s).
Thread 2: program stack frame(s), thread_yield, thread_switch, switch frame.

3 Synchronization

3.1 Thread Synchronization

All threads in a concurrent program *share access* to the program's global variables and the heap.
The part of a concurrent program in which a shared object is accessed is called a *critical section.*

`volatile` keyword: Without it, the compiler could optimize the code on the

variable.
`volatile` forces the compiler to load and store the value on every use. Otherwise, we may have a variable that is loaded before a loop and stored after the loop terminates which may not be what we want/expect.

3.2 Mutual Exclusion

To prevent race conditions, we can enforce *mutual exclusion* on critical sections in the code.

A *race condition* is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

3.2.1 Enforcing Mutual Exclusions With Locks

Acquire/Release must ensure that only one thread at a time can hold the lock, even if both attempt to Acquire at the same time.
If a thread cannot Acquire the lock immediately, it must wait until the lock is available.

3.3 Hardware-Specific Synchronization Instructions

Used to implement synchronization primitives like locks.

4 Processes and System Calls

5 Assignment 2A Review

6 Virtual Memory

6.1 Physical Memory and Addresses

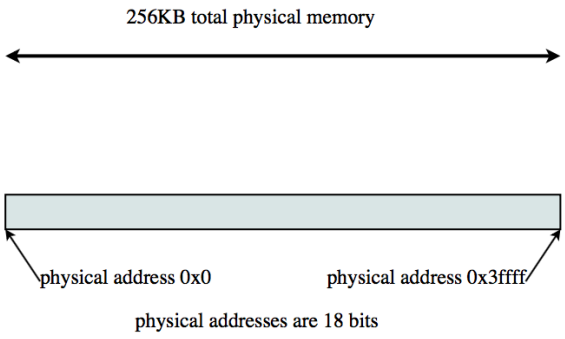


Figure 2: An example physical memory, $P = 18$

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.

6.2 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.,

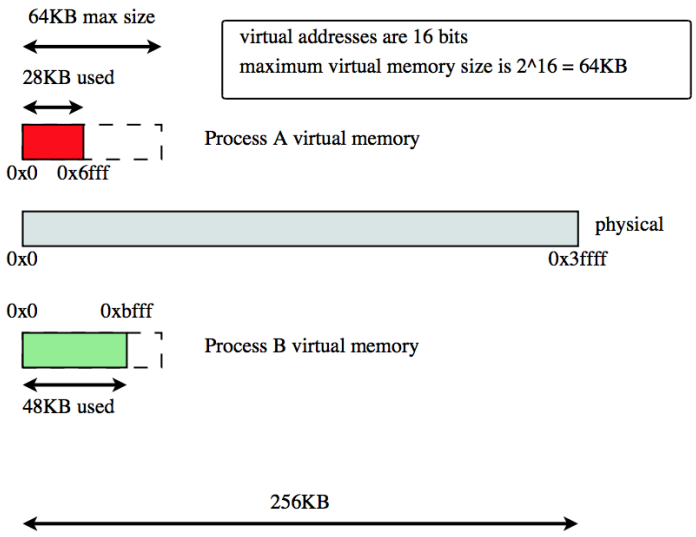


Figure 3: An example of virtual memory, $V = 16$

- 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.

6.3 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.

6.4 Address Translation for Dynamic Relocation

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

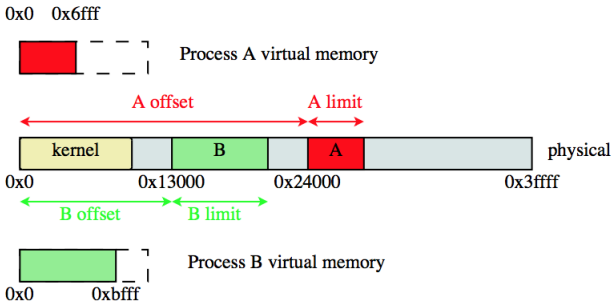


Figure 4: Example of dynamic relocation

- 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`

6.5 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

6.6 Paging: Physical Memory

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

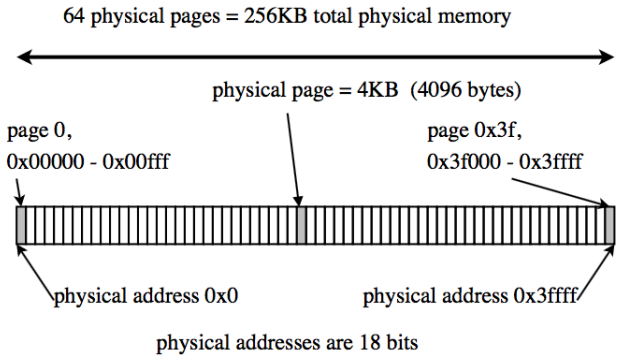


Figure 5: The frame size if 2^{12} bytes (4KB)

6.7 Paging: Virtual Memory

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

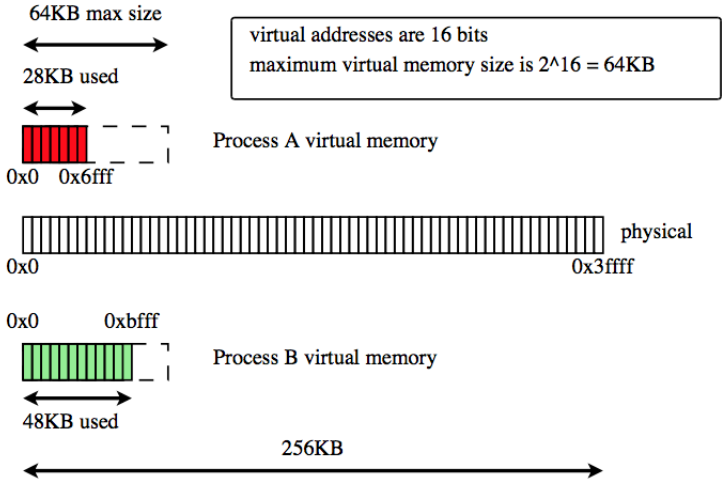
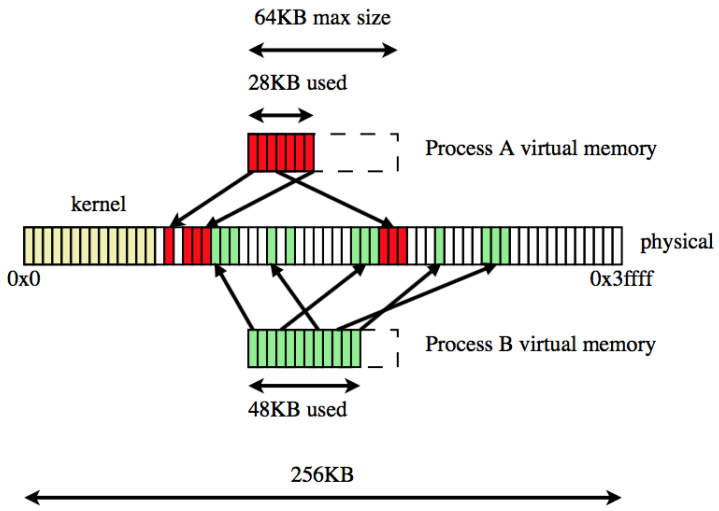


Figure 6: Page size is 4KB here

6.8 Paging: Address Translation



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

6.9 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

6.10 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

6.11 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.

6.12 Page Tables: Where?

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

6.13 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

6.14 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

6.15 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

6.16 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.

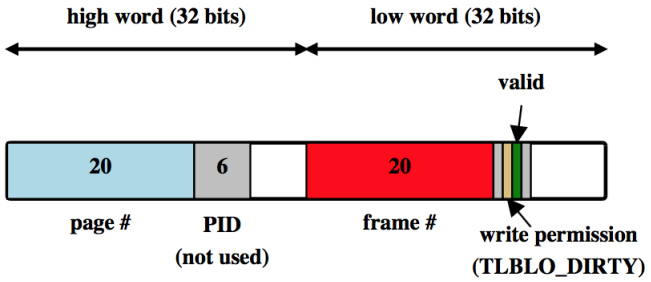


Figure 7: The MIPS R3000 TLB

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

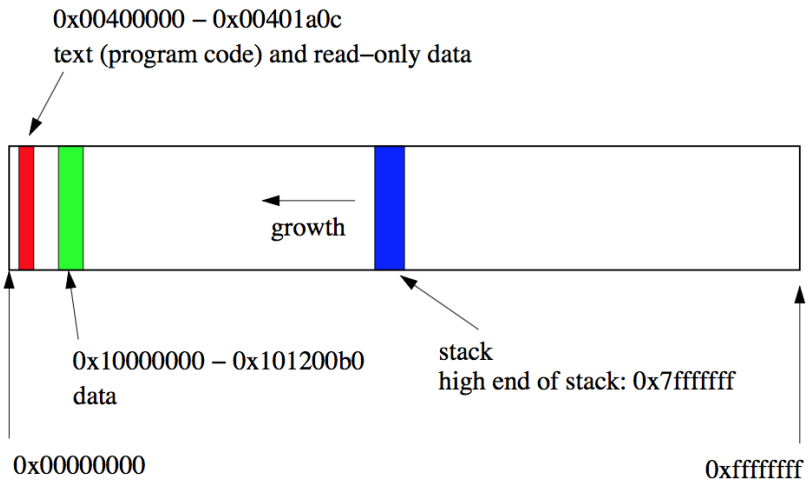


Figure 8: A more realistic virtual memory; this is a layout of the virtual address space for `user/testbin/sort` in OS/161

6.17 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 8)!

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!

6.18 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

6.19 Segmentation

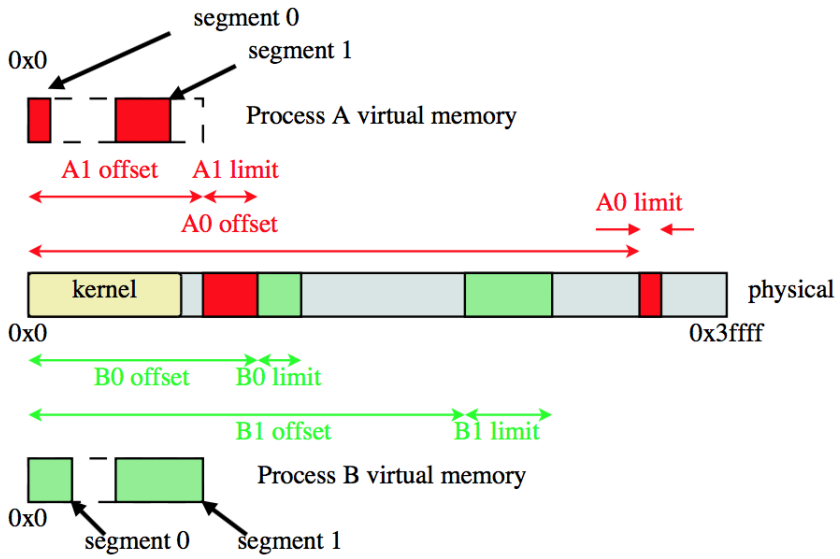


Figure 9: Example of segment address space

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!

6.20 Translating Segmented Virtual Addresses

The MMU needs a relocation register and a limit register for each segment,

- let R_i be the relocation offset for the ith segment
- let L_i be the limit of 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.

6.21 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 larger, contiguous table, we have multiple smaller tables
- If all PTEs in a small table are invalid, we can avoid creating that table entirely

Each virtual address has three parts:

1. Level one page number: used to index the directory
2. Level two page number: used to index a page table
3. Offset within the page

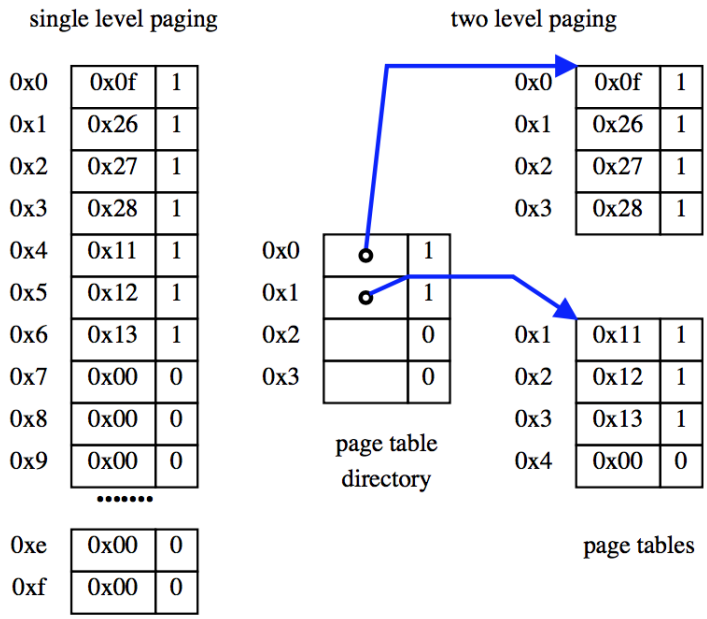


Figure 10: Single vs. Two Level Paging

6.22 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 translate 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)

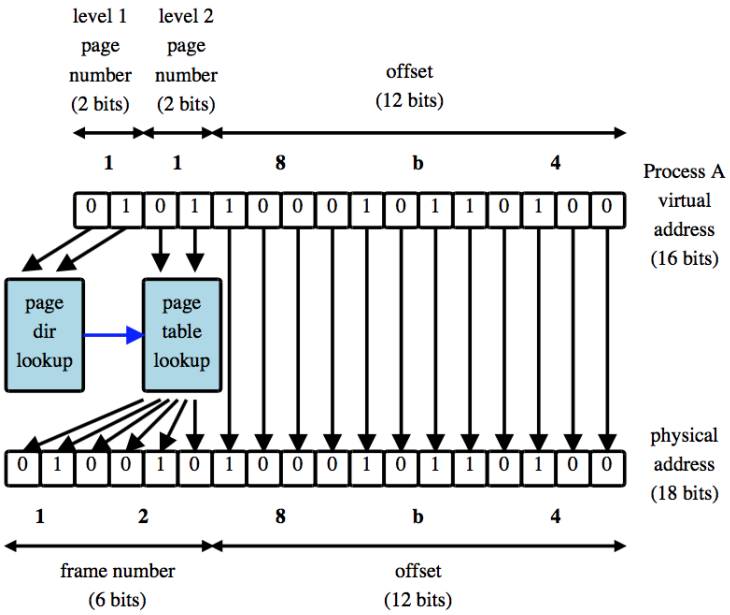


Figure 11: Example of two-level address translation

6.23 Limits of Two-Level Paging

A 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!

6.24 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 table/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

6.25 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

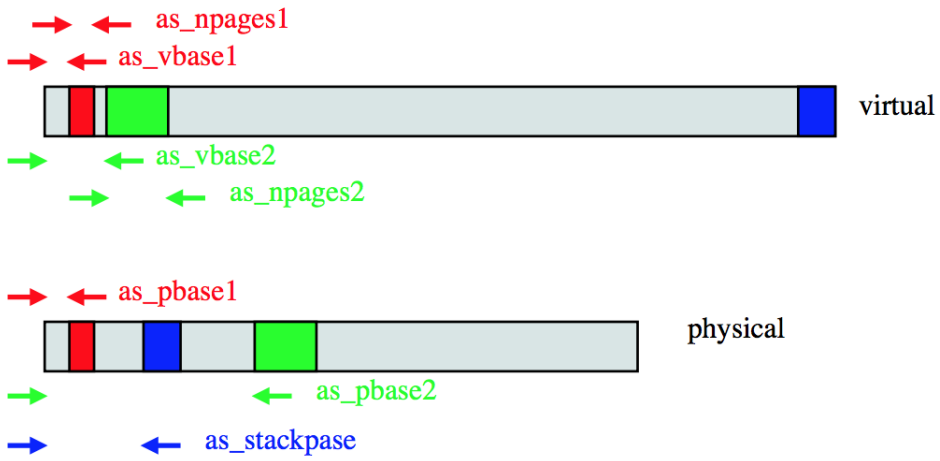


Figure 12: `addrspace` diagram

6.25.1 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 */
};
```

6.25.2 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_bpages1	0x0000 0008	0x0000 0002
as_vbase2	0x1000 0000	0x1000 0000
as_pbase2	0x0080 0000	0x00A0 0000
as_npages	0x0000 0010	0x0000 0008
as_stackpbase	0x0010 0000	0x00B0 0000

7 Scheduling

8 Devices and Device Management

9 File Systems

10 Interprocess Communications and Networking