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 1
1.1 Application View of an Operating System 1
1.2 System View of an Operating System 1
1.3 Implementation View of an Operating System 2
1.4 Operating System Abstractions 2

2 Threads and Concurrency 3
2.1 OS/161's Thread Interface 3
2.2 Why Threads? 3
2.3 Some Reviews 4
2.4 Implementing Concurrent Threads 4
2.5 Timesharing and Context Switches 5
2.6 What Causes a Context Switch? 6
2.7 Preemption 7
2.8 Review on Interrupts 7
2.9 Preemptive Scheduling 8
2.10 Two-Thread Example 8

3 Synchronization 9
3.1 Thread Synchronization 9
3.2 Mutual Exclusion 10
3.2.1 Enforcing Mutual Exclusions With Locks 10
3.3 Hardware-Specific Synchronization Instructions 10
3.3.1 Lock Acquire and Release with Xchg 11
3.3.2 Other Synchronization Instructions 11
3.4 Spinlocks in OS/1616 11
3.5 OS/161 Locks 12
3.6 Thread Blocking 13
3.7 Wait Channels in OS/161 13
3.8 Thread States, Revisited 13
3.9 Semaphores 14
3.10 Condition Variables in OS/161 14
3.10.1 Using Condition Variables 14
3.10.2 Waiting on Condition Variables 15
3.10.3 Example 15
3.11 Deadlocks 15
3.11.1 Two Techniques for Deadlock Prevention 16

4 Processes and System Calls 17
4.1 What is a Process? 17
4.2 System Calls 17
4.3 Kernel Privilege 17
4.4 How System Calls Work 18
4.4.1 Interrupts 18
4.4.2 Exceptions 18
4.4.3 Performing a Syscall 19
4.4.4 System Call Timeline 20
4.4.5 Which Syscall? 20
4.4.6 Syscall Parameters 21
4.5 User and Kernel Stacks 21
4.6 Exception Handling in OS/161 22
4.6.1 mips_trap 22
4.7 Multiprocessing 23
4.8 fork, _exit, and waitpid 23
4.9 The execv system call 24

5 Assignment 2A Review 25
5.1 fork 25
5.2 waitpid 25
5.3 getpid 26
5.4 _exit 26

6 Assignment 2B Review 27

7 Virtual Memory 28
7.1 Physical Memory and Addresses 28
7.2 Virtual Memory and Addresses 28
7.3 Address Translation 29
7.4 Address Translation for Dynamic Relocation 29
7.5 Properties of Dynamic Relocation 30
7.6 Paging: Physical Memory 31
7.7 Paging: Virtual Memory 31
7.8 Paging: Address Translation 32
7.9 Paging: Address Translation 32
7.10 Other Information Found in PTEs 33
7.11 Page Tables: How Big? 33
7.12 Page Tables: Where? 33
7.13 Summary: Roles of the Kernel and the MMU 33

7.14 TLBs 34
7.15 TLB Use 34
7.16 Software-Managed TLBs 35
7.17 Large, Sparse Virtual Memories 36
7.18 Limitations of Simple Address Translation Approaches 36
7.19 Segmentation 37
7.20 Translating Segmented Virtual Addresses 38
7.21 Two-Level Paging 38
7.22 Address Translation with Two-Level Paging 39
7.23 Limits of Two-Level Paging 40
7.24 Multi-Level Paging 41
7.25 Virtual Memory in OS/161 on MIPS: `dumbvm` 41
7.25.1 The `addrspace` Structure 42
7.25.2 Address Translation: OS/161 dumbvm Example 42

8 Scheduling 43

9 Devices and Device Management 44

10 File Systems 45

11 Interprocess Communications and Networking 46

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)

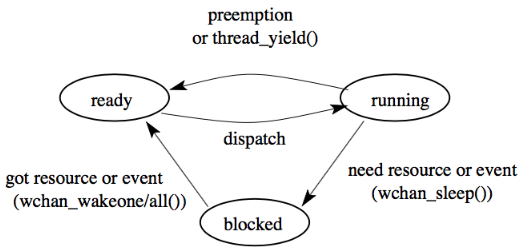
The running thread calls `thread_exit`, running thread is terminated.

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.
Provide a way to *test and set* a lock in a single *atomic* (indivisible) operation.

Example. x86 `xchg` instruction:

```
xchg  src, addr
```

where `src` is typically a register, and `addr` is a memory address.
Value in register `src` is written to memory at address `addr`, and the old value at `addr` is placed into `src`.

Logical behaviour of `xchg` can be thought of as an *atomic* function that behaves like this:

```
Xchg(value,addr) {
  old = *addr;
  *addr = value;
  return(old);
}
```

3.3.1 Lock Acquire and Release with Xchg

```
Acquire(bool *lock) {
  while (Xchg(true,lock) == true) ;
}
Release(book *lock) {
  *lock = false; /* give up the lock */
}
```

If `Xchg` returns `true`, the lock was already set, and we must continue to loop. If `Xchg` returns `false`, then the lock was free, and we have now acquired it.

This construct is known as a *spin lock*, since a thread busy-waits (loops) in Acquire until the lock is free!

3.3.2 Other Synchronization Instructions

SPARC `cas` instruction

```
cas   addr, R1, R2
```

if value at `addr` matches value in `R1` then swap contents of `addr` and `R2`.

Compare-And-Swap

```
CompareAndSwap(addr, expectedval, newval)
  old = *addr;            // get old value at addr
  if (old == expectedval)
    *addr = newval;
  return old;
```

MIPS load-linked and store-conditional
Load-linked returns the current value of a memory location, while a subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to that location since the load-linked.

3.4 Spinlocks in OS/1616

```
struct spinlock {
  volatile spinlock_data_t lk_lock;
  struct cpu *lk_holder;
};
void spinlock_init(struct spinlock *lk);
void spinlock_acquire(struct spinlock *lk);
void spinlock_release(struct spinlock *lk);
```

`spinlock_acquire` calls `spinlock_data_testandset` in a loop until the lock is acquired.
Using Load-Linked/Store-Conditional:

```
/* return value 0 indicates lock was acquired */
spinlock_data_testandset(volatile spinlock_data_t *sd) {
  spinlock_data_t x,y;
  y = 1;
  __asm volatile(
    ".set push;"      /* save assembler mode */
    ".set mips32;"    /* allow MIPS32 instructions */
    ".set volatile;"  /* avoid unwanted optimization */
    "ll %0, 0(%2);"   /* x = *sd */
    "sc %1, 0(%2);"   /* *sd = y; y = success? */
    ".set pop"        /* restore assembler mode */
    : "=r" (x), "+r" (y) : "r" (sd));
  if (y == 0) { return 1; }
return x; }
```

3.5 OS/161 Locks

In addition to spinlocks, OS/161 also has *locks*.
Like spinlocks, locks are used to enforce mutual exclusion.

```
struct lock *mylock = lock_create("LockName");

lock_acquire(mylock);
  critical section
lock_release(mylock);
```

Spinlock *spins*, locks *blocks*:

- a thread that calls `spinlock_acquire` spins until the lock can be acquired

- a thread that called `lock_acquire` *blocks* until the lock can be acquired

3.6 Thread Blocking

Sometimes a thread will need to wait for something, e.g.:

- wait for a lock to be released by another thread
- wait for data from a (relatively) slow device
- wait for input from a keyboard
- wait for busy device to become idle

When a thread blocks, it stops running:

i. the scheduler chooses a new thread to run
ii. a context switch from the blocking thread to the new thread occurs
iii. the blocking thread is queued in a *wait queue* (not on the ready list)

Eventually, a blocked thread is signalled and awakened by another thread.

3.7 Wait Channels in OS/161

Wait channels are used to implement thread blocking in OS/161.

- `void wchan_sleep (struct wchan *wc);`
  blocks calling thread on wait channel `wc`
  causes a context switch, like `thread_yield`
- `void wchan_wakeall (struct wchan *wc);`
  unblocks all threads sleeping on waiting channel `wc`
- `void wchan_wakeone (struct wchan *wc);`
  unblocks one thread sleeping on waiting channel `wc`
- `void wchan_lock (struct wchan *wc);`
  prevent operations on wait channel `wc`

There can be many different wait channels, holding threads that are blocked for different reasons.

3.8 Thread States, Revisited

Refer to Figure 1.
Ready threads are queued on the ready queue, blocked threads are queued on wait channels.

3.9 Semaphores

A semaphore is a synchronization primitive that can be used to enforce mutual exclusion requirements. It can also be used to solve other kinds of synchronization problems.

A semaphore is an object that has an integer value, and that supports two operations (*atomic* by definition):

P: if the semaphore value is greater than 0, decrement the value.
Otherwise, wait until the value is greater than 0 and then decrement it.

V: increment the value of the semaphore

3.10 Condition Variables in OS/161

Each cv is purposed to work together along with a lock: cvs are used *from within the critical section protected by the lock.*
Supported operations:

wait: causes calling thread to block, and it releases the lock associated within the cv.
Once the thread is unblocked, it reacquires the lock.

signal: one of the threads blocked on the signalled cv is unblocked

broadcast: all threads blocked on the cv are unblocked

3.10.1 Using Condition Variables

Cvs allow threads to wait for arbitrary conditions to become true inside of a critical section.
By convention, each cv corresponds to a particular condition of interest to an application.
When a condition is not true, a thread can `wait` on the corresponding cv until it becomes true.
When a thread detects that a condition is true, it uses `signal` or `broadcast` to notify any threads that may be waiting.

Note: signalling (or broadcasting to) a cv that has no waiting threads has *no effect.* Signals *do not* accumulate.

3.10.2 Waiting on Condition Variables

The OS/161 condition variables follows the Mesa-style condition variables. When a blocked thread is unblocked, it reacquires the lock before returning from the `wait` call.
A thread is in the critical section when it calls `wait`, and it will be in the critical section when `wait` returns. In between the call and the return, while the caller is blocked, the caller is *out* of the critical section, and other threads may enter!
The thread that calls `signal` (or `broadcast`) to wake up the waiting thread will itself be in the critical section when it signals. The waiting threads need to wait until the signaller releases the lock before it unblocks and return from the `wait` call.

3.10.3 Example

```
int volatile count = 0;
struct lock *mutex;
struct cv *notfull, *notempty;
/* Initialization Note: the lock and cvâĂŹs must be created
 * using lock_create() and cv_create() before Produce()
 * and Consume() are called
 */
Produce(itemType item) {            itemType Consume() {
  lock acquire(mutex);                lock_acquire(mutex);
  while (count == N) {                while (count == 0)
    cv_wait(notfull, mutex);            cv_wait(nonempty, mutex);
  }                                   remove item from buffer
  add item to buffer                  count = count - 1;
  count = count + 1;                  cv_signal(notfull, mutex);
  cv_signal(notempty, mutex);         lock_release(mutex);
  lock release(mutex);                return(item);
}                                   }
```

3.11 Deadlocks

Threads are *deadlocked* when none of the threads can make progress (i.e. waiting on a lock indefinitely).
Waiting does not resolve the deadlock.
Those threads are permanently stuck.

3.11.1 Two Techniques for Deadlock Prevention

No Hold and Wait: prevent a thread from requesting resources if it currently has resources allocated to it.
A thread may hold several resources, but to do so it must make a single request for all of them.

Resource Ordering: Order the resource types, and require that each thread acquire resources in increasing resource type order.
That is, a thread may make no request for resources of type less than or equal to i if it is holding resources of type i.

4 Processes and System Calls

4.1 What is a Process?

A *process* is an environment in which an application program runs.
A process includes virtualized *resources* that its program can use:

- one (or more) threads
- virtual memory, used for the program's code and data
- other resources

Processes are created and managed by the *kernel*.
Each program's process *isolates* it from other programs in other processes.

4.2 System Calls

System calls (syscalls) are the interface between processes and the kernel.
A process uses syscalls to request operating system services.

Table 2: Syscall examples

Services	OS/161 Examples
create, destroy, manage processes	`fork, execv, waitpid, getpid`
create, destroy, read, write files	`open, close, remove, read, write`
manage file system and directories	`mkdir, rmdir, link, sync`
interprocess communication	`pipe, read, write`
manage virtual memory	`sbrk`
query, manage system	`reboot, _time`

4.3 Kernel Privilege

Kernel code runs at a higher level of *execution privilege* than application code.
Privilege levels are implemented by the CPU.

The kernel's higher privilege level allows it to do things that the CPU prevents less-privileged (application) programs from doing, like

- application programs cannot modify the page tables that the kernel uses to implement process virtual memories

- application programs cannot halt the CPU

These restrictions allow the kernel to keep processes isolated from one another — and from the kernel.

Note: application programs cannot directly call kernel functions or access kernel data structures.

4.4 How System Calls Work

There are only *two* things that make kernel code run!

On the MIPS, the same mechanism handles exceptions and interrupts, and there is a single handler for both in the kernel. The handler uses these codes to determine what triggered it to run.

4.4.1 Interrupts

Interrupts are generated by devices.
An interrupt means a device (hardware) needs attention.

Recall that an interrupt causes the hardware to transfer control to a fixed location in memory, where an *interrupt handler* is located.

Interrupt handlers are part of the kernel.
If an interrupt occurs while an application program is running, control will jump from the application to the kernelâĂŹs interrupt handler.

When an interrupt occurs, the processor switches to privileged execution mode when it transfers control to the interrupt handler. This is how the kernel gets its execution privilege.

4.4.2 Exceptions

Exceptions are caused by instruction execution.
An exception means that a running program needs attention.

Exceptions are conditions that occur during the execution of a program instruction.
Examples: arithmetic overflows, illegal instructions, or page faults.

Exceptions are detected by the CPU during instruction execution.

The CPU handles exceptions like it handles interrupts:

- control is transferred to a fixed location, where an *exception handler* is located

- the processor is switched to privileged execution mode

The exception handler is part of the kernel

MIPS Exception Types:

```
EX_IRQ    0     /* Interrupt */
EX_MOD    1     /* TLB Modify (write to read-only page) */
EX_TLBL   2     /* TLB miss on load */
EX_TLBS   3     /* TLB miss on store */
EX_ADEL   4     /* Address error on load */
EX_ADES   5     /* Address error on store */
EX_IBE    6     /* Bus error on instruction fetch */
EX_DBE    7     /* Bus error on data load *or* store */
EX_SYS    8     /* Syscall */
EX_BP     9     /* Breakpoint */
EX_RI     10    /* Reserved (illegal) instruction */
EX_CPU    11    /* Coprocessor unusable */
EX_OVF    12    /* Arithmetic overflow */
```

4.4.3 Performing a Syscall

To perform a syscall, the application program needs to cause an exception to make the kernel execute!
The kernel's exception handler checks the exception code (set by the CPU when the exception is generated) to distinguish syscall exceptions from other types of exceptions.

On the MIPS, `EX_SYS` is the syscall exception.
To cause this exception on the MIPS, the application executes a special purpose instruction: `syscall`. Other processor instruction sets include similar instructions, e.g., `syscall` on x86.

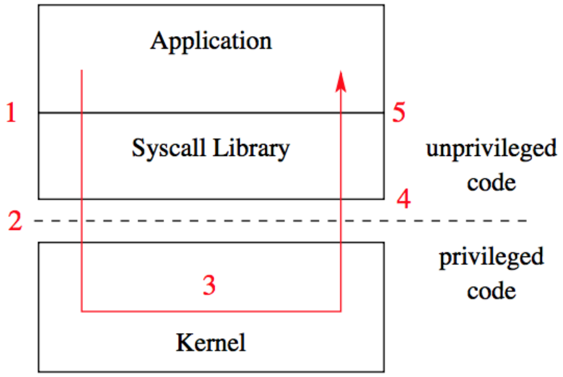


Figure 2: System call software stack

4.4.4 System Call Timeline

1. application calls library wrapper function for desired system call
2. library function performs `syscall` instruction
3. kernel exception handler runs
   - creates a trap frame to save application program state
   - determines that this is a syscall exception
   - determines which syscall is being requested
   - does the work for the requested system call
   - restores the application program state from the trap frame
   - returns from the exception
4. library wrapper function finishes and returns from its call
5. application continues execution

4.4.5 Which Syscall?

The kernel uses system call codes to determine which syscall the application is requesting,

- the kernel defines a code for each syscall it understands
- the kernel expects the application to place a code in a specified location before executing the `syscall` instruction.
  For OS/161 on the MIPS, the code goes in register `v2`

- the kernel's exception handler checks this code to determine which system call has been requested

- the codes and and code location are part of the *kernel ABI* (Application Binary Interface)

4.4.6 Syscall Parameters

The application places parameter values in kernel-specified locations before the `syscall`, and looks for return values in kernel-specified locations after the exception handler returns,

- The locations are part of the kernel ABI

- Parameter and return value placement is handled by the application system call library functions

- On the MIPS
  - parameters go in registers `a0, a1, a2, a3`
  - result success/fail code is in `a3` on return
  - return value or error code is in `v0` on return

4.5 User and Kernel Stacks

Every OS/161 process thread has two stacks, although it only uses one at a time.

User (Application) Stack: used while application code is executing,

- this stack is located in the application's virtual memory

- it holds activation records for application functions

- the kernel creates this stack when it sets up the virtual address memory for the process

Kernel Stack: used while the thread is executing kernel code, after an exception or interrupt,

- this stack is a kernel structure

- in OS/161, the `t_stack` field of the `thread` structure points to this stack

- this stack holds activation records for kernel functions
- this stack also holds *trap frames* and *switch frames* (because the kernel creates trap frames and switch frames)

4.6 Exception Handling in OS/161

First to run is careful assembly code that

- saves the application stack pointer
- switches the stack pointer to point to the thread's kernel stack
- carefully saves application state and the address of the instruction that was interrupted in a trap frame on the thread's kernel stack
- calls `mips_trap`, passing a pointer to the trap frame as a parameter

After `mips_trap` is finished, the handler will

- restore application state (including the application stack pointer) from the trap frame on the thread's kernel stack
- jump back to the application instruction that was interrupted, and switch back to unprivileged execution mode

4.6.1 mips_trap

`mips_trap` determines what type of exception this is by looking at the exception code.
There is a separate handler in the kernel for each type of exception:

- interrupt? call `mainbus_interrupt`
- address translation exception? call `vm_fault`
- system call? call `syscall` (kernel function), passing it the trap frame pointer

4.7 Multiprocessing

Multiprocessing (or multitasking) means having multiple processes existing at the same time.

All processes share the available hardware resources, with the sharing coordinated by the OS:

- Each process' virtual memory is implemented using some of the available physical memory.
  The OS decides how much memory each process gets.
- Each process' threads are scheduled onto the available CPUs (or CPU cores) by the OS.
- Processes share access to other resources (e.g., disks, network devices, I/O devices) by making syscalls.
  The OS controls this sharing.

The OS ensures that processes are isolated from one another.
Interprocess communication should be possible, but only at the explicit request of the processes involved.

Table 3: Syscalls for Process Management

	Linux	OS/161
Creation	`fork, execv`	`fork, execv`
Destruction	`_exit, kill`	`_exit`
Synchronization	`wait, waitpid, pause, ...`	`waitpid`
Attribute Mgmt	`getpid, getuid, nice, getrusage, ...`	`getpid`

4.8 `fork`, `_exit`, and `waitpid`

`fork` creates a new process (the *child*) that is a clone of the original (the *parent*),

- after `fork`, both parent and child are executing copies of the same program
- virtual memories of parent and child are identical at the time of the fork, but may diverge afterwards

- `fork` is called by the parent, but returns in *both* the parent and the child! Parent and child see different return values from `fork`, parent gets child's pid and child gets 0.

`_exit` terminates the process that calls it,

- process can supply an exit status code when it exits
- kernel records the exit status code in case another process asks for it (via `waitpid`)

`waitpid` lets a process wait for another to terminate, and retrieve its exit status code.

4.9 The `execv` system call

`execv` changes the program that a process is running.
The calling process's current virtual memory is destroyed.
The process gets a new virtual memory, initialized with the code and data of the new program to run.
After `execv`, the new program starts executing.

5 Assignment 2A Review

5.1 fork

i. Create process structure for child process.
Use proc_create_runprogram to create the process structure, sets up the VFS and console.

ii. Create and copy address space (and data) from parent to child.
Child process must be identical to the parent process.
as_copy creates a new address space, and copies the pages from the old address space to the new one.
Address space is not associated with the new process yet.

iii. Attach the newly created address space to the child process structure.

iv. Assign PID to child process and create the parent/child relationship.
PIDs should be unique (no two processes should have the same PID).
PIDs should be reusable.

v. Create thread for child process (need a safe way to pass the trapframe to the child thread).
Use thread_fork to create a new thread.
Need to pass trapframe to the child thread.

vi. Child thread needs to put the trapframe onto the stack and modify it so that it returns the current value (and executes the next instruction)

vii. Call mips_usermode in the child to go back to userspace

Note: need to ensure that the new thread does not go to user mode until its address space and trap frame have been set up (requires synchronization)! In addition, the parent process must not return to user mode until its address space and trap frame have been copied.

5.2 waitpid

Only the parent can call waitpid on its children.
If waitpid is called before the child process exits, then the parent must wait/block.
If waitpid is called after the child process has exited, then the parent should immediately get the exit status and exit code.

PID cleanup should not rely on `waitpid`. Parent process is not guaranteed to call `waitpid` when it exits.

5.3 getpid

Returns the PID of the current process.
Need to perform process assignment even without/before any fork calls. The first user process might call `getpid` before creating any children. `getpid` needs to return a valid PID for this process.

5.4 _exit

Causes the current process to exit.
Exit code is passed to the parent process

6 Assignment 2B Review

7 Virtual Memory

7.1 Physical Memory and Addresses

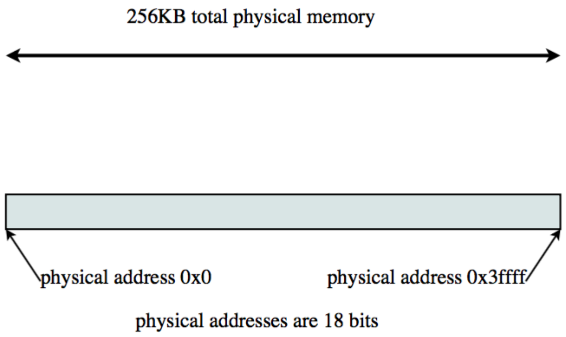


Figure 3: An example physical memory, $P = 18$

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.

7.2 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.,

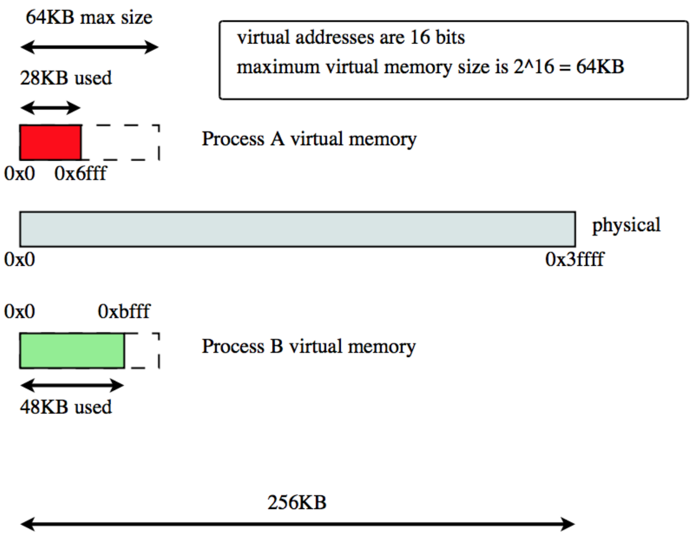


Figure 4: An example of virtual memory, $V = 16$

- 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.

7.3 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.

7.4 Address Translation for Dynamic Relocation

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

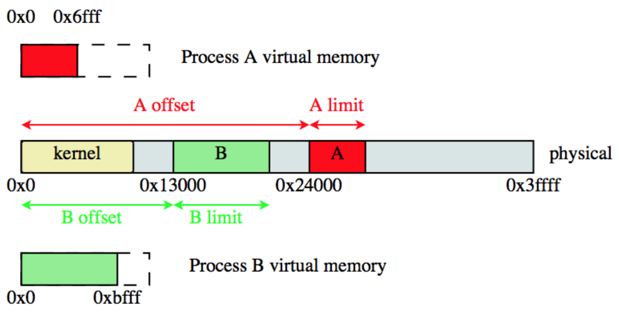


Figure 5: Example of dynamic relocation

- 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 $v \geq L$ then generate exception
else
    $p \leftarrow v + R$
```

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. $v = \texttt{0x102c}$ $\quad p = \texttt{0x102c} + \texttt{0x24000} = \texttt{0x2502c}$
$v = \texttt{0x8800}$ $\quad p = \texttt{exception}$ since $\texttt{0x8800} \geq \texttt{0x7000}$

7.5 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

7.6 Paging: Physical Memory

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

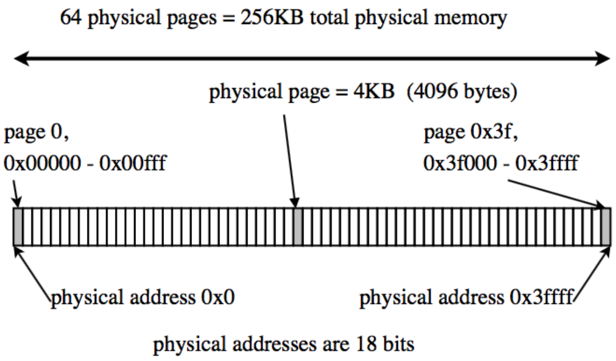

64 physical pages = 256KB total physical memory
physical page = 4KB (4096 bytes)
page 0,
0x00000 - 0x00fff
page 0x3f,
0x3f000 - 0x3ffff
physical address 0x0
physical address 0x3ffff
physical addresses are 18 bits

Figure 6: The frame size if 2^{12} bytes (4KB)

7.7 Paging: Virtual Memory

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

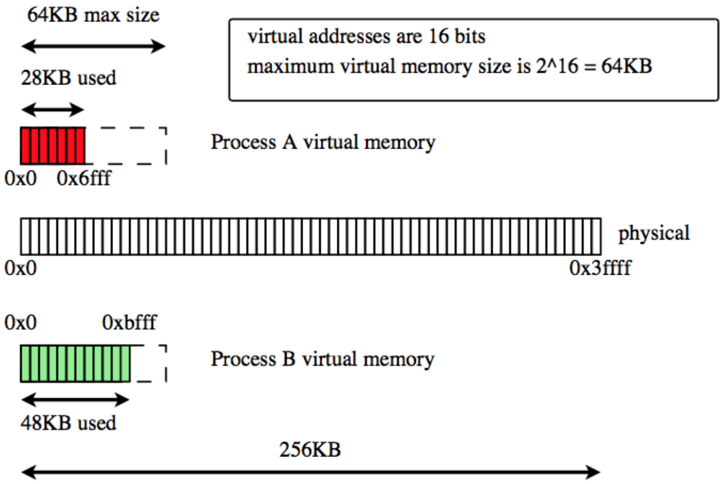
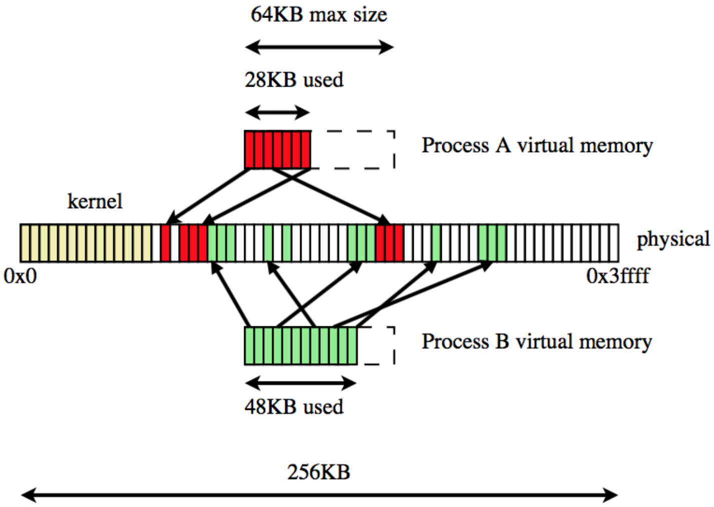


Figure 7: Page size is 4KB here

7.8 Paging: Address Translation



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

7.9 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 · frame size) + offset

7.10 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

7.11 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.

7.12 Page Tables: Where?

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

7.13 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

7.14 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 ⇒ $framenumber$) mapping

7.15 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

7.16 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.

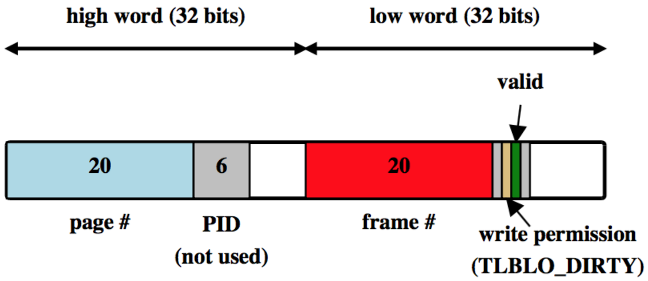


Figure 8: The MIPS R3000 TLB

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

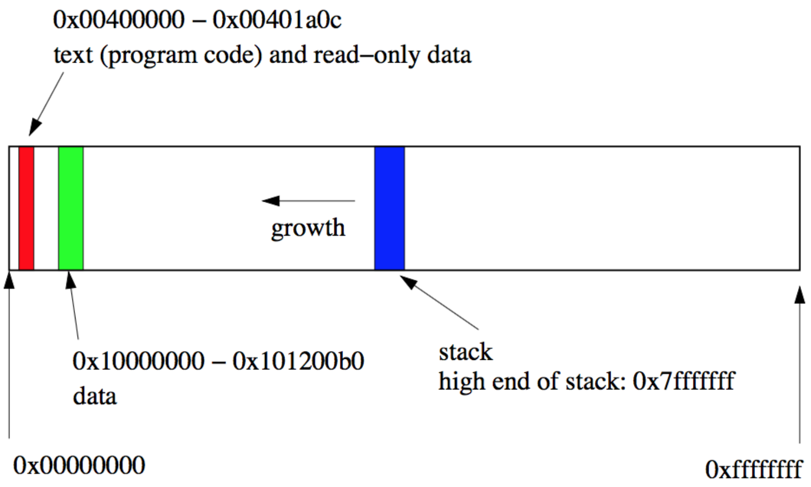


Figure 9: A more realistic virtual memory; this is a layout of the virtual address space for `user/testbin/sort` in OS/161

7.17 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 9)!

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!

7.18 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

7.19 Segmentation

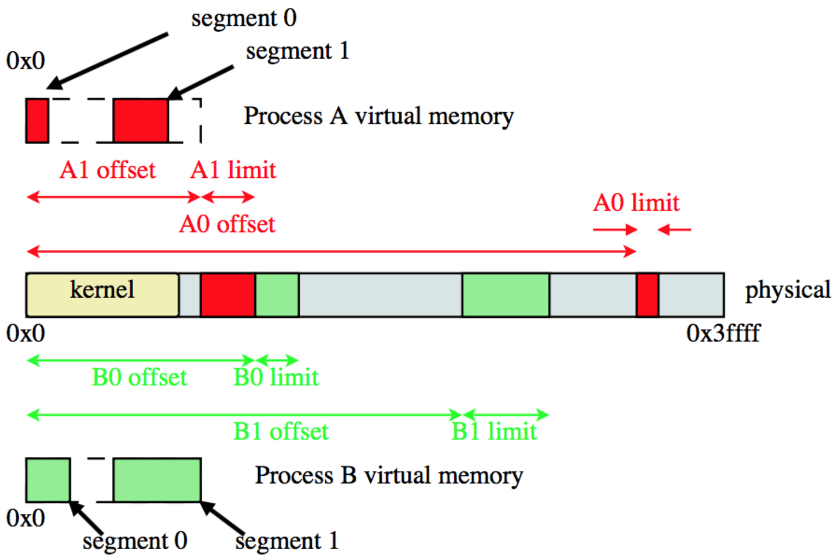


Figure 10: Example of segment address space

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!

7.20 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 $a \geq L_s$ then generate exception
else
  $p \leftarrow a + R_i$
```

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.

7.21 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

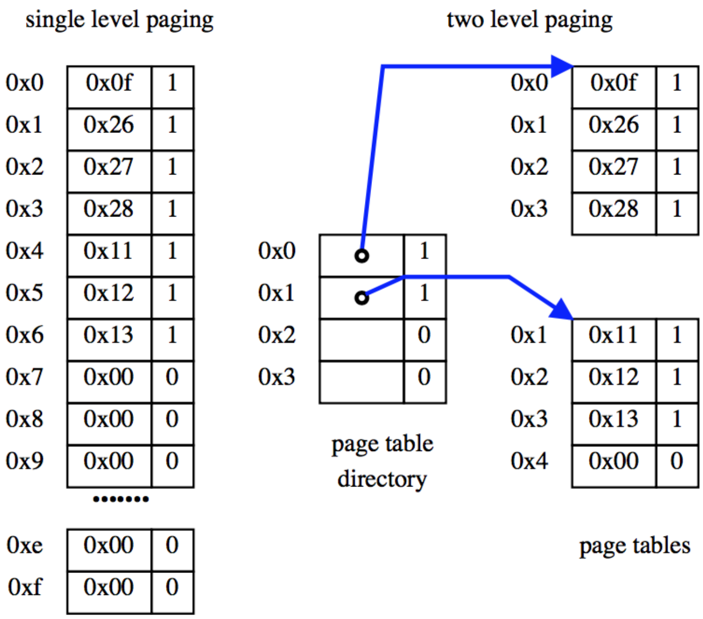


Figure 11: Single vs. Two Level Paging

7.22 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)

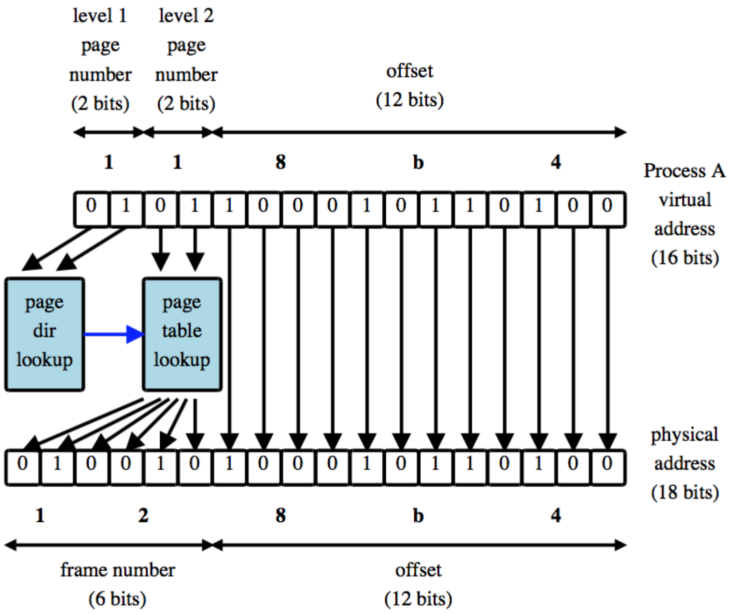


Figure 12: Example of two-level address translation

7.23 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!

7.24 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

7.25 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

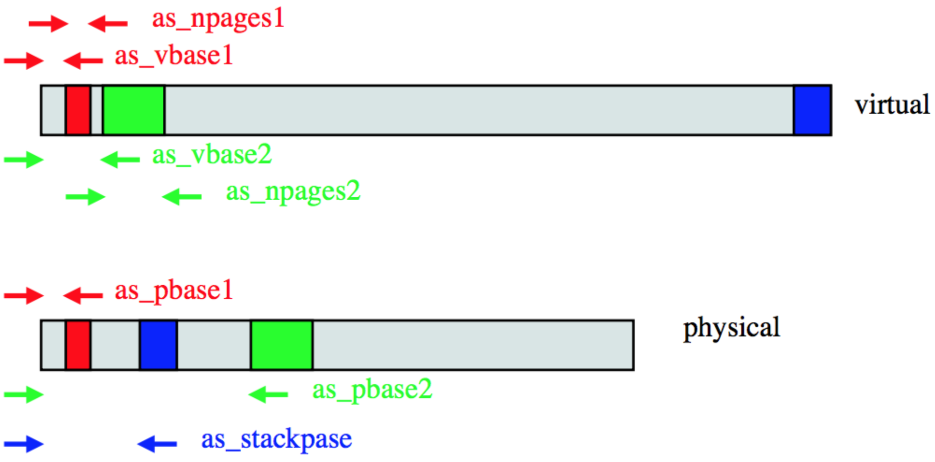


Figure 13: `addrspace` diagram

7.25.1 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 */
};
```

7.25.2 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

8 Scheduling

9 Devices and Device Management

10 File Systems

11 Interprocess Communications and Networking