SistOp Summary

Lautaro Bachmann

Contents

Contents

Introduction to Operating Systems

Introduction

So what happens when a program runs?

Well, a running program does one very simple thing: it executes instructions. the processor fetches an instruction from memory, decodes it and executes it **we have just described the basics of the Von Neumann model of computing2.**

the operating system

is in charge of making sure the system operates correctly and efficiently in an easy-to-use manner.

virtualization.

the OS takes a physical resource and transforms it into a more general, powerful, and easy-to-use virtual form of itself. **we sometimes refer to the operating system as a virtual machine.**

tell the OS what to do

the OS provides some interfaces (APIs) that you can call. A typical OS, in fact, exports a few hundred **system calls** that are available to applications. we also sometimes say that the OS provides a **standard library** to applications.

resource manager.

Each of the CPU, memory, and disk is a **resource** of the system; it is the operating system's role to **manage** those resources, doing so efficiently or fairly

Virtualizing The CPU

Turning a single CPU (or a small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing the CPU,** the ability to run multiple programs at once raises all sorts of new questions. if two programs want to run at a particular time, which should run? **This question is answered by a policy of the OS;** policies are used in many different places within an OS to answer these types of questions,

Virtualizing Memory

model of physical memory

Memory is just an array of bytes; to **read** memory, one must specify an **address** to be able to access the data stored there; to **write** memory, one must also

specify the data to be written to the given address.

virtualizing memory.

Each process accesses its own private virtual address space (sometimes just called its address space), which the OS somehow maps onto the physical memory of the machine. A memory reference within one running program does not affect the address space of other processes

Concurrency

We use this conceptual term to refer to a host of problems that arise, and must be addressed, when working on many things at once (i.e., concurrently) in the same program.

2.4 Persistence

In system memory, data can be easily lost, Thus, we need hardware and software to be able to store data persistently; such storage is thus critical to any system

file system;

it is responsible for storing any files the user creates in a reliable and efficient manner on the disks of the system.

2.5 Design Goals

One of the most basic goals is to build up some abstractions in order to make the system convenient and easy to use. One goal in designing and implementing an operating system is to provide high performance;

Another goal will be to provide protection between applications, as well as between the OS and applications. isolating processes from one another is the key to protection and thus underlies much of what an OS must do. operating systems often strive to provide a high degree of reliability.

The Abstraction: The Process

The Abstraction: A Process

process.

a process is simply a running program; at any instant in time, we can summarize a process by taking an inventory of the different pieces of the system it accesses or affects during the course of its execution.

machine state:

what a program can read or update when it is running.

memory.
Instructions lie in memory; the data that the running program reads and writes sits in memory as well. Thus the memory that the process can address (called its **address space)** is part of the process.

registers;
many instructions explicitly read or update registers and thus clearly they are important to the execution of the process.

special registers
program counter (PC) tells us which instruction of the program will execute next;

stack pointer and frame pointer are used to manage the stack for function parameters, local variables, and return addresses.

persistent storage devices
Such I/O information might include a list of the files the process currently has open.

Process API

Create:

method to create new processes.

Destroy:

an interface to destroy processes forcefully.

Wait:

wait for a process to stop running;

Miscellaneous Control:

For example, some kind of method to suspend a process and then resume it

Status:

get status information such as how long it has run for, or what state it is in.

Process Creation: A Little More Detail

load

the process of loading a program and static data from disk and place them in memory somewhere modern OSes perform the process **lazily,** by loading pieces of code or data only as they are needed during program execution.

run-time stack

Some memory must be allocated for the program's run-time stack

C programs use the stack for local variables, function parameters, and return addresses; the OS allocates this memory and gives it to the process. The OS will also likely initialize the stack with arguments; specifically, it will fill in the parameters to the main() function, i.e., argc and the argv array.

heap.

The OS may also allocate some memory for the program's heap

the heap is used for explicitly requested dynamically-allocated data; programs request such space by calling malloc() and free it explicitly by calling free().

other initialization tasks,

particularly as related to input/output (I/O).

UNIX systems,
each process by default has three open file descriptors, for standard input, output, and error; these descriptors let programs easily read input from the terminal and print output to the screen.

set the stage for program execution.

one last task: to start the program running at the entry point, namely main(). the OS transfers control of the CPU to the newly-created process, and thus the program begins its execution.

Process States

states a process can be in at a given time.

Running:

a process is running on a processor. This means it is executing instructions.

Ready:

a process is ready to run but for some reason the OS has chosen not to run it at this given moment.

Blocked:

a process has performed some kind of operation that makes it not ready to run until some other event takes place.

ready and running states

Being moved from ready to running means the process has been **scheduled;** being moved from running to ready means the process has been **descheduled.**

a process has become blocked

the OS will keep it as such until some event occurs. at that point, the process moves to the ready state again

Data Structures

process list

all processes that are ready and some additional information to track which process is currently running. The OS must also track, in some way, blocked processes;

register context

When a process is stopped, its registers will be saved to this memory location;

other states a process can be in,

initial
state that the process is in when it is being created.

final
state where it has exited but has not yet been cleaned up (in UNIX-based systems, this is called the zombie state).

This final state can be useful as it allows other processes (usually the parent that created the process) to examine the return code of the process and see if the just-finished process executed successfully

ASIDE: KEY PROCESS TERMS

process

the major OS abstraction of a running program.

the process can be described by

its state: the contents of memory in its address space, the contents of CPU registers and information about I/O

process API

consists of calls programs can make related to processes. Typically, this includes creation, destruction, and other useful calls.

process states,

running, ready to run, and blocked. Different events transition a process from one of these states to the other.

process list

contains information about all processes in the system. Each entry is found in what is sometimes called a **process control block (PCB),** which is really just a structure that contains information about a specific process.

Interlude: Process API

The fork() System Call

The fork() system call is used to create a new process

process identifier,

also known as a PID. the PID is used to name the process if one wants to do something with the process, such as (for example) stop it from running.

fork() system call,

the process that is created is an (almost) exact copy of the calling process. The newly-created process doesn't start running at main(), rather, it just comes into life as if it had called fork() itself.

nondeterminism,

leads to some interesting problems, particularly in multi-threaded programs;

The wait() System Call

Sometimes, as it turns out, it is quite useful for a parent to wait for a child process to finish what it has been doing. This task is accomplished with **the wait() system call** (or its more complete sibling waitpid()); the parent process calls **wait()** to delay its execution until the child finishes executing. When the child is done, **wait()** returns to the parent.

The exec() System Call

This system call is useful when you want to run a program that is different from the calling program.

What it does:

given the name of an executable and some arguments it loads code (and static data) from that executable and overwrites its current code segment (and current static data) with it;

it does not create a new process; rather, it transforms the currently running program into a different running program

a successful call to exec() never returns.

Why? Motivating The API

the separation of fork() and exec() is essential in building a **UNIX shell**, because it lets the shell run code after the call to fork() but before the call to exec();

The shell

- It shows you a **prompt** and then waits for you to type something into it.
- You then type a command into it; in most cases, the shell then figures out where in the file system the executable resides, calls **fork()** to create a new child process to run the command, calls some variant of **exec()** to run the command, and then waits for the command to complete by calling **wait()**.
- When the child completes, the shell returns from **wait()** and prints out a prompt again, ready for your next command.

Redirecting Output

The way the shell accomplishes this task is quite simple: when the child is created, before calling **exec()**, the shell closes standard output and opens the file By doing so, any output from the soon-to-be-running program are sent to the file instead of the screen.

file descriptors.

UNIX systems start looking for free file descriptors at zero. In this case, **STDOUT_FILENO** will be the first available one and thus get assigned when open() is called.

UNIX pipes

UNIX pipes are implemented in a similar way, but with the pipe() system call. In this case, the output of one process is connected to an in-kernel pipe (i.e., queue), and the input of another process is connected to that same pipe; thus, the output of one process seamlessly is used as input to the next,

Process Control And Users

there are a lot of other interfaces for interacting with processes in UNIX systems.

For example,

the kill() system call is used to send signals to a process, including directives to pause, die, and other useful imperatives.

UNIX shells,

certain keystroke combinations are configured to deliver a specific signal to the currently running process;

for example, control-c sends a SIGINT (interrupt) to the process (normally terminating it) and control-z sends a SIGTSTP (stop) signal thus pausing the process in mid-execution

signals subsystem

provides a rich infrastructure to deliver external events to processes, including ways to receive and process those signals within individual processes, and ways to send signals to individual processes as well as entire process groups.

signal() system call

It is used to "catch" various signals; doing so ensures that when a particular signal is delivered to a process, it will suspend its normal execution and run a particular piece of code in response to the signal.

who can send a signal to a process, and who cannot?

modern systems include a strong conception of the notion of a user.

The user,
after entering a password to establish credentials, logs in to gain access to system resources. The user may then launch one or many processes, and exercise full control over them **Users generally can only control their own processes;** it is the job of the operating system to parcel out resources to each user to meet overall system goals.

Useful Tools

ps command

allows you to see which processes are running;

The tool top

it displays the processes of the system and how much CPU and other resources they are eating up.

The command kill

can be used to send arbitrary

signals to processes, as can the slightly more user friendly killall.

KEY PROCESS API TERMS

- Each process has a name; in most systems, that name is a number known as a **process ID (PID).**
- The **fork()** system call is used in UNIX systems to create a new process. The creator is called the **parent;** the newly created process is called the **child.** the child process is a nearly identical copy of the parent.

- The **wait()** system call allows a parent to wait for its child to complete execution.
- The **exec()** family of system calls allows a child to break free from its similarity to its parent and execute an entirely new program.
- A UNIX **shell** commonly uses fork(), wait(), and exec() to launch user commands; the separation of fork and exec enables fea- tures like **input/output redirection, pipes,**
- Process control is available in the form of **signals,** which can cause jobs to stop, continue, or even terminate.
- Which processes can be controlled by a particular person is encapsulated in the notion of a **user;** the operating system allows multiple users onto the system, and ensures users can only control their own processes.
- A **superuser** can control all processes this role should be assumed infrequently and with caution for security reasons.

Mechanism: Limited Direct Execution

Introduction

Overview

In order to virtualize the CPU, the operating system needs to somehow share the physical CPU among many jobs running seemingly at the same time. run one process for a little while, then run another one, and so forth. By time sharing the CPU in this manner, virtualization is achieved.

challenges,

performance:
how can we implement virtualization without adding excessive overhead to the system?

control:
how can we run processes efficiently while retaining control over the CPU? without control, a process could simply run forever and take over the machine, or access information that it should not be allowed to access. **Obtaining high performance while maintaining control is thus one of the central challenges in building an operating system.**

Basic Technique: Limited Direct Execution

Description

just run the program directly on the CPU. when the OS wishes to start a program running, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory, locates its entry point, jumps to it, and starts running the user's code.

a few problems

The first
if we just run a program, how can the OS make sure the program doesn't do anything that we don't want it to do, while still running it efficiently?

The second:
when we are running a process, how does the operating system stop it from running and switch to another process,

the "limited" part of the name

without limits on running programs, the OS wouldn't be in control of anything and thus would be "just a library" — a very sad state of affairs for an aspiring operating system!

Problem #1: Restricted Operations

Direct execution

has the obvious advantage of being fast; the program runs natively on the hardware CPU and thus executes as quickly as one would expect.

THE CRUX: HOW TO PERFORM RESTRICTED OPERATIONS

A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system. How can the OS and hardware work together to do so?

ASIDE: WHY SYSTEM CALLS LOOK LIKE PROCEDURE CALLS

You may wonder why a call to a system call, looks exactly like a typical procedure call in C; The simple reason: it is a procedure call, but hidden inside that procedure call is the famous trap instruction. the library uses an agreed-upon calling convention with the kernel to put the arguments in well-known locations puts the system-call number into a well-known location as well and then executes the aforementioned trap instruction. The code in the library after the trap unpacks return values and returns control to the program that issued the system call. the parts of the C library that make system calls are hand-coded in assembly,

processor mode,

user mode;
code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can't issue I/O requests;

kernel mode,
The mode that the operating system runs in. In this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.

perform privileged operation as a user

To enable this, virtually all modern hardware provides the ability for user programs to perform a **system call.** system calls allow the kernel to carefully expose certain key pieces of functionality to user programs,

How to execute a system call

a program must execute a special trap instruction. This instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode; once in the kernel, the system can now perform whatever privileged operations are needed (if allowed), When finished, the OS calls a special **return-from-trap**

instruction, which, as you might expect, returns into the calling user program while simultaneously reducing the privilege level back to user mode.

what code executes upon a trap.

The kernel does so by setting up a **trap table** at boot time. When the machine boots up, it does so in privileged (kernel) mode, and thus is free to configure machine hardware as need be. One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur.

The OS informs the hardware of the locations of these **trap handlers,** Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do when system calls and other exceptional events take place.

system-call number

is usually assigned to each system call. The user code is thus responsible for placing the desired system-call number in a register or at a specified location on the stack;

the OS, when handling the system call inside the trap handler, examines this number, ensures it is valid, and, if it is, executes the corre- sponding code.

protection;
user code cannot specify an exact address to jump to, but rather must request a particular service via number. being able to execute the instruction to tell the hardware where the trap tables are is a very powerful capability. Thus, it is a **privileged operation.**

Kernel Stack
We assume each process has a kernel stack where registers are saved to and restored from when transitioning into and out of the kernel.

limited direct execution protocol phases

Boot time
The kernel initializes the trap table, and the CPU remembers its location for subsequent use. The kernel does so via a privileged instruction

Running a process
the kernel sets up a few things before using a return-from-trap instruction to start the execution of the process; this switches the CPU to user mode and begins running the process. When the process wishes to issue a system call, it traps back into the OS, which handles it and once again returns control via a return-from-trap to the process.

Problem #2: Switching Between Processes

Overview

if a process is running on the CPU, this by definition means the OS is not running. If the OS is not running, how can it do anything at all? (hint: it can't)

THE CRUX: HOW TO REGAIN CONTROL OF THE CPU

How can the operating system regain control of the CPU so that it can switch between processes?

A Cooperative Approach: Wait For System Calls

the cooperative approach.

In this style, the OS trusts the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task.

in a cooperative scheduling system, the OS regains control of the CPU by waiting for a system call or an illegal operation of some kind to take place.

A Non-Cooperative Approach: The OS Takes Control

Without some additional help from the hardware, it turns out the OS can't do much at all when a process refuses to make system calls (or mistakes) and thus return control to the OS.

THE CRUX: HOW TO GAIN CONTROL WITHOUT COOPERATION

How can the OS gain control of the CPU even if processes are not being cooperative? What can the OS do to ensure a rogue process does not take over the machine?

The answer

A timer device can be programmed to raise an interrupt every so many milliseconds; when the interrupt is raised, the currently running process is halted, and a pre-configured interrupt handler in the OS runs. At this point, the OS has regained control of the CPU, and thus can do what it pleases:

TIP: DEALING WITH APPLICATION MISBEHAVIOR

Operating systems often have to deal with misbehaving processes, In modern systems, the way the OS tries to handle such malfeasance is to simply terminate the offender.

the hardware has some responsibility
when an interrupt occurs, in particular to save enough of the state of the program that was running when the interrupt occurred such that a subsequent return-fromtrap instruction will be able to resume the running program correctly.

Saving and Restoring Context

Now that the OS has regained control, a decision has to be made: whether to continue running the currently-running process, or switch to a different one. This decision is made by a part of the operating system known as the **scheduler;**

If the decision is made to switch,
the OS then executes a low-level piece of code which we refer to as a **context switch.**

Context Switch
A context switch is conceptually simple: all the OS has to do is save a few register values for the currently-executing process and restore a few for the soon-to-be-executing process By doing so, the OS thus ensures that when the return-from-trap instruction is finally executed, instead of returning to the process that was running, the system resumes execution of another process.

save the context of the currently-running process,
the OS will execute some low-level assembly code to save the general purpose registers, and then restore said registers, and switch to the kernel stack for the soon-to-be-executing process. By switching stacks, the kernel enters the call to the switch code in the context of one process and returns in the context of another

two types of register saves/restores

Timer interrupts
when the timer interrupt occurs; in this case, the user registers of the running process are implicitly saved by the hardware, using the kernel stack of that process.

Switch
when the OS decides to switch from A to B; in this case, the kernel registers are explicitly saved by the software but this time into memory in the process structure of the process.

Worried About Concurrency?

the OS does indeed need to be concerned as to what happens if, during interrupt or trap handling, another interrupt occurs. One simple thing an OS might do is **disable interrupts** during interrupt processing;

the OS has to be careful disabling interrupts for too long could lead to lost interrupts, which is (in technical terms) bad. Operating systems also have developed a number of sophisticated **locking schemes** to protect concurrent access to internal data structures.

Summary

limited direct execution.

The basic idea is straightforward: just run the program you want to run on the CPU, but first make sure to set up the hardware so as to limit what the process can do without OS assistance.

the concept of baby proofing a room:
locking cabinets containing dangerous stuff and covering electrical sockets. When the room is thus readied, you can let your baby roam freely, secure in the knowledge that the most dangerous aspects of the room have been restricted.

the OS "baby proofs" the CPU,
by first (during boot time) setting up the trap handlers and starting an interrupt timer, and then by only running processes in a restricted mode.

ASIDE: KEY CPU VIRTUALIZATION TERMS (MECHANISMS)

modes of execution:

a restricted user mode and a privileged (non-restricted) kernel mode.

system call

Typical user applications run in user mode, and use a system call to trap into the kernel to request operating system services.

The trap instruction

saves register state carefully, changes the hardware status to kernel mode, and jumps into the OS to a pre-specified destination: the **trap table.**

return-from-trap instruction,

When the OS finishes servicing a system call, it returns to the user program via another special return-from-trap instruction, which reduces privilege and returns control to the instruction after the trap that jumped into the OS.

trap tables

must be set up by the OS at boot time, and make sure that they cannot be readily modified by user programs.

timer interrupt.

Once a program is running, the OS must use hardware mechanisms to ensure the user program does not run forever, namely the **timer interrupt**

This approach is a non-cooperative approach to CPU scheduling.

context switch.

Sometimes the OS, during a timer interrupt or system call, might wish to switch from running the current process to a different one, a low-level technique known as a **context switch.**

Scheduling: Introduction

Workload Assumptions

workload.

processes running in the system, are called the workload

fully-operational scheduling discipline

We will make the following assumptions about the processes, sometimes called jobs, that are running in the system:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

some assumptions are more unrealistic than others in this chapter.

Scheduling Metrics

What is a metric?

A metric is just something that we use to measure something, and there are a number of different metrics that make sense in scheduling.

turnaround time.

The turnaround time of a job is defined as the time at which the job completes minus the time at which the job arrived in the system.

You should note that turnaround time is a **performance** metric,

More formally

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Definitions

convoy effect

Definition
where a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer.

Example
This scheduling scenario might remind you of a single line at a grocery store and what you feel like when you see the person in front of you with **three carts full of provisions**

non-preemptive schedulers

run each job to completion before considering whether to run a new job.

preemptive,

Virtually all modern schedulers are preemptive and quite willing to stop one process from running in order to run another.

two types of schedulers.

The first type (SJF, STCF)
optimizes turnaround time, but is bad for response time.

The second type (RR)
optimizes response time but is bad for turnaround.

First In, First Out (FIFO)

Behaviour

The name says it all. The first job that appears is executed first.

properties:

it is clearly simple and thus easy to implement.

Shortest Job First (SJF)

Behaviour

it runs the shortest job first, then the next shortest, and so on.

SJF performs much better with regards to average turnaround time than FIFO.

assume that jobs can arrive at any time

suffer the same convoy problem as FIFO.

Shortest Time-to-Completion First (STCF)

Behaviour

Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one.

The result is a much-improved average turnaround time

A New Metric: Response Time

We define response time as the time from when the job arrives in a system to the first time it is scheduled.

More formally:

$$T_{response} = T_{firstrun} - T_{arrival}$$

STCF and related disciplines

are not particularly good for response time. While great for turnaround time, this approach is quite bad for response time and interactivity.

Round Robin

Round-Robin (RR) scheduling

The basic idea is simple: instead of running jobs to completion, RR runs a job for a **time slice** (sometimes called a **scheduling quantum**) and then switches to the next job in the run queue.

the length of a time slice

must be a multiple of the timer-interrupt period;

making the time slice too short

is problematic: suddenly the cost of context switching will dominate overall performance.

turnaround time?

RR is one of the **worst policies** if turnaround time is our metric. any policy (such as RR) that is **fair,** will perform poorly on metrics such as turnaround time.

Incorporating I/O

the currently-running job won't be using the CPU during the I/O; the scheduler should probably schedule another job on the CPU at that time.

how a scheduler might incorporate I/O.

treating each CPU burst as a job, the scheduler makes sure processes that are "interactive" get run frequently. While those interactive jobs are performing I/O, other CPU-intensive jobs run, thus better utilizing the processor.

Summary

We have still not solved the problem of the fundamental inability of the OS to see into the future. Shortly, we will see how to overcome this problem, by building a scheduler that uses the recent past to predict the future. This scheduler is known as the **multi-level feedback queue,** and it is the topic of the next chapter.

Scheduling: The Multi-Level Feedback Queue

Definitions

starvation:

if there are "too many" interactive jobs in the system, they will combine to consume all CPU time, and thus long-running jobs will never receive any CPU time (they starve).

game the scheduler.

Gaming the scheduler generally refers to the idea of doing something sneaky to trick the scheduler into giving you more than your fair share of the resource.

voo-doo constants,

Constants that seem to require some form of black magic to set them correctly.

TIP: USE ADVICE WHERE POSSIBLE

it is often useful to provide interfaces to allow users or administrators to provide some hints to the OS.

We often call such hints **advice**, as the OS need not necessarily pay attention to it, but rather might take the advice into account in order to make a better decision.

MLFQ: Summary

why it is called that:

it has multiple levels of queues, and uses feedback to determine the priority of a given job.

History is its guide:

Pays attention to how jobs behave over time and treat them accordingly.

refined set of MLFQ rules,

Rule 1:
If Priority(A) > Priority(B), A runs (B doesn't).

Rule 2:
If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.

Rule 3:
When a job enters the system, it is placed at the highest priority (the topmost queue).

Rule 4:
Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

Rule 5:
After some time period S, move all the jobs in the system to the topmost queue.

MLFQ usefulness

it manages to achieve the best of both worlds: it can deliver excellent overall performance for short-running interactive jobs, and is fair and makes progress for long-running CPU-intensive workloads.

The Abstraction: Address Spaces

The Address Space

Definition

it is the running program's view of memory in the system. The address space of a process contains all of the memory state of the running program.

The memory state elements

the code
The code of the program (the instructions) have to live in memory somewhere,

stack
The program uses a stack to keep track of where it is in the function call chain as well as to allocate local variables and pass parameters and return values to and from routines.

the heap
is used for dynamically-allocated, user-managed memory,

there are other things
There are other things (e.g., statically-initialized variables), but for now let us just assume those three components: **code, stack, and heap.**

Parts that may grow (and shrink)

The parts that may grow (and shrink) are the heap (at the top) and the stack (at the bottom).

We place them like this because each wishes to be able to grow, and by putting them at opposite ends of the address space, we can allow such growth: they just have to **grow in opposite directions.**

this placement of stack and heap is just a convention;

the abstraction

The program really isn't in memory at physical addresses 0 through 16KB; rather it is loaded at some arbitrary physical address(es).

Goals

transparency.

The OS should implement virtual memory in a way that is invisible to the running program. the program behaves as if it has its own private physical memory.

efficiency.

The OS should strive to make the virtualization as efficient as possible, both in terms of time and space

protection.

The OS should make sure to **protect** processes from one another as well as the OS itself from processes.

Summary

The VM system

is responsible for providing the illusion of a large, sparse, private address space to programs, which hold all of their instructions and data therein.

The OS,

The OS, with some serious hardware help, will take each of these virtual memory references, and turn them into physical addresses, which can be presented to the physical memory in order to fetch the desired information.

The OS will do this for many processes at once, making sure to protect programs from one another, as well as protect the OS.

Interlude: Memory API

Types of Memory

stack memory,

allocations and deallocations of it are managed implicitly by the compiler for you, for this reason it is sometimes called **automatic** memory.

heap memory,

all allocations and deallocations are explicitly handled by you,

A couple of notes
both stack and heap allocation occur when using the heap

first the compiler knows to make room for a pointer when it sees your declaration of said pointer. Subsequently, when the program calls malloc(), it requests space on the heap;

heap memory presents more challenges to both users and systems.

Common Errors

Forgetting To Allocate Memory

Not Allocating Enough Memory

Forgetting to Initialize Allocated Memory

Forgetting To Free Memory

Freeing Memory Before You Are Done With It

Freeing Memory Repeatedly

Calling free() Incorrectly

Underlying OS Support

malloc() and free() aren't system calls,

but rather library calls. Thus the malloc library manages space within your virtual address space, but itself is built on top of some system calls which call into the OS to ask for more memory or release some back to the system.

brk system call

is used to change the location of the program's **break:** the location of the end of the heap.

It increases or decreases the size of the heap based on whether the new break is larger or smaller than the current break.

sbrk

is passed an increment but otherwise serves a similar purpose as brk.

mmap() call.

mmap() can create an anonymous memory region within your program

This memory can then also be treated like a heap and managed as such.

Mechanism: Address Translation

Introduction

address translation

With address translation, the hardware transforms each memory access changing the **virtual** address provided by the instruction to a **physical** address where the desired information is actually located.

the hardware alone cannot virtualize memory,

The OS must get involved at key points to set up the hardware so that the correct translations take place; it must thus **manage memory,** keeping track of which locations are free and which are in use, and intervening to maintain control over how memory is used.

create a beautiful illusion:

The goal is to create the illusion that the program has its own private memory, where its own code and data reside.

Assumptions

- the user's address space must be placed **contiguously** in physical memory.
- the size of the address space is **less than the size of physical memory.**
- each address space is exactly the **same size.**

Static Relocation

crude form of relocation purely via software methods.

a piece of software known as the **loader** takes an executable that is about to be run and rewrites its addresses to the desired offset in physical memory.

It does not provide protection.

Dynamic (Hardware-based) Relocation

base and bounds;

the technique is also referred to as **dynamic relocation**;

we'll need two hardware registers within each CPU: the base register, and the bounds register

In this setup,

each program is written and compiled as if it is loaded at address zero.

when a program starts running,

the OS decides where in physical memory it should be loaded and sets the **base register** to that value.

when the process is running.

When any memory reference is generated by the process, it is translated by the processor in the following manner:

physical address = virtual address + base

Each memory reference

generated by the process is a virtual address;

address translation;

the hardware takes a virtual address the process thinks it is referencing and transforms it into a physical address which is where the data actually resides.

the bounds register

is there to help with protection. the processor will first check that the memory reference is **within bounds** to make sure it is legal;

If a process generates a virtual address that is greater than the bounds, or one that is negative, the CPU will raise an exception,

memory management unit (MMU);

Sometimes people call the part of the processor that helps with address translation the **memory management unit (MMU)**;

bound registers can be defined two ways.

In one way
it holds the size of the address space, and thus the hardware checks the virtual address against it first before adding the base.

In the second way,
it holds the physical address of the end of the address space, and thus the hardware first adds the base and then makes sure the address is within bounds.

Hardware Support: A Summary

Hardware Requirements

- **Privileged mode:** Needed to prevent user-mode processes from executing privileged operations

- **Base/bounds registers:** Need pair of registers per CPU to support address translation and bounds checks
- **Ability to translate virtual addresses and check if within bounds:** Circuitry to do translations and check limits; in this case, quite simple
- **Privileged instruction(s) to update base/bounds:** OS must be able to set these values before letting a user program run
- **Privileged instruction(s) to register exception handlers:** OS must be able to tell hardware what code to run if exception occurs
- **Ability to raise exceptions:** When processes try to access privileged instructions or out-of-bounds memory

Operating System Issues

OS Requirements

- **Memory management:** Need to allocate memory for new processes; Reclaim memory from terminated processes; Generally manage memory via free list
- **Base/bounds management:** Must set base/bounds properly upon context switch
- **Exception handling:** Code to run when exceptions arise; likely action is to terminate offending process

Summary

address translation.

With address translation, the OS can control each and every memory access from a process, ensuring the accesses stay within the bounds of the address space.

hardware support,

helps to perform the translation quickly for each access, turning virtual addresses into physical ones

All of this is performed in a way the process has no idea its memory references are being translated,

base and bounds charasteristics

Efficiency
Base-and-bounds virtualization is quite efficient,

protection;
the OS and hardware combine to ensure no process can generate memory references outside its own address space.

Space inefficiencies.
internal fragmentation, the space inside the allocated unit is not all used (i.e., is fragmented) and thus wasted.

Segmentation

Segmentation: Generalized Base/Bounds

The idea

instead of having just one base and bounds pair in our MMU, why not have a base and bounds pair per logical **segment** of the address space?

A segment

is just a contiguous portion of the address space of a particular length,

we have three logically-different segments:
code, stack, and heap.

What segmentation allows the OS to do

place each one of those segments in different parts of physical memory, and thus avoid filling physical memory with unused virtual address space.

The hardware structure in our MMU

required to support segmentation is a set of three base and bounds register pairs.

Which Segment Are We Referring To?

The hardware uses segment registers during translation. How does it know the offset into a segment, and to which segment an address refers?

explicit approach,

chop up the address space into segments based on the top few bits of the virtual address; we have three segments; thus we need two bits to accomplish our task.

one segment of the address space goes unused.

Another issue

with using the top so many bits to select a segment is that it limits use of the virtual address space.

the implicit approach,

the hardware deter-

mines the segment by noticing how the address was formed.

the address was generated from the program counter

then the address is within the code segment;

if the address is based off of the stack or base pointer,
it must be in the stack segment;

any other address
must be in the heap.

What About The Stack?

little extra hardware support.

the hardware also needs to know which way the segment grows

Support for Sharing

to save memory,

share certain memory

segments between address spaces. In particular, code sharing is common and still in use in systems today. To support sharing, we need a little extra support from the hardware,

protection bits.

Basic support adds a few bits per segment, indicating whether or not a program can read or write a segment, or perhaps execute code that lies within the segment.

the hardware algorithm

would also have to change. In addition to checking whether a virtual address is within bounds, the hardware also has to check whether a particular access is permissible.

Fine-grained vs. Coarse-grained Segmentation

coarse-grained,

it chops up the address space into relatively large, coarse chunks. large number of smaller segments,

fine-grained

segment table

support the creation of a very large number of segments, and thus enable a system to use segments in more flexible ways

OS Support

how segmentation works.

Pieces of the address space are relocated into physical memory as the system runs, and thus a huge savings of physical memory is achieved

segmentation

issues

what should the OS do on a context switch?
the segment registers must be saved and restored. Clearly, each process has its own virtual address space, and the OS must make sure to set up these registers correctly before letting the process run again.

interaction when segments grow
In some cases, the existing heap will be able to service the request,

In others, the heap segment itself may need to grow.

managing free space in physical memory.
When a new address space is created, the OS has to be able to find space in physical memory for its segments. we have a number of segments per process, and each segment might be a different size.

The general problem

is that physical memory quickly becomes full of little holes of free space, making it difficult to allocate new segments, or to grow existing ones.

external fragmentation

One solution

would be to **compact** physical memory by rearranging the existing segments. However, compaction is expensive, as copying segments is memory-intensive and generally uses a fair amount of processor time;

use a free-list management algorithm

tries to keep large extents of memory available for allocation.

hundreds of approaches
best-fit (which keeps a list of free spaces and returns the one closest in size that satisfies the desired allocation to the requester), **worst-fit, first-fit,** and more complex schemes like **buddy algorithm**

Summary

Segmentation

solves a number of problems, and helps us build a more effective virtualization of memory. segmentation can **better support sparse address spaces,** by avoiding the huge potential waste of memory between logical segments of the address space. **It is also fast,** as doing the arithmetic segmentation requires is easy and well-suited to hardware; **the overheads of translation are minimal.** A fringe benefit arises too: **code sharing.** If code is placed within a separate segment, such a segment could potentially be shared across multiple running programs.

allocating variable-sized segments

in memory leads to some problems

The first,
is external fragmentation. Because segments are variablesized, free memory gets chopped up into odd-sized pieces, and thus satisfying a memory-allocation request can be difficult.

The second
is that segmentation still isn't flexible enough to support our fully generalized, sparse address space. if our model of how the address space is being used doesn't exactly match how the underlying segmentation has been designed to support it, segmentation doesn't work very well.

Free-Space Management

Definitions

free-space management.

It is easy when the space you are managing is divided into fixed-sized units; Where free-space management becomes more difficult is when the free space you are managing consists of variable-sized units;

external fragmentation:

the free space gets chopped into little pieces of different sizes and is thus fragmented; subsequent requests may fail because there is no single contiguous space that can satisfy the request, even though the total amount of free space exceeds the size of the request.

internal fragmentation;

if an allocator hands out chunks of memory bigger than that requested, any unasked for space in such a chunk is considered internal fragmentation once memory is handed out to a client, it cannot be relocated to another location in memory.

Assumptions

- We assume a basic interface such as that provided by malloc() and free().
  - This implies that the user, when freeing the space, does not inform the library of its size;
- We assume that primarily we are concerned with external fragmentation,

Low-level Mechanisms

Splitting and Coalescing

splitting:

find a free chunk of memory that can satisfy the request and split it into two. The first chunk it will return to the caller; the second chunk will remain on the list.

the split is commonly used in allocators when requests are smaller than the size of any particular free chunk.

coalescing of free space.

when returning a free chunk in memory, if the newlyfreed space sits right next to existing free chunks, merge them into a single larger free chunk.

Tracking The Size Of Allocated Regions

header block

To accomplish this task, most allocators store a little bit of extra information in a **header block** which is kept in memory, usually just before the handed-out chunk of memory.

The header minimally contains the size of the allocated region; it may also contain additional pointers to speed up deallocation,

Embedding A Free List

you need to build the list inside the free space itself.

the library will first find a chunk that is large enough to accommodate the request;

Then, the chunk will be split into two: one chunk big enough to service the request and the remaining free chunk.

out of the existing one free chunk, returns a pointer to it, stashes the header information immediately before the allocated space for later use upon free(),

When the calling program returns some memory via free()

The library immediately figures out the size of the free region, and then adds the free chunk back onto the free list.

Growing The Heap

what should you do if the heap runs out of space?

- The simplest approach is just to fail.
- Most traditional allocators start with a small-sized heap and then request more memory from the OS when they run out.

Basic Strategies

Best Fit

search through the free list and find chunks of free memory that are as big or bigger than the requested size. Then, return the one that is the smallest in that group of candidates;

naive implementations pay a heavy performance penalty when performing an exhaustive search for the correct free block.

Worst Fit

The worst fit approach is the opposite of best fit; find the largest chunk and return the requested amount; keep the remaining (large) chunk on the free list.

a full search of free space is required, and thus this approach can be costly.

First Fit

The first fit method simply finds the first block that is big enough and returns the requested amount to the user. First fit has the advantage of speed but sometimes pollutes the beginning of the free list with small objects.

address-based ordering;
One approach is to use address-based ordering; by keeping the list ordered by the address of the free space, coalescing becomes easier, and fragmentation tends to be reduced.

Next Fit

the next fit algorithm keeps an extra pointer to the location within the list where one was looking last. The idea is to spread the searches for free space throughout the list more uniformly,

The performance of such an approach is quite similar to first fit, as an exhaustive search is once again avoided.

Other Approaches

Segregated Lists

The basic idea
if a particular application has one (or a few) popular-sized request that it makes, keep a separate list just to manage objects of that size; all other requests are forwarded to a more general memory allocator.

By having a chunk of memory dedicated for one particular size of requests, fragmentation is much less of a concern;

New complications
how much memory should one dedicate to the pool of memory that serves specialized requests?

the slab allocator
when the kernel boots up, it allocates a number of **object caches** for kernel objects that are likely to be requested frequently;

the object caches thus are each segregated free lists of a given size and serve memory allocation and free requests quickly.

When a given cache is running low on free space, it requests some **slabs** of memory from a more general memory allocator

when the reference counts of the objects within a given slab all go to zero, the general allocator can reclaim them from the specialized allocator,

The slab allocator also goes beyond most segregated list approaches by keeping free objects on the lists in a pre-initialized state. the slab allocator thus avoids frequent initialization and destruction cycles per object and thus lowers overheads noticeably.

Buddy Allocation

binary buddy allocator
free memory is first conceptually thought of as one big space of size 2^N

When a request for memory is made,
the search for free space recursively divides free space by two until a block that is big enough to accommodate the request is found. At this point, the requested block is returned to the user.

The beauty of buddy allocation
is found in what happens when that block is freed. The allocator checks if the buddy of the block is still free; if so, it coalesces those two blocks. This recursive coalescing process continues up the tree, either restoring the entire free space or stopping when a buddy is found to be in use.

Other Ideas

One major problem
with many of the approaches described above is their lack of **scaling.**

more complex data structures
searching lists can be quite slow. advanced allocators use **more complex data structures** to address these costs, trading simplicity for performance.

Data structures examples

- balanced binary trees,
- splay trees,
- partially-ordered trees

multiple processors
a lot of effort has been spent making allocators work well on multiprocessor-based systems.

Paging: Introduction

paging,

Paging refers to chop up space into fixed-sized pieces. each of which we call a **page.**

physical memory

an array of fixed-sized slots called **page frames;** each of these frames can contain a single virtual-memory page.

Overview

Paging advantages

flexibility:
Probably the most important improvement is flexibility.

With a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space;

simplicity
When the OS wants to place the adress space into physical memory it simply has to find some free pages to use;

perhaps the OS keeps a **free list** of all free pages for this, and just grabs the first free pages off of this list.

page table.

To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a per-process data structure known as a page table.

major role
store **address translations** for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides.

important to remember
this page table is a **per-process** data structure

Translate a virtual adress

To translate a virtual address that the process generated, we have to first split it into two components: **the virtual page number (VPN),** and the **offset**

Formulas

$$\text{VPN} = \text{address} \,/\, \text{pagesize}$$

$$\text{offset} = \text{address} \,\%\, \text{pagesize}$$

$$\text{physical} = \text{VPN} * \text{pagesize} + \text{offset}$$

Where Are Page Tables Stored?

Page tables can get terribly large,

we store the page table for each process in **memory** somewhere. Let's assume for now that the page tables live in physical memory that the OS manages;

What's Actually In The Page Table?

page table organization.

The page table is just a data structure that is used to map virtual addresses to physical addresses. Thus, any data structure could work.

linear page table,

The simplest form is called a linear page table, which is just an array.

The OS **indexes** the array by the **virtual page number** (VPN), and looks up the **page-table entry** (PTE) at that index in order to find the desired **physical frame number** (PFN).

contents of each PTE,

A valid bit
is common to indicate whether the particular translation is valid; All the unused space in-between the address space will be marked **invalid,**

by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages and thus save a great deal of memory.

protection bits,
indicating whether the page could be read from, written to, or executed from.

present bit
indicates whether this page is in physical memory or on disk

dirty bit
indicating whether the page has been modified since it was brought into memory.

reference bit (a.k.a. accessed bit)
is sometimes used to track whether a page has been accessed, and is useful in determining which pages are popular and thus should be kept in memory;

Paging: Also Too Slow

With page tables in memory, we already know that they might be too big. As it turns out, they can slow things down too.

Paging: Faster Translations (TLBs)

Introduction

translation-lookaside buffer, or TLB

To speed address translation, we are going to add what is called a translation-lookaside buffer, or TLB

A TLB is part of the chip's **memory-management unit (MMU),** and is simply a hardware **cache** of popular virtual-to-physical address translations;

Upon each virtual memory reference,

the hardware first checks the TLB to see if the desired translation is held therein; if so, the translation is performed (quickly) **without** having to consult the page table

TLB Basic Algorithm

Assumptions

- We are using a simple linear page table
- We are using a hardware-managed TLB

first,

extract the virtual page number (VPN) from the virtual address and check if the TLB holds the translation for this VPN

TLB hit,
which means the TLB holds the translation.

We can now extract the page frame number (PFN) from the relevant TLB entry, concatenate that onto the offset from the original virtual address, and form the desired physical address (PA), and access memory assuming protection checks do not fail

TLB miss
This happens when the CPU does not find the translation in the TLB

the hardware accesses the page table to find the translation and, updates the TLB with the translation

These set of actions are costly,

Finally,

once the TLB is updated, the translation is found in the TLB, and the memory reference is processed quickly.

The TLB premise

The TLB is built on the premise that in the common case, translations are found in the cache

When a miss occurs,

the high cost of paging is incurred;

TIP: USE CACHING WHEN POSSIBLE

locality

temporal locality,
the idea is that an instruction or data item that has been recently accessed will likely be re-accessed soon in the future.

spatial locality,
the idea is that if a program accesses memory at address x, it will likely soon access memory near x.

Hardware caches,

Hardware caches take advantage of locality by keeping copies of memory in small, fast on-chip memory.

If you want a fast cache,

it has to be small, Any large cache by definition is slow, and thus defeats the purpose.

Who Handles The TLB Miss?

Hardware-managed TLB

In the olden days, the hardware would handle the TLB miss entirely.

To do this, the hardware would "walk" the page table, find the correct page-table entry and extract the desired translation, update the TLB with the translation, and retry the instruction.

software-managed TLB.

the hardware simply raises an exception which pauses the current instruction stream, raises the privilege level to kernel mode, and jumps to a **trap handler.**

When run,
the code will lookup the translation in the page table, use special "privileged" instructions to update the TLB, and return from the trap; at this point, the hardware retries the instruction

couple of important details.

the return-from-trap instruction
needs to be a little different than the return-from-trap when servicing a system call. when returning from a TLB miss-handling trap, the hardware must resume execution at the instruction that **caused** the trap;

when running the TLB miss-handling code,
the OS needs to be extra careful not to cause an infinite chain of TLB misses to occur.

advantage of the software-managed approach

flexibility:
the OS can use any data structure it wants to implement the page table, without necessitating hardware change.

simplicity,
The hardware doesn't do much on a miss: just raise an exception and let the OS TLB miss handler do the rest.

ASIDE: TLB VALID BIT $\neq$ PAGE TABLE VALID BIT

Page table valid bit

If the PTE is marked invalid, it means that the page has not been allocated by the process, and should not be accessed by a correctly-working program.

TLB valid bit,

refers to whether a TLB entry has a valid translation within it.

TLB Contents: What's In There?

Typical size

A typical TLB might have 32, 64, or 128 entries

fully associative.

this just means that any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation.

TLB entry

A TLB entry might look like this:

VPN | PFN | other bits

Note that both the **VPN** and **PFN** are present in each entry,

"other bits".

valid bit,
  says whether the entry has a valid translation or not.

protection bits,
  determine how a page can be accessed

For example, code pages might be marked read and execute, whereas heap pages might be marked read and write.

other fields,
  including an **address-space identifier,** a **dirty bit,** and so forth;

TLB Issue: Context Switches

issues when switching between processes

the TLB contains virtual-to-physical translations that are only valid for the currently running process; As a result, when switching from one process to another, the hardware or OS (or both) must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process.

possible solutions

One approach is to simply **flush** the TLB on context switches, thus emptying it before running the next process.

However, there is a cost:
  each time a process runs, it must incur TLB misses as it touches its data and code pages. If the OS switches between processes frequently, this cost may be high.

To reduce this overhead,
  some systems add hardware support to enable sharing of the TLB across context switches.

In particular, some hardware systems provide an **address space identifier (ASID)** field in the TLB. You can think of the ASID as a **process identi-**

fier (PID), only the ASID field is needed to differentiate otherwise identical translations.

Issue: Replacement Policy

Specifically, when we are installing a new entry in the TLB, we have to **replace** an old one, and thus the question: **which one to replace?**

a few typical policies.

least-recently-used
One common approach is to evict the least-recently-used entry

LRU tries to take advantage of locality in the memory-reference stream, assuming it is likely that an entry that has not recently been used is a good candidate for eviction.

random policy,
Another typical approach is to use a random policy which evicts a TLB mapping at random.

Summary

TLB coverage,

If the number of pages a program accesses in a short period of time exceeds the number of pages that fit into the TLB, the program will generate a large number of **TLB misses**, and thus run quite a bit more slowly. We refer to this phenomenon as exceeding the **TLB coverage**

other TLB issue

TLB access can easily become a bottleneck in the CPU pipeline, in particular with what is called a **physically-indexed cache.** With such a cache, address translation has to take place before the cache is accessed, which can slow things down quite a bit.

Paging: Smaller Tables

Simple Solution: Bigger Pages

Effect

If we use bigger pages, the page table size reduction exactly mirrors the increase in page size.

The Problem

is that big pages lead to internal fragmentation

Hybrid Approach: Paging and Segments

our hybrid approach:

instead of having a single page table for the entire address space of the process, why not have one per logical segment? we might thus have three page tables, one for the code, heap, and stack parts of the address space.

with segmentation,

we had a **base** register that told us where each segment lived in physical memory, and a **bound** register that told us the size of said segment.

In our hybrid,

we still have those structures in the MMU; we use the base not to point to the segment itself but rather to hold the **physical address of the page table** of that segment. **The bounds register** is used to indicate the end of the page table

On a TLB miss

the hardware uses the segment bits (SN) to determine which base and bounds pair to use. The hardware then takes the physical address therein and combines it with the VPN to form the address of the page table entry (PTE):

The critical difference in our hybrid

is the presence of a bounds register per segment; each bounds register holds the value of the maximum valid page in the segment. unallocated pages between the stack and the heap no longer take up space in a page table

Problems

Use of segmentation,
it still requires us to use segmentation;

External fragmentation,
this hybrid causes external fragmentation to arise again.

Multi-level Page Tables

The basic idea

First, chop up the page table into page-sized units; then, if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all. To track whether a page of the page table is valid use a new structure, called the page directory.

page directory.

The page directory thus either can be used to tell you where a page of the page table is, or that the entire page of the page table contains no valid pages.

Explained Behaviour

it just makes parts of the linear page table disappear and tracks which pages of the page table are allocated with the page directory.

page directory entries

The page directory, in a simple two-level table, contains one entry per page of the page table. It consists of a number of page directory entries (PDE).

A PDE (minimally) has a **valid bit** and a **page frame number** (PFN), similar to a PTE.

the meaning of this valid bit is slightly different:

if the PDE is valid, it means that at least one of the pages of the page table that the entry points to (via the PFN) is valid, If the PDE is not valid the rest of the PDE is not defined.

advantages

the multi-level table only allocates page-table space in proportion to the amount of address space you are using;

each portion of the page table fits neatly within a page, making it easier to manage memory; the OS can simply grab the next free page when it needs to allocate or grow a page table.

Disadvantages

on a TLB miss, two loads from memory will be required to get the right translation information from the page table

Another obvious negative is increased complexity.

More Than Two Levels

to make each piece of the page table fit within a single page if the page directory gets too big we build a further level of the tree, by splitting the page directory itself into multiple pages, and then adding another page directory on top of that, to point to the pages of the page directory.

The Translation Process: Remember the TLB

process of address translation

before any of the complicated multilevel page table access occurs, the hardware first checks the TLB;

upon a hit,

the physical address is formed directly without accessing the page table at all,

upon a TLB miss

the hardware need to perform the full multi-level lookup.

The cost of our traditional two-level page table: two additional memory accesses to look up a valid translation.

Inverted Page Tables

Description

instead of having many page tables we keep a single page table that has an entry for each **physical page** of the system.

The entry

tells us which process is using this page, and which virtual page of that process maps to this physical page.

Finding the correct entry

is now a matter of searching through this data structure. a hash table is often built over the base structure to speed up lookups.

Swapping the Page Tables to Disk

the size of page tables, may be too big to fit into memory all at once. Thus, some systems place such page tables in **kernel virtual memory,** thereby allowing

the system to **swap** some of these page tables to disk when memory pressure gets a little tight.

Notas clase

CR3

Apunta base tabla paginas (page directory)

Se encarga
Cambiar memoria cambios de contexto

Muy facil
Hacer context switch. Solo hay que cambiar un registro

Cambiar esquema memoria virtual de un proceso a otro desde el sistema operativo?

Se cambia a donde apunta el CR3

Memoria Kernel

Por mas que los procesos cambien direccion memoria virtual, la memoria del kernel siempre debe estar en el mismo lugar

Por ende
Todos los procesos mapean el kernel en el mismo lugar

Traduccion memoria paginas 1 nivel

Virtual a física

1. Tomar primeros 5 digitos (hexa) y ver a que entrada corresponde.
2. Reemplazar los primeros 5 digitos de la memoria virtual con la direccion del marco físico de la entrada
3. Revisar demas bits entrada para ver acceso y si está presente

Fisica a virtual

1. Ver primeros 5 digitos (hexa) y ver a que marco físico corresponde.
2. Reemplazar 5 digitos de la memoria física por el número de entrada

Nota: Puede producir mas de una direccion

Mapear multiples virtuales a una fisica

Muy util para ahorrar memoria

Ejemplos

- El fork usa las mismas paginas a no ser que se escriba algo nuevo
- bibliotecas compartidas
- Calloc, usa marco pagina física llena de 0 y le pone RWX = 100
  - Cuando se escribe salta trap y se cambia RWX

Otros nombres
COW: copy on write

Caso bomba fork
Solo se pide memoria para las tablas de página. Pero como el crecimiento es exponencial se termina rompiendo todo.

Paginacion 2 niveles

Se representa
(BITS_PAGE_DIR, BITS_PAGE_TABLE, BITS_OFFSET)

Pasar de virtual a física

1. Tomar primeros n bits especificados y ver a que entrada de la page directory corresponde
2. Ver a que page table apunta la entrada de la page directory
3. Tomar el número de entrada de la page table del segundo grupo de m bits especificados
4. Reemplazar los primeros 5 digitos de la memoria virtual con la direccion del marco físico de la entrada
5. Revisar demas bits entrada para ver acceso y si está presente