Introduction to Operating Systems

David Marchant
Based on slides by Troels Henriksen

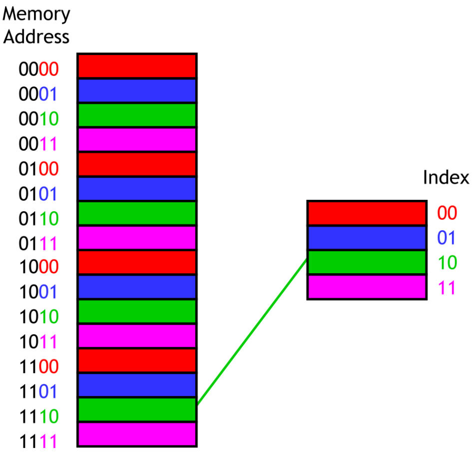
2022-09-27

Incorporating material by Mathias Payer
(`https://github.com/HexHive/OSTEP-slides`).

A bit more on cache

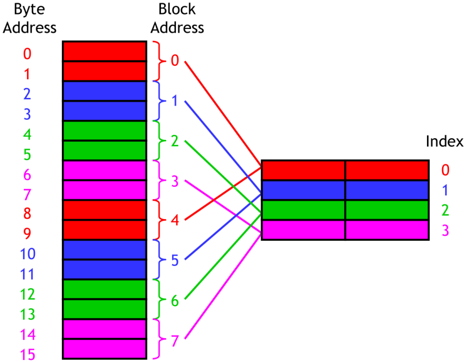
- Got a lot of questions so I don't think I explained this very well...
- Blocks vs bytes vs lines
- What are tags and how are the derived.
- Note that some confusion seems to come from the distinction between direct-mapped cache (shown in the book), fully associative mapping (most online examples), and set-associative mapping (the last lecture)
- Time was limited on adding these slides so pics shamelessly taken from: https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec15.pdf

Direct Mapping



- Each memory address maps to one cache block.
- Least significant bits of the address gives us the *tag* in the cache.
- Gives rise to conflict misses if we keep using say memory address 0000, 1000, 0000, 1000.
- But its quick and easy to implement (the quick is often the enemy of the good).

Block sizes



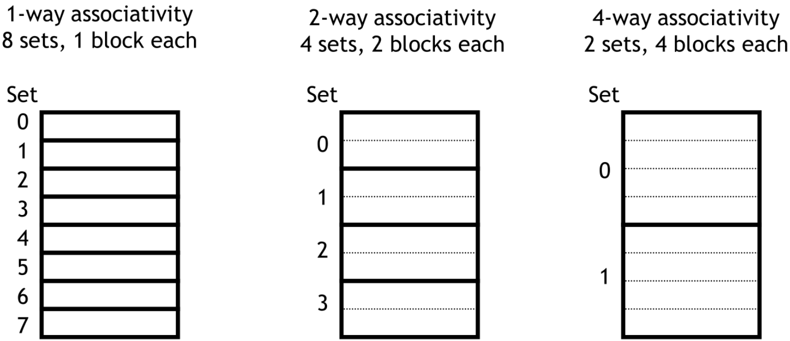
- We always store more than a single byte at a time in practice.
- Typically 4-32 KiBs, though this can differ by hardware.
- If we only stored single bytes, then spatial locality would mean nothing...
- Most examples present as though only a single byte for space/simplicity.
- Note the distinction between the memory address and the block address.

Fully associative Mapping

- Lets remove conflict misses by saying any block can be stored anywhere in cache.
- But now there's no clever indexing or tagging, so the entire memory address must be used as the tag.
- And we also now have to check the tag of every cache entirely
- This makes it very expensive to implement (both in hardware and clock cycles)
- So we don't tend to use it (hence no picture)

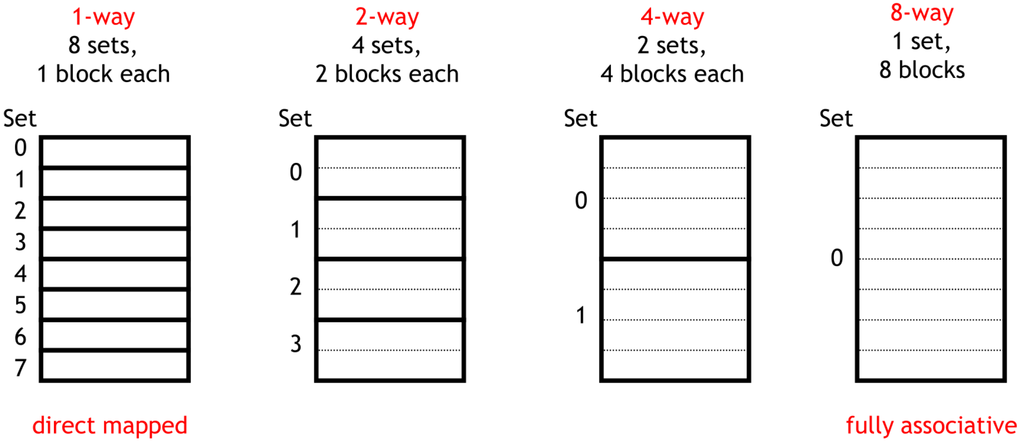
Set-associative Mapping

- Combination of direct and fully associative
- We group blocks into *sets*
- Each memory address maps to exactly one cache set, but can be placed in any block within the set
- Can be organised multiple ways.



Everything is Sets

Depending on how many sets we use, we can actually create anything from direct mapping to fully associative caches



Addressing within Sets

- Note this differs from the explanation in the book pg 400 (direct mapping).
- Look to pg 420 instead.

When $S = 2^n$, $B = 2^m$ we can easily split a w-bit address into *fields*, writing x_i for bit i.

$$\underbrace{x_{w-1} \cdots x_{m+n+1}}_{\text{tag}}\ \underbrace{x_{m+n} \cdots x_m}_{\text{set index}}\ \underbrace{x_{m-1} \cdots x_0}_{\text{block offset}}$$

Addressing within Sets

- Note this differs from the explanation in the book pg 400 (direct mapping).
- Look to pg 420 instead.

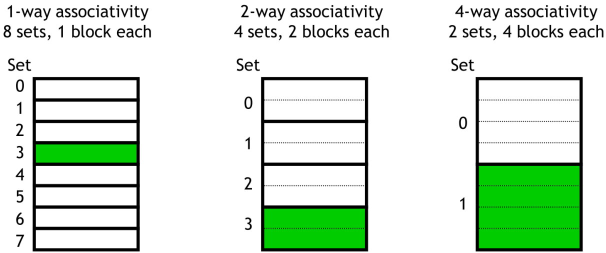
When $S = 2^n$, $B = 2^m$ we can easily split a w-bit address into *fields*, writing x_i for bit i.

$$\underbrace{x_{w-1} \cdots x_{m+n+1}}_{\text{tag}} \underbrace{x_{m+n} \cdots x_m}_{\text{set index}} \underbrace{x_{m-1} \cdots x_0}_{\text{block offset}}$$

- Lets work out where we might cache something stored at memory address 6195
- 6195 in binary is 00...0110000110011
- The offest is used to find a byte within the block, so must address all bytes in a block
- Assuming each block has 16 bytes, the lowest 4 bits are the offset
- Note only 16 bytes would be low for a real world example.

Addressing within Sets

- 6195 in binary is 00...0110000110011
- Now determining our set index depends on how many sets we have in our cache
  - For 1-way associative cache, the next 3 bits (011) are the set index.
  - For 2-way associative cache, the next 2 bits (11) are the set index.
  - For 4-way associative cache, the next bit (1) is the set index.
- Whatever is left is used as the tag to identify the block within the set (see previous lecture)



A Written Example (Hint hint)

Problem:
A computer uses 32-bit byte addressing. The computer uses a 2-way associative cache with a capacity of 32KB. Each cache block contains 16 bytes. Calculate the number of bits in the TAG, SET, and OFFSET fields of a main memory address.

A Written Example (Hint hint)

Problem:
A computer uses 32-bit byte addressing. The computer uses a 2-way associative cache with a capacity of 32KB. Each cache block contains 16 bytes. Calculate the number of bits in the TAG, SET, and OFFSET fields of a main memory address.
Solution:
Since there are 16 bytes in a cache block, the OFFSET field must contain 4 bits (24 = 16). To determine the number of bits in the SET field, we need to determine the number of sets. Each set contains 2 cache blocks (2-way associative) so a set contains 32 bytes. There are 32KB bytes in the entire cache, so there are 32KB/32B = 1K sets. Thus the set field contains 10 bits (210 = 1K).
Finally, the TAG field contains the remaining 18 bits (32 - 4 - 10). Thus a main memory address is decomposed as shown below.

TAG = 18 bits	SET = 10 bits	OFFSET = 4 bits

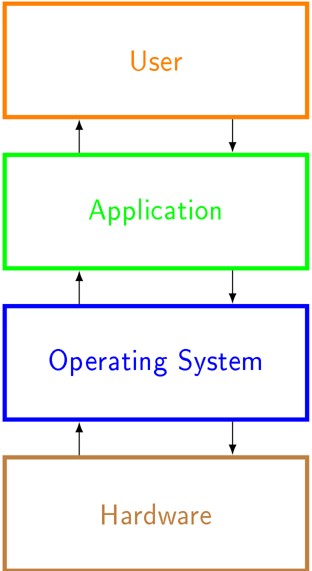
Introduction to Operating Systems

David Marchant
Based on slides by Troels Henriksen

2022-09-27

Incorporating material by Mathias Payer
(`https://github.com/HexHive/OSTEP-slides`).

What is an operating system?



OS is middleware between applications and hardware.

- Provides standardized interface to resources.
- Manages hardware.
- Orchestrates currently executing processes.
- Responds to resource access requests.
- Handles access control.

Why study operating systems?

Inspirational

- One of the most potent *abstractions* in computing.
  - Each process thinks it has machine to iself.
  - Controlled communication.
  - Abstracts over hardware differences.

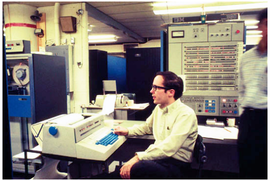
Practical

- You almost always use an operating system.
- Its performance characteristics are important to understand.
- It often determines what is fundamentally possible.

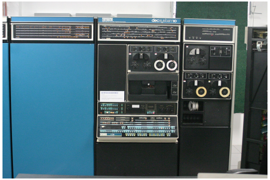
They are where the magic happens.

In the old days

Each brand of machine would have its own operating system.



IBM System/360 running OS/360 (man not part of computer)

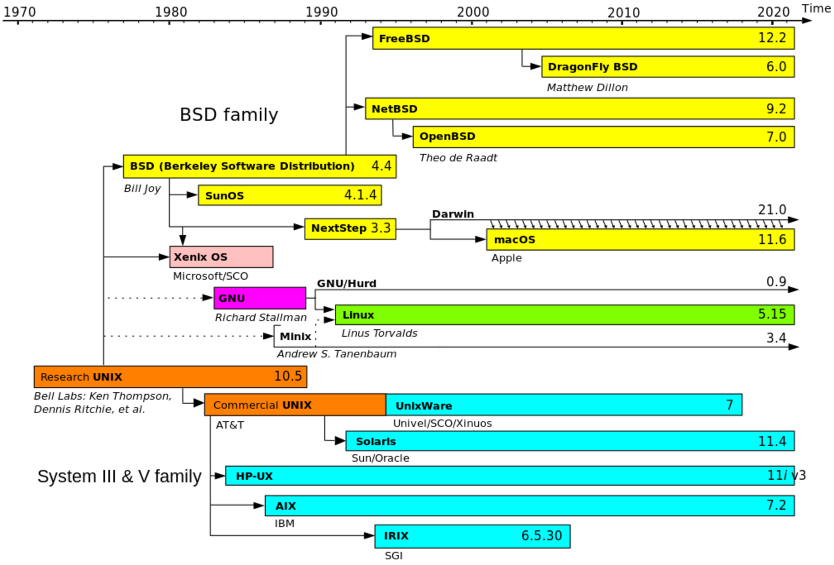


DEC PDP-10 running TENEX

VAX 11/780 running VMS

- PDP-11 running early UNIX, written in C.
- Developed at AT&T, who were banned from selling UNIX.
- Shared it (almost) freely with others, who *ported* UNIX to every machine under the sun.
- Unix was *popular*, *good enough*, and *cheap*.

Unix lifetime



What is Unix

What?

- Unix is a family of operating systems derived from original UNIX developed in the 1970s by Ken Thompson and Dennis Ritchie.
- Most modern operating systems are heavily influenced by Unix (even Windows).
- Many operating systems are *direct descendants*: Linux, iOS, macOS, the BSDs, etc.

Why?

- We teach Unix because it is *simple* and *representative* of modern systems.
- We use Unix designs for all examples.
- ...does not mean Unix is always *good* design.

The purpose of operating systems

Kernel and Processes

System calls

Process management in Unix

Main takeaways

More than one process

- So far we've made strictly *sequential* code that does one thing after another until it is done.
- This is a useful (and not inaccurate) concept for how our programs run.
- But is also not entirely true...

More than one process

- So far we've made strictly *sequential* code that does one thing after another until it is done.
- This is a useful (and not inaccurate) concept for how our programs run.
- But is also not entirely true...
- Modern computers run many processes at once, to manage different devices and services.
- If all processes ran entirely sequentially then we'd have to wait for the clock to finish before we could browse the internet
- A key abstraction the OS provides us is the concept of a process, to allow different sequential systems to be interleved.

Processes contra programs

Program: is a file containing code. Stateless and **dead**.

Process: a running *instance* of a program. The same program can be running in multiple instances. Stateful and **alive**.

They are not the same thing, although we informally often say *program* when we really mean *process*.

Processes contra programs

Program: is a file containing code. Stateless and **dead**.

Process: a running *instance* of a program. The same program can be running in multiple instances. Stateful and **alive**.

They are not the same thing, although we informally often say *program* when we really mean *process*.

- Operating systems manage *processes*.
  - Switching between multiple *concurrent* processes.
  - Handling process termination.
  - Starting new processes (perhaps from a given *program file*).

The kernel

- Technically, *operating systems* encompass lots of parts: shell, GUI, C library, maybe bundled applications (mail reader etc).
- In this course, when we say *operating system* we really mean the **kernel**.

The kernel

- Technically, *operating systems* encompass lots of parts: shell, GUI, C library, maybe bundled applications (mail reader etc).
- In this course, when we say *operating system* we really mean the **kernel**.

Kernel

Always-resident code that services requests from the hardware and manages processes.
It is not a process.

- The kernel uses the same CPU, memory, and other hardware as ordinary code.
- When running process code, the CPU is in *unprivileged state*, and many operations are restricted (e.g. access to hardware devices).
  - When an *interrupt* happens, the CPU switches to *privileged state* and jumps to kernel code, which handles it and then resumes the previously running process.
    - Think of it like a sudden and unplanned procedure call.
  - Interrupts can be outside events (keyboard press, network traffic) or special instructions (invalid memory accesses, *system calls*).

Virtualising the CPU

Goal Give each process the illusion of exclusive CPU access.

Reality: the CPU is a shared resource.

Virtualising the CPU

Goal Give each process the illusion of exclusive CPU access.

Reality: the CPU is a shared resource.

Solution: **context switching**

- Only one process gets to run at a time.
- ...but we regularly switch between available processes.
- Doing this often and rapidly creates the illusion of simultaneous execution.

Context switching

Intuition Pausing a process, saving its entire *state*, then resuming some other process based on its saved state.

Context switching

Intuition Pausing a process, saving its entire *state*, then resuming some other process based on its saved state.

So what do we need to save?

1. All registers, including control registers.
2. Contents of memory.

Context switching

Intuition Pausing a process, saving its entire *state*, then resuming some other process based on its saved state.

So what do we need to save?

1. All registers, including control registers.
2. Contents of memory.

So when do we do this?

- Regular *timer interrupts* transfer control to the kernel, whose *scheduler* decides the next process to run.
  - Scheduling is a big and interesting topic that we don't have time to go into.

We've seen this before...

- Recall from our time with Assembler we discussed procedures
- In that context we presented the shift in control as between different function calls
- The same principle applies here, control is shifted from one process to another, though *usually* lacks a direct communication between them.
- But the same principle applies, memory must be saved and maintainted between switches.
- But in this context, EVERYTHING must be saved.

System calls

- Only the kernel has direct access to hardware and system memory.
- Whenever we want to do IO we have to perform a *system call*.

System calls

A request by a process that the kernel carries out some operation on its behalf.

- Much like a function call, but implemented very differently.

System calls in RISC-V

- The `ecall` (*environment call*) instruction transfers control to the kernel.
  - Kernel then inspects registers (mostly `a0-a7`) to see what it has been asked to do.
  - Specific interpretation varies between operating systems.
  - System call identified by a number.

System calls in RISC-V

- The `ecall` (*environment call*) instruction transfers control to the kernel.
  - Kernel then inspects registers (mostly `a0`-`a7`) to see what it has been asked to do.
  - Specific interpretation varies between operating systems.
  - System call identified by a number.

In RARS

`https://github.com/TheThirdOne/rars/wiki/Environment-Calls`

- System call number passed in `a7`.
- Excerpt;

Human name	**Number**	**Description**	**Reads**	**Writes**
`PrintInt`	1	Prints int to console.	`a0`	
`ReadInt`	5	Reads int from console.		`a0`

How does that work?

- These system calls are both very basic, but also 'higher level' than assembly calls;
- Recall that most input/output really is just file manipulation.
- Many of these OS calls are really just shortcuts to reading or writing to certain files.

Used in A0's `io.s` (Not any more :P)

```
read_loop:
        beq     t1, t2, read_done
        li      a7, 5
        ecall                           # read int
        sw      a0, 0(t0)
        addi    t0, t0, 4               # next output addr
        addi    t1, t1, 1               # increment count
        jal     zero, read_loop
```

System calls in C

- C exposes system calls as functions.
  - Internally use `ecall` instruction or architecture-specific equivalent.
- Not observably distinct from other C functions, except typically more primitive and tedious to use.

System calls in C

- C exposes system calls as functions.
  - Internally use `ecall` instruction or architecture-specific equivalent.
- Not observably distinct from other C functions, except typically more primitive and tedious to use.

Example

```
// system calls
int open(const char *pathname, int flags);
ssize_t write(int fd, const void *buf, size_t count);

// stdio functions
FILE *fopen(const char *pathname, const char *mode);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

File descriptors

```
int open(const char *pathname, int flags);

ssize_t write(int fd, const void *buf, size_t count);
```

- The `int` returned by `open` is a *file descriptor*.
- Has no significance in itself, but allows the kernel to recognise the open file when passed to other system calls.
  - Typically an index into some kernel-side table.
  - Such values are known as *handles*.
- Passing complex data structures or pointers between *kernel space* and *user space* is annoying and fragile, so we usually use numeric identifiers instead.

Basic principles

- Each process in Unix has a *process ID* (PID).
- Each process has a *parent*.
  - ...except the *initial process* (`init`) with PID 1.
- A process may have multiple *children*.
- Implies processes are organised as a *tree* (`pstree` command shows it).
- **Creating processes:** `fork()`.
- **Terminating current process:** `exit()`.
- **Loading program code from disk into current process:** `exec()`.
- **Waiting for a specific child to die:** `waitpid()`.
- **Getting PID of running process:** `getpid()`.

Main takeaways

- Hardware provides *mechanisms* such as interrupts, privileged/unprivileged mode, and *virtual memory* (next week).
  - Kernels implement *policy* and *abstractions* on top.
- Processes are a *purely virtual concept*—CPU has no idea what they are.
- Processes are *isolated* from each other.
- Processes can only directly interact with the outside world through *system calls*, mediated by the kernel.