

# LECTURE 6

# APPLICATION BINARY INTERFACES (ABI)

- most Windows laptops, Linux laptops and pre-M1 Macs share the same ISA: [x86\\_64](#)
- iPhones, Android phones, M1 and M2 Macs share the same ISA: [AArch64](#)

**Q:** Why, then, do applications need to be recompiled separately for each platform?  
e.g. iPhone vs. Android phone

**A:** Because platforms have different OSs and ABIs.

# What is an ABI?

An application binary interfaces (ABI) defines:

- file format for
  - object files
  - dynamically-linked files (shared objects / dll)
  - and executable files
- convention for function calls
- convention for system calls

It is called **binary** because it is independent of the language in which applications are written  
(i.e. it is related to the machine code, not to the source code)

# ABI: function calls (x86\_64)

```
#include <stdio.h>

int main()
{
    puts("Hello\n");
    return 0;
}
```

clang / Linux / x86\_64

```
main:
    push    rax
    lea    rdi, [rip + .L.str]
    call   puts@PLT
    xor    eax, eax
    pop    rcx
    ret

.L.str:
    .asciz "Hello\n"
```

MSVC / Windows / x86\_64

```
_DATA SEGMENT
$SG9391 DB      'Hello', 0aH, 00H
_DATA ENDS

main PROC
$LN3:
    sub    rsp, 40
    lea    rcx, OFFSET FLAT:$SG9391
    call   puts
    xor    eax, eax
    add    rsp, 40
    ret    0
main ENDP
```

# ABI: function calls (AArch64)

```
#include <stdio.h>

int main()
{
    puts("Hello\n");
    return 0;
}
```

clang / MacOS / [AArch64](#)

```
main:
    stp    x29, x30, [sp, #-16]!
    mov    x29, sp
    adrp   x0, .L.str
    add    x0, x0, :lo12:.L.str
    bl     puts
    mov    w0, wzr
    ldp    x29, x30, [sp], #16
    ret

.L.str:
    .asciz "Hello\n"
```

MSVC / Windows / [AArch64](#)

```
IMPORT |puts|

|main| PROC
|$LN3|

    stp    fp,lr, [sp, #-0x10]!
    mov    fp, sp
    adrp   x8, |$SG4901|
    add    x0, x8, |$SG4901|
    bl     puts
    mov    w0, #0
    ldp    fp,lr, [sp], #0x10
    ret

ENDP
```

Try it for yourself:  
[godbolt.org](http://godbolt.org)

# PORTABLE CODE



How do we ship code that work across all platforms?

## Option 1: interpreters

- use interpreted languages, ship source
  - Python, Javascript, ...
- languages that compile to virtual machine code
  - ship VM code
  - optionally, ship VM interpreter
  - Java, C#

## Option 2: multiple compilations

- compile one executable on each platform
- in some cases, cross-compilation is possible
  - MacOS → iOS
  - Linux → Android

What if we cannot (or do not want to) recompile?

## Option 3: Translation

**Use case:** same OS, different ISA

- Translation is a form of compilation
- From machine code
- To machine code (of a different ISA)

Example: Apple Rosetta 2 translates `x86_64` into `AArch64`

# Option 4: Compatibility layers

**Use case:** different OSs, same ISA

- add OS support for a foreign ABI
  - foreign file formats (for objects, DLLs and executables)
  - foreign convention for system calls
- add libraries for foreign ABI
  - foreign convention for function calls

Examples:

- Wine allows running Windows apps on Linux.
- WSLv1 allows running Linux apps on Windows.

## Option 5: emulation

- an emulator is an **interpreter** for machine code (e.g. QEmu)
- much slower than running the code
- JIT can mitigate slowness, to some extent
- typically, a full-blown **operating system runs inside** the **interpreter!**

**LOAD** 0.3 **PROCS/MIN**

**CPU0** 0% **USR/NICE/SYS/FREE**

**MEM** 1.0G **USED+SHAR/BUFF/CACHE**

**SWAP** 0 **USED/FREE**

**PAGE** 0 **IN/OUT/IDLE**

**NET** 0 **IN/OUT/IDLE**

**DISK** 133K **READ/WRITE/IDLE**

**INTS** **INTs (0-228)**

```

stefan@tuxstation:~/...-Qemu/ReactOS-Qemu
stefan@tuxstation:~$ cd os/ReactOS-Qemu/ReactOS-Qemu/
stefan@tuxstation:~/os/ReactOS-Qemu/ReactOS-Qemu$ qemu -hda c.img -m 256

```

**QEMU**

My Computer    Command Prompt

Reactos Explorer

File View Window Tools Help

Explore Shell Web NT Obj

bin

system32

lake.bmp

inf

explorer...

logo.bmp

media

explorer...

regedit.exe

file://C:\

Command Prompt

```

C:\ReactOS>dir
Volume in drive C has no label.
Volume Serial Number is 933E-20E7

Directory of C:\ReactOS

08/27/2006  05:25p  <DIR>      .
08/27/2006  05:25p  <DIR>      ..
08/27/2006  05:25p  <DIR>      system32
08/27/2006  05:25p  <DIR>      media
08/27/2006  05:25p  <DIR>      bin
08/27/2006  05:25p  <DIR>      inf
08/27/2006  05:25p                315,392 regedit.exe
08/27/2006  05:25p            3,588,096 explorer.exe
08/27/2006  05:25p                1,115 explorer-cfg-template.
08/27/2006  05:30p            921,654 logo.bmp
08/27/2006  05:31p            921,654 lake.bmp
               5 File(s)    5,747,911 bytes
               6 Dir(s)   163,708,928 bytes

C:\ReactOS>

```

Start

1 2 3 4

Command Prompt    Reactos Explorer

1:18 PM

stefan@tuxstation:~/...-Qemu/Rea    The GIMP

QEMU

xosview@tuxstation

15:17

Freitag

13.09.2006



xos1

LOAD	0.3	PROCS/MIN
CPU0	0%	USR/PRV/SYS/FREE
MEM	1.0G	USED+SHAR/BUFF/CACHE
SWAP	0	USED/FREE
PAGE	0	IN/OUT/IDLE
NET	0	IN/OUT/IDLE
DISK	133K	READ/WRITE/IDLE
INTS		INTs (0-228)

```
stefan@tuxstation:/...-Qemu/ReactOS-Qemu
stefan@tuxstation:~$ cd os/ReactOS-Qemu/ReactOS-Qemu/
stefan@tuxstation:~/os/ReactOS-Qemu/ReactOS-Qemu$ qemu -hda c.img -m 256
```

QEMU

ReactOS Explorer

File View Window Tools Help

Explore Shell Web NT Obj

bin system32 lake.bmp

inf explorer... logo.bmp

media explorer... regedit.exe

file://C:\

Command Prompt

```
C:\ReactOS>dir
Volume in drive C has no label.
Volume Serial Number is 933E-20E7

Directory of C:\ReactOS

08/27/2006  05:25p    <DIR>          .
08/27/2006  05:25p    <DIR>          ..
08/27/2006  05:25p    <DIR>          system32
08/27/2006  05:25p    <DIR>          media
08/27/2006  05:25p    <DIR>          bin
08/27/2006  05:25p    <DIR>          inf
08/27/2006  05:25p                315,392 regedit.exe
08/27/2006  05:25p                3,588,096 explorer.exe
08/27/2006  05:25p                 1,115 explorer-cfg-template.
08/27/2006  05:30p                 921,654 logo.bmp
08/27/2006  05:31p                 921,654 lake.bmp
           5 File(s)      5,747,911 bytes
           6 Dir(s)     163,708,928 bytes

C:\ReactOS>
```

Start

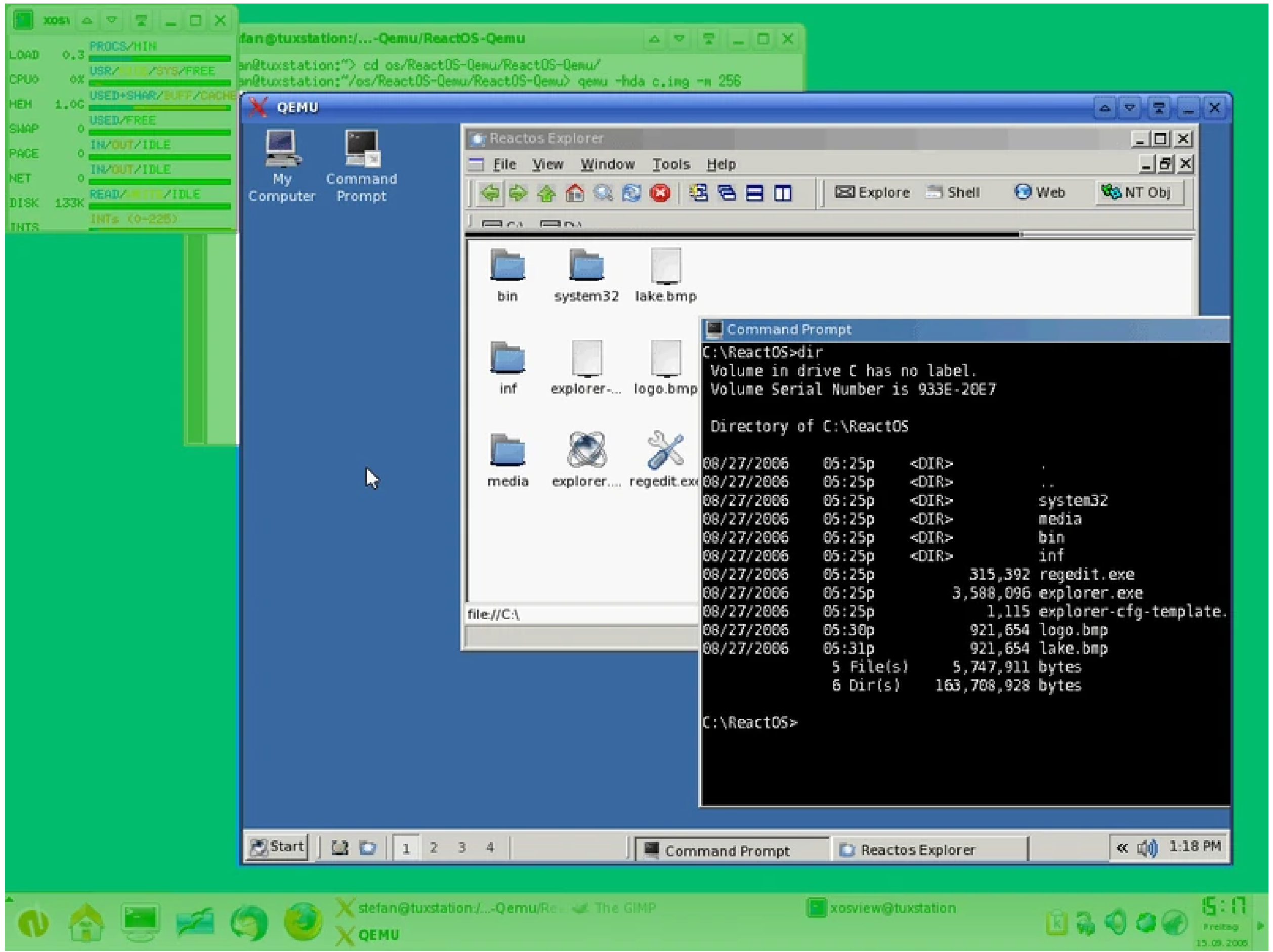
1 2 3 4

Command Prompt ReactOS Explorer

15:17 Freitag 13.09.2006

stefan@tuxstation:/...-Qemu/Re The GIMP xosview@tuxstation

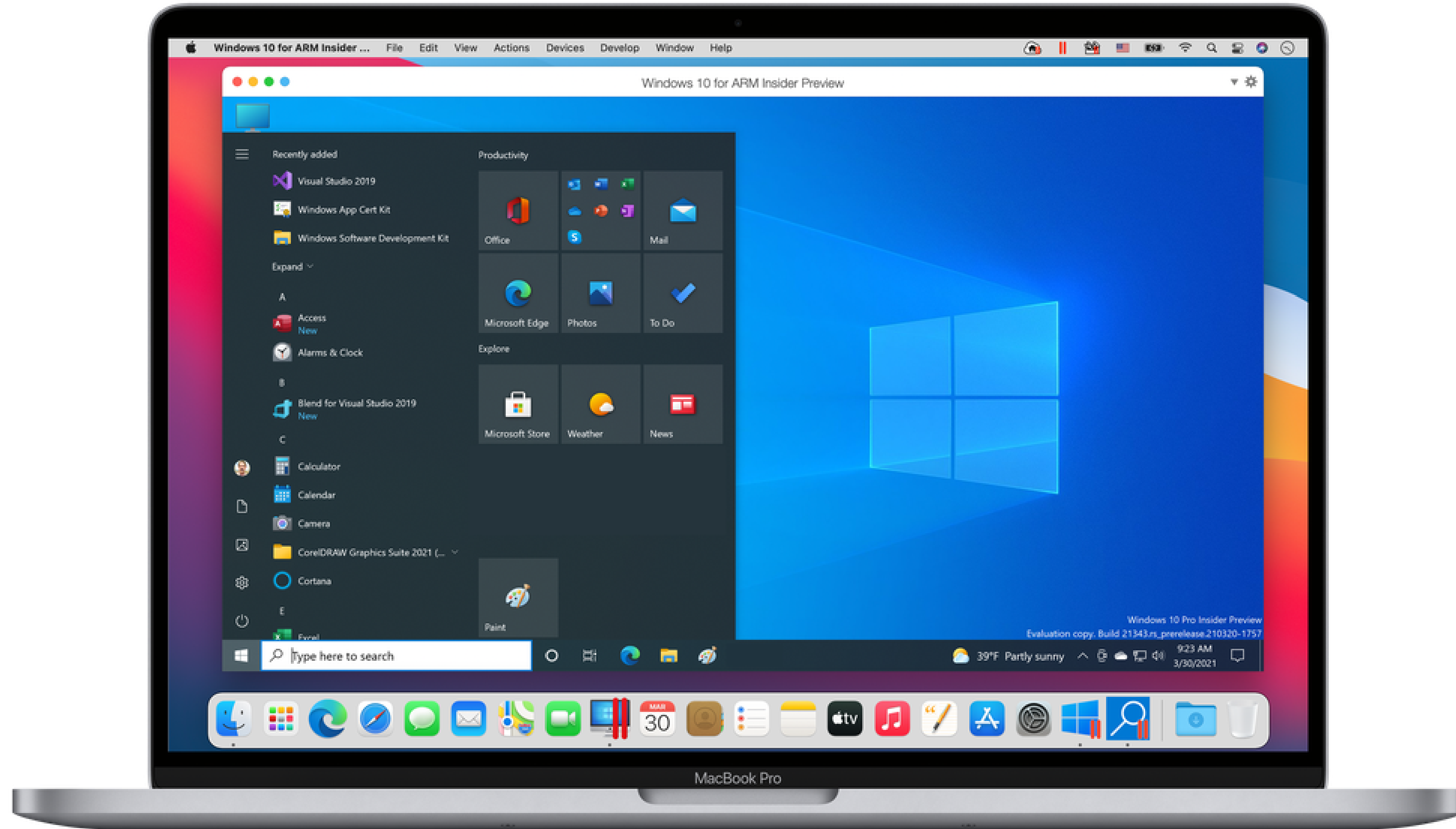
QEMU



## Option 6: virtualization

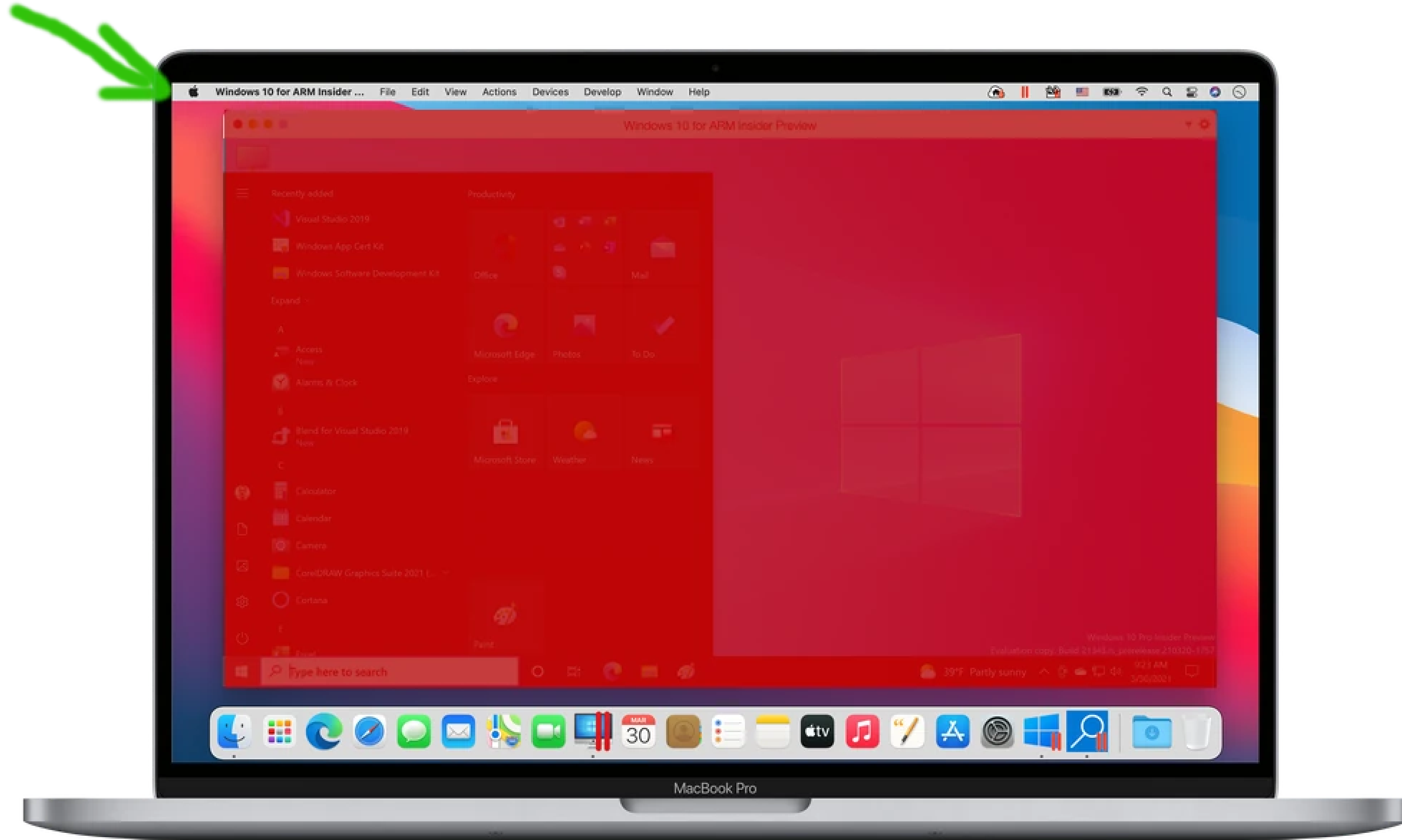
- virtualization is essentially hardware-assisted emulation  
(e.g. Xen, KVM, VirtualBox, VMWare, Apple Parallels, WSLv2)
- virtualized software must target the **same ISA** as hardware
- like emulation, runs a full-blown **operating system**

# Example: Apple Parallels



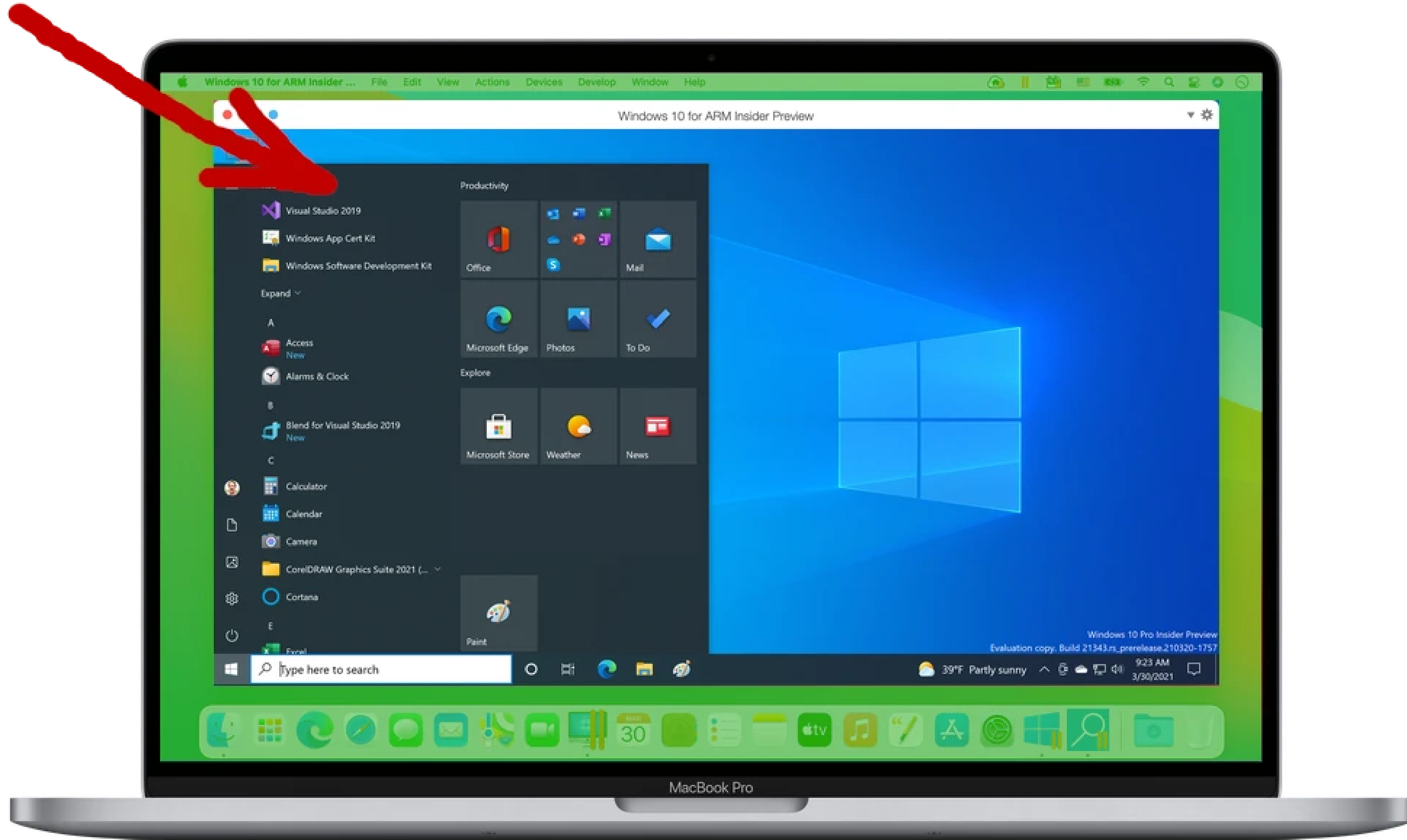


# Host OS





# Guest OS







# Definitions

- The *hypervisor* is the software that manages the **guest OS**.
- It can be the **host OS** itself (“Type 1”: Xen, KVM)
- It can be a process within the **host OS** (“Type 2”: Apple Parallels)

Virtualization is mainly deals with security:

Let **guest OSs** believe they have direct access to hardware...

... but every hardware access is tightly controlled by the **hypervisor**

Virtualization is the main technology enabling “cloud computing”.

- Amazon Web Services runs **Xen**
- Google Cloud Platform runs **KVM**
- Customers rent a virtual machine in a datacenter
  - They can connect (remotely) to this machine
  - It runs their (**guest**) OS of choice
  - It acts as if it was physical hardware

# Option 7: containers

**Use case:** Same ISA, same kernel, different OS.

- Containers are a lightweight form of virtualization.
- The **host**'s kernel also acts as a kernel for the **guest**.
- Mainly: filesystems, libraries and applications are separated.

Examples:

- A Debian Linux **guest** on a Fedora Linux **host**
- A Debian 11 Linux **guest** on a Debian 12 **host**
- A Debian 12 **guest** with specific libraries installed, on a Debian 12 **host**

# APPLICATION PROGRAMMING INTERFACES (API)

# Definition

An API defines how a library (or any other service) is to be used.

# Library API

```
FILE *fopen(const char *path, const char *mode);
```

```
open(file, mode='r', buffering=- 1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
```

# Web API

```
GET https://www.google.com/search?q=<query>
```

Example:

```
google-chrome https://www.google.com/search?q=Software%20Engineering
```



```
GET https://cloudflare.com/cdn-cgi/trace
```

Example:

```
curl -4s "https://cloudflare.com/cdn-cgi/trace"
```

```
PUT https://api.cloudflare.com/client/v4/zones/{zone_idenfier}/dns_records/{identifier}
```

## Example:

```
curl --request PUT \  
  --url https://api.cloudflare.com/client/v4/zones/zone_idenfier/dns_records/identifier \  
  --header 'Content-Type: application/json' \  
  --header 'X-Auth-Email: ' \  
  --data '{  
    "content": "198.51.100.4",  
    "name": "example.com",  
    "proxied": false,  
    "type": "A",  
    "comment": "Domain verification record",  
    "tags": [  
      "owner:dns-team"  
    ],  
    "ttl": 3600  
  }'
```

# APIs and portability

- many APIs are cross-platform
  - C standard library
  - Almost all Python modules
  - Qt, Electron, Flutter, ... (frameworks for GUI applications)
  - WEB APIs only depend on an internet connection
- some are specific to a platform
  - Windows UI Library, MacOS Cocoa

# DEPENDENCIES

- your code requires libA version  $\geq 1.1$ , lib B version  $\geq 4.5$ 
  - lib B version 4.5 requires libX version 2.0 and libA version 0.8
  - lib B version 4.7 requires libX version 2.0 and libA version 1.1
  - lib B version 4.6 requires libX version 2.0 and libA version 2.0
  - lib X version 2.0 requires libA version  $\leq 1.9$

How do we install all this?

Which version do we install?

# Package managers

Package managers solve this problem for you.

They can solve it...

- at the OS level:
  - MacOS: `brew install <package>`
  - Debian/Ubuntu Linux: `apt-get install <package>`
  - Fedora/Suse Linux: `dnf install <package>`
- at the language level:
  - Python: `pip install <module>`
  - JavaScript/Node: `npm install <package>`

# Limitations

- package selection may be limited (packaging is labor-intensive)
- security and trust

# TUTORIAL



# TREATING INTEGERS AS STRINGS OF BITS

## Binary OR

```
>>> x = 0b110000 | 0b000011
```

```
>>> x = 0b110000 \  
      | 0b000011
```

```
>>> x = 0b110000 \  
      | 0b000011  
>>> f"{x:06b}"      # format(x, "06b")
```

```
>>> x = 0b110000 \
      | 0b000011
>>> f"{x:06b}"          # format(x, "06b")
'110011'
```

## Binary AND

```
>>> x = 0b111100 \  
      & 0b001111  
>>> f"{x:06b}"  
'001100'
```

## Binary XOR

```
>>> x = 0b101010 \  
      ^ 0b110011  
>>> f"{x:06b}"  
'011001'
```

## Binary shift left

```
>>> x = 0b000110 << 2  
>>> f"{x:06b}"  
'011000'
```



```
>>> x = 1 << 16
>>> f"{x:b}"
'100000000000000000'
>>> x
65536
>>> 2**16
65536
```

## Binary shift right

```
>>> x = 0b011000 >> 2  
>>> f"{x:06b}"  
'000110'
```

