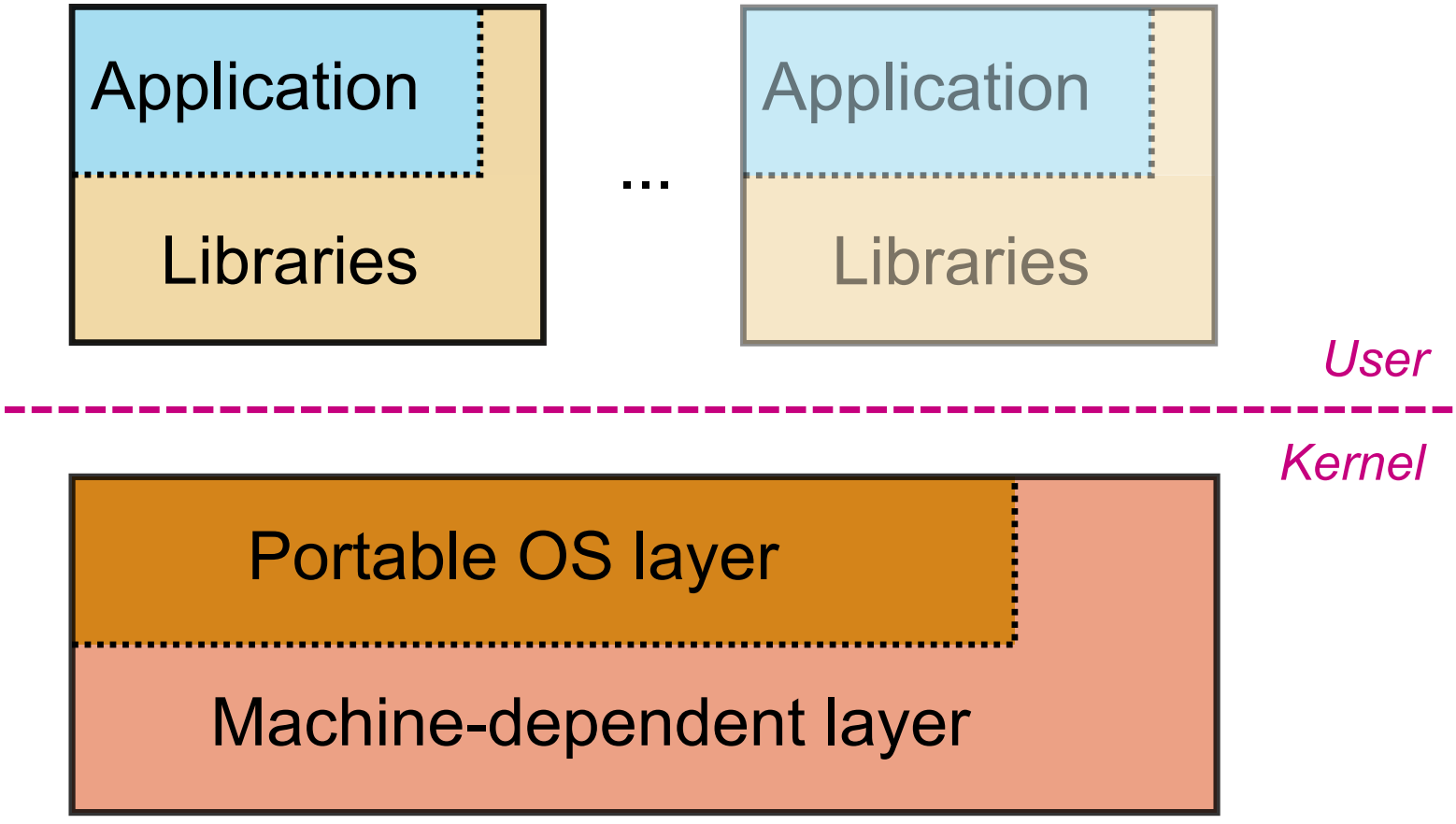
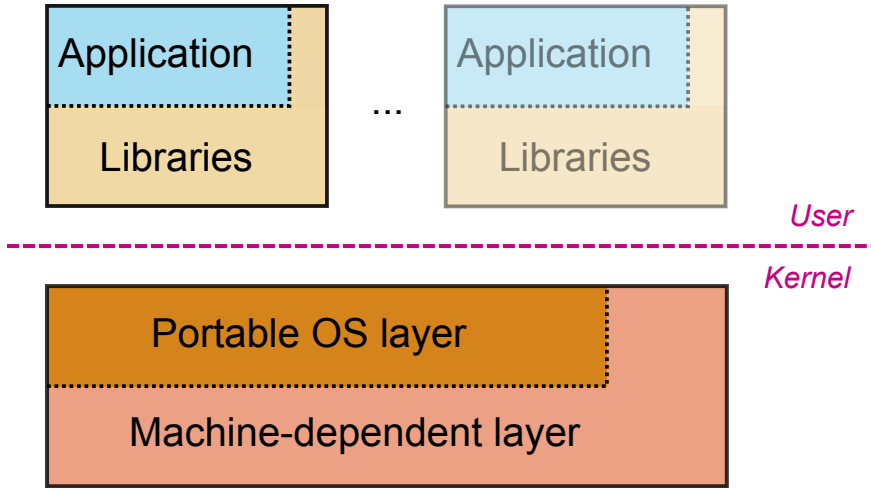
OS Structure

OS Layers: overview

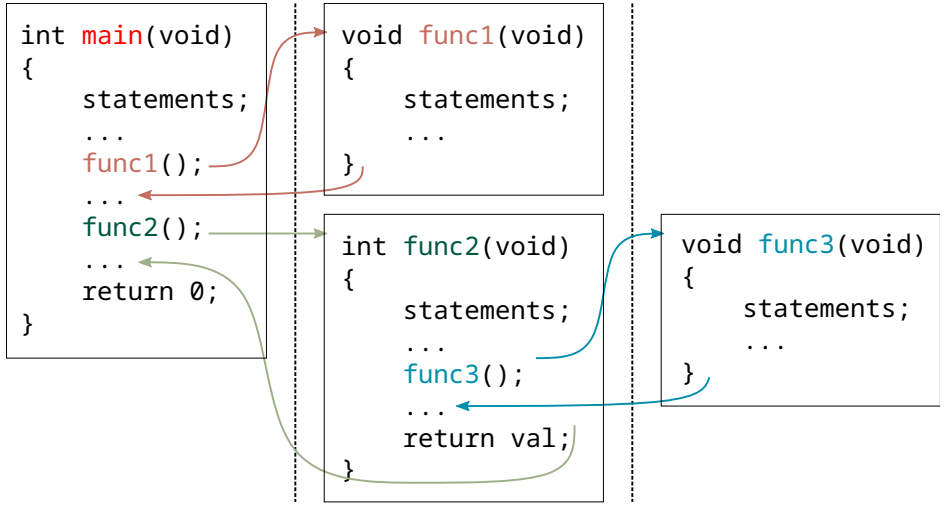


OS Layers: details

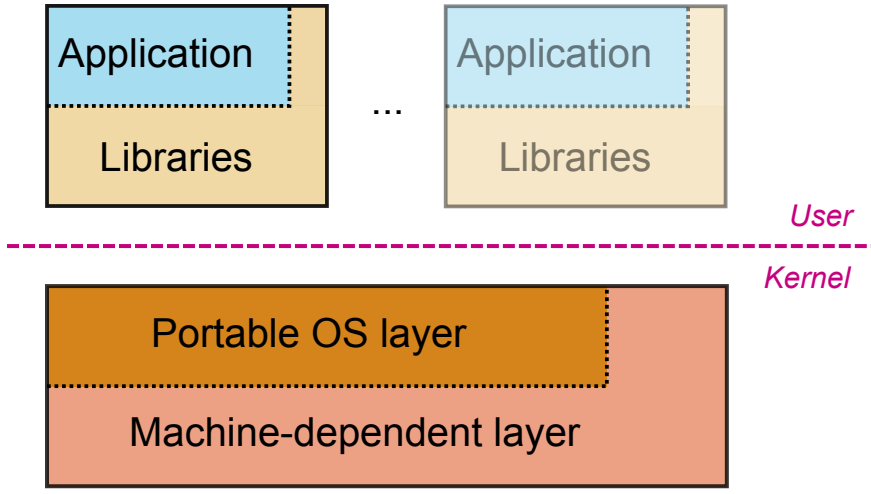


Application(s)

- User function calls
- Written by programmers
- Compiled by programmers



OS Layers: details

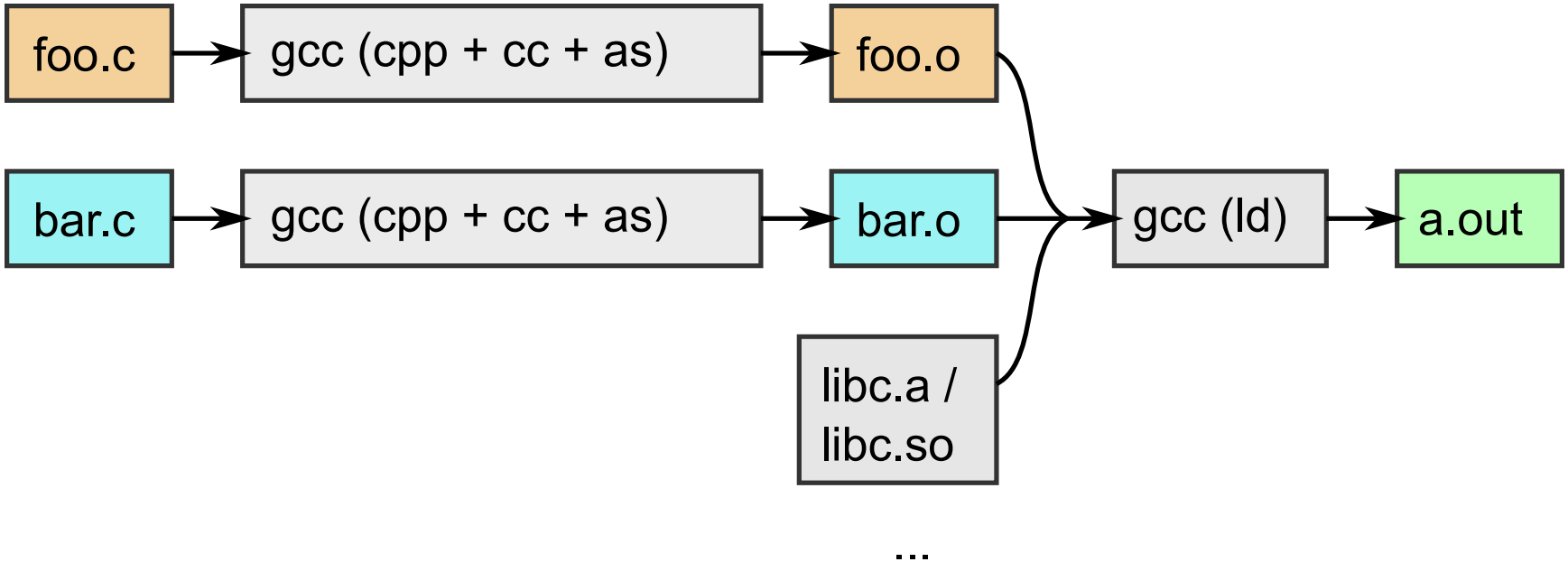


Libraries

- Definition
  - Via standard headers (e.g. `stdio.h`, `stdlib.h`, `math.h`)
  - Used like regular functions
- Declaration
  - Pre-compiled objects (e.g. `libc.so.6`, `libc.a`, `libm.so`)
  - Input to linker (e.g. `gcc -lc -lm`)
- Code inclusion
  - Included in executable directly
  - Or *resolved* at load-time

All about applications + libraries

Application compilation



GCC can pre-process, compile, assemble and link together

- Preprocessor (`cpp`) transform program before compilation
- Compiler (`cc`) compiles a program into assembly code
- Assembler (`as`) compiles assembly code into relocatable object file
- Linker (`ld`) links object files into an executable

All about applications + libraries

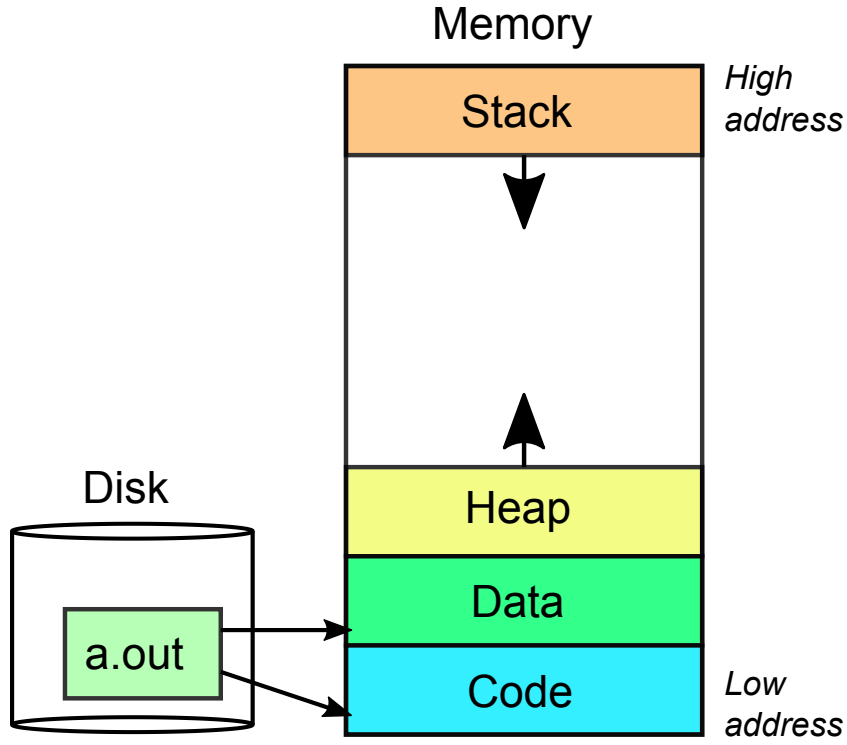
Application loading 101

Directly from executable

- Code (a.k.a. text)
  - Instructions
- Data
  - Global variables

Created at runtime

- Stack
  - Local variables
- Heap
  - `malloc()` area



Segment characteristics

- Separate code and data for permissions and sharing reasons
- Maximize space for stack/heap

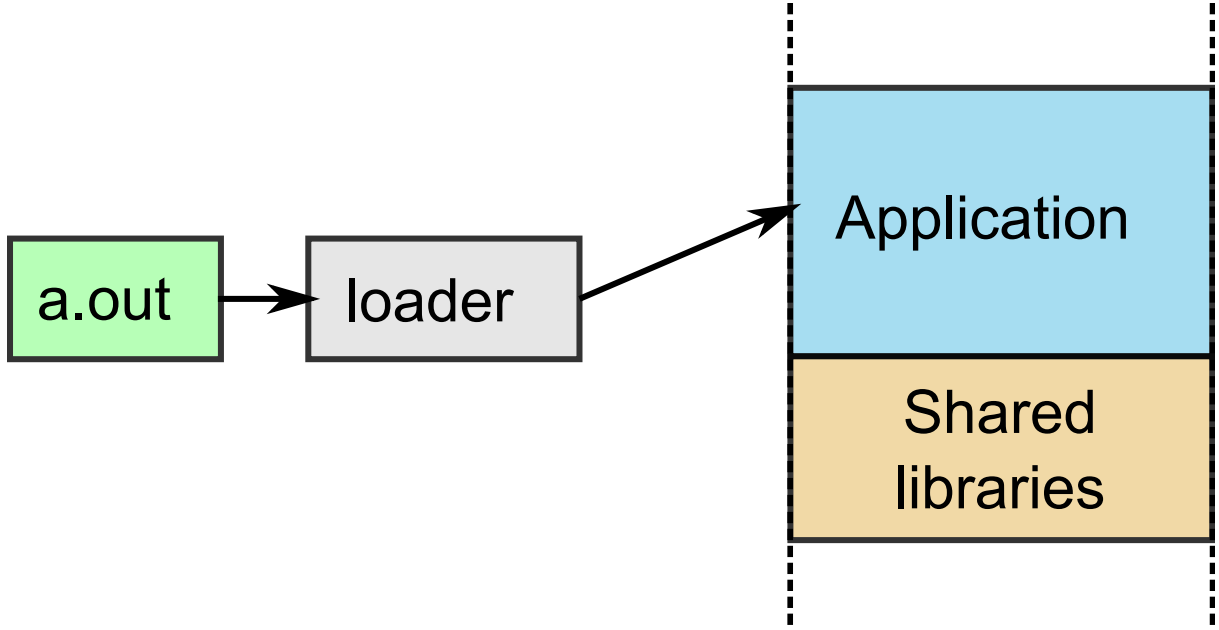
All about applications + libraries

Application dynamic loading

By default, *loader* dynamically prepares application for execution

- (unless compiled with `-static`)
- Loaded before the application by the kernel
- Read the executable file, and lays out the code, data (using syscalls)
- Dynamically links to shared libraries

```
$ ldd a.out
        libc.so.6 => /usr/lib/libc.so.6 (0x00007fab5382b000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fab53bc9000)
```



All about applications + libraries

Static and dynamic libraries

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    printf ("%f\n", cos(2.0));
    return 0;
}
```

Dynamic

```
$ gcc main.c -lm
$ ldd a.out
    linux-vdso.so.1
    libm.so.6
    libc.so.6
    /lib64/ld-linux-x86-64.so.2
$ ./a.out
-0.416147
```

- Math code will be loaded upon execution, by *loader*

Static

```
$ gcc main.c /usr/lib/libm-2.28.a
$ ldd a.out
    linux-vdso.so.1
    libc.so.6
    /lib64/ld-linux-x86-64.so.2
$ ./a.out
-0.416147
```

- Math code is inserted as part of the executable

All about applications + libraries

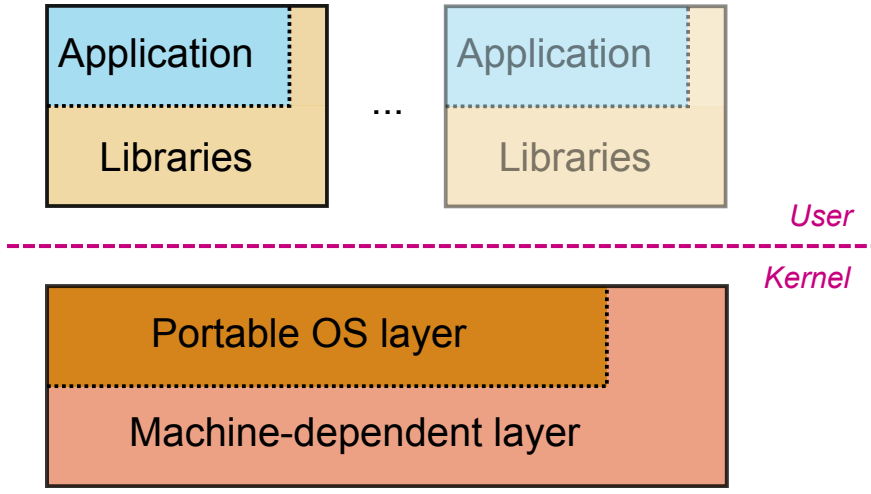
Dynamically loaded libraries

```c
#include <dlfcn.h>
#include <stdio.h>

int main(void)
{
    void *handle;
    double (*cosine)(double);
    char *error;

    handle = dlopen ("/lib/libm.so.6",
                     RTLD_LAZY);
    if (!handle)
        return 1;

    cosine = dlsym(handle, "cos");
    if (!cosine)
        return 1;

    printf ("%f\n", (*cosine)(2.0));

    dlclose(handle);
    return 0;
}
```

```
$ gcc main.c -ldl
$ ldd a.out
    linux-vdso.so.1
    libdl.so.2
    libc.so.6
    /lib64/ld-linux-x86-64.so.2
$ ./a.out
-0.416147
```

- Math code is neither part of the executable, nor is it referenced
- Loaded at runtime only if specific code is executed
  - Handle case where library doesn't exist
  - Great for plugins

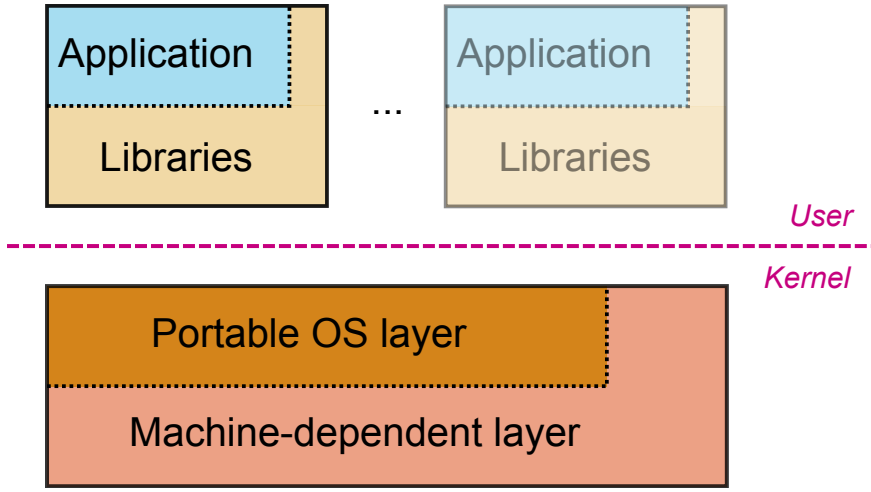
OS Layers: details



Portable OS layer

- Implementation of most system calls
- *High-level* kernel code (i.e., top-half) for most subsystems
  - Virtual File System (VFS)
  - Inter-Process Communication (IPC)
  - Process scheduler
  - Virtual memory
  - Networking, Sound, Cryptography, etc.

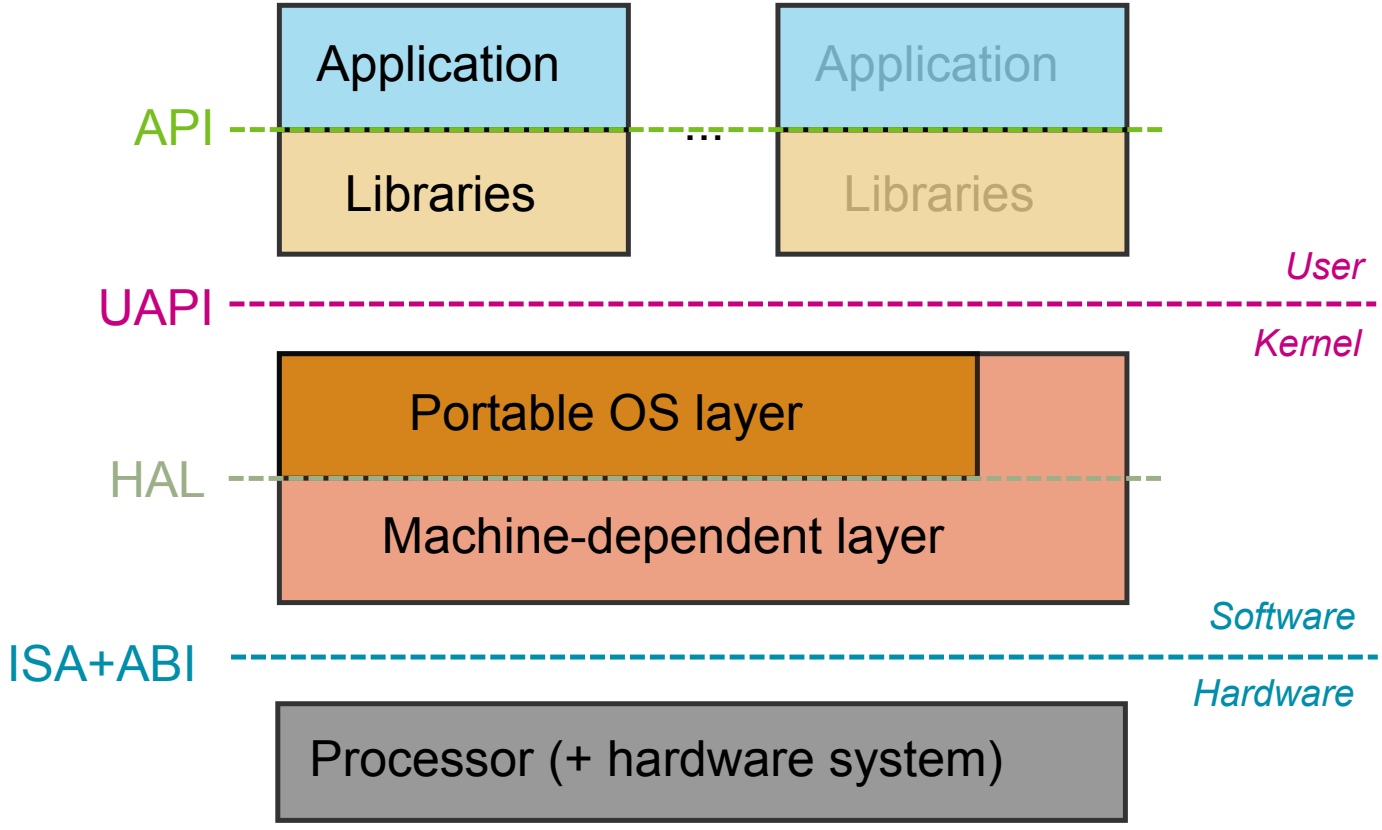
OS Layers: details



Machine-dependent layer

- Bootstrap
- System initialization
- Exception handler (exceptions, interrupts and syscalls)
- I/O device drivers
- Memory management
- Processor mode switching
- Processor management

OS Interfaces



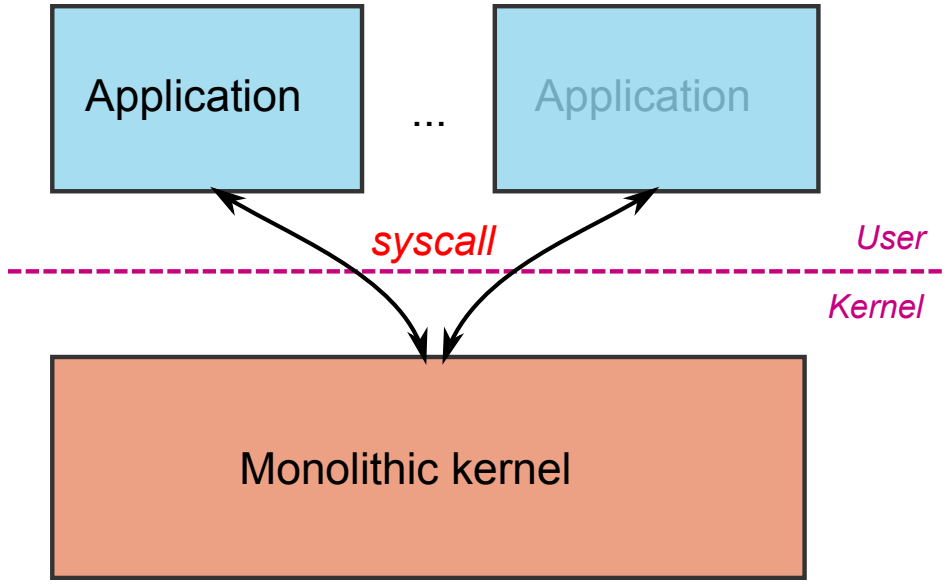
- **API** (Application Programming Interface): interface between pieces of code
- **UAPI** (User API): syscall interface between apps and kernel
- **HAL** (hardware-abstraction layer), interface inside kernel between arch-independent code and arch-dependent code
- **ISA** (Instruction Set Architecture): list of processor instructions
- **ABI** (Application Binary Interface): interface between code and processor

Kernel structure

Monolithic kernel

Concept

- Entire kernel code *linked* together in a single large executable
- System call interface between kernel and applications



Examples

- GNU/Linux
- Unix
- BSD

Pros and cons

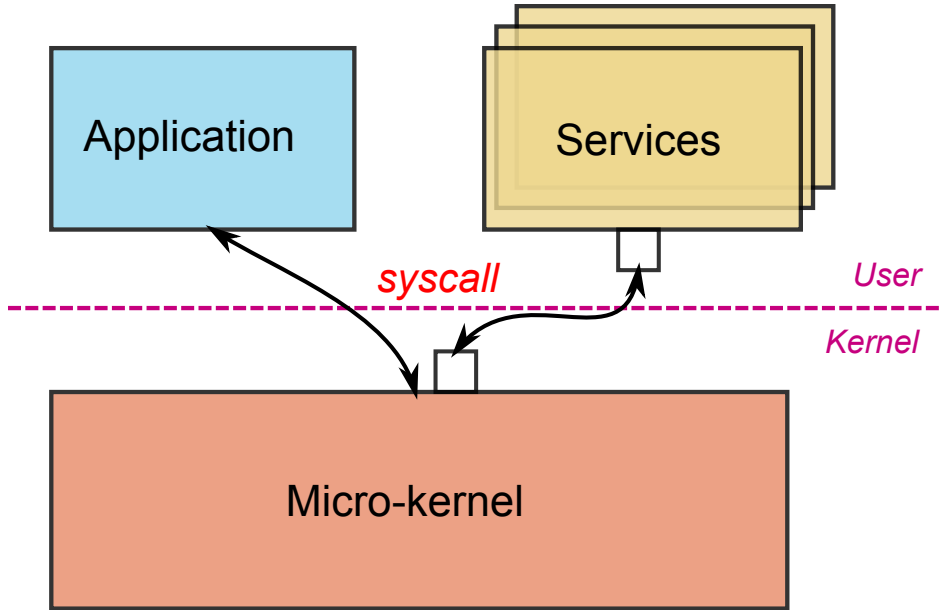
- Great performance
- But increased potential for instability
  - Crash in any function brings the whole system down
  - `kernel panic`

Kernel structure

Microkernel

Concept

- Most kernel services implemented as regular user-space processes
- Microkernel communicates with services using message passing



Examples

- Minix
- Mach
- L4

Pros and cons

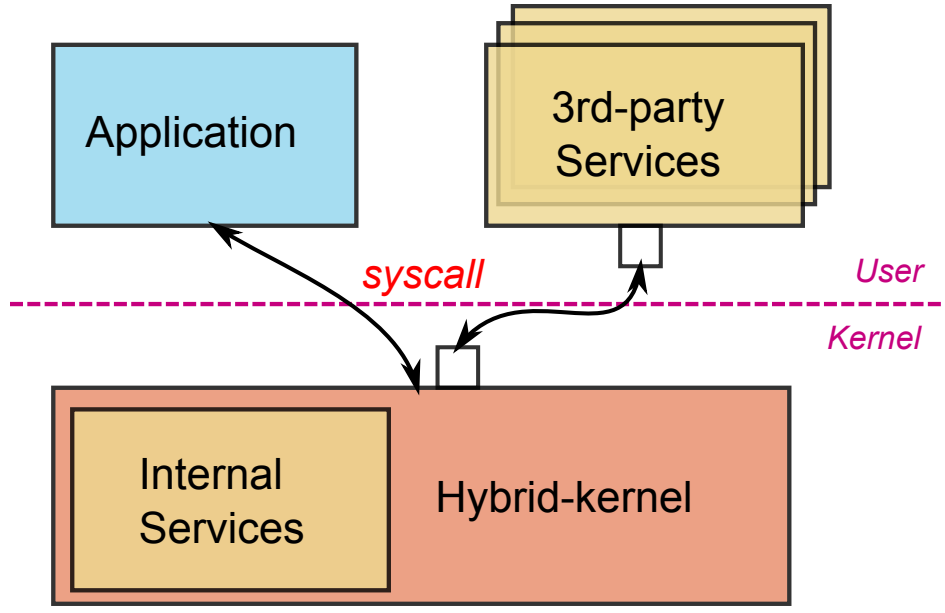
- Great fault isolation
- But inefficient (boundary crossings)

Kernel structure

Hybrid kernel

Concepts

- Trusted OS services implemented in kernel
- Non-trusted OS services implemented as regular user-space processes
- Best of both worlds?



Examples

- Windows
- macOS

Pros and cons

- Monolithic kernel for the most part
- But user-space device drivers

Linux Kernel

Simplified internal structure