TDDB68/TDDE47 Concurrent Programming and Operating Systems

Lecture 9:
Virtualization + Synchronization II

Klas Arvidsson, Mikael Asplund, Adrian Pop
Department of Computer and Information Science

OS structures

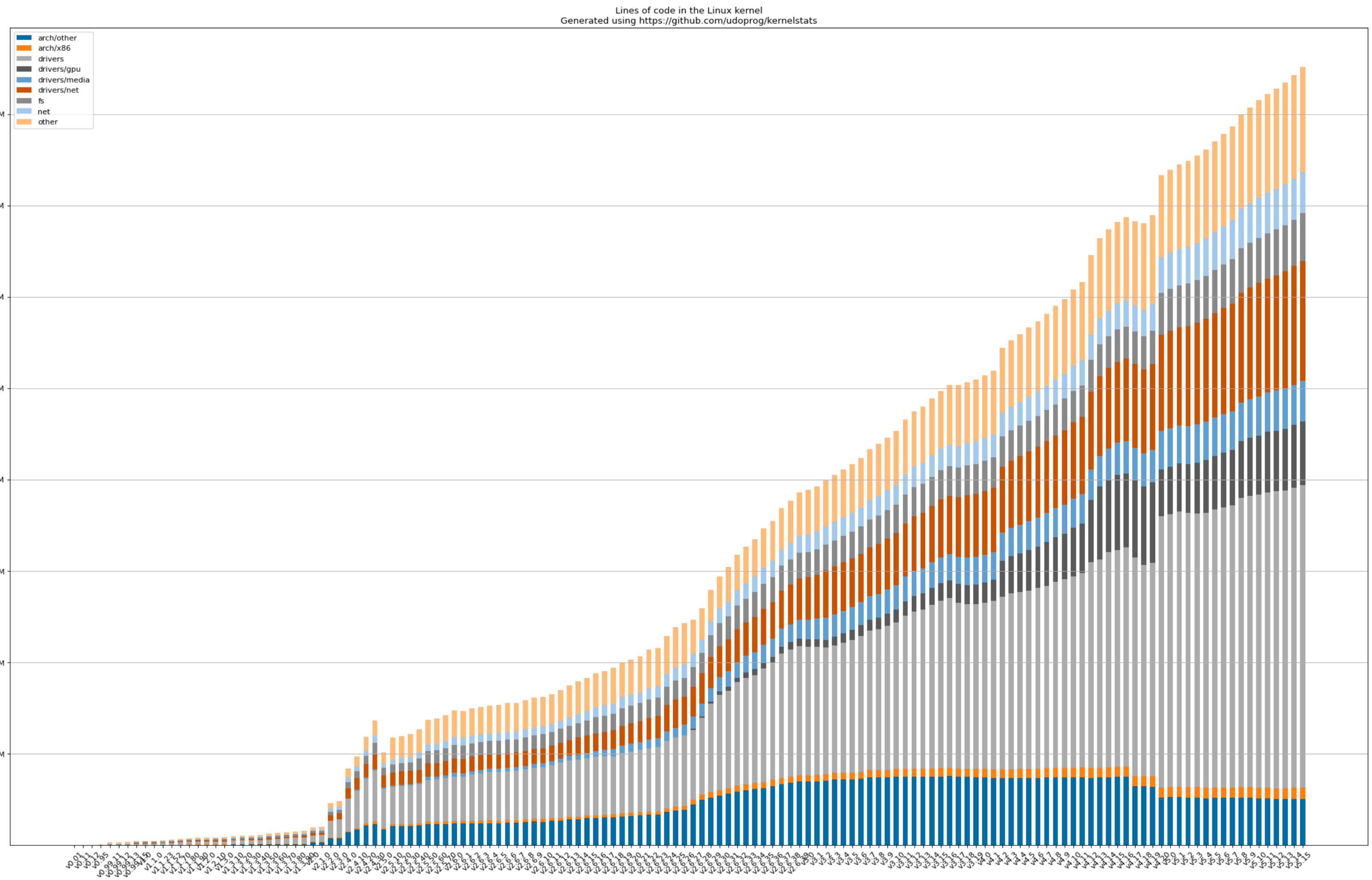
Linux kernel source code size

20M

15M

10M

5M



https://github.com/udoprog/kernelstats

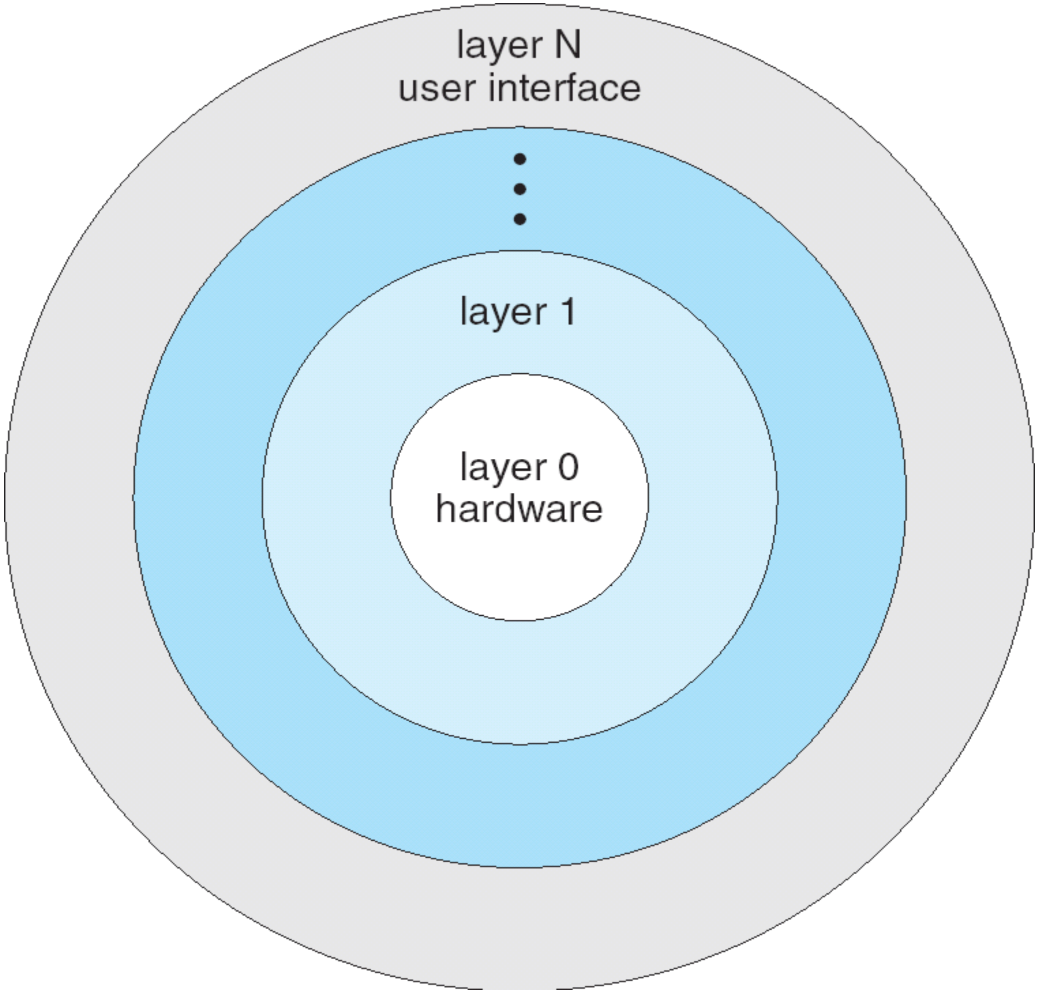
Operating System Structures

- How to manage OS complexity?
- Divide-and-conquer!
- Decompose into smaller components with well-defined interfaces and dependences
  - Layered Approach
  - Microkernels
  - Modules
  - Virtual Machines

Layered Approach

- The operating system is divided into a number of layers (levels, rings), each built on top of lower layers.

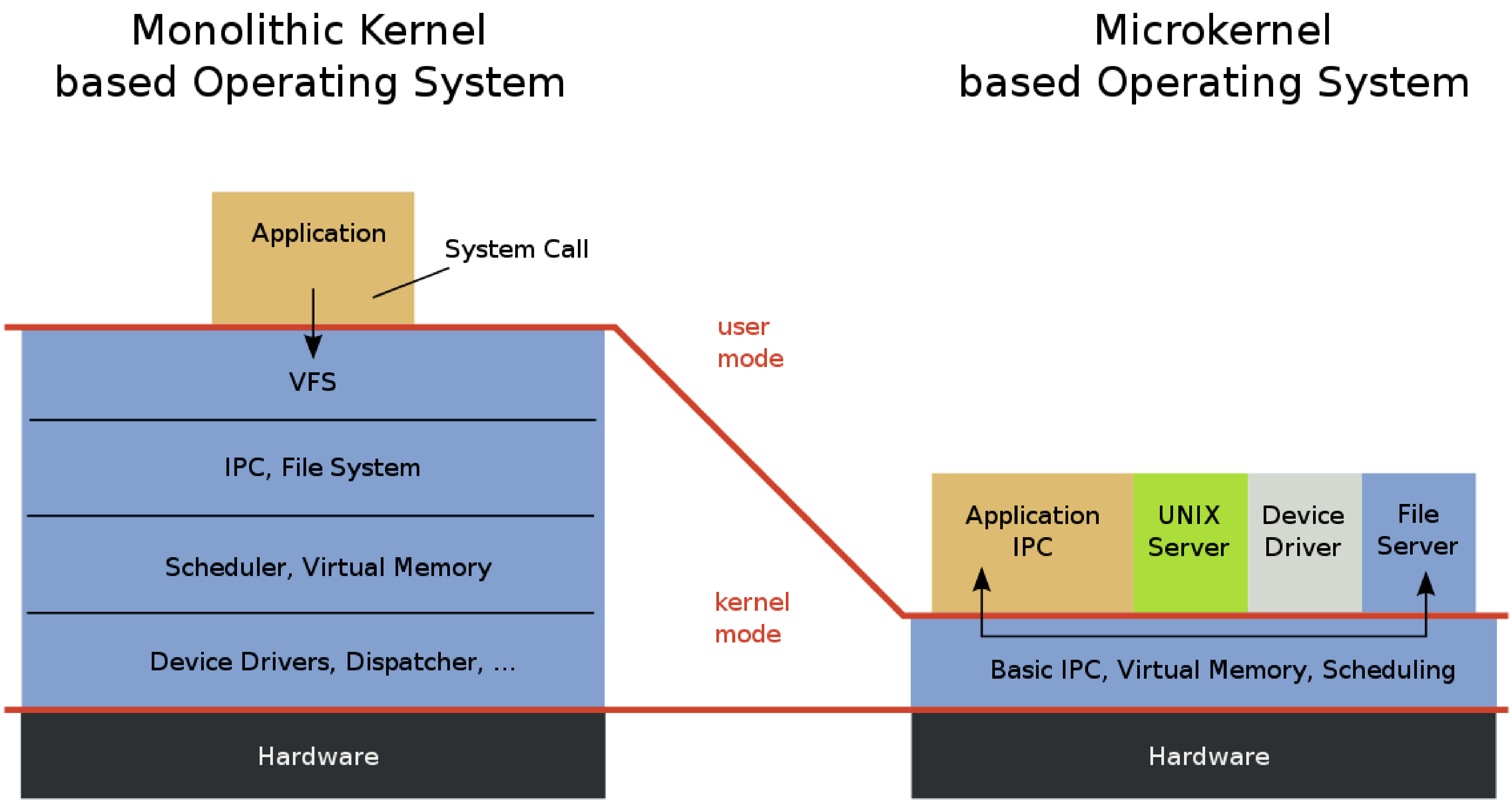
- Functions in layer *i* call only functions/services in layers $\leq i$ (**strict layering**: only in *i* or *i-1*)



Problems of the layered approach

- **Cyclic dependences** between different OS components
- **Less efficient**
  - Long call chains (e.g. I/O) down to system calls, possibly with parameter copying/modification at several levels
- Compromise solution: Have few layers

Microkernels



Source: wikipedia

Microkernel Pros and Cons

- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication
  - More complicated synchronization

Modules

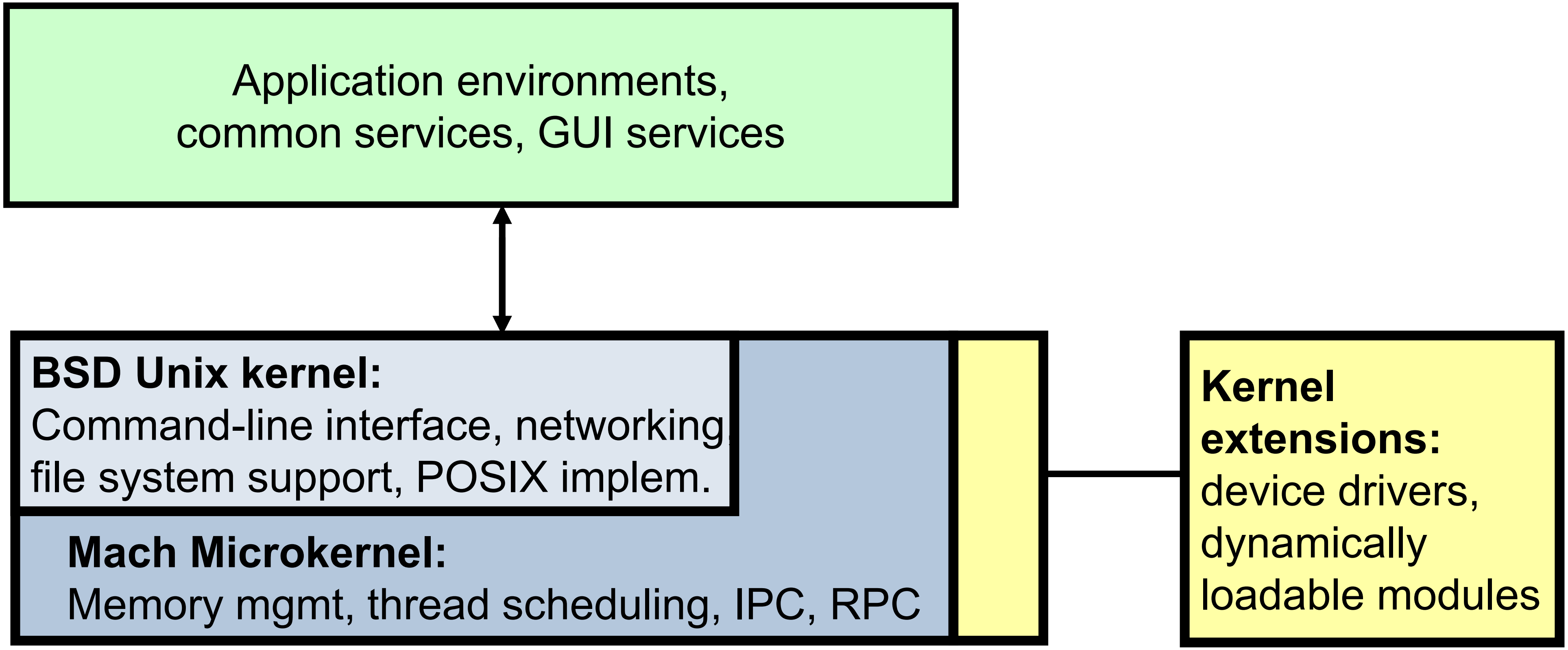
- Most modern operating systems implement kernel modules

- Component-based approach:
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel

- Overall, similar to layers
but more flexible

Example: MacOS - ”Darwin”

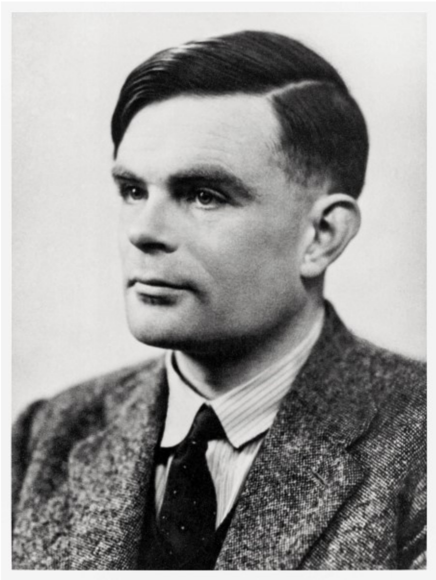
- Hybrid structure: Layering + Microkernel + Modules



Virtual Machines

”It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the standard description of some computing machine M, then U will compute the same sequence as M.”

Alan Turing 1936



Why?

Functional
vs.
Non-functional

NF properties to consider

- Resource efficiency
- Security and fault tolerance
  - (through protection)
- Flexibility
- Responsiveness for an individual unit

Implementation of virtualization

- Emulation
  - HW-independent
- Hypervisor-based virtualization
  - Often HW-assisted
- Paravirtualization
  - Requires modification of guest OS
- Programming environment virtualization
- Application containment

Implementation of virtualization

- Emulation
  - HW-independent

1 slide

- Hypervisor-based virtualization
  - Often HW-assisted

10 slides

- Paravirtualization
  - Requires modification of guest OS

2 slides

- Programming environment virtualization

1 slide

- Application containment

1 slide

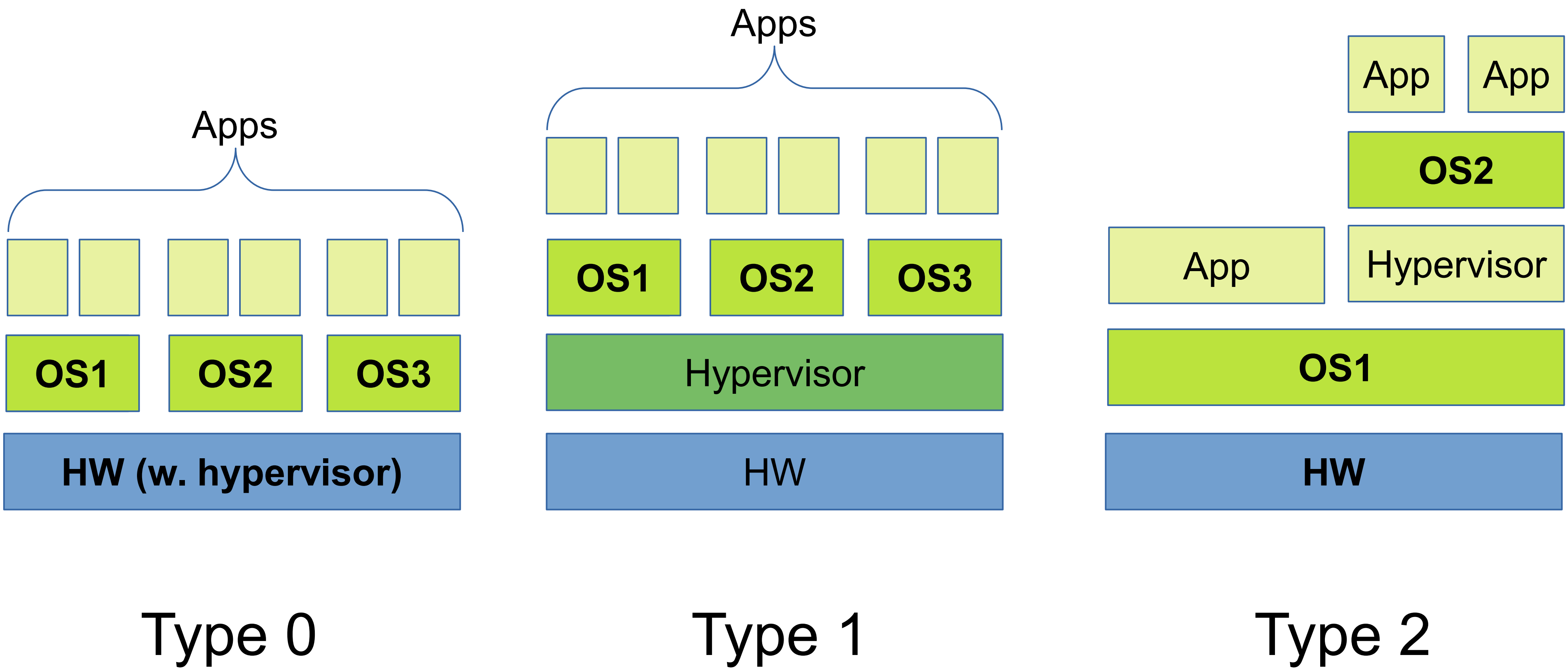
Emulation

- Emulation allows guest to run on different CPU
- Necessary to translate all guest instructions from guest CPU to native CPU
  - Performance challenge
- Examples when useful:
  - Company replacing outdated servers
  - Gaming (e.g., playing old Nintendo games)

Hypervisor-based virtualization

- Type 0 hypervisor
  - Hypervisor implemented in firmware – full separation between guest OSs
- Type 1 hypervisor
  - Basic OS that just provides OS switching capabilities
- Type 2 hypervisor
  - Virtualization at software level (runs as a process)

Hypervisors



Example - VMware/Virtualbox

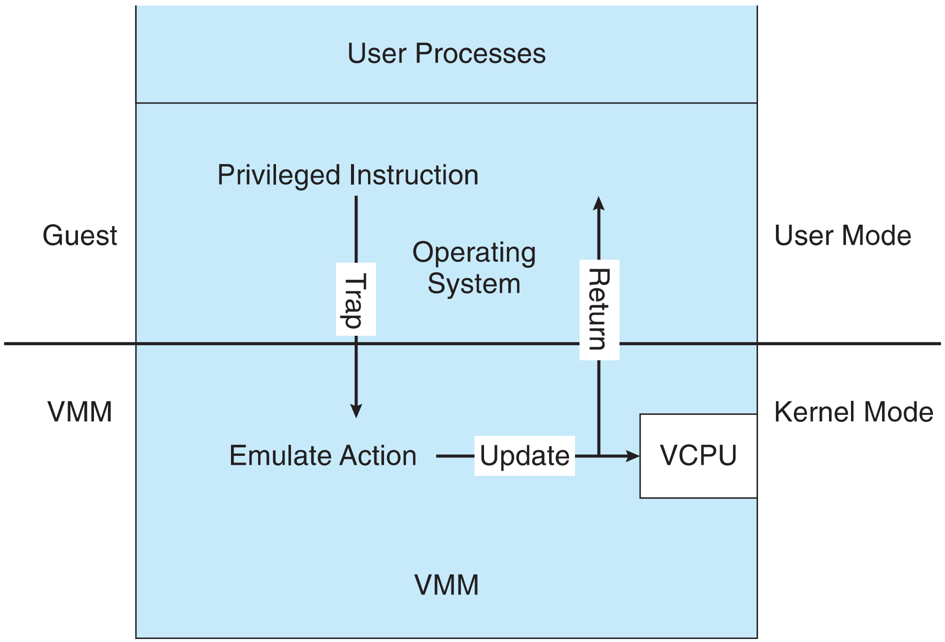
- Provides Virtual Machine Manager (VMM) for guests
- Runs as application on other native, installed host operating system -> Type 2
- Lots of guests possible, including Windows, Linux, etc. all runnable concurrently (as resources allow)
- Virtualization layer abstracts underlying HW, providing guest with is own virtual CPUs, memory, disk drives, network interfaces, etc.
- Physical disks can be provided to guests, or virtual physical disks (just files within host file system)

Virtualization building blocks

- Trap and emulate
- Binary translation
- Nested page tables
- More HW assistance

Trap and emulate

- Guest OS will need to execute privileged instructions
- Not safe to let Guest OS run in kernel mode
- Solution: trap privileged instructions and emulate them

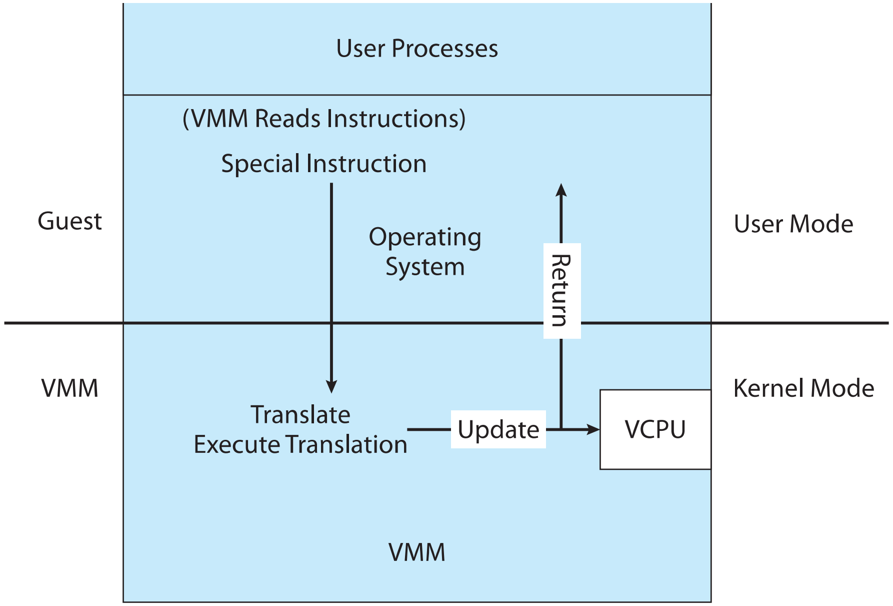


Problems with trap & emulate

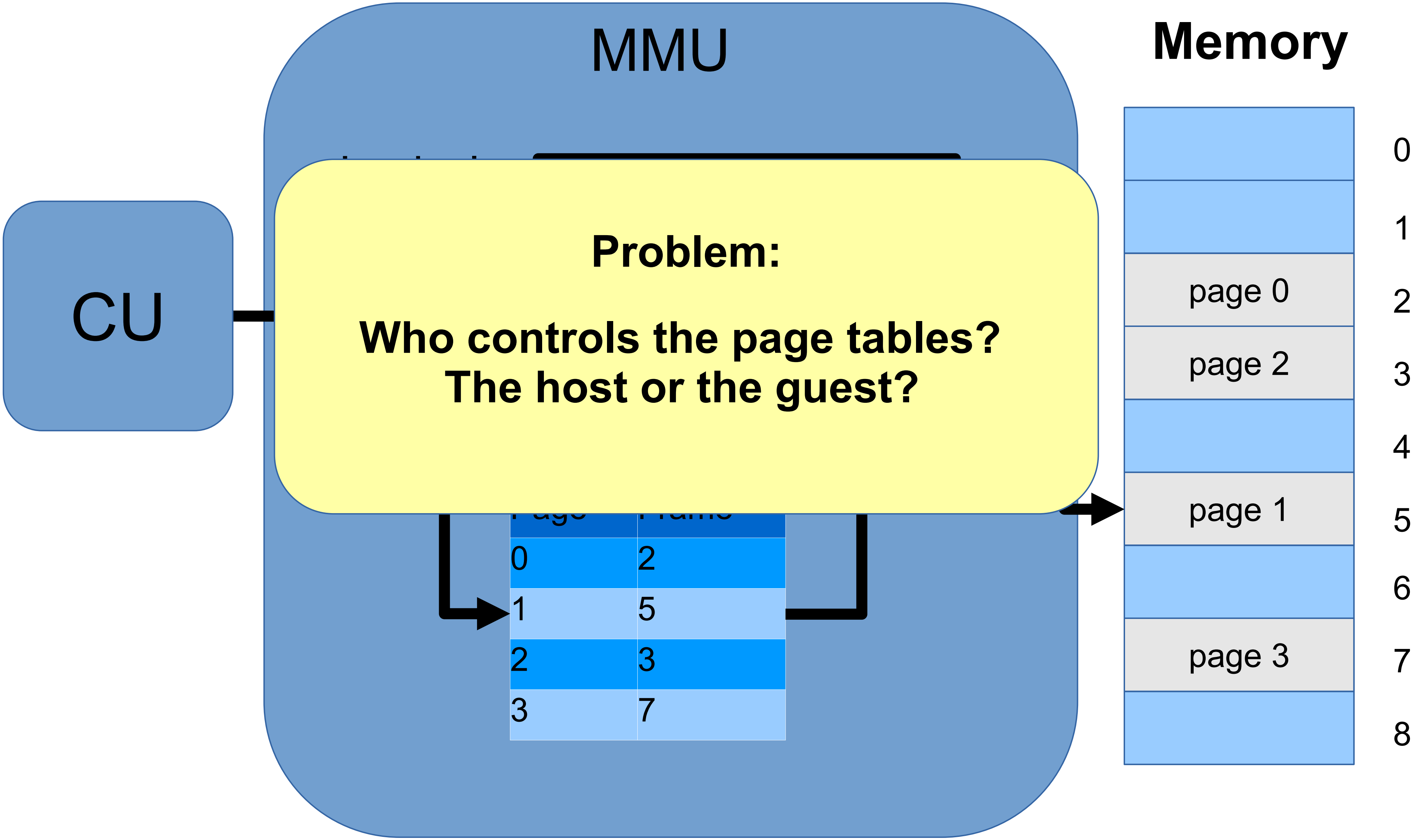
- CPU architectures often not so clean
- Example: x86 **popf** instruction
  - Loads CPU flags register from contents of the stack
  - If CPU in privileged mode -> all flags replaced
  - If CPU in user mode -> some flags replaced
  - No trap is generated!
- Also other such special instructions

Binary translation

- If guest VCPU is in user mode
  - run instructions natively
- If guest VCPU in kernel mode
  - VMM examines instructions in advance
  - Non-special-instructions run natively
  - Special instructions translated into equivalent instructions



Recall


MMU
Memory
CU
Problem:
Who controls the page tables?
The host or the guest?
0 2
1 5
2 3
3 7
page 0
page 2
page 1
page 3
0
1
2
3
4
5
6
7
8

Nested Page Tables (NPT)

- Each guest maintains its own (per-process) page tables
- VMM maintains per guest NPTs to represent guest’s page-table state
  - Just as VCPU stores guest CPU state
- Shadow page tables can be kept in software (very slow)
- Hardware support with one more level of nesting

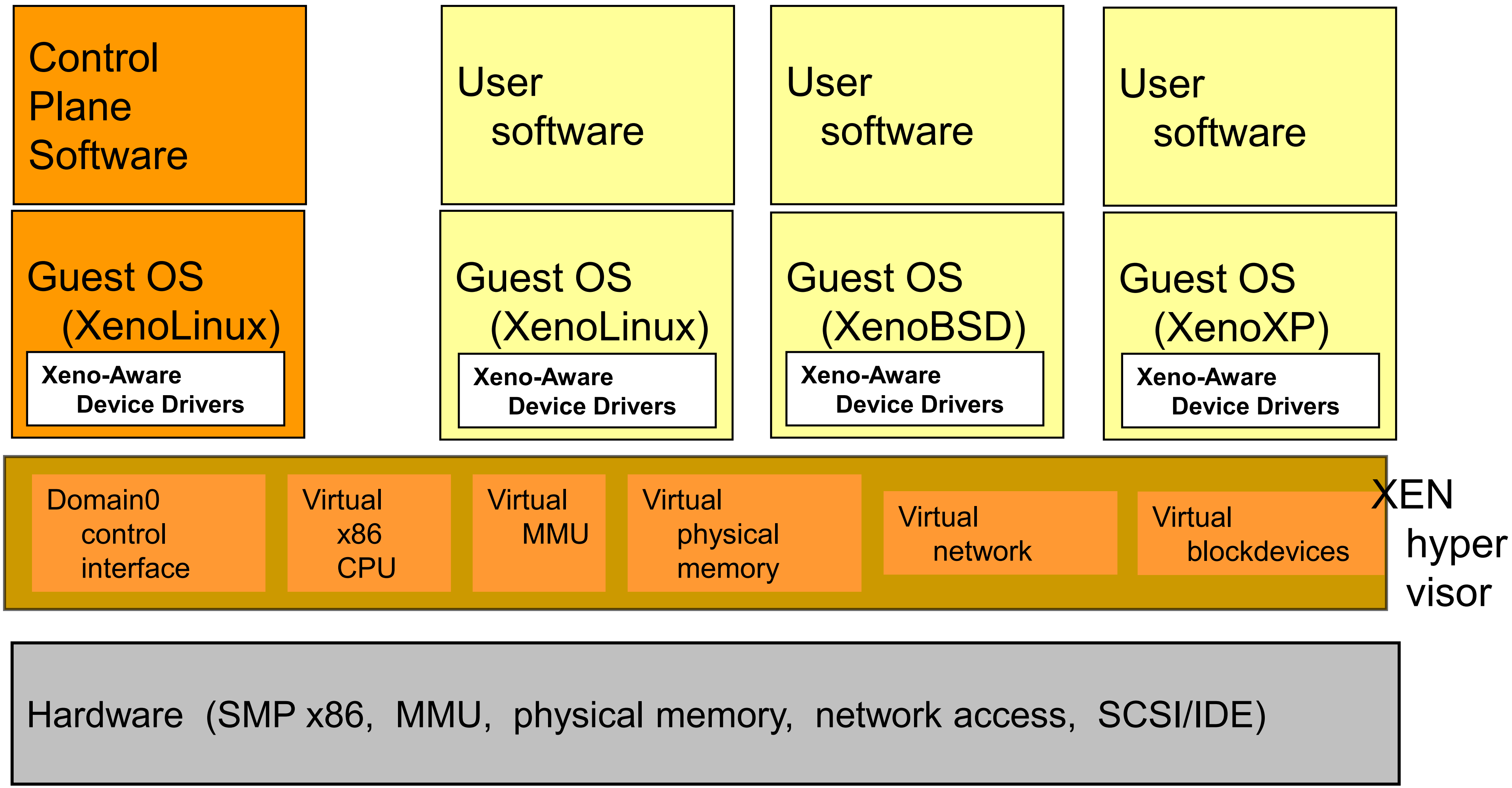
More HW assistance

- More support -> more feature rich, stable, better performance of guests
- Intel added new VT-x instructions in 2005 and AMD the AMD-V instructions in 2006
  - Removes the need for binary translation
  - Generally define more CPU modes – Guest/host, VCPU states
  - In guest mode, guest OS thinks it is running natively
- New examples and variants appear over time

Paravirtualization

- Does not fit the definition of virtualization – VMM not presenting an exact duplication of underlying hardware
- VMM provides services that guest must be modified to use
- Leads to increased performance (compared to emulation)
- Less needed as hardware support for VMs grows

Paravirtualization Example: Xen



Adapted from: P. Barham et al.: Xen and the Art of Virtualization. Proc. SOSP 2003

Programming environment virtualization

- Programming language is designed to run within custom-built virtualized environment
- For example Oracle Java has many features that depend on running in Java Virtual Machine (JVM)
  - Virtualization through API
- Programs written in Java run in the JVM no matter the underlying system
- Similar to interpreted languages

Application Containment

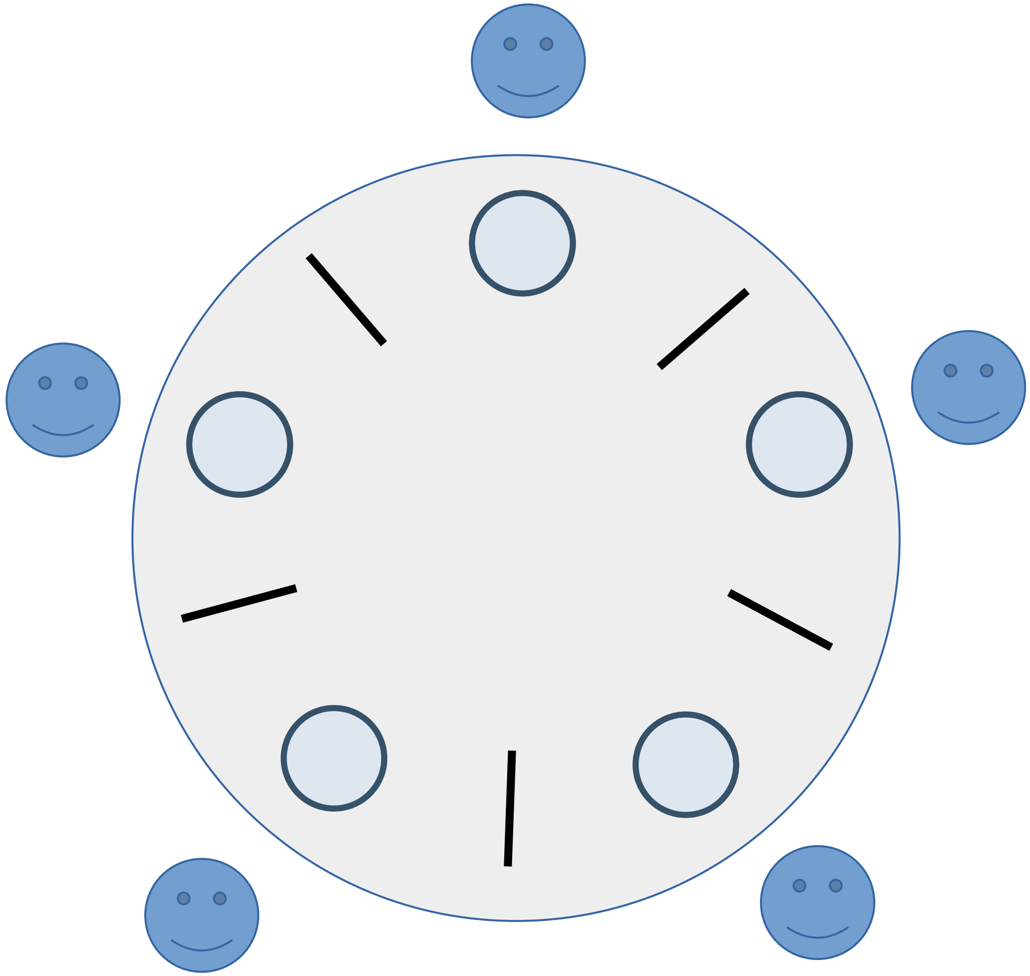
- Virtualization still costly!
- Oracle **containers** / **zones** for example create virtual layer between OS and apps
  - Only one kernel running – host OS
  - Virtual environment through different zones
  - Applications run in a zone
- Popular today: Docker

Synchronization II

Remaining topics

- Monitors
- Reader-writer synchronization
- Lock-free synchronization

Example:Dining-Philosophers Problem



A philosopher can be either:

Thinking (happy)

Hungry (cannot think, wants to eat)

Eating (also happy)

Eating requires 2 chopsticks

A chopstick can only be used by one philosopher at a time

Potential solution

```
Process philosopher {
    while (True) {
        think();
        if hungry() {
            pickup_left();
            pickup_right();
            eat();
        }
    }
}
```

What if this fails?

Three bad options

- Program crashes when trying to pickup a chopstick which is already taken
- Pickup operation waits until the chopstick is free
  - Risk of deadlock
- Pickup operation fails if chopstick is taken
  - Eat operation will also fail (need two chopsticks)
  - Risk of starvation (will only get to eat if lucky)

Good design

- Synchronization mechanisms with queues
  - Avoids starvation
- Prevent deadlocks
  - Enforce global order of locking resources
  - Either take both chopsticks or neither

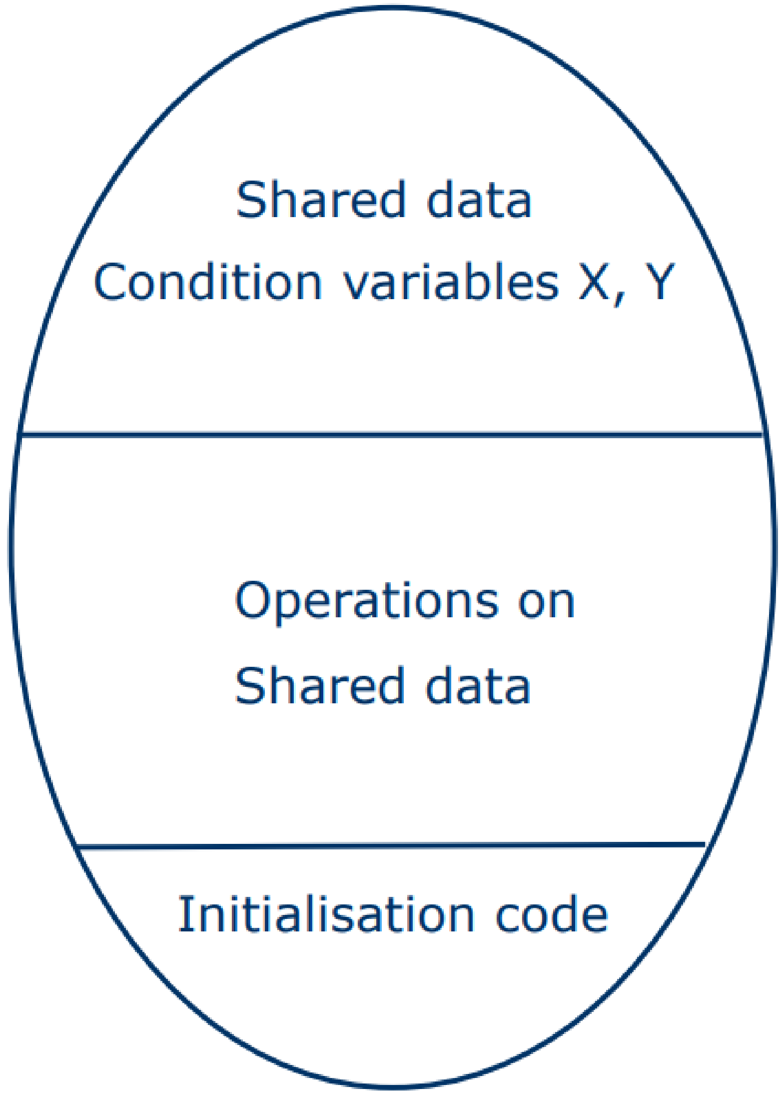
Monitors

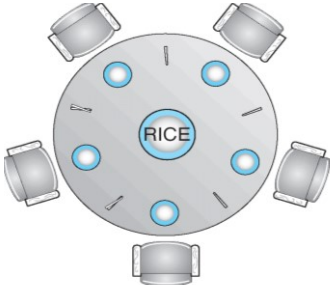
What is a monitor?

- A programming abstraction consisting of:
  - A data structure on which programmer can define operations – which can only be run one at a time
  - Condition variables for synchronisation
- Encapsulates shared data that several processes can operate upon
- All access is with mutual exclusion
- Pre object-orientation!

[Hoare 74]

Monitor overview





Monitor Solution to Dining Philosophers

```
monitor DP {
  enum {THINKING, HUNGRY, EATING} state
    [5];
  condition self [5];

  void pickup ( int i ) {
    state[i] = HUNGRY;
    test ( i );
    if (state[i] != EATING)
       self [i].wait();
  }

  void putdown ( int i ) {
    state[i] = THINKING;
    test((i+4)%5);   // left neighbor
    test((i+1)%5);   // right neighbor
  }
...
```

```
...
  void test ( int i ) {
    if ((state[(i+4)%5] != EATING)
         && (state[i] == HUNGRY)
         && (state[(i+1)%5] != EATING)) {
      state[i] = EATING ;
      self[i].signal () ;
    }
  }

  initialization_code() {
    for (int i = 0; i < 5; i++) {
      state[i] = THINKING;
    }
  }
}
```

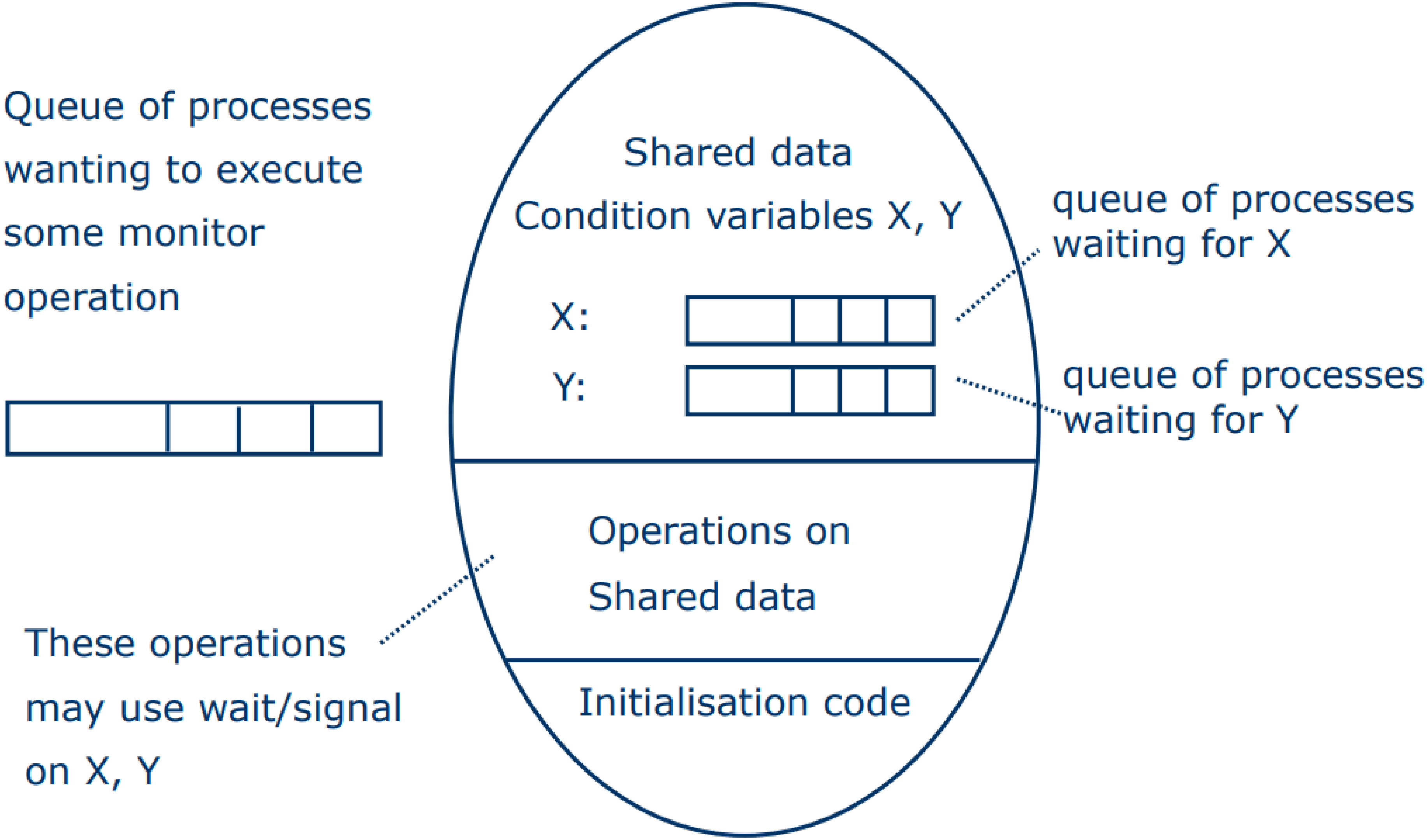
Observations

- Programmer uses wait and signal inside the code that applies the operations on the shared data structure

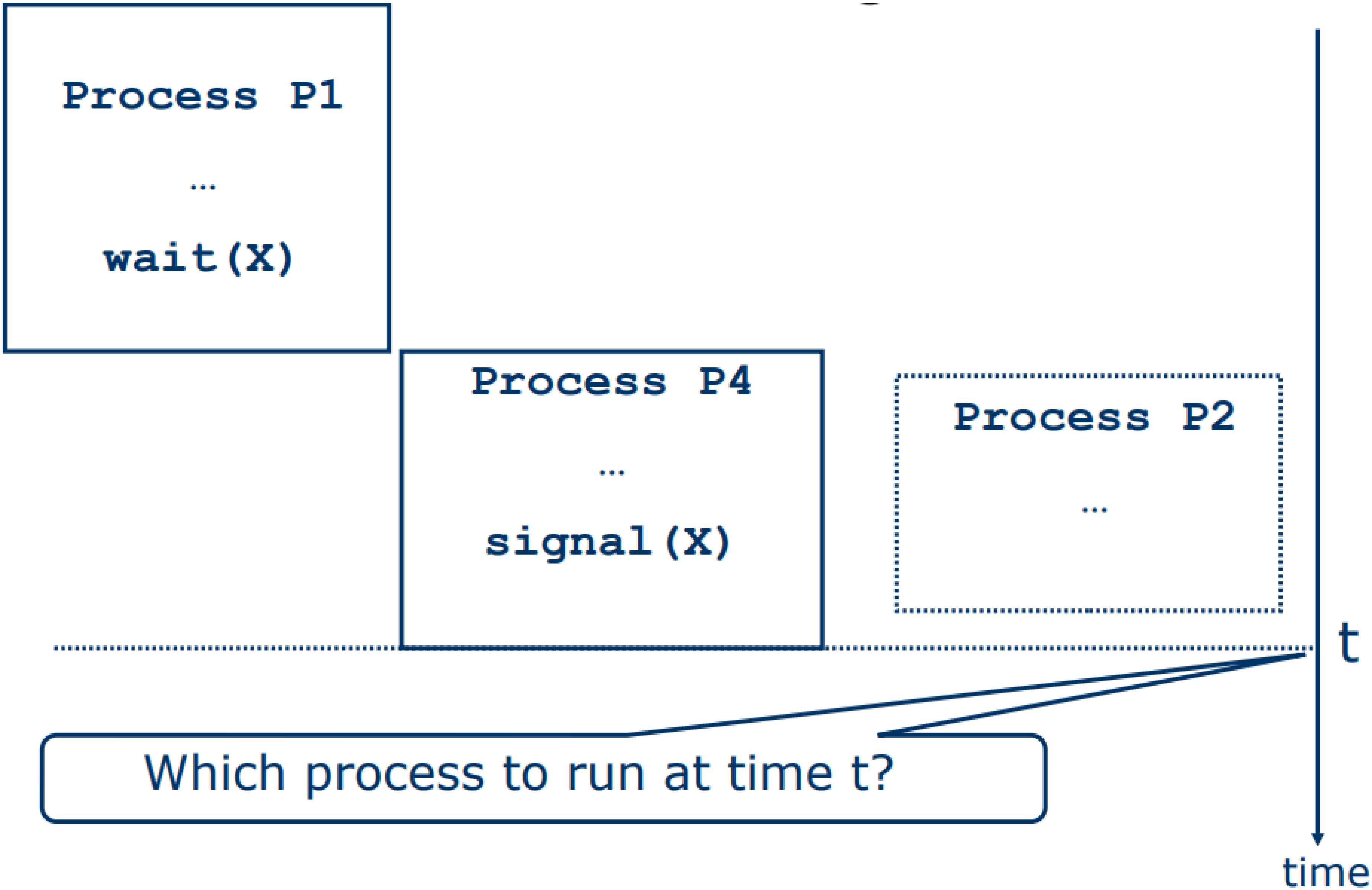
Note:

- The condition variable has no values assigned to it
- The queue associated with each variable is the main synchronisation mechanism
- Different semantics from semaphore operations for wait and signal

Process queues



Who wakes on signal?



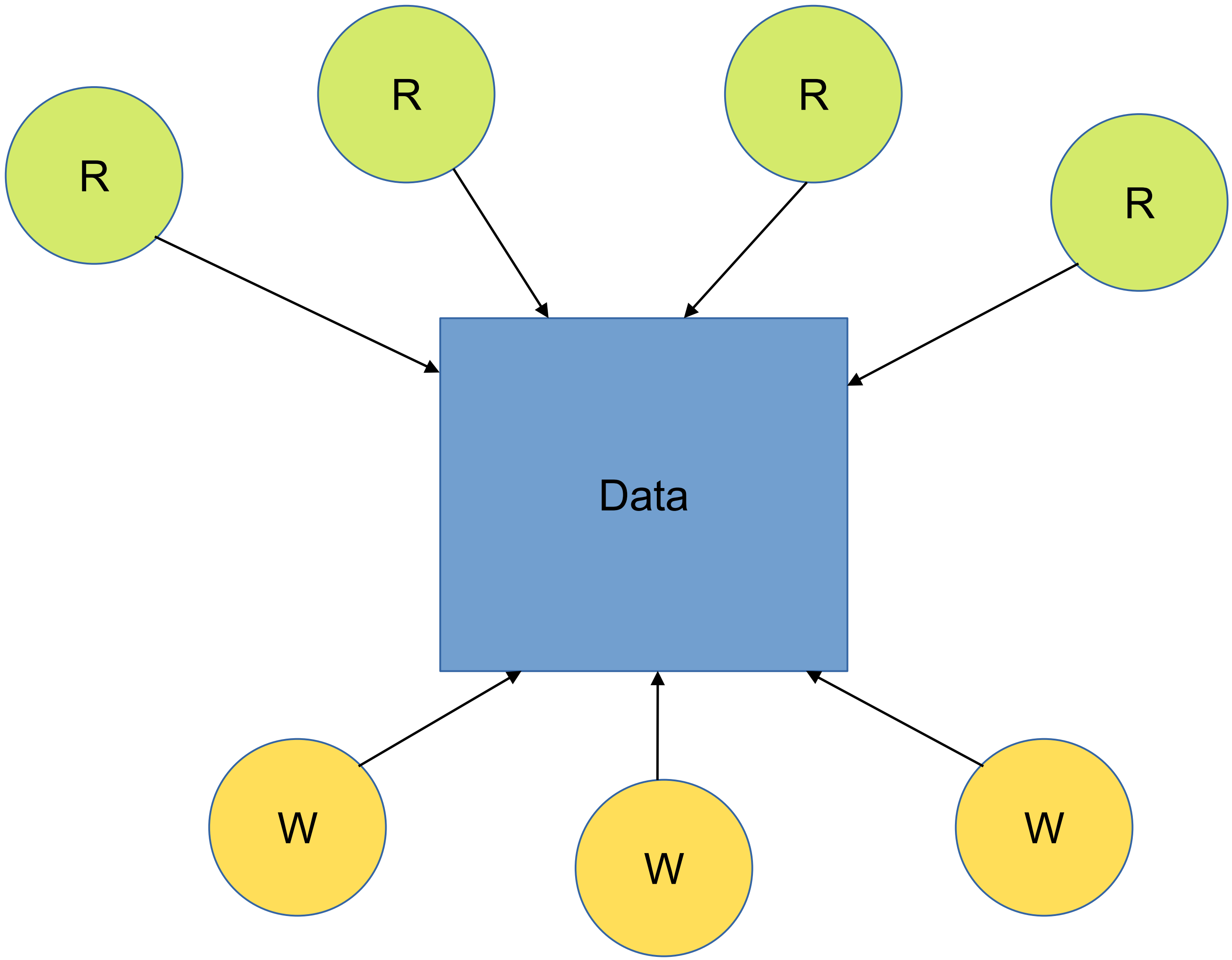
Options for waking up

- Original Hoare monitor: let the woken up process (P1) continue
  - What if there are several processes waiting on X?

- Pragmatic solution (Java): let the signalling process continue, and wake up P1 once P4 is suspended/exits
  - P1 has to check for condition X when woken up!

Readers-writer problem

Who gets access?



Reader-writer solution

Shared data

- Data
- Semaphore rw_mutex initialized to 1
- Semaphore mutex initialized to 1
- Integer read_count initialized to 0

Writer process

```
while (true) {
  wait(rw_mutex);
  /* WRITE */
  signal(rw_mutex);
}
```

Reader process

```
while (true){
  wait(mutex);
  read_count++;
  if (read_count == 1) /* first reader */
    wait(rw_mutex);
  signal(mutex);
  /* READ */
  wait(mutex);
  read_count--;
  if (read_count == 0) /* last reader */
    signal(rw_mutex);
  signal(mutex);
}
```

Readers-Writers Problem Variations

- **First reader-writer problem**
  - Once a reader has access, readers will be prioritized over writers
  - Writers starve
- **Second reader-writer problem**
  - Once a writer is ready to write, no “newly arrived reader” is allowed to read.
  - Readers starve
- **Third reader-writer problem**
  - Implement a service queue
  - Complex
- **Problem is solved on some systems by kernel providing reader-writer locks**

Lock-free concurrent programming

Lock-based solutions

Process P1

```
while true {
  acquire(lock)
  critical_section
  release(lock)
}
```

Process P2

```
while true {
  acquire(lock)
  critical_section
  release(lock)
}
```

What happens if a process that holds a semaphore/monitor is killed?

Lock-free algorithms!

Basic definitions

- A **lock-free** algorithm guarantees that at least one process can make progress within a finite time
  - Also called non-blocking
- A lock-free algorithm is **wait-free** if every process makes progress within some finite time

Primitive

- Compare and Swap (CAS), two flavors:

```
// bool version

bool CAS(int *p, int old, int new)
{
  if (*p != old)
  {
    return false;
  }
  else
  {
    *p = new;
    return true;
  }
}
```

```
// old value version
// (and simpler(?) logic)
int CAS(int *p, int cmp, int v)
{
  int old = *p;
  if (old == cmp)
  {
    *p = v;
  }
  return old;
}
```

Simple lock-free stack algorithm

- Due to Treiber 1986
- This presentation based on Michael and Scott 1998 (JPDC)

Data structures

```
struct pointer_t {
    node_t* ptr;
    uint count;
}
struct node_t {
    int value;
    pointer_t next;
}
struct stack_t {
    pointer_t top;
}
```

```
// Why not:
struct node_t {
    int value;
    node_t* next;
}
struct stack_t {
    node_t top;
}
```

Flawed Push (Why?)

```
push(stack_t* S, int value) {
    node_t* node = malloc(sizeof(node_t));
    node->value = value;
    node->next = NULL;
    repeat
        pointer_t top = S->top;
        node->next = top;
    until CAS(&S->top, top, node);
}
```

ABA problem

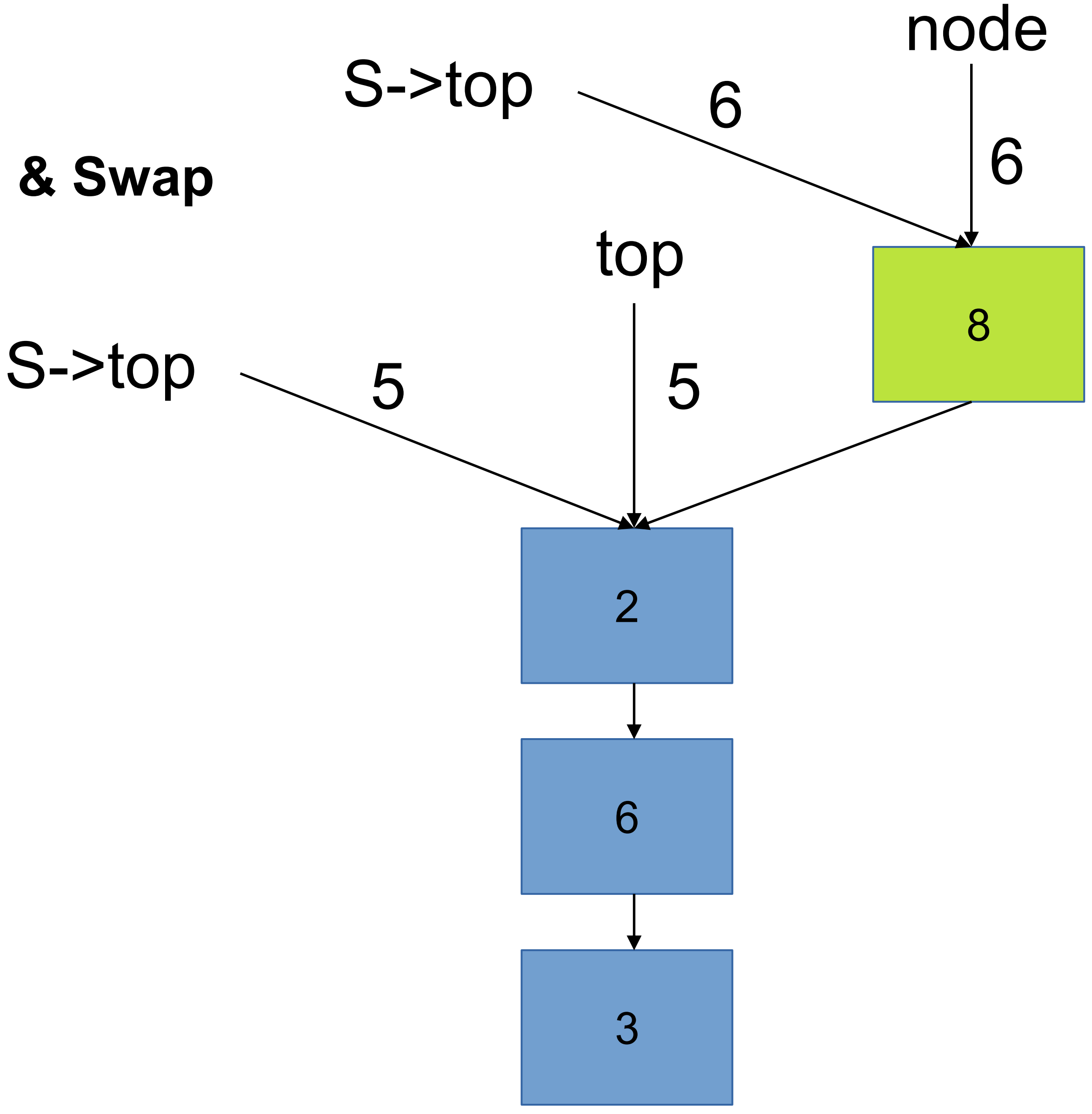
```
I: malloc node X

I: top = S->top

A: malloc node Y

B: pop node and free top

A: push node Y

A: malloc and push node Z
  ! malloc may reuse space of last free which
  ! will yield same adress as previous top

I: push node X
  ! Stack have changed since read of top but
  ! adress of new top Z will be same as old top
  ! Compare and swap will not detect the change!
  ! Nodes Y and Z are lost!
```

Push (Corrected!)

```
push(stack_t* S, int value) {
    node_t* node = malloc(sizeof(node_t));
    node->value = value;
    node->next.ptr = NULL;
    repeat
        pointer_t top = S->top;
        node->next.ptr = top.ptr;
    until CAS(&S->top, top, [node, top.count+1]);
} // Solves ABA-problem
```

Push

Compare & Swap



Pop

```
pop(stack_t* S, int *pvalue) {
    repeat
        top = S->top;
        if top.ptr == NULL
            return False;
    until CAS(&S->top, top, [top.ptr->next.ptr, top.count+1]);
    *pvalue = top.ptr->value;
    free(top.ptr);
    return True;
}
```

CAS-based spinlock

```
// tempting to assume HW-solutions are faster and use it

void aquire(bool* lock) {
  while !CAS(lock, false, true)
    ; /* busy wait, spinlock */
}

void release(bool* lock) {
  CAS (lock, true, false);
}
```

Threads holding the lock but not executing (on ready queue due to lack of available CPU:s) will cause threads executing and waiting for the lock to waste their entire time slice!

On the other hand, *if* critical section is small and fast, *and* threads are guaranteed it's own CPU, the wait time will be really small.

```
// use example
bool mutex = false;

aquire(&mutex);
 critical_section();
release(&mutex);
```

Regular wait-queue based locks let other threads run during wait, but have higher overhead in locking, queue management and context switching

In general

- Some lock-free algorithms provide reasonable performance
- Wait-free algorithms have low performance
- Complex to create
- More library support is coming