KVM/ARM

Linux Symposium 2010

Christoffer Dall and Jason Nieh
{cdall,nieh}@cs.columbia.edu

Slides: http://www.cs.columbia.edu/~cdall/ols2010-presentation.pdf

We like KVM

- It’s Fast, Free, Open, and Simple!
- Integrates well with Linux
- Always maintained
- Supports x86, ia64, PowerPC, and s390

ARM devices are everywhere

Google Nexus One Specifications

Processor	Qualcomm Snapdragon QSD8250
CPU Core	Qualcomm Scorpion
Architecture	ARM v7
Clock speed	1000 MHz
Technology	65 nm
Memory	512 MB

...and they are getting really powerful

KVM relies on hardware support

- x86 and ia64 (Itanium)
- PowerPC, and s390

KVM relies on hardware support

- x86 and ia64 (Itanium)
- PowerPC, and s390

Virtualization Extensions

KVM relies on hardware support

- x86 and ia64 (Itanium)
- PowerPC, and s390

Virtualization Extensions

Virtualizable

Hardware Support for Virtualization

- Guest kernel runs in user mode
- Sensitive instructions are instructions that depend on CPU mode
- Virtualizable if all sensitive instructions trap
- Trap-and-emulate
- Hardware virtualization features provide extra mode where all sensitive instructions trap

Problem

- ARM is not virtualizable
- ARM has no hardware virtualization extensions

31 Sensitive instructions

CPS	LDRT	STC	RSBS
MRS	STRBT	ADCS	RSCS
MSR	STRT	ADDS	SBCS
RFE	CDP	ANDS	SUBS
SRS	LDC	BICS	
LDM (2)	MCR	EORS	
LDM (3)	MCRR	MOVS	
STM (2)	MRC	MVNS	
LDRBT	MRRC	ORRS	

31 Sensitive instructions

CPS	LDRT	STC	RSBS
MRS	STRBT	ADCS	RSCS
MSR	STRT	ADDS	SBCS
RFE	CDP	ANDS	SUBS
SRS	LDC	BICS	
LDM (2)	MCR	EORS	
LDM (3)	MCRR	MOVS	
STM (2)	MRC	MVNS	
LDRBT	MRRC	ORRS	

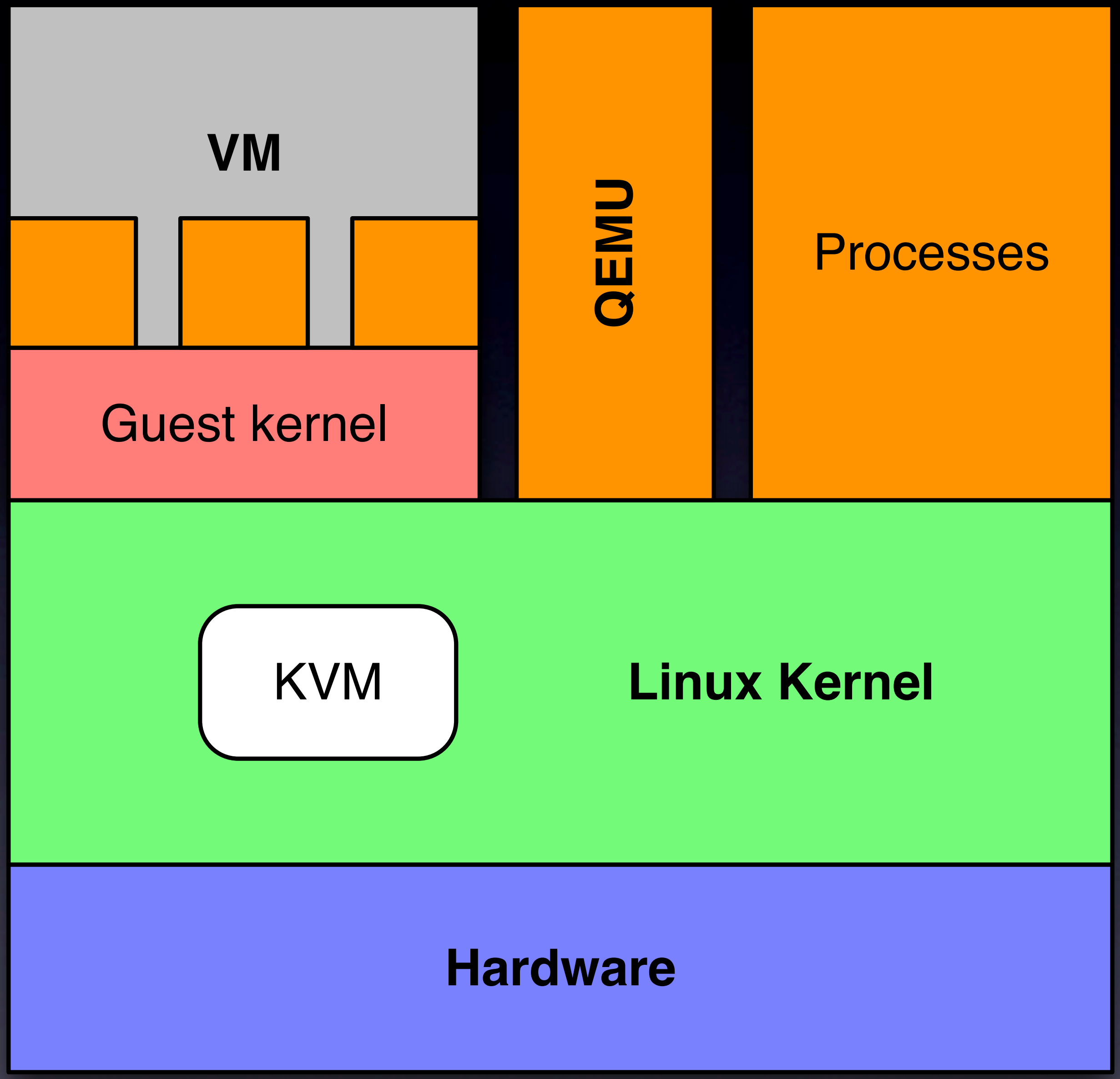
and 25 of them are non-privileged

Solution

- We use lightweight paravirtualization
- Retains simplicity of KVM architecture
- Minimally intrusive to KVM and the Kernel
- Uses on QEMU for device emulation

- KVM
- CPU virtualization on ARM
- Memory virtualization on ARM
- World Switch details
- Implementation status

KVM Architecture

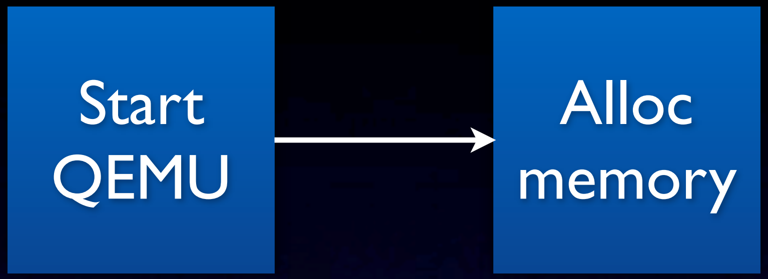


KVM execution flow

Start QEMU

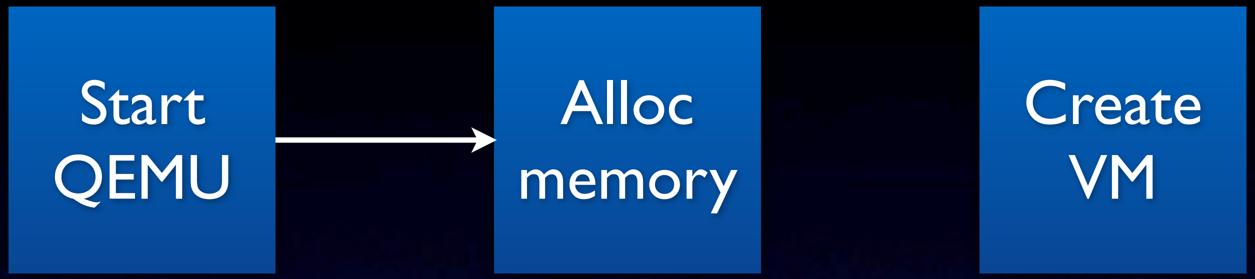
Start QEMU

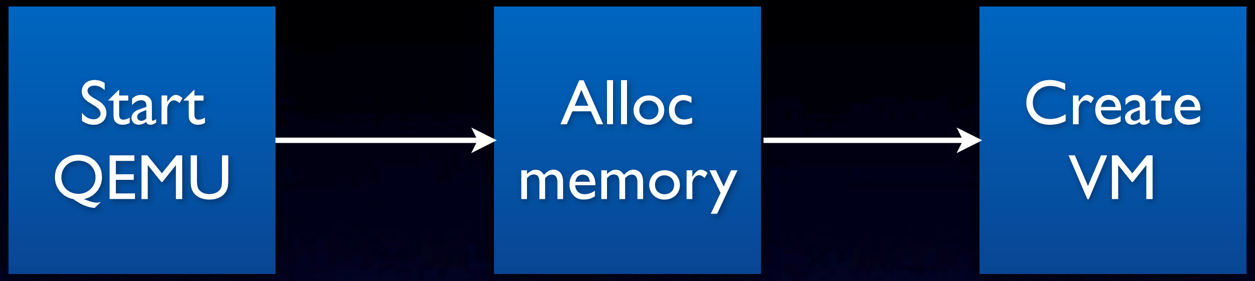
Alloc memory



Start QEMU

Alloc memory

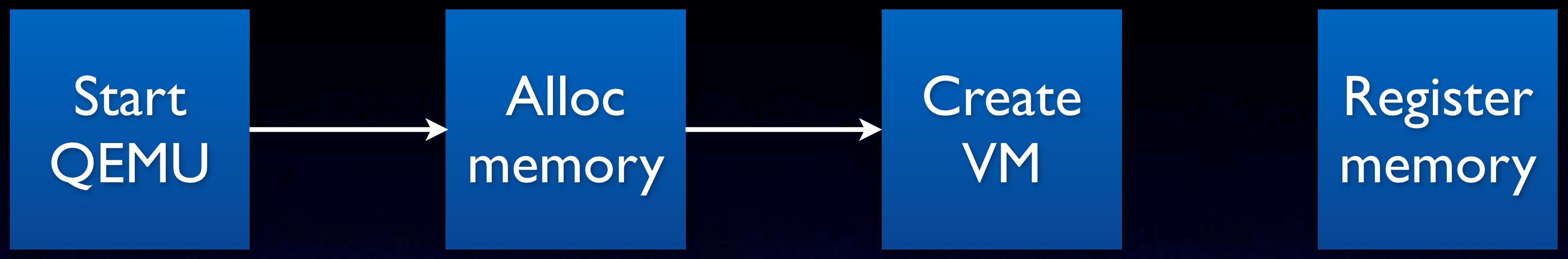

Start QEMU
Alloc memory
Create VM



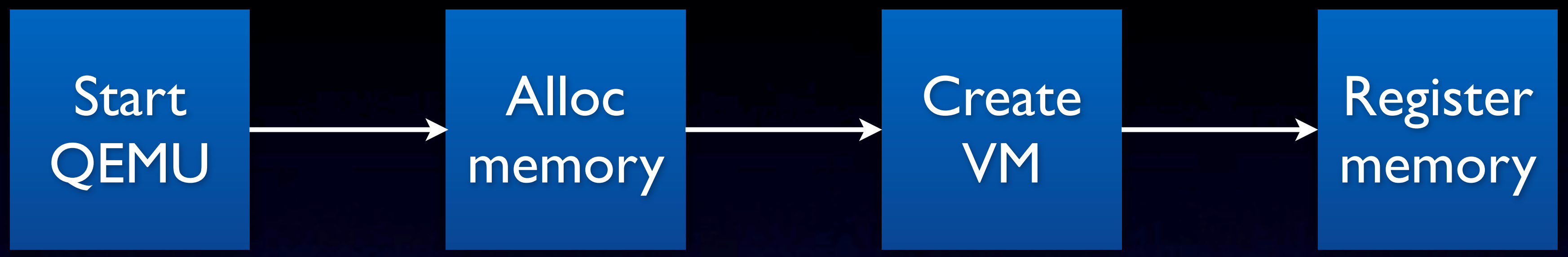
Start QEMU

Alloc memory

Create VM



Start QEMU
Alloc memory
Create VM
Register memory

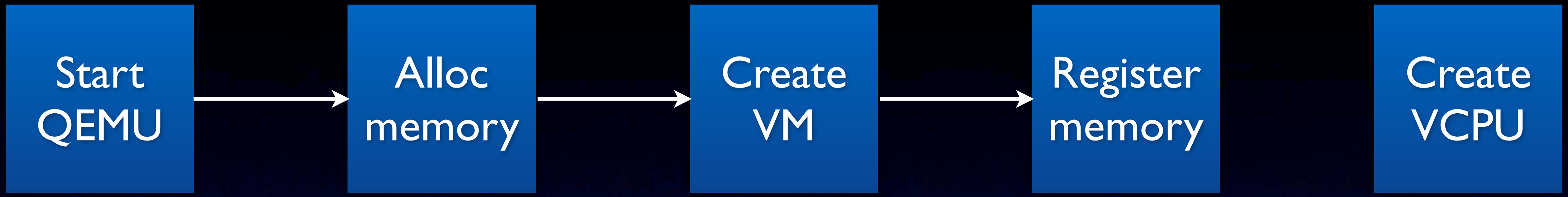


Start QEMU

Alloc memory

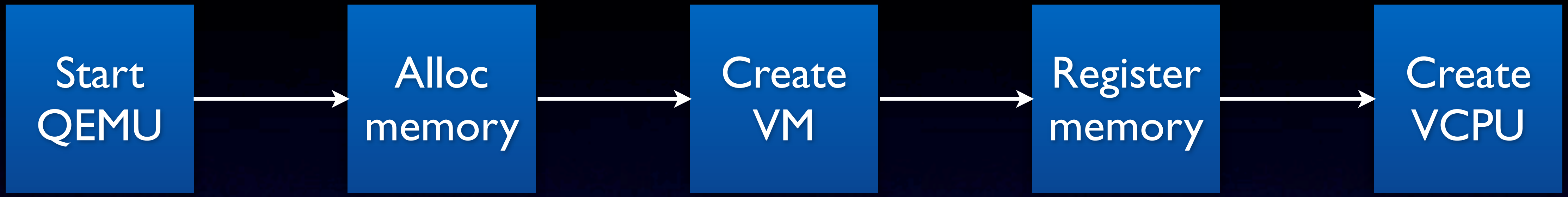
Create VM

Register memory



Start QEMU → Alloc memory → Create VM → Register memory

Create VCPU



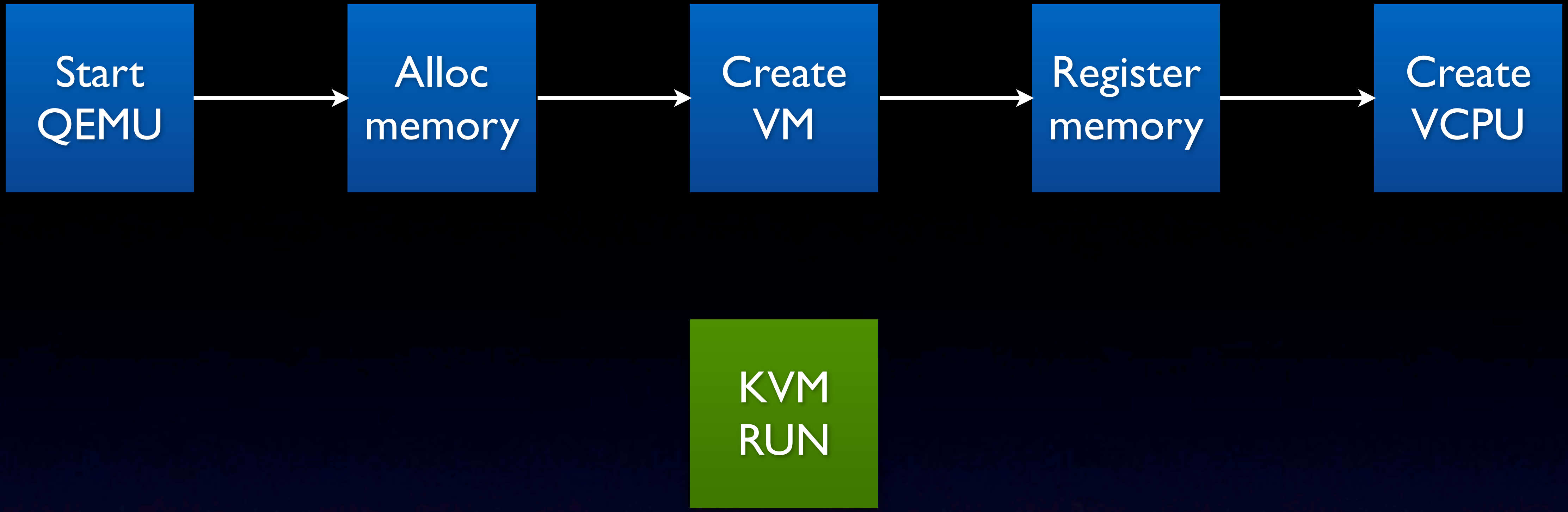
Start QEMU

Alloc memory

Create VM

Register memory

Create VCPU



Start QEMU

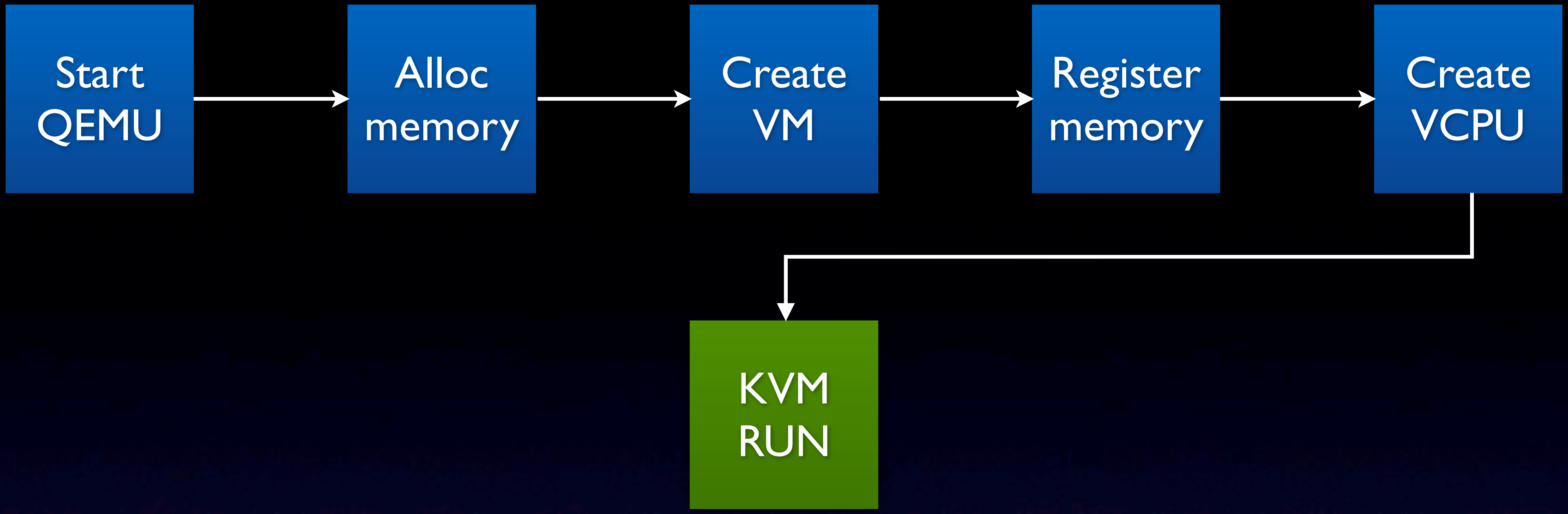
Alloc memory

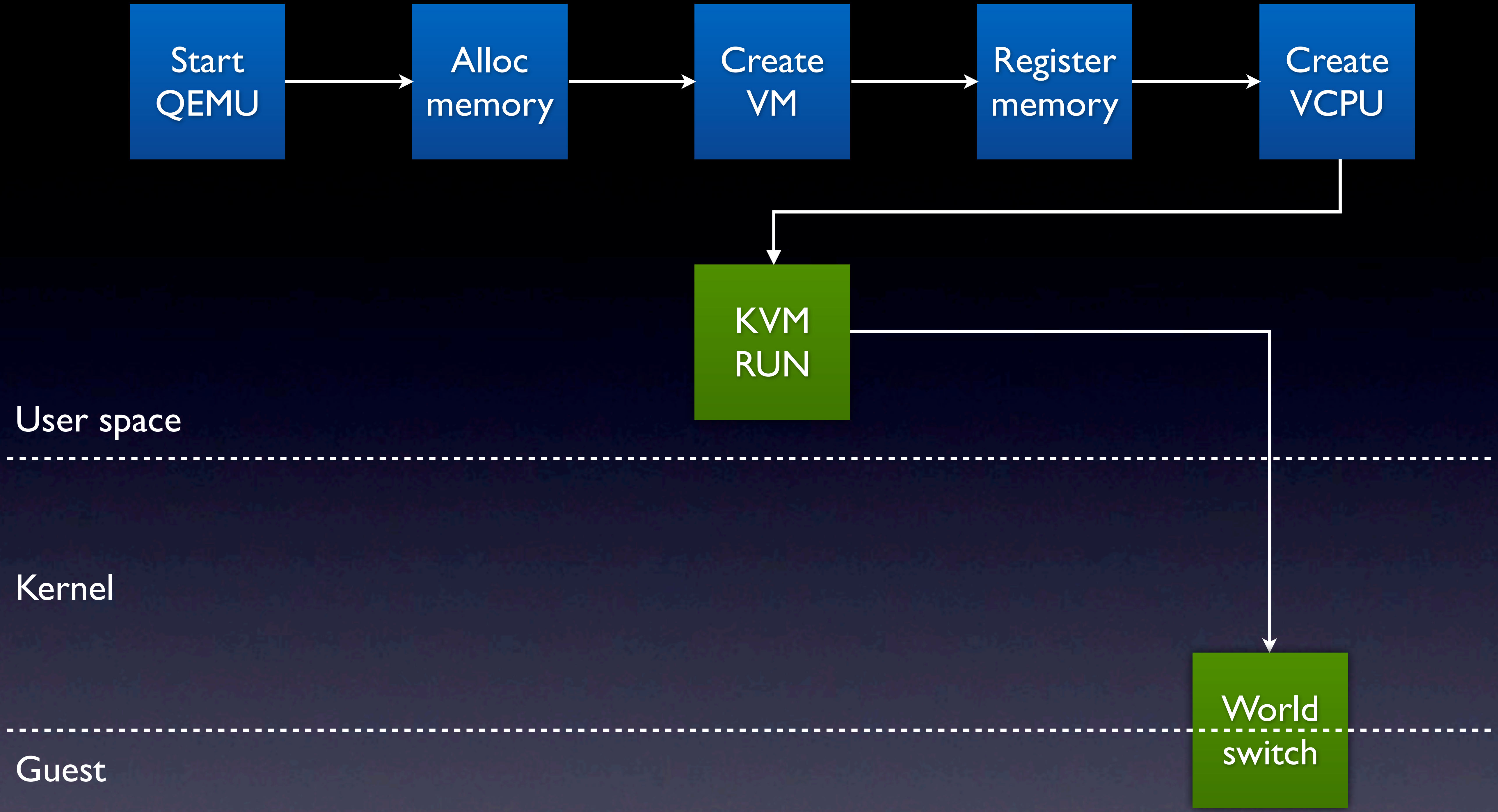
Create VM

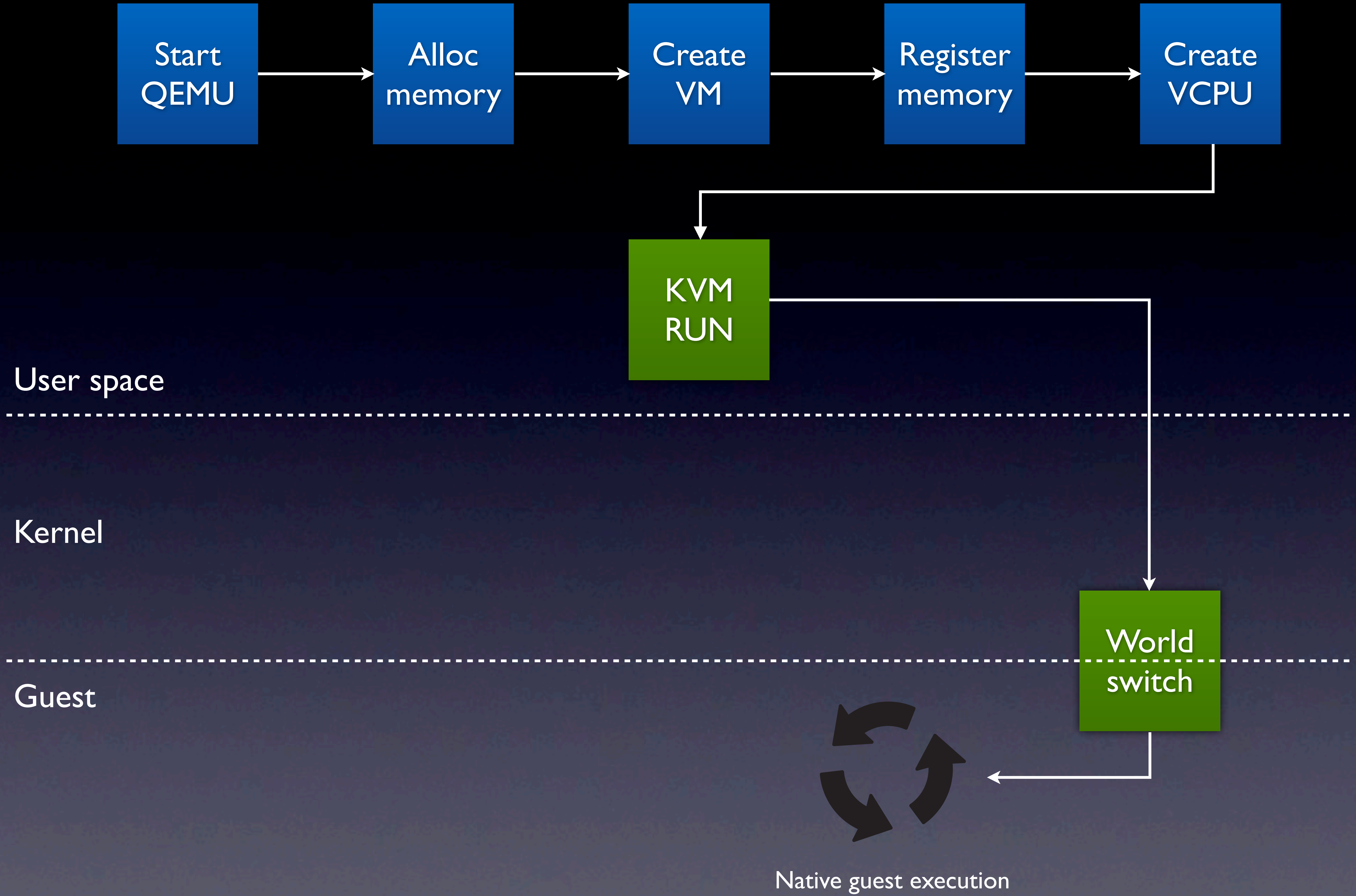
Register memory

Create VCPU

KVM RUN


Start QEMU
Alloc memory
Create VM
Register memory
Create VCPU
KVM RUN


Start QEMU
Alloc memory
Create VM
Register memory
Create VCPU
KVM RUN
World switch
User space
Kernel
Guest



Start QEMU

Alloc memory

Create VM

Register memory

Create VCPU

KVM RUN

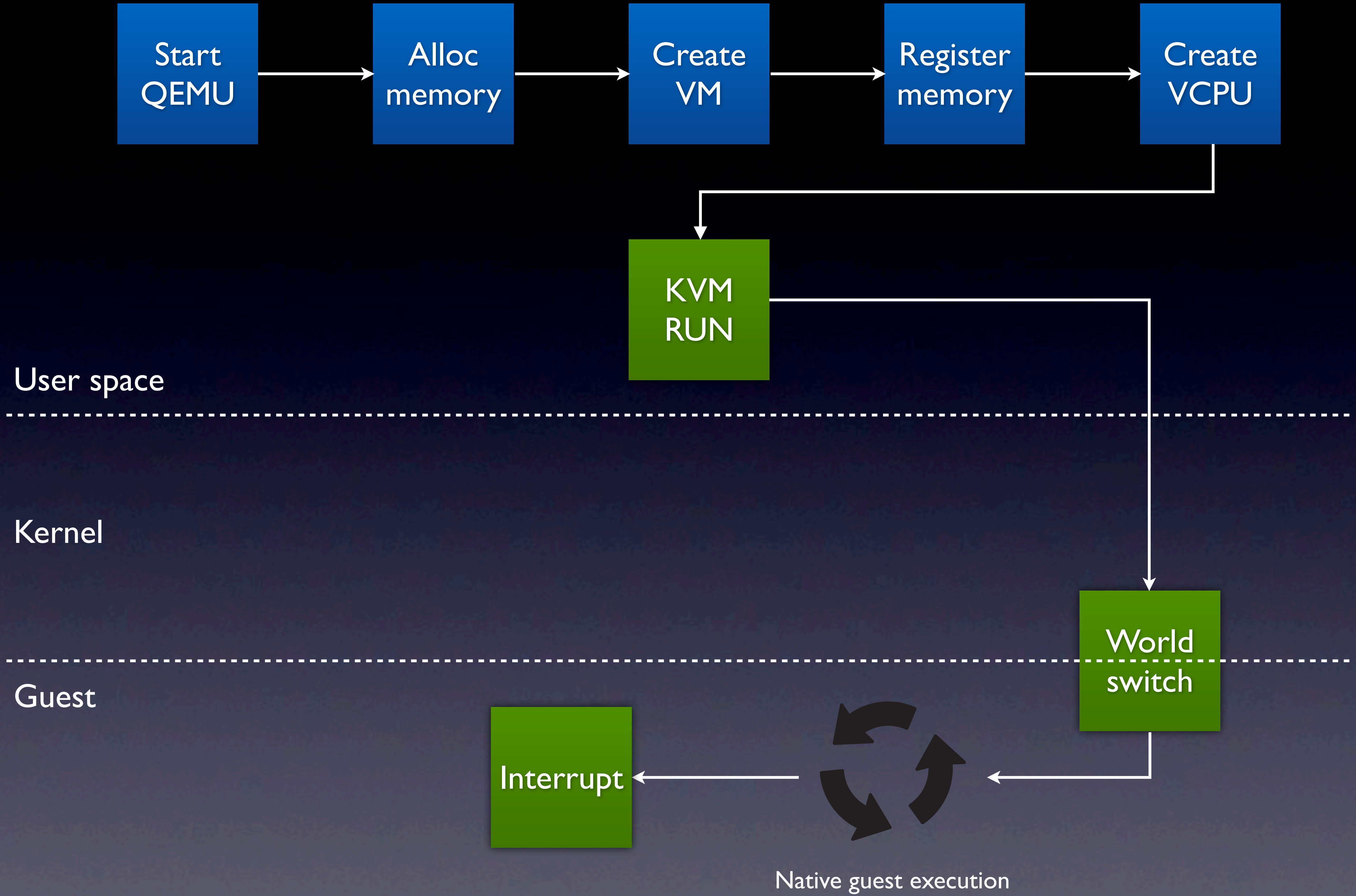
User space

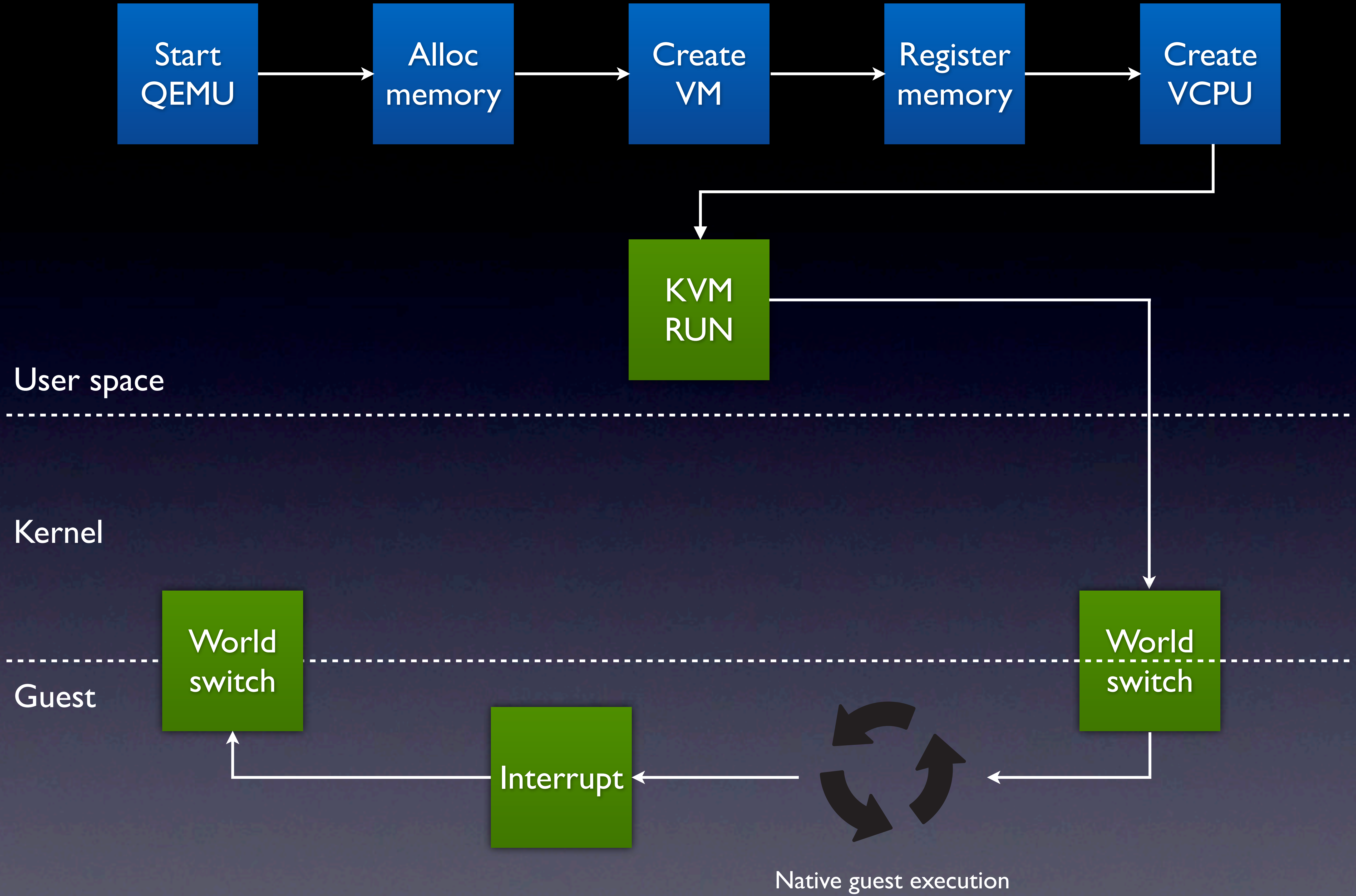
Kernel

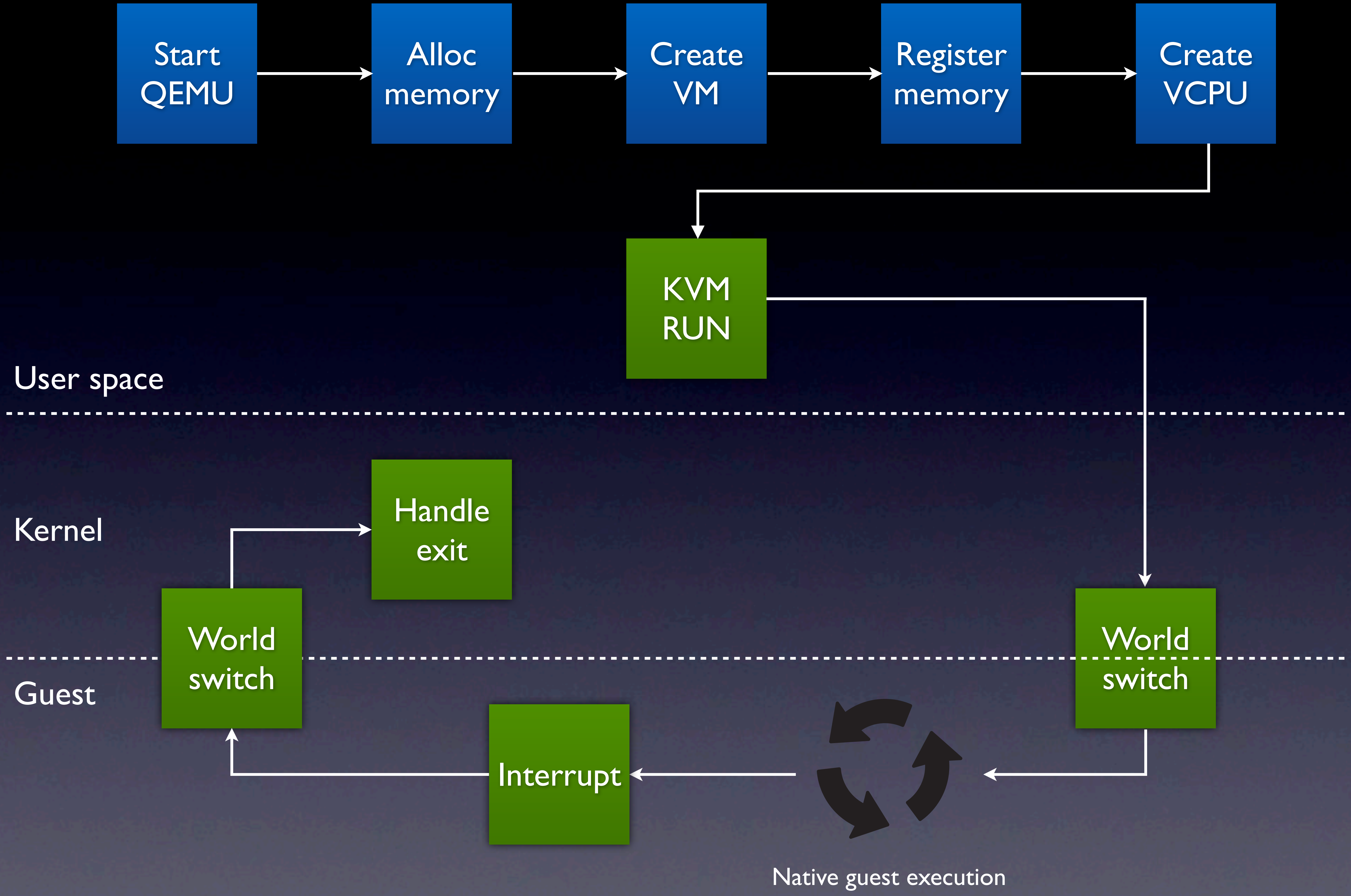
World switch

Guest

Native guest execution


Start QEMU
Alloc memory
Create VM
Register memory
Create VCPU
KVM RUN
User space
Kernel
World switch
Guest
Interrupt
Native guest execution


Start QEMU
Alloc memory
Create VM
Register memory
Create VCPU
KVM RUN
User space
Kernel
Guest
World switch
World switch
Interrupt
Native guest execution



Start QEMU

Alloc memory

Create VM

Register memory

Create VCPU

KVM RUN

User space

Kernel

Handle exit

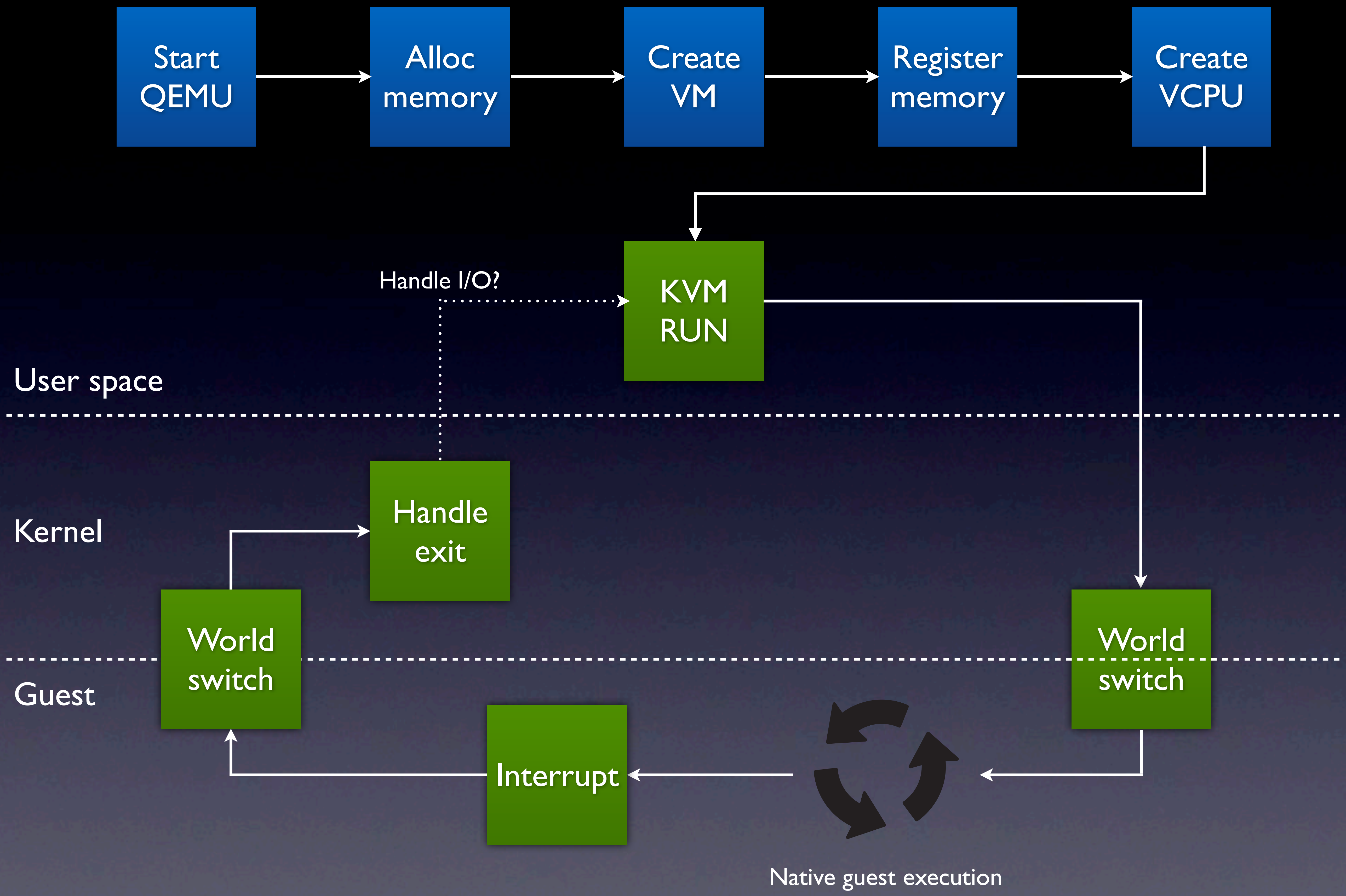
World switch

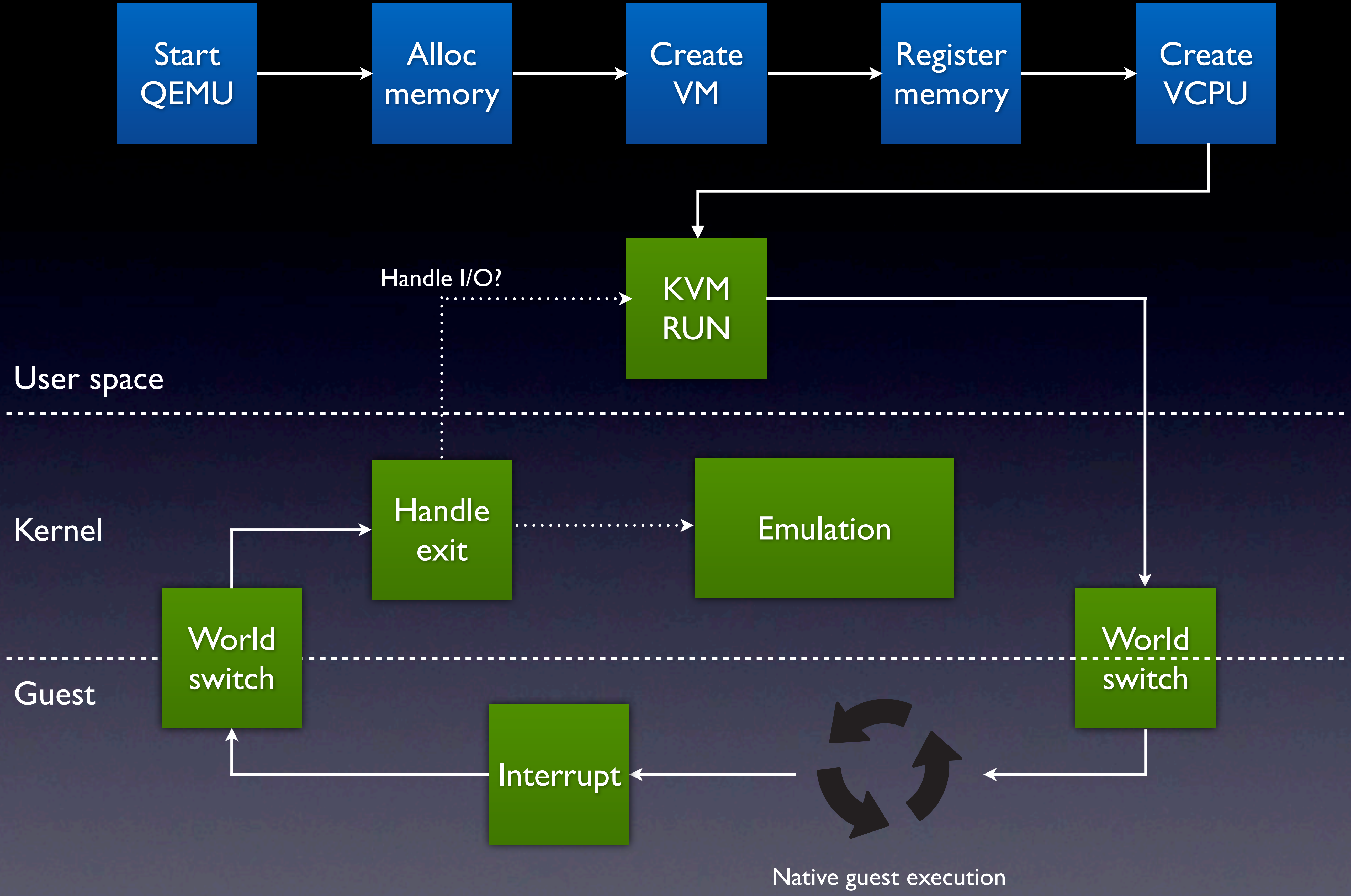
World switch

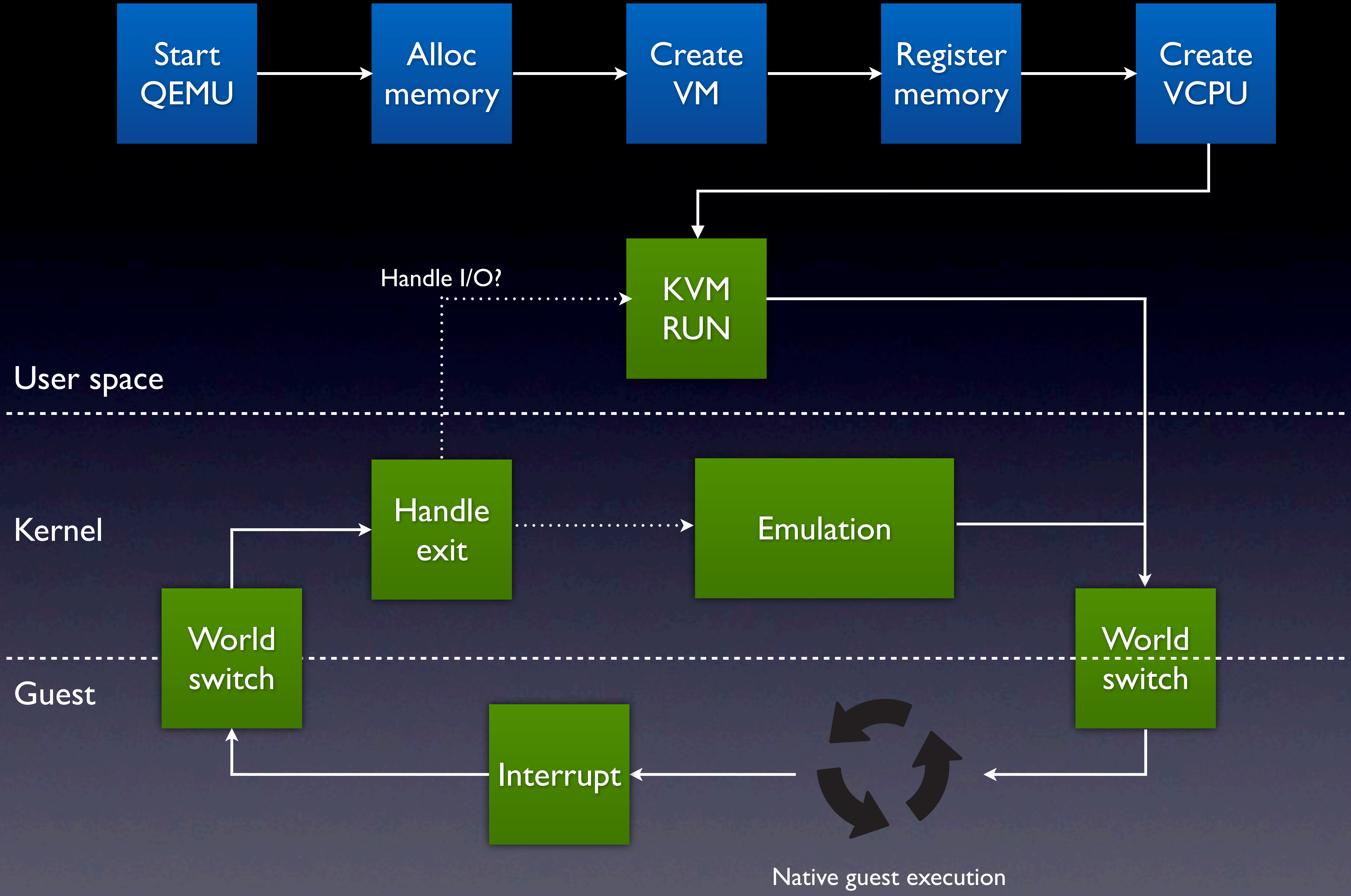
Guest

Interrupt

Native guest execution


Start QEMU
Alloc memory
Create VM
Register memory
Create VCPU
Handle I/O?
KVM RUN
User space
Kernel
Handle exit
World switch
World switch
Guest
Interrupt
Native guest execution


Start QEMU
Alloc memory
Create VM
Register memory
Create VCPU
KVM RUN
Handle I/O?
User space
Kernel
Handle exit
Emulation
World switch
World switch
Guest
Interrupt
Native guest execution


Start QEMU
Alloc memory
Create VM
Register memory
Create VCPU
KVM RUN
Handle I/O?
User space
Kernel
Handle exit
Emulation
World switch
World switch
Guest
Interrupt
Native guest execution

New KVM architecture

- Logical separation of architecture dependent and independent code
  - `kvm_arch_XXX`
  - `kvm_XXX`

- KVM
- CPU virtualization on ARM
- Memory virtualization on ARM
- World Switch details
- Implementation status

ARM virtualization

- ARM is not virtualizable - nor does it have hardware virtualization support
- Possible solutions:
  - binary translation
  - or paravirtualization

Binary Translation

- Traditionally done out-of-place with a translation cache
- Difficult to make it fast
- Contradicts idea of KVM

Paravirtualization

- Changes the guest kernel to replace code with sensitive instructions with hypercalls
- Guest kernel is modified by hand
- Hard to merge changes with upstream Kernel versions

Lightweight-paravirtualization (LPV)

Original code:

```
mrs r2, cpsr  @ get current mode
tst r2, #3    @ not user?
bne not_angel
```

Lightweight-paravirtualization (LPV)

Original code:

```
mrs r2, cpsr  @ get current mode
tst r2, #3    @ not user?
bne not_angel
```

Lightweight-paravirtualization (LPV)

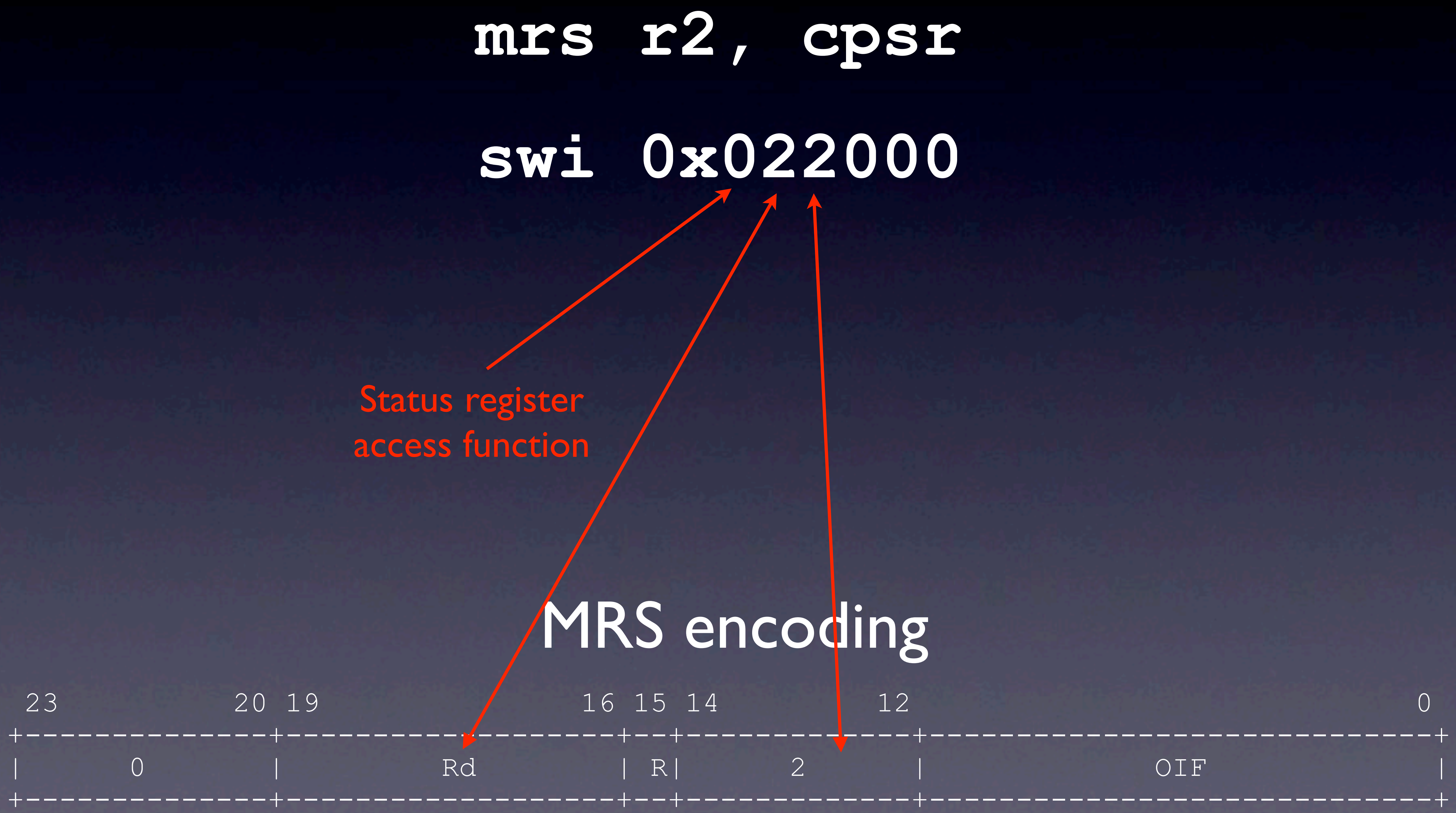
Original code:

```
swi 0x022000  @ get current mode
tst r2, #3    @ not user?
bne not_angel
```

Lightweight-paravirtualization (LPV)

- Replace sensitive instructions with traps
- Traps encode original instruction and operands
- Emulate replaced instructions in KVM
- Script-based solution applicable to any vanilla kernel tree

LPV encoding example



```
mrs r2, cpsr

swi 0x022000
```

Status register access function

MRS encoding

```
 23          20 19               16 15 14         12                             0
+--------------+-------------------+--+-------------+-----------------------------+
|      0       |         Rd        | R|      2      |             OIF             |
+--------------+-------------------+--+-------------+-----------------------------+
```

LPV implementation

- Uses regular expressions to search for sensitive assembly instructions
- ~150 lines (written in Python)
- Supports inline assembler, preprocessor macros and assembler files.

LPV requirements

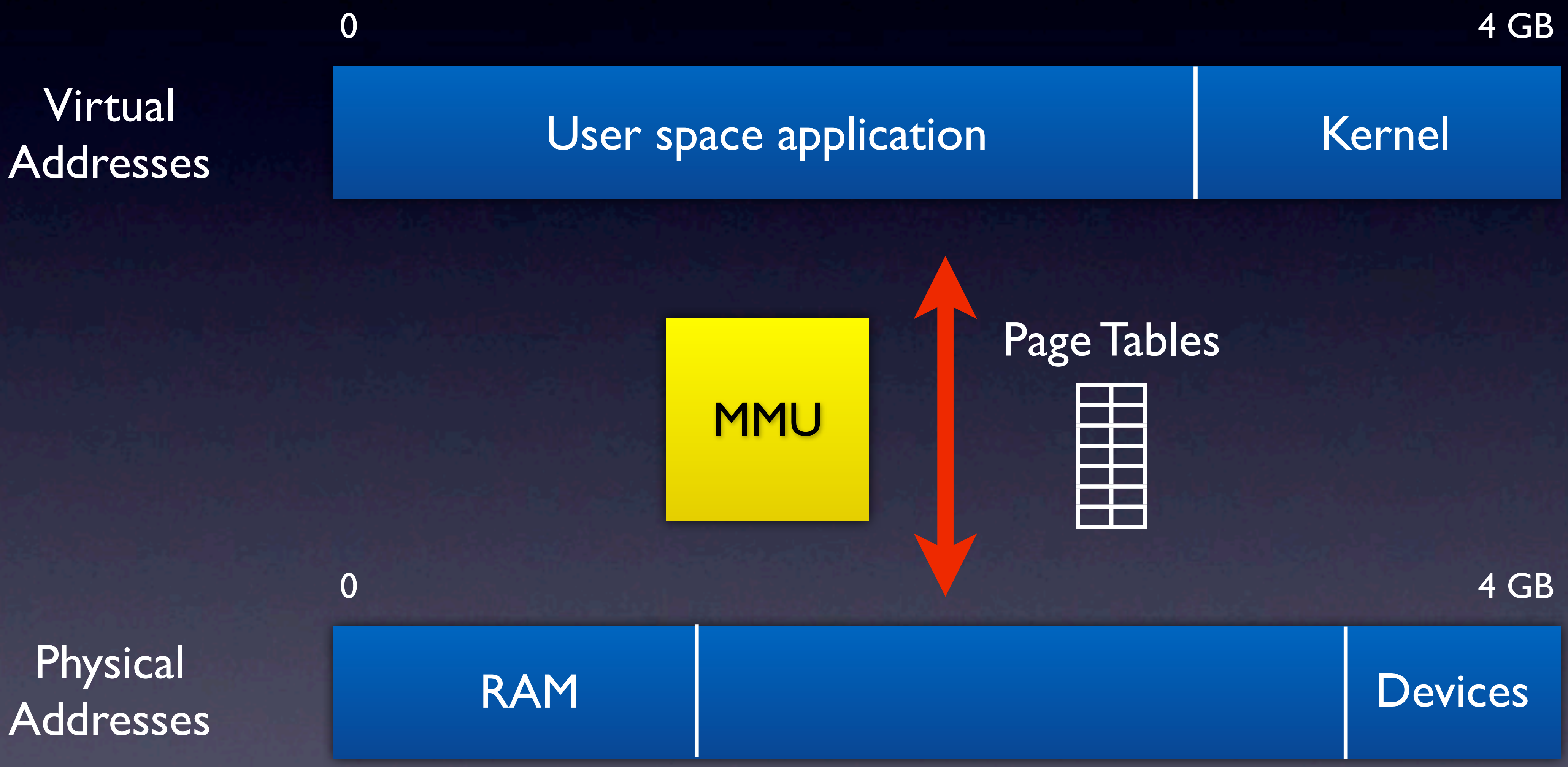
- Assumes guest kernel does not make system calls to itself
- Module source code must also be handled
- GCC does not generate sensitive instructions from C-code

LPV key points

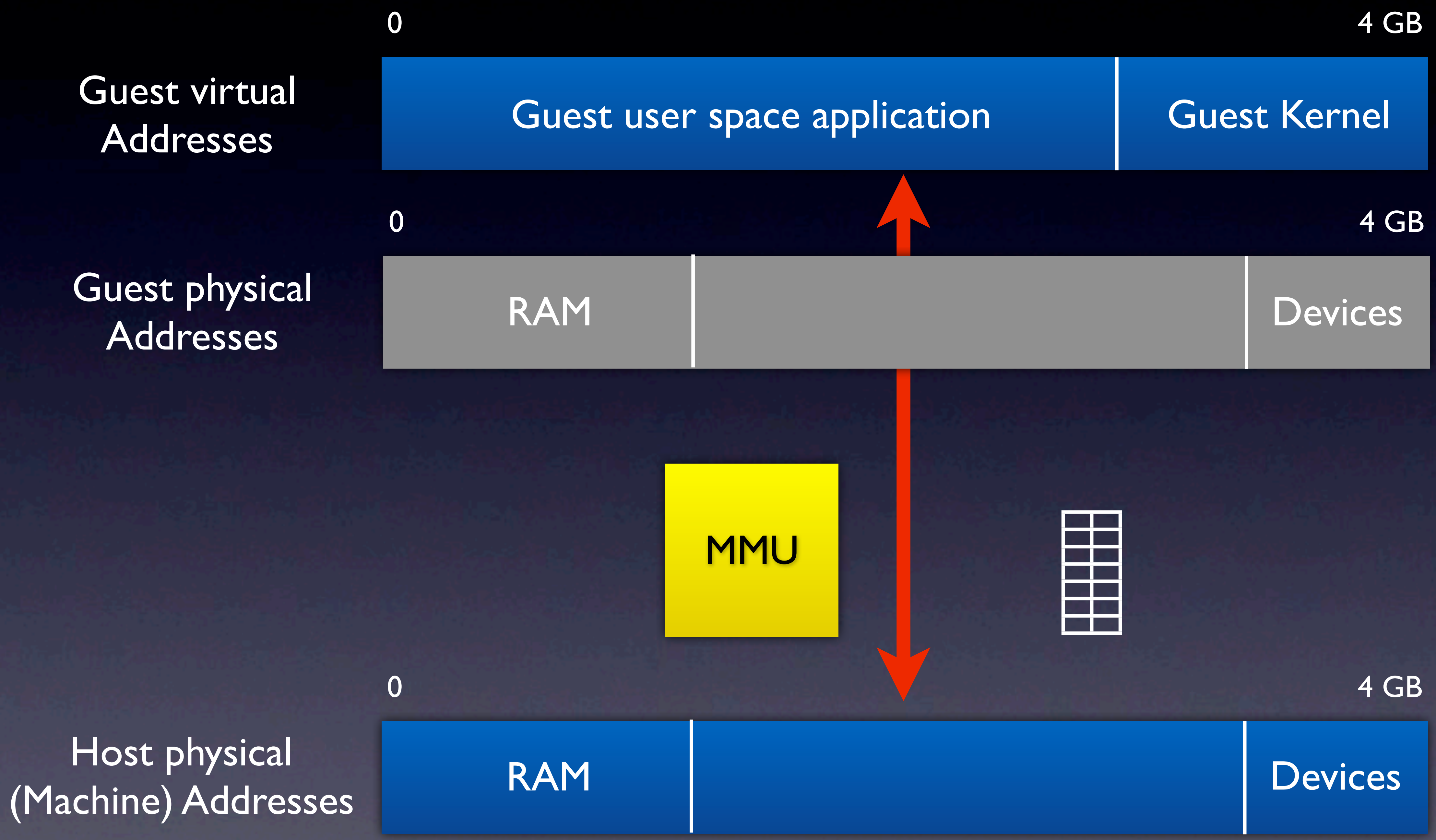
- Encodes each sensitive instructions to a single trap
- As efficient as trap-and-emulate
- Fully automated
- Doesn’t affect kernel code size

- KVM
- CPU virtualization on ARM
- Memory virtualization on ARM
- World Switch details
- Implementation status

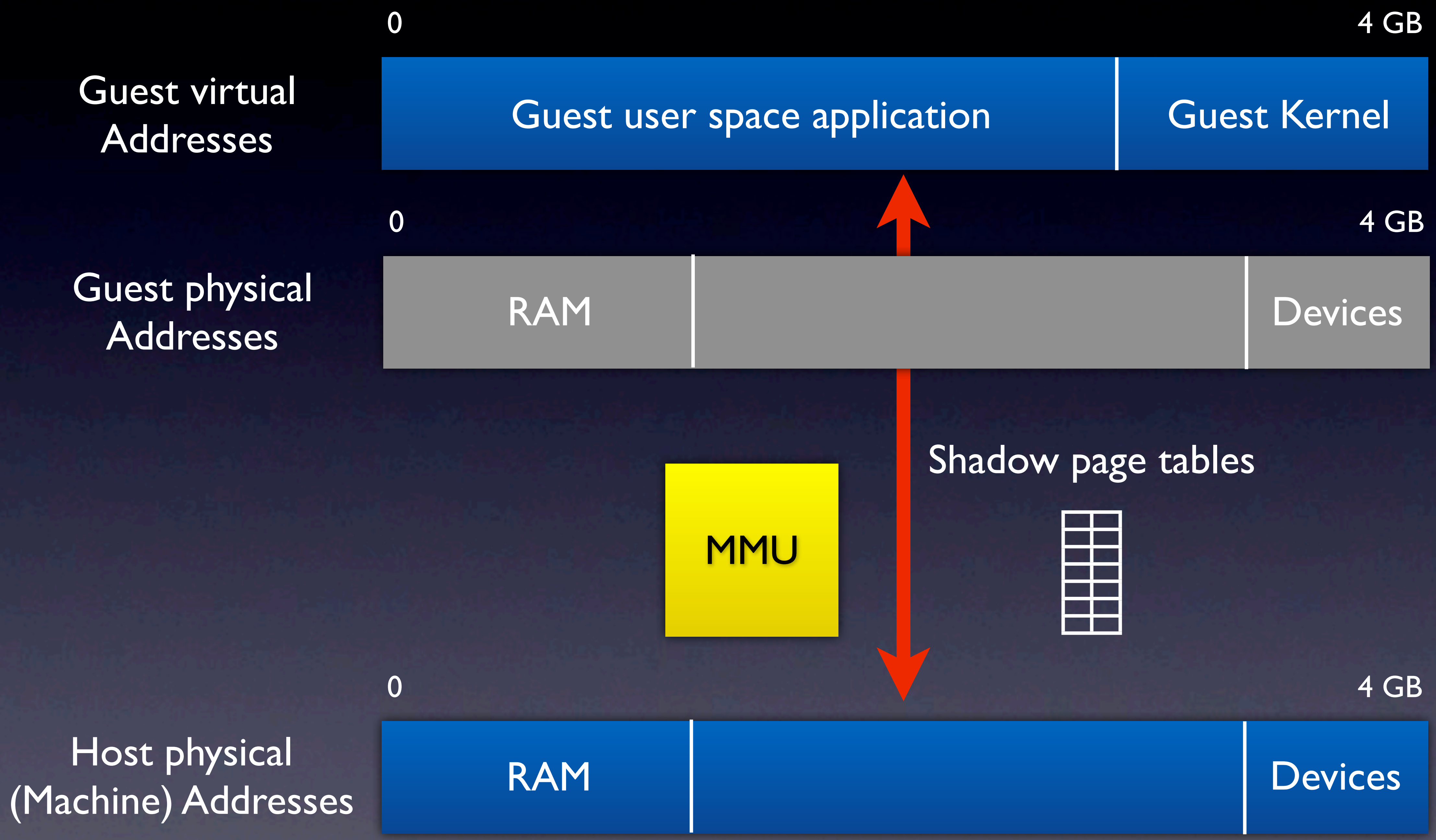
Virtual memory



New address space



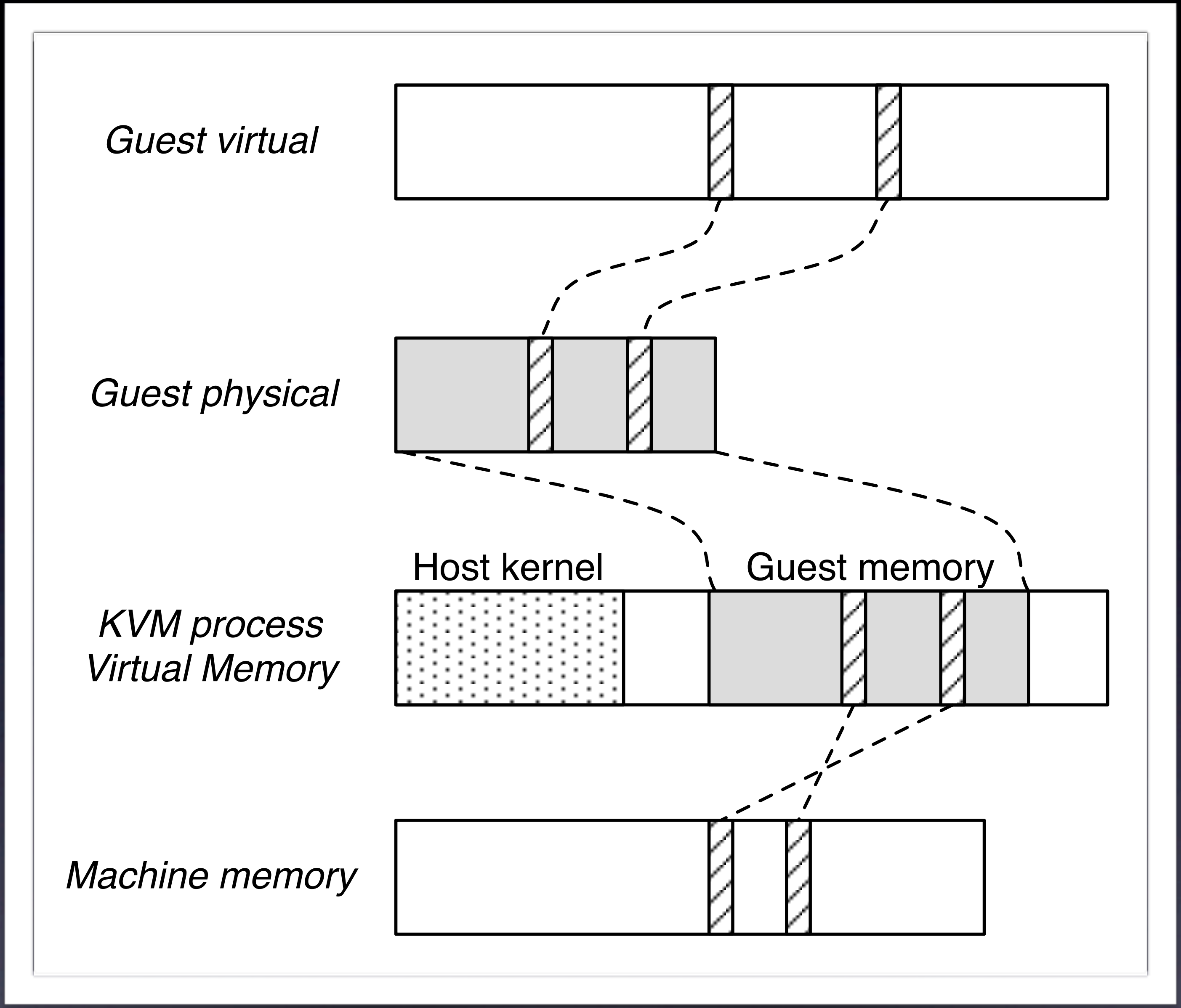
New address space



Shadow page tables

- Map
  - Guest Virtual Addresses to
  - Host Physical Addresses
- One per guest page table (process)
- Start out empty and add entries on page faults (on demand)

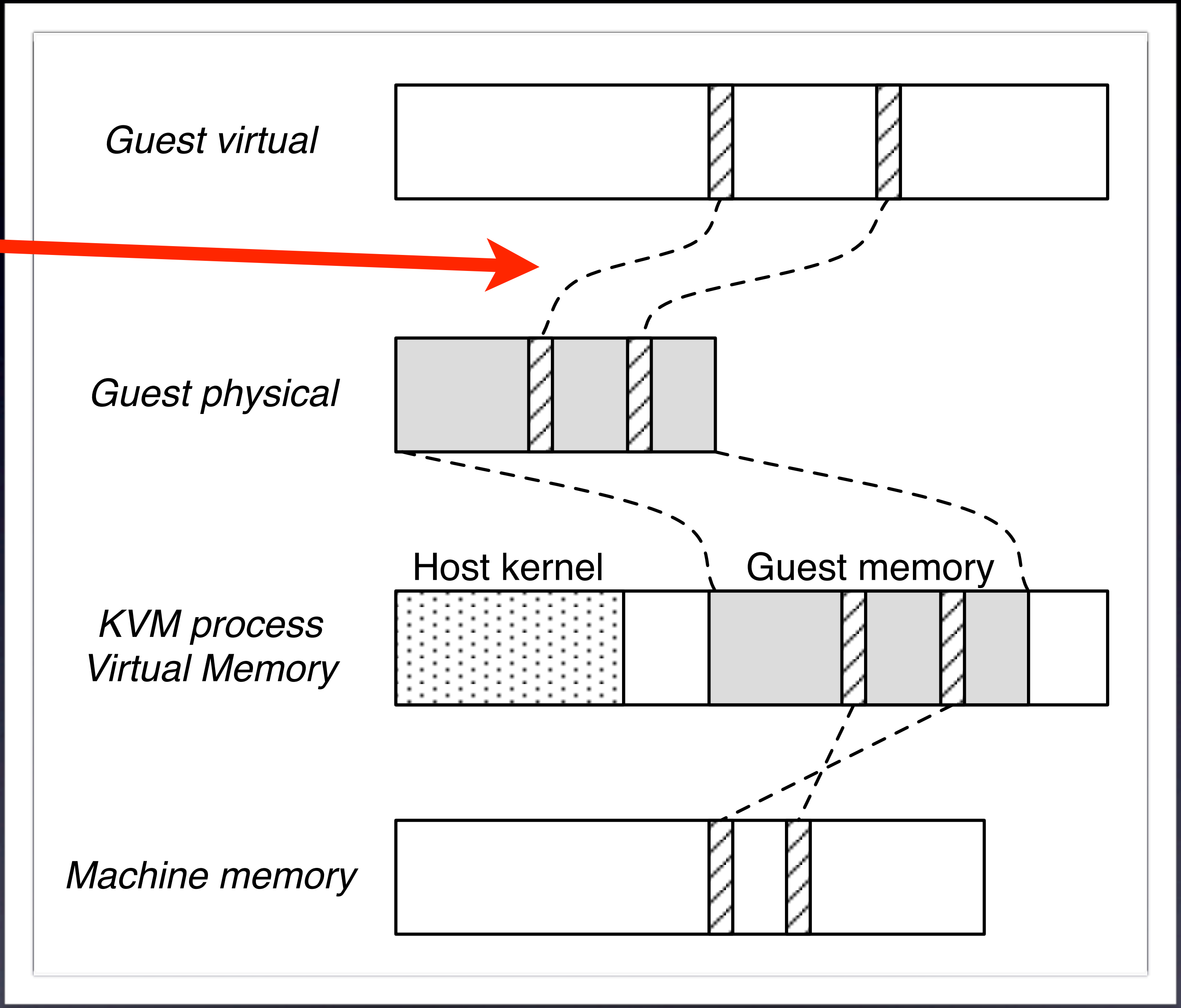
Address translation



Address translation

Walk guest page tables in software:

```
gva_to_gfn(...);
```



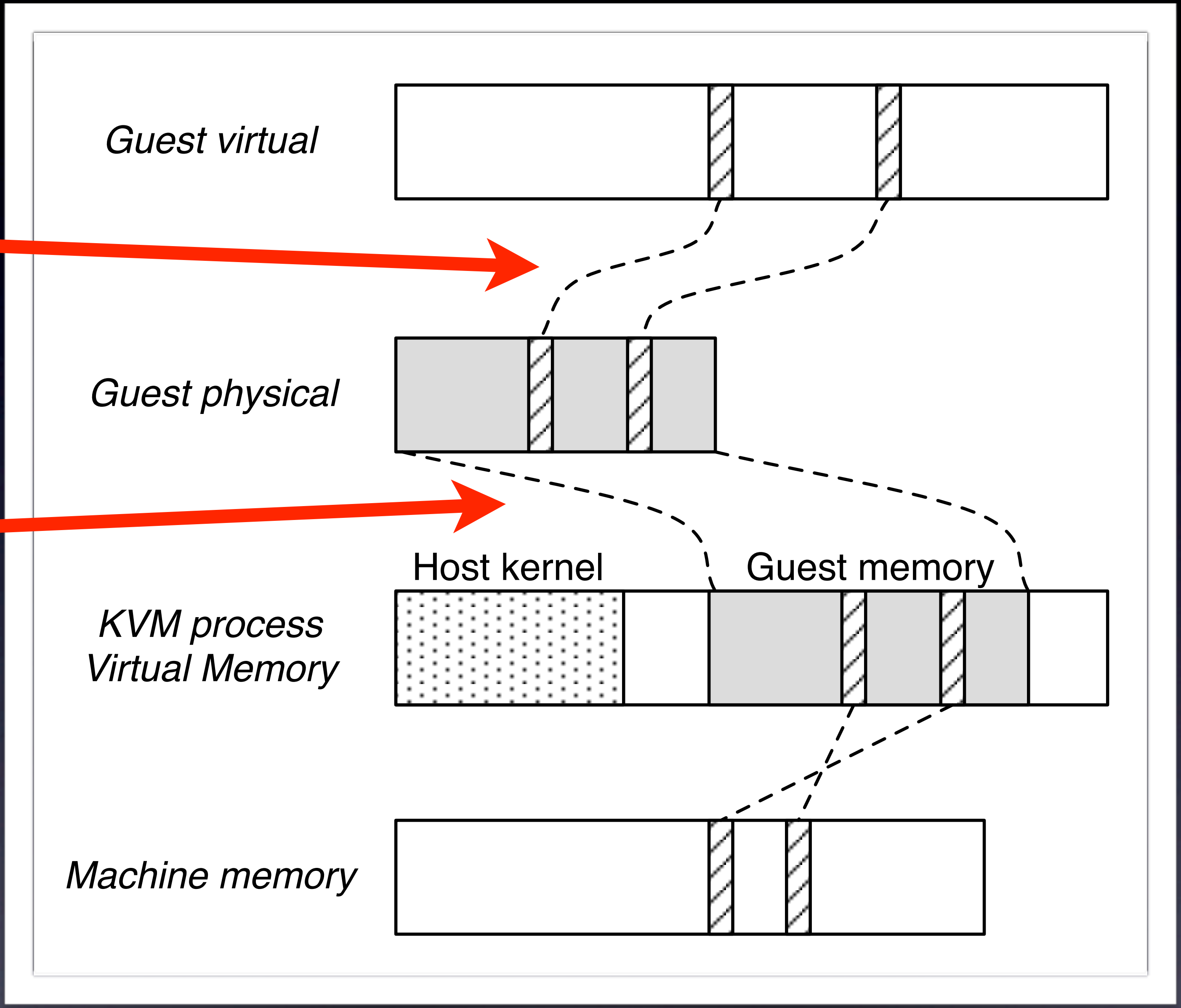
Address translation

Walk guest page tables in software:

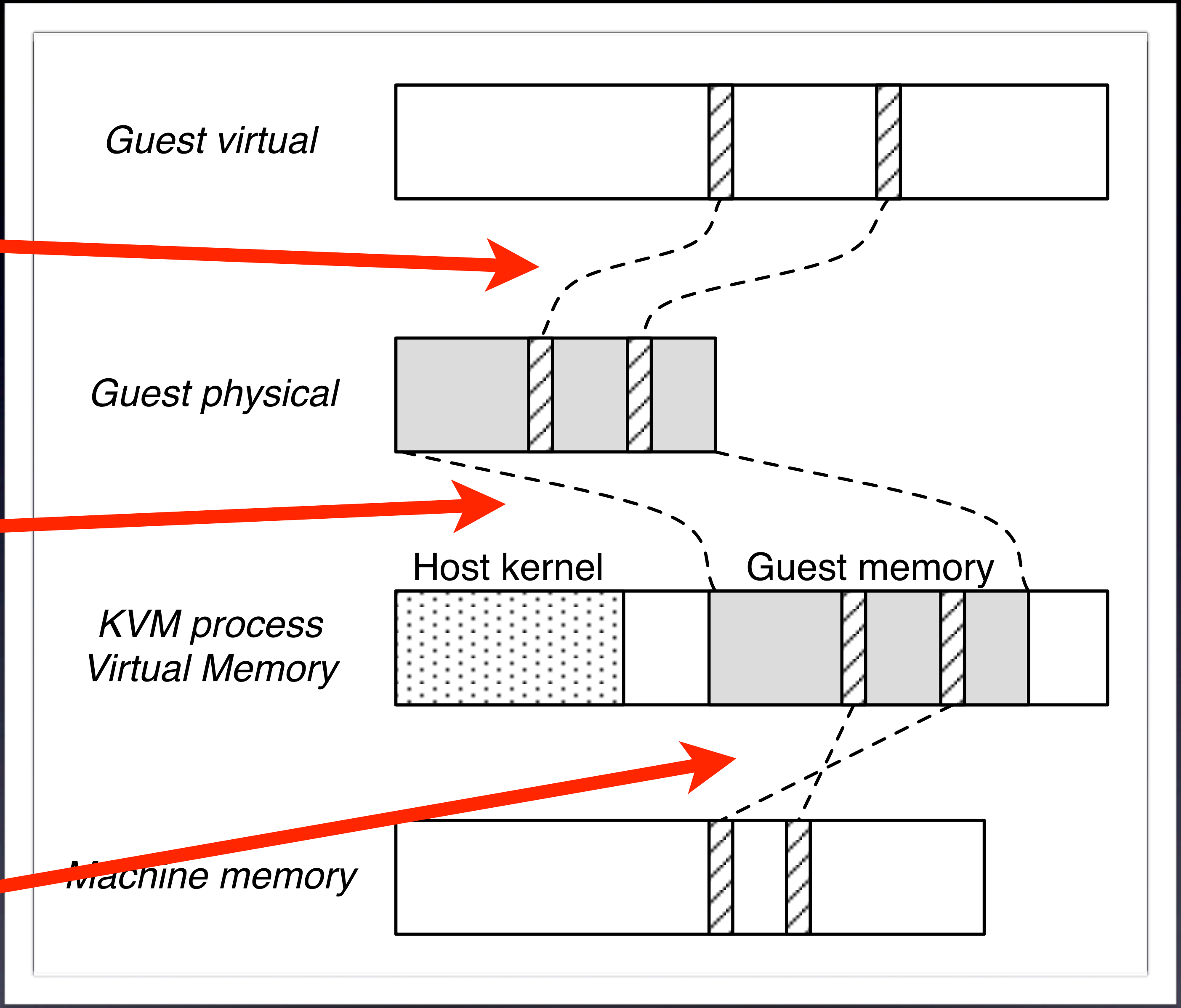
```
gva_to_gfn(...);
```

Built-in KVM functionality:

```
gfn_to_hva(...);
```



Address translation



Walk guest page tables in software:
```
gva_to_gfn(...);
```

Built-in KVM functionality:
```
gfn_to_hva(...);
```

Kernel functionality:
```
page = virt_to_page(...);
pfn = page_to_pfn(page);
```

Shadow page table consistency

- Caching shadow page tables is an optimization
- Keep cached page tables in sync by protecting guest page tables and tracking updates

Memory Protection

- Goal
  - Protect host from guest
  - Honor intended guest protection
- ARM provides flexible protection methods
- Access is specified per CPU privilege level

Access Protection Bits

AP	Privileged	User
00	None	None
01	R/W	None
10	R/W	R/O
11	R/W	R/W

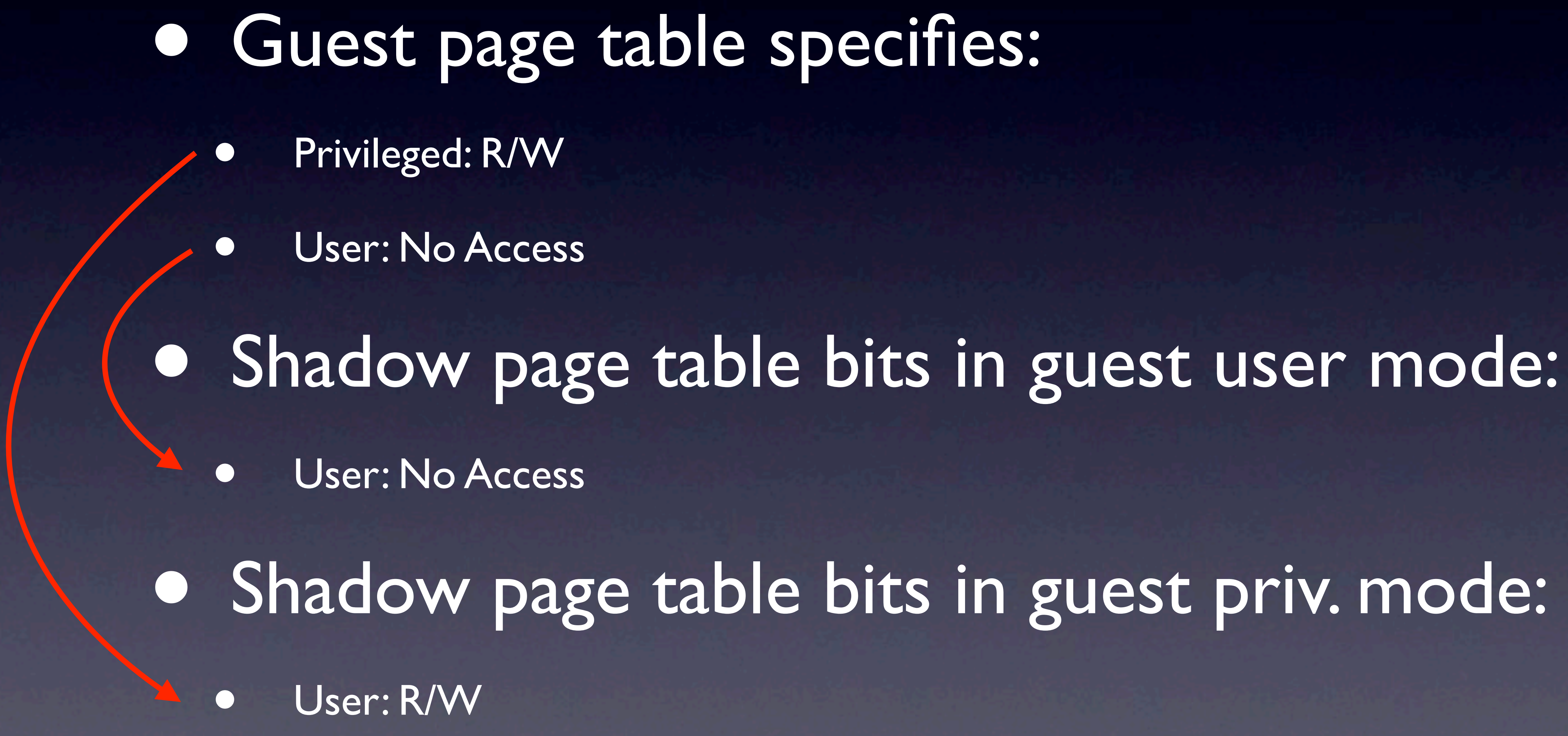
Access mapping example

- Guest page table specifies:
  - Privileged: R/W
  - User: No Access
- Shadow page table bits in guest user mode:
  - User: No Access
- Shadow page table bits in guest priv. mode:
  - User: R/W

Access mapping example

- Guest page table specifies:
  - Privileged: R/W
  - User: No Access
- Shadow page table bits in guest user mode:
  - User: No Access
- Shadow page table bits in guest priv. mode:
  - User: R/W

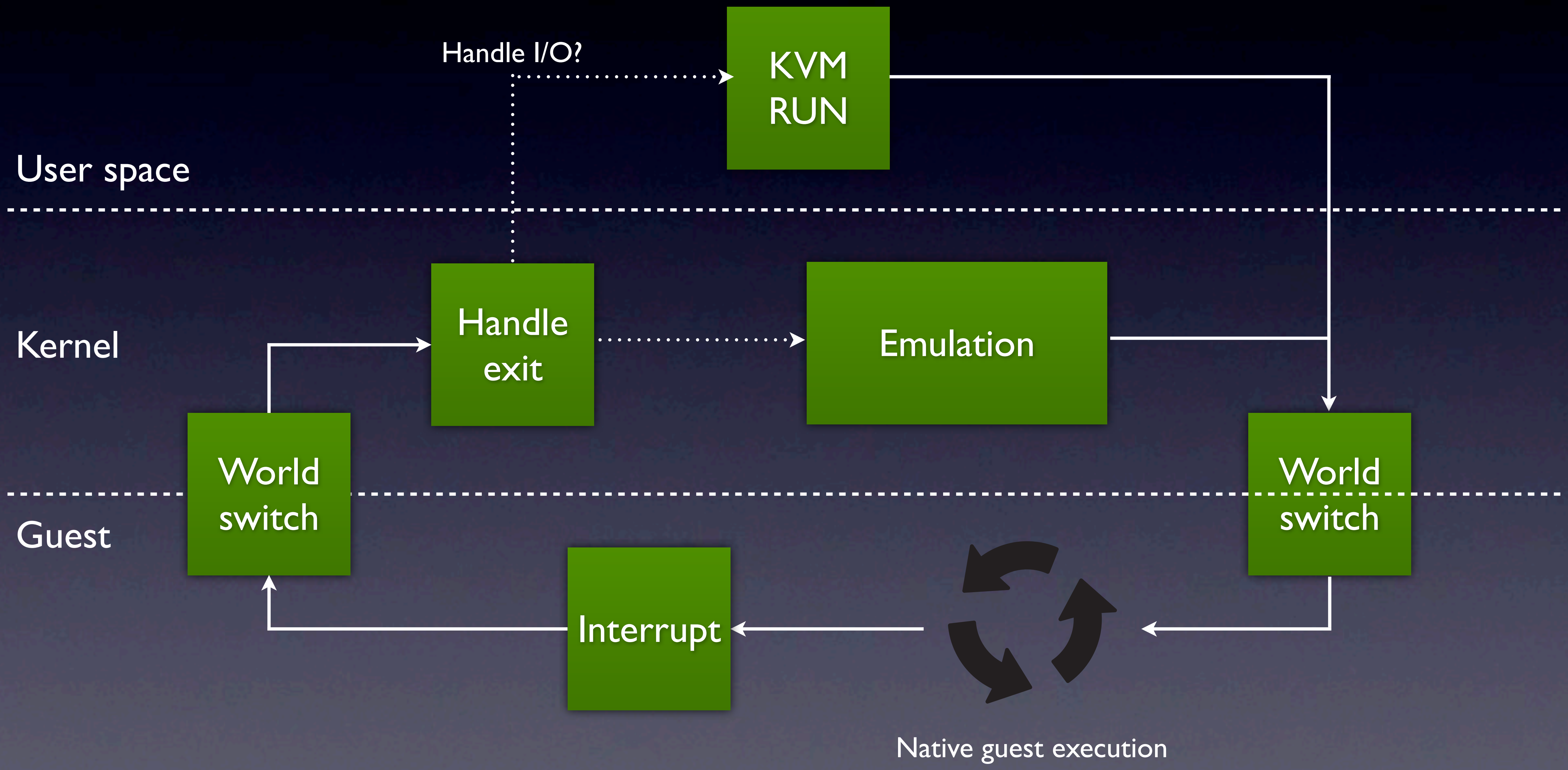
Access mapping example



- Guest page table specifies:
  - Privileged: R/W
  - User: No Access
- Shadow page table bits in guest user mode:
  - User: No Access
- Shadow page table bits in guest priv. mode:
  - User: R/W

- KVM
- CPU virtualization on ARM
- Memory virtualization on ARM
- World Switch details
- Implementation status

World Switches



World switch

To guest

- Disable interrupts
- Store host state
- Switch page tables
- Load guest state
- Enable interrupts
- Jump to guest code

From guest

- Store exit state
- Switch page tables
- Restore host state
- (Host kernel IRQ handler)
- Enable interrupts
- Return to ioctl call

World switch

To guest

- Disable interrupts
- Store host state
- Switch page tables
- Load guest state
- Enable interrupts
- Jump to guest code

From guest

- Store exit state
- Switch page tables
- Restore host state
- (Host kernel IRQ handler)
- Enable interrupts
- Return to ioctl call

World switch

To guest

- Disable interrupts
- Store host state
- Switch page tables
- Load guest state
- Enable interrupts
- Jump to guest code

From guest

- Store exit state
- Switch page tables
- Restore host state
- (Host kernel IRQ handler)
- Enable interrupts
- Return to ioctl call

World switch

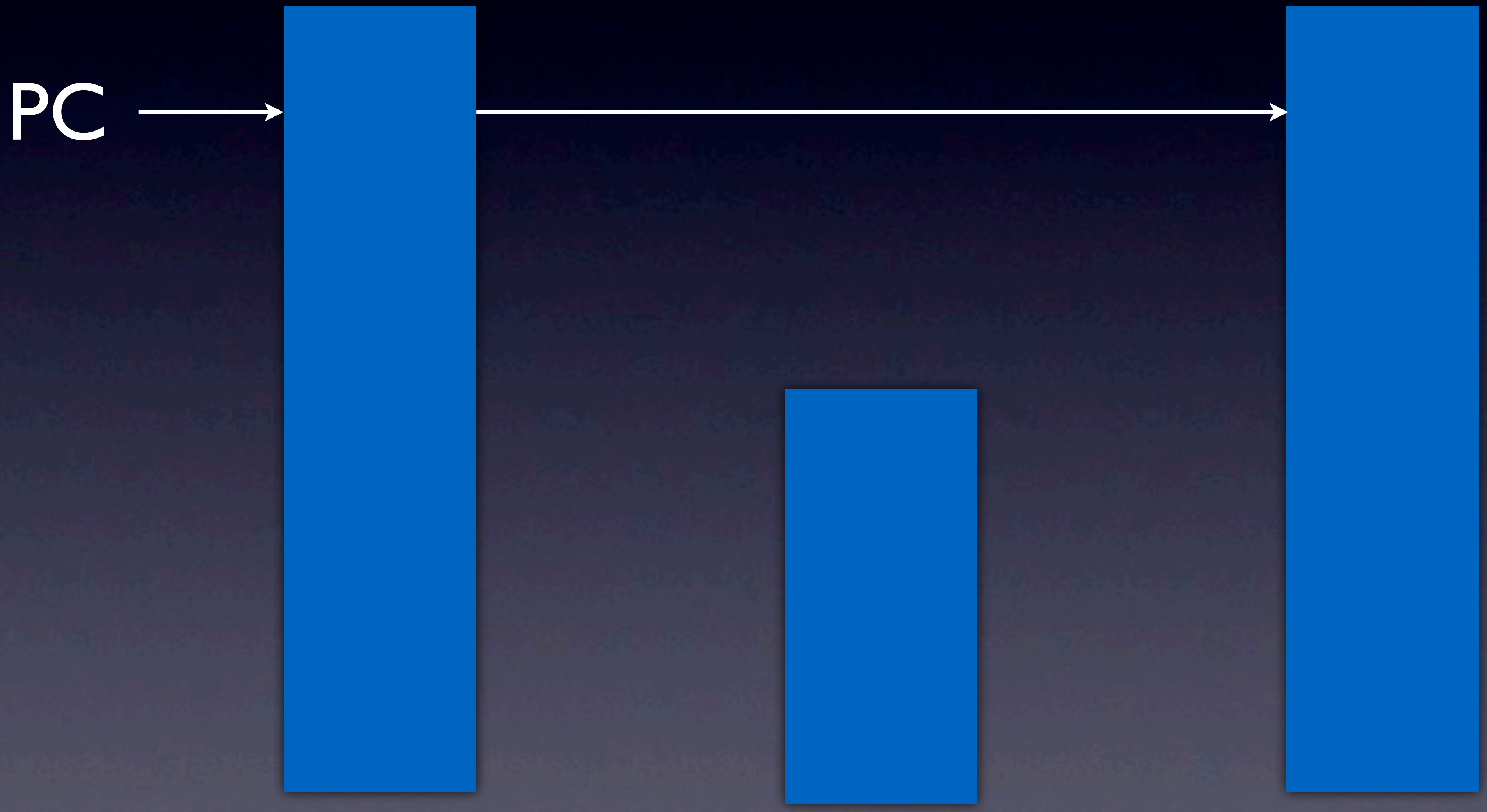
To guest

- Disable interrupts
- Store host state
- Switch page tables
- Load guest state
- Enable interrupts
- Jump to guest code

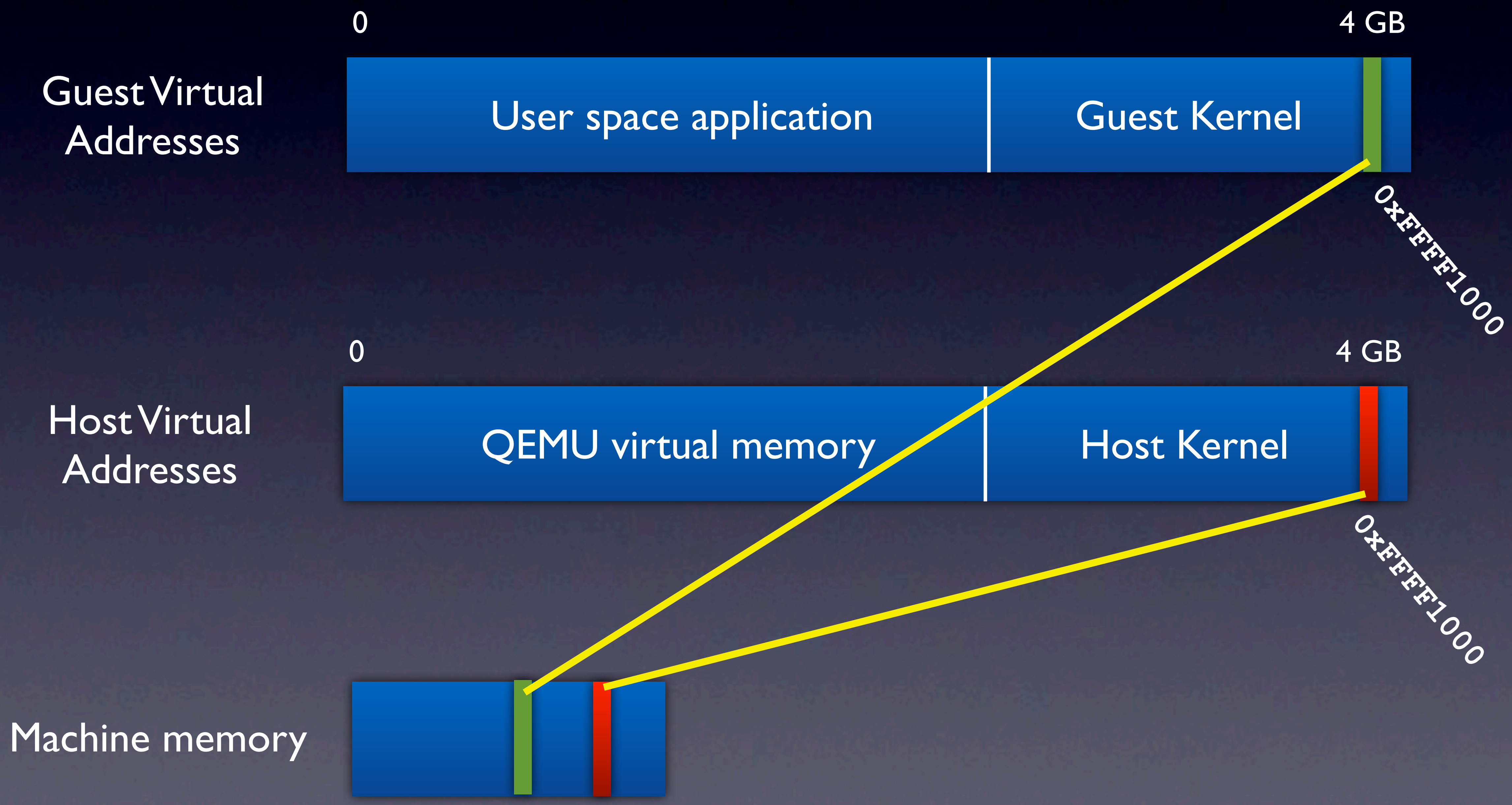
From guest

- Store exit state
- Switch page tables
- Restore host state
- (Host kernel IRQ handler)
- Enable interrupts
- Return to ioctl call

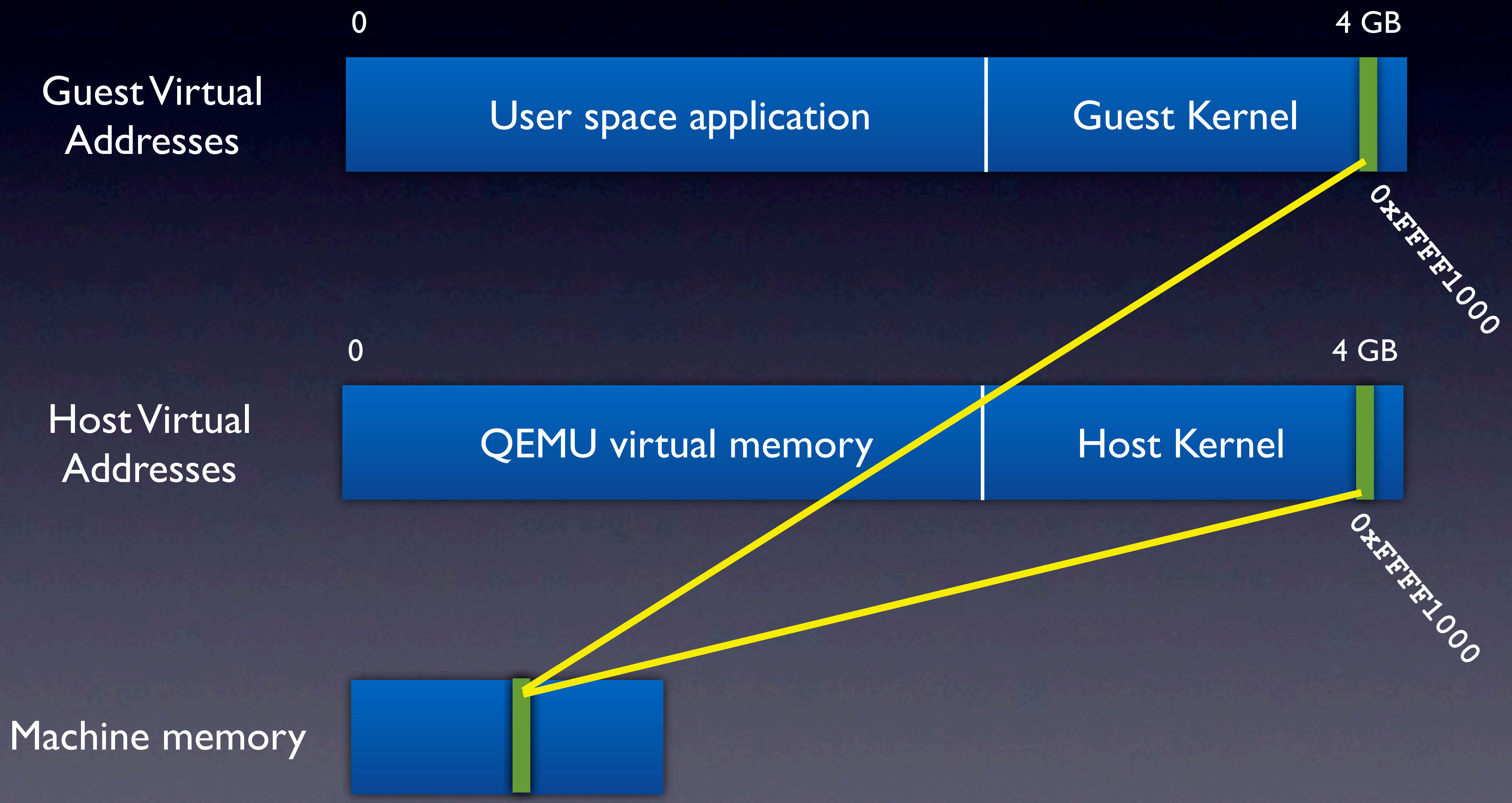
Switch page tables



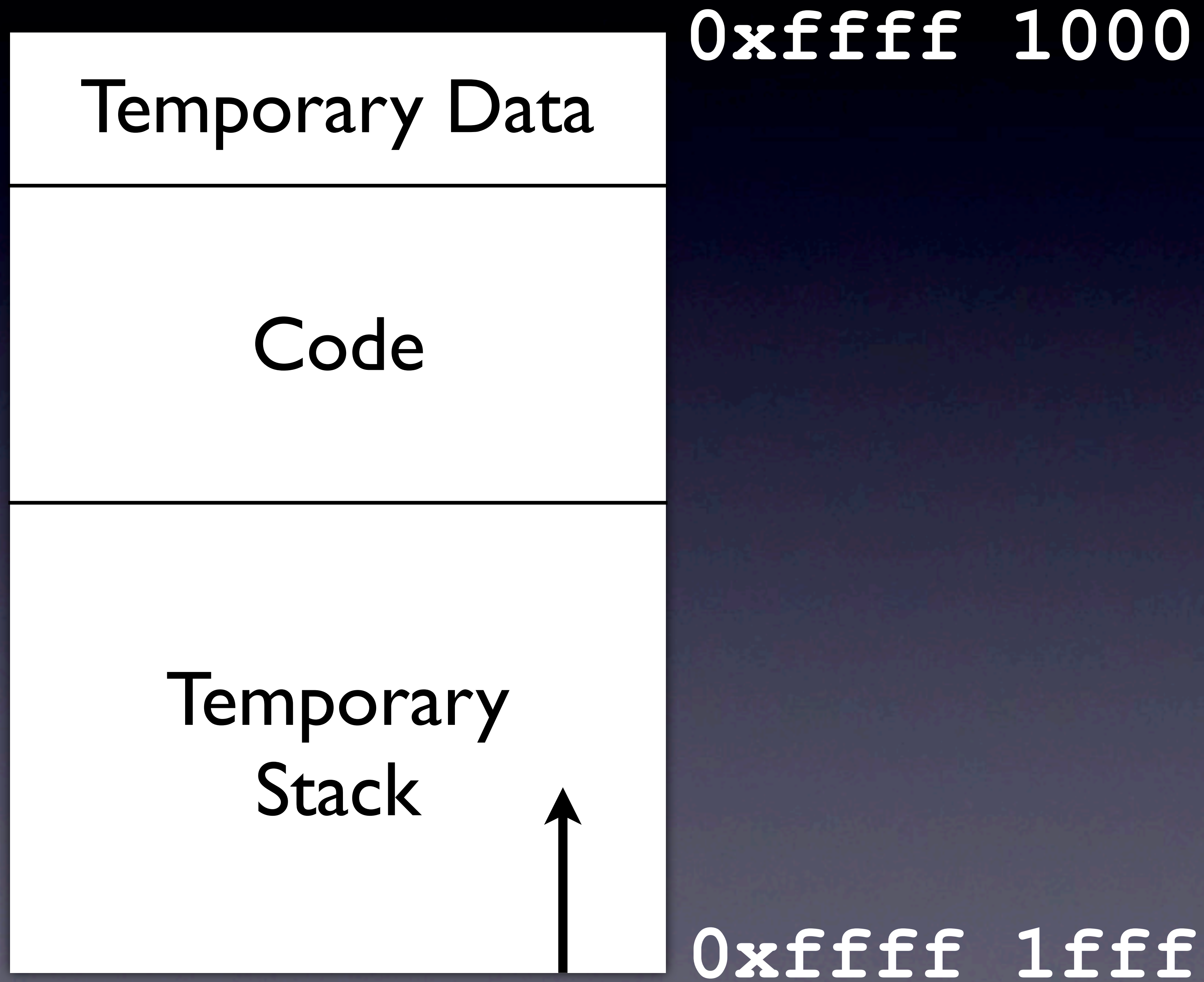
Shared Page


0
4 GB
Guest Virtual Addresses
User space application
Guest Kernel
0xFFFF1000
0
4 GB
Host Virtual Addresses
QEMU virtual memory
Host Kernel
0xFFFF1000
Machine memory

Shared Page


0
4 GB
Guest Virtual Addresses
User space application
Guest Kernel
0xFFFF1000
0
4 GB
Host Virtual Addresses
QEMU virtual memory
Host Kernel
0xFFFF1000
Machine memory

Shared Page Internals



- KVM
- CPU virtualization on ARM
- Memory virtualization on ARM
- World Switch details
- Implementation status

Status

- Successfully boots Linux VMs
- Host built on Android Kernel 2.6.27
- Tested guest kernels from 2.6.17 to 2.6.33

Future work

- Improve performance
  - Cache shadow page tables
  - Avoid unnecessary world-switches
- Binary patching
- Test device support
- Upstream!

ARMv6

- Physically tagged caches
- TLB “Application Space Identifiers” (ASID’s)
- New instructions

Related Work

- Commercial solutions:
  - VMWare MVP, OK Labs, VirtualLogix, ...
- Open-source:
  - QEMU
  - XenARM

Conclusions

- ARM virtualization is important
- With LPV we now have KVM/ARM
- LPV is simple, fully automated, and efficient
- Minimally intrusive
- It works!

Tasks

- Caching of shadow page tables
- Moving things to shared page
- Coalesced MMIO
- GDB support
- Testing devices (on BeagleBoards, IGEPv2 boards etc.)
- ...

Want to contribute?

- Mailing list: android-virt@lists.columbia.edu
- WIKI: http://wiki.ncl.cs.columbia.edu
- Source code: http://git.ncl.cs.columbia.edu/git

Extra Material

Use cases

- Same as on x86:
  - Test and Development
  - OS freedom
- Multiple Personas
- Virtualization features

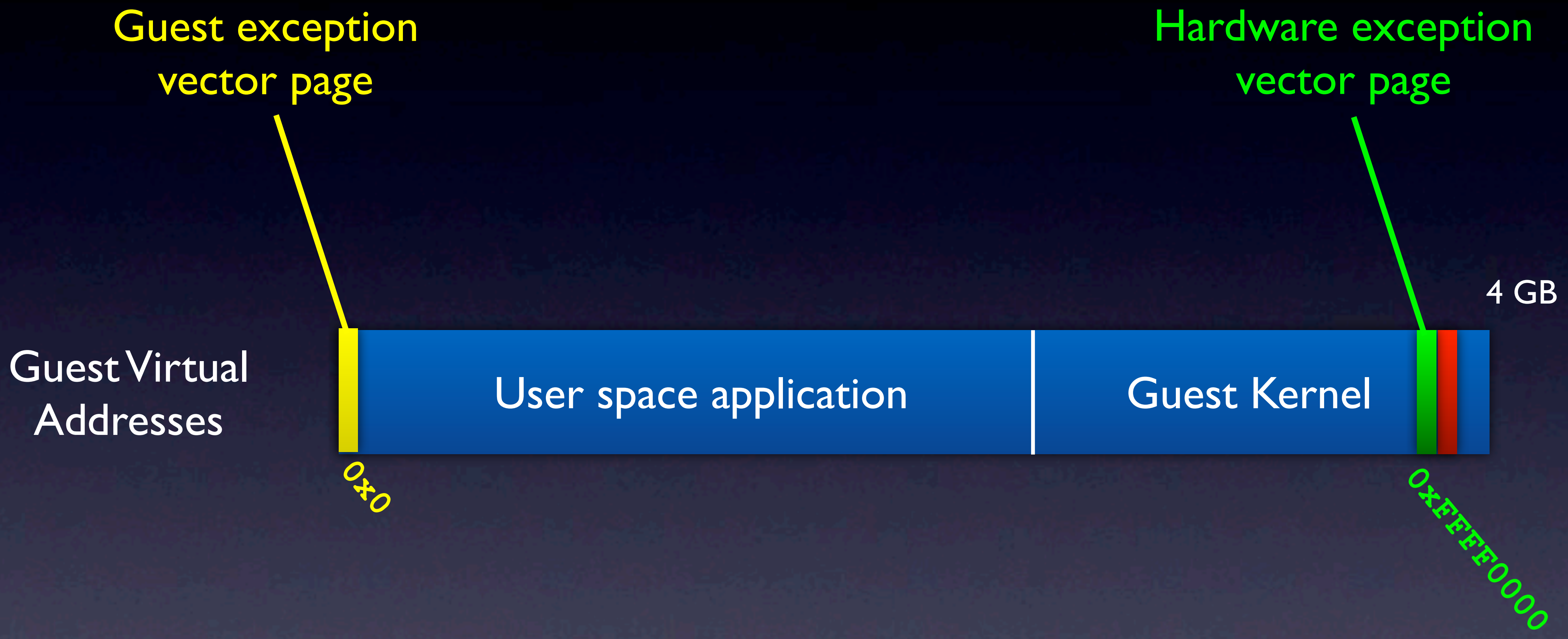
Exceptions

- Traps & Interrupts
- CPU changes mode and execution starts from “vectors” at either:
  - 0x00000000 + offset
  - or 0xFFFF0000 + offset

Exceptions and KVM/ARM

- KVM/ARM uses custom handlers to handle exceptions while executing guest
- Exceptions are the only way to:
  "exit from the guest"
- IRQ's are forwarded to the host kernel handlers
- Traps are handled by KVM/ARM

Guest exceptions

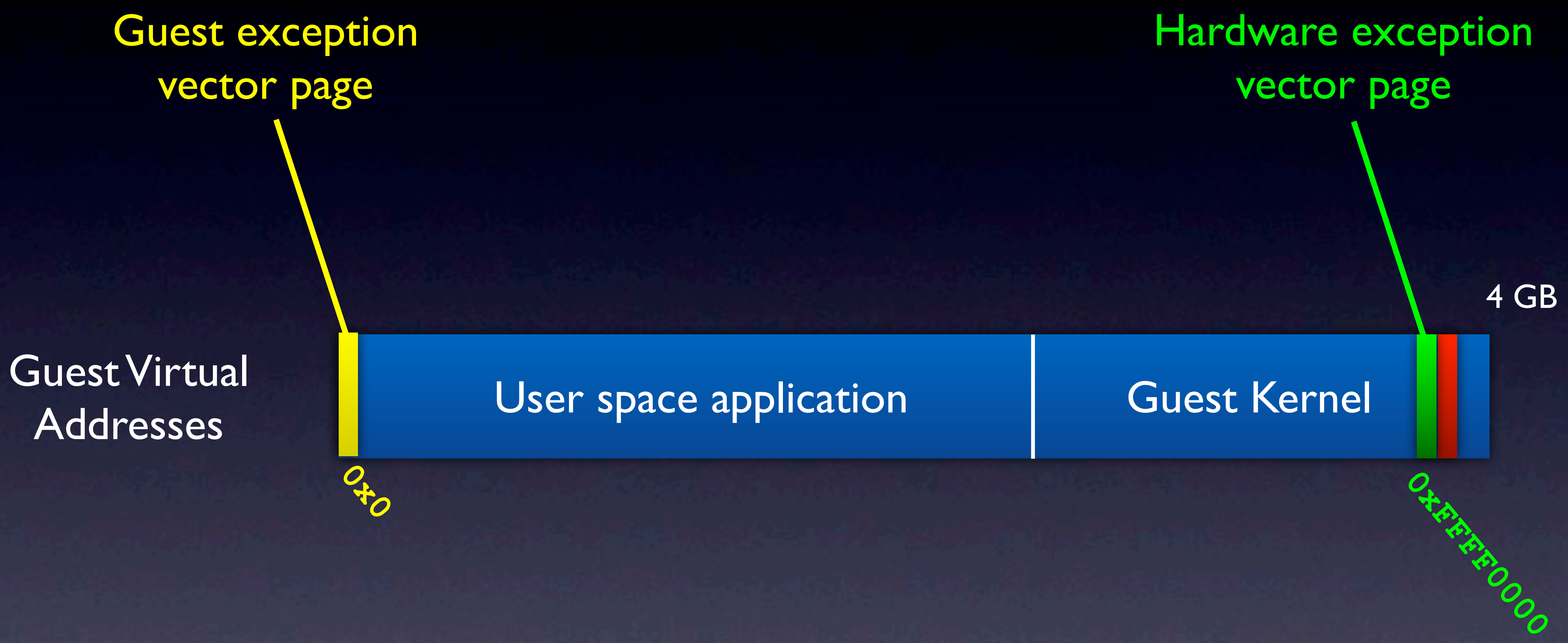


Guest uses "low" vectors

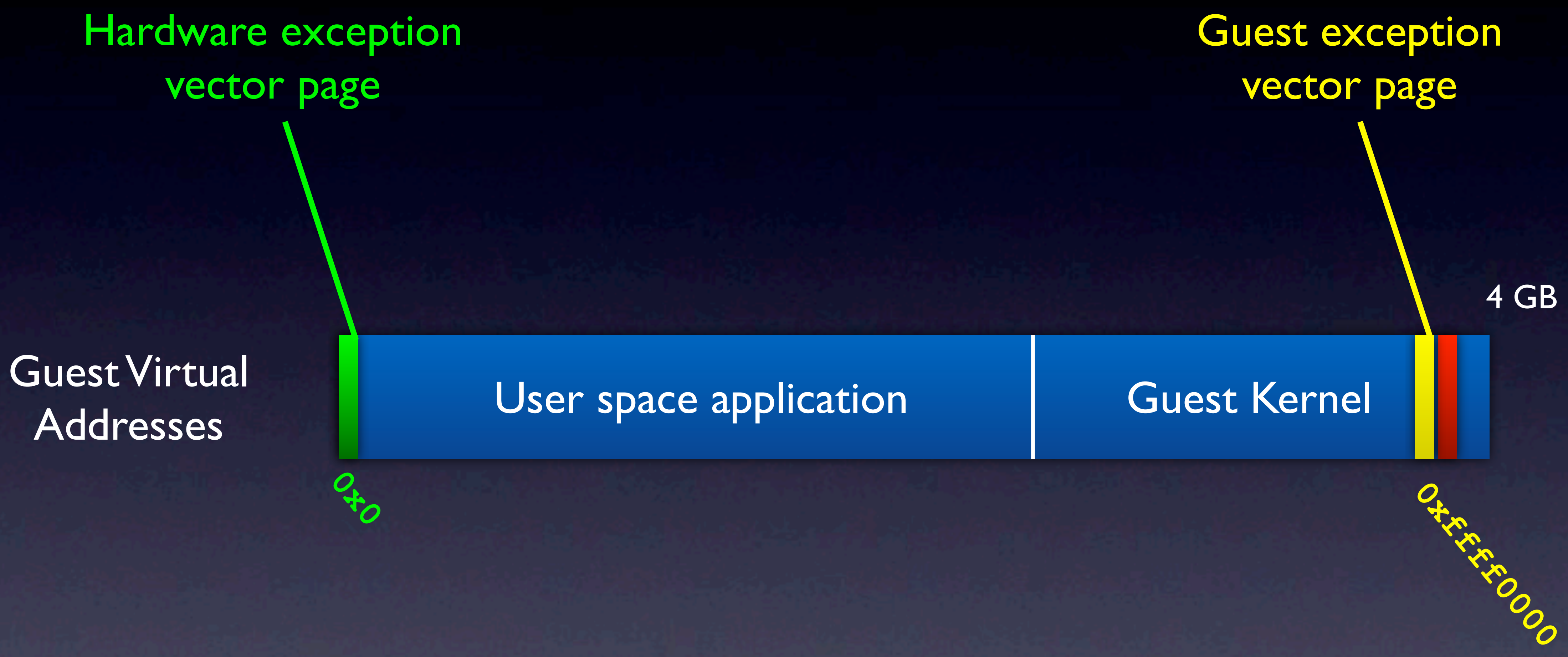
What happens at a conflict?

- KVM/ARM's vectors are mapped with no-access for user mode code at 0xffff0000
- The guest tries to access 0xffff0000 page
- KVM/ARM handles the permission fault

Exception page conflict

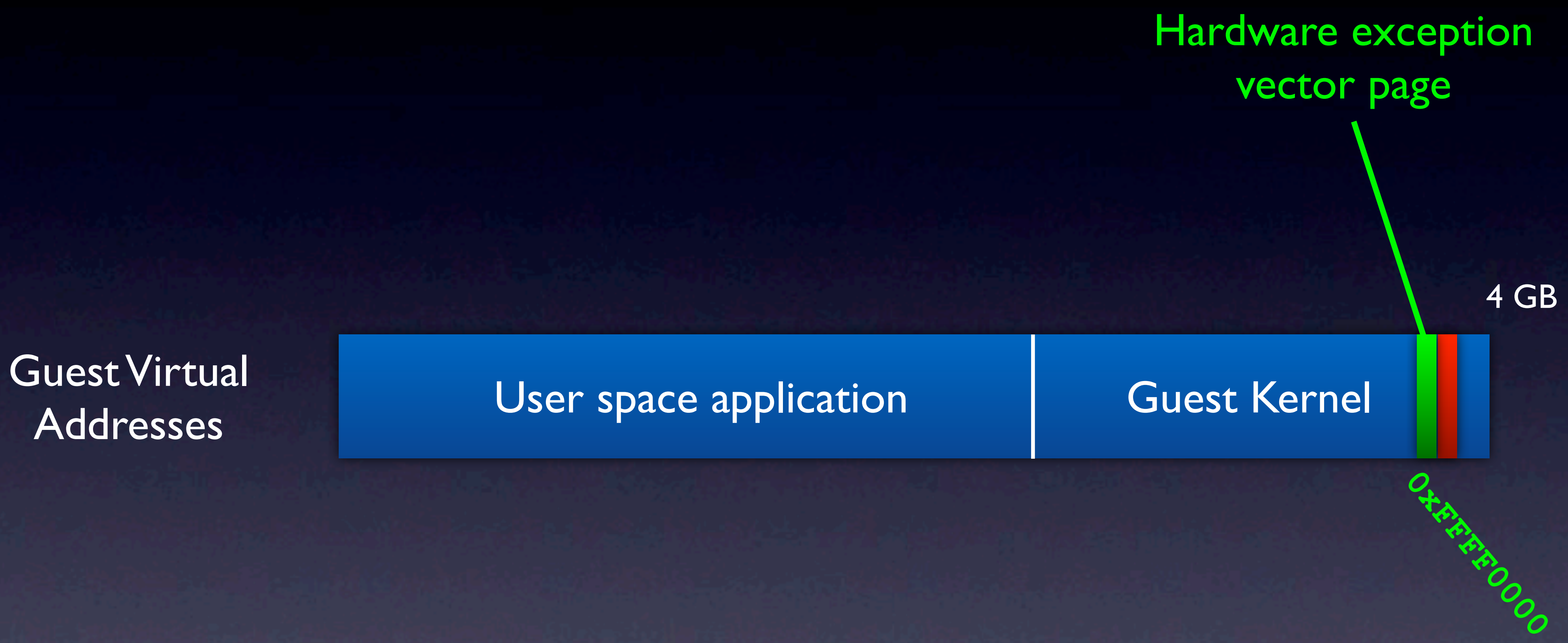


Exception page conflict



Guest uses “high” vectors

Exception page conflict



Guest uses “high” vectors, but needs access to page 0