Real-Time Virtualization
Partitioning Hypervisors

Real-Time Industrial Systems

Prof. Marcello Cinque
Prof. Luigi De Simone

Roadmap

- **What is Jailhouse?**
- **Jailhouse under the hood**
- **Jailhouse demos**

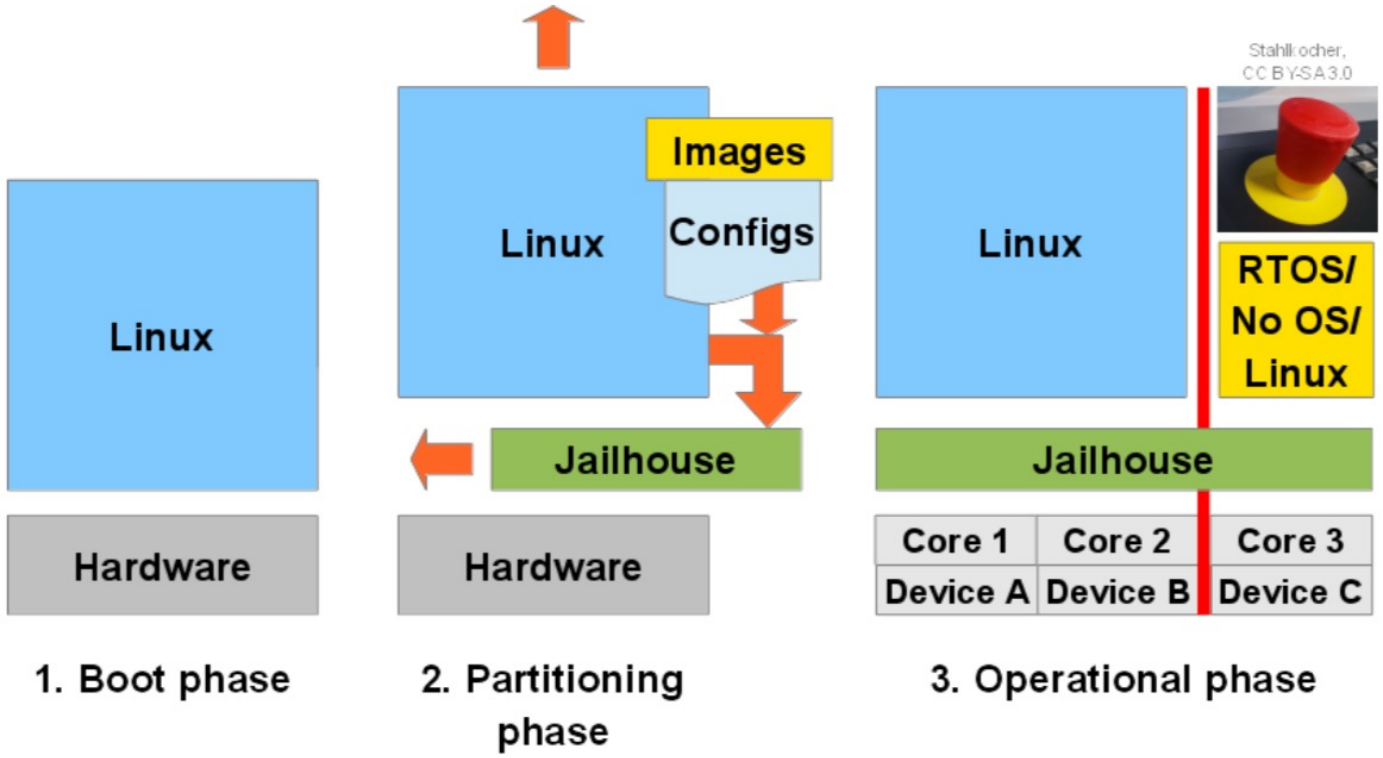
- **References**
  - **Jailhouse Documentation**. https://github.com/siemens/jailhouse/tree/master/Documentation
  - **Understanding the Jailhouse hypervisor, Part 1.** https://lwn.net/Articles/578295/
  - **Understanding the Jailhouse hypervisor, Part 2.** https://lwn.net/Articles/578852/
  - **Jailhouse demo documentation.** https://github.com/ldesi/virtualization_technologies_course/tree/master/5_real_time_virtualization

Jailhouse

- **Developed at Siemens in 2013 as partitioning hypervisor**
  - Currently at v0.12, Open source (GPLv2): **https://github.com/siemens/jailhouse**
- **Build static partitions on AMP/SMP systems**
  - ➢Flexible partitioning, runtime controlled
- **Use hardware-assisted virtualization**
  - ➢Supports x86/x86-64 (Intel, AMD) and ARM (32/64 bit)
- **Does not schedule VM on cores**
  - ➢Very thin hypervisor layer (low overhead)
  - ➢1:1 device assignment, memory mapped
- **Splits up running Linux systems**
  - ➢Starts native, ”migrates” to be virtualized after boot
- **Simplicity over features**
  - ➢<10k lines of code (kernel module, suitable for certification purposes)
  - ➢Assumes multicore, at least one guest per core

Jailhouse activation

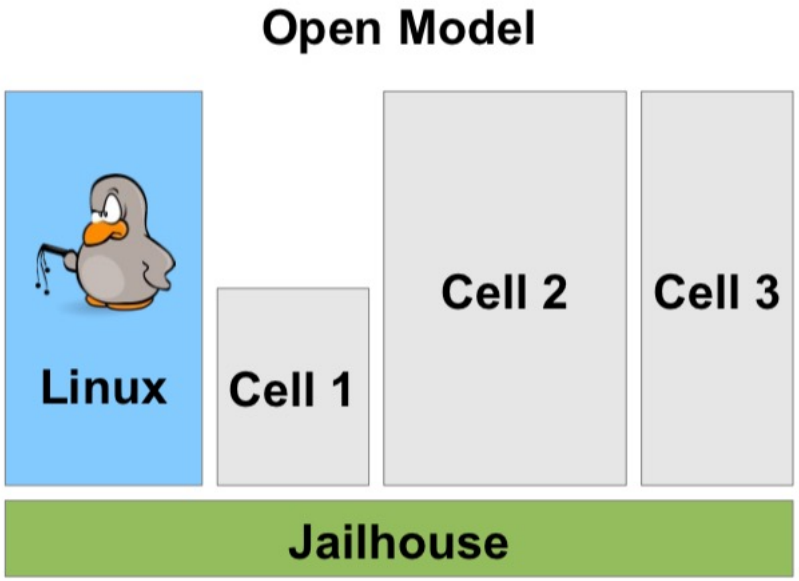
- **Jailhouse doesn't meld with the kernel** as KVM (which is a kernel module)
- It is **loaded as a firmware image** and resides in a **dedicated memory region**
- **`jailhouse.ko`** loads the firmware and creates **`/dev/jailhouse`** device
- **`/dev/jailhouse`** is used by the Jailhouse user-space tool for handling partitions lifecycle, but it doesn't contain any hypervisor logic



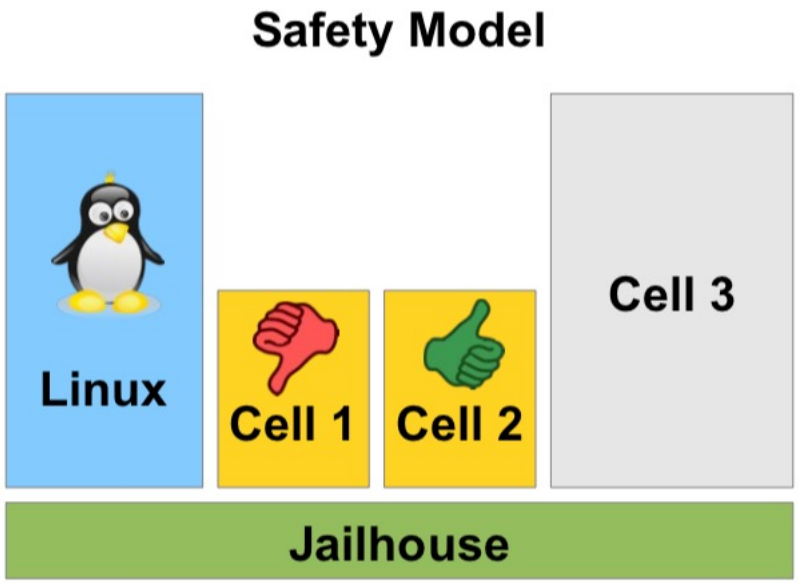
Jailhouse elements

- Jailhouse **doesn't emulate resources you don't have**, instead…
  - Splits physical hardware into **isolated compartments**
    - **Cells** that are fully dedicated to guest OS (could be also for bare-metal app.)
    - **Inmates** for software programs
  - **Two kind of cells**
    - **Root cell**: runs the ‘’host’’ Linux OS
    - **Non-root cells**: borrow CPUs and devices from the root cell as they are created

Jailhouse modes



- **Linux (root cell) is in control**
- Cells not involved in management decisions
- Sufficient if root cell is trusted



- **Linux controls, but...**
- Cells can be configured to vote over management decisions
- Building block for safe operation

Jailhouse memory reservation

- The hypervisor requires a **contiguous piece of RAM** for itself and each additional cell

- This currently has to be **pre-allocated during boot-up**
  - **x86 family**
    - Use **`memmap=`** as parameter to the command line of the virtual machine's kernel (e.g., **`memmap=82M$0x3a000000)`**
  - **ARM family**
    - Reducing the amount of memory seen by the kernel (through the **`mem=`** kernel boot parameter) or by **modifying the Device Tree** (i.e. the reserved-memory node)

Jailhouse cell configuration

- **Cell configs include**
  - Reservation of at least one CPU
  - Amount of RAM
  - PCI devices
  - IRQ lines
  - Cache
  - etc.

- Reserved CPUs will be ***turned off*** by the root cell that will invoke *Jailhouse operations* to create the cell

- **CPU virtualization extensions will be used to guarantee spatial isolation**
  - E.g., access violation among cells will trap to hypervisor that halts the execution if needed

Jailhouse cell configuration

- Each cell (either root or non-root) must be **statically configured before it launches**
- This configuration determines which **hardware resources can the cell access**
- Jailhouse uses *.c files where parameters have to be assigned as **fields of special C structures**, which are defined into **`hypervisor/include/jailhouse/cell-config.h`**
  - **Root-cell ->** we need to define the `struct jailhouse_system`
  - **Non-root-cell ->** we need to define the `struct jailhouse_cell_desc`

Jailhouse cell configuration

- **jailhouse_memory mem_regions[N]** specifies the number of memory areas to be used
- **hypervisor_memory** specifies the memory area where the hypervisor binary must be placed. This memory must be reserved when Linux starts and must be the same specified in memmap (x86) or mem (ARM) directives. This field does not make sense with non-root cells
- **.cpus** indicates how many and which CPUs are assigned to a cell. E.g.:
  0x4 = 0b0000 0011
  2 CPU, CPU Core 0 e Core 1

```
struct jailhouse_memory {
        __u64 phys_start;
        __u64 virt_start;
        __u64 size;
        __u64 flags;
} __attribute__((packed));
```

```
                                                der;

                                                regions[33];
                                                qchips[1];
                                                pci_devices[1];
        .header = {
                        ...
                        .hypervisor_memory = {
                                        .phys_start = 0x7c000000,
                                        .size = 0x4000000,
                        },
        .root_cell = {
                                        .name = "Banana-Pi",
                                        .cpu_set_size = sizeof(config.cpus),
                                        .num_memory_regions =
__ARRAY_SIZE(config.mem_regions),
                                        .num_irqchips =
__ARRAY_SIZE(config.irqchips),
                                        .num_pci_devices =
__ARRAY_SIZE(config.pci_devices),

                                        .vpci_irq_base = 108,
                        },
        },
        . . .
        .cpus = {
                        0x3,
        },
```

Jailhouse cell configuration

- **`.mem_regions`** indicates the memory areas the cell could have access and with which rights (e.g., `JAILHOUSE_MEM_READ`, `JAILHOUSE_MEM_WRITE`, flags)
- **`.irqchips`** allows specifying several irq chips that could be assigned to the cell. Further, **`pin_bitmap`** indicates the allowed irqs

```
.mem_regions = {
        . . .
        /* Timer */ {
                .phys_start = 0x01c20c00,
                .virt_start = 0x01c20c00,
                .size = 0x400,
                .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                         JAILHOUSE_MEM_IO | JAILHOUSE_MEM_IO_32,
        },
        /* UART0-3 */ {
                .phys_start = 0x01c28000,
                .virt_start = 0x01c28000,
                .size = 0x1000,
                .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                         JAILHOUSE_MEM_IO,
        },
        . . .
.irqchips = {
        /* GIC */ {
        .address = 0x01c81000,
        .pin_base = 32,
        .pin_bitmap = {
                0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
                },
        },
},
. . .
```

Jailhouse cell configuration

- **`.pci_devices`** indicates the PCI devices assignment. **`.bdf`** specifies the bus, Device, and Function address.
- **`.pci_caps`** indicates the list of capabilities
  - It is possible to add capabilities to a device. The index of the first entry for this device in the array `.pci_caps`

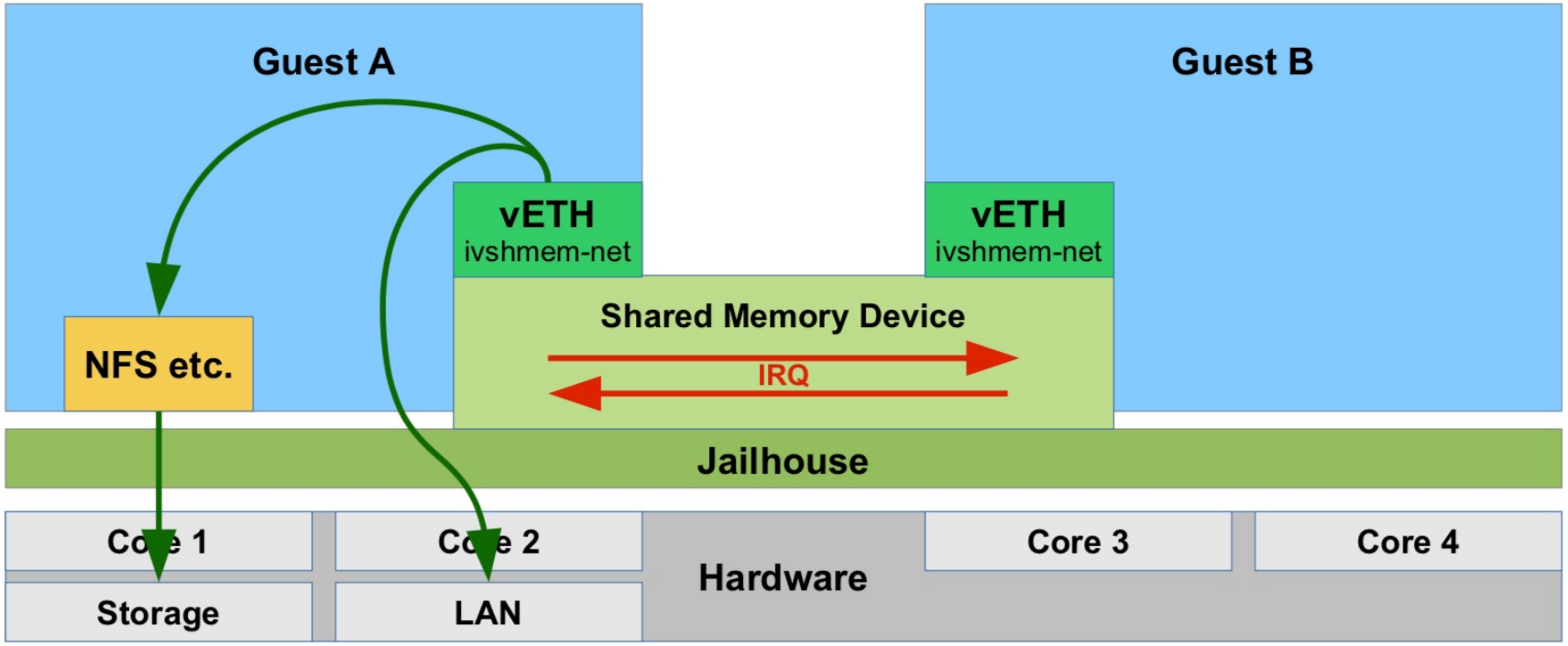
```
.pci_devices = {
            {
                        .type = JAILHOUSE_PCI_TYPE_DEVICE,
                        .domain = 0x0000,
                        /*Bus, Device, and Function address*/
                        .bdf = 0x00d8,
                        .caps_start = 0,
                        .num_caps = 2,
                        .num_msi_vectors = 1,
                        .msi_64bits = 1,
            },
}

/*list of capabilities */
.pci_caps = {
            {
                        .id = 0x5,
                        .start = 0x60,
                        .len = 14,
                        .flags = JAILHOUSE_PCICAPS_WRITE,
            },
}
```

Jailhouse intervention

- Jailhouse comes into play when
  - **Memory access violations**
  - **Re-injection of interrupt** (in some cases the interrupt could be directly routed towards guest)
  - **Intercept hardware resource that are not virtualized** (e.g., part of Generic Interrupt Controller (GIC) for ARM CPUs)
  - **Emulation of privileged instructions**

Jailhouse inter-cell communication

- Use **Inter-VM Shared Memory** (IVSHMEM) **device model**
  - Shared memory, memory-mapped areas for devices, and signaling between cells
  - An IVSHMEM device appears as a PCI device to its users



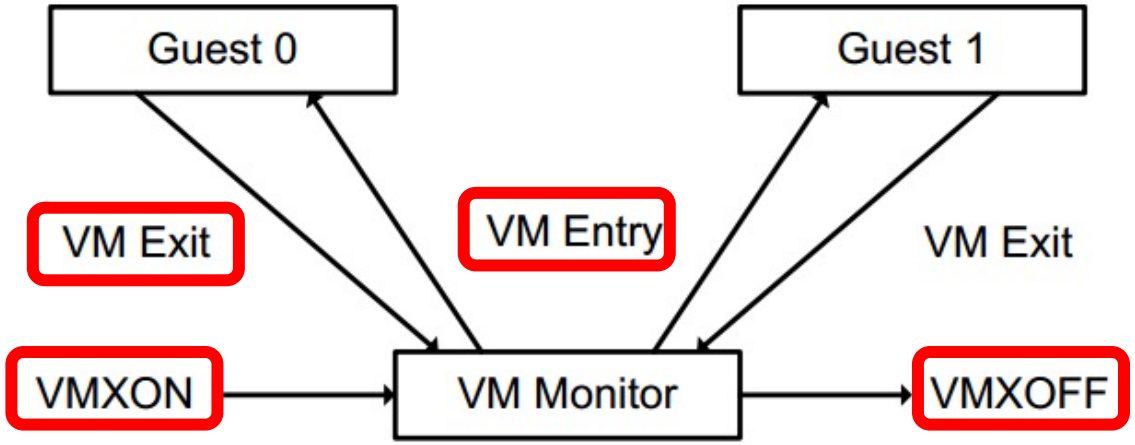
Jailhouse user-space commands

- `jailhouse enable SYSCONFIG`
- `jailhouse cell create CELLCONFIG`
- `jailhouse cell load ID IMAGE`
- `jailhouse cell start ID`
- `jailhouse cell shutdown ID`
- `jailhouse cell destroy ID`
- `jailhouse cell list`

Jailhouse enabling steps

- Jailhouse is enabled (root cell is started) by running the command
  - **`jailhouse enable <path/to/rootcell.cell>`**
- Sends the **`JAILHOUSE_ENABLE`** request to the **`/dev/jailhouse`** device, which signals driver to call **`jailhouse_cmd_enable()`** (driver/main.c)
  - Checks virtualization technology this CPU uses (Intel's VMX or AMD's SVM)
  - Basic validation of a configuration binary (signature in the header).
  - Calls **`request_firmware()`** function which searches for **`jailhouse-intel.bin`** or **`jailhouse-amd.bin`** in **`/lib/firmware`** folder
  - Remaps memory region reserved to the kernel address space memory (using **`ioremap_page_range(...)`**), so hypervisor could be accessed from the user-space
  - Copies the configuration binary at the start of this memory area and cell configuration right after it
  - Calls **`jailhouse_cell_create()`** function

Intel VMX brief operations recap.



- Software enters VMX operation by executing a **`VMXON`** instruction
- **VM Entry**: the VMM turns guests into VMs (one at a time). The VMM effects a VM entry using instructions **`VMLAUNCH`** and **`VMRESUME`**
- **VM Exit**: the guest transfers control to an entry point specified by the VMM (e.g., by a **`VMCALL`** instruction). The VMM can take action appropriate to the cause of the VM exit and can then return to the VM using a VM entry
- Eventually, the VMM may decide to shut itself down and leave VMX operation. It does so by executing the **`VMXOFF`** instruction

Jailhouse enabling (CPU Initialization)

- Hypervisor calls **`entry_hypervisor()`** function for every CPU
- Jailhouse needs to become an **interface between cells** (root cell with Linux at this early boot time**) and CPU cores**
- Jailhouse **saves system's state** and then **sets up** its **environment**
  - When the **CPU0** initializes
    - Sets up paging for the hypervisor and APIC
    - Creates the Interrupt Description Table (IDT) (aka *interrupt vector table*)
    - Creates root cell and remapping Linux memory regions and devices
    - Configures Virtual Machine Extensions (VME)
    - Sets up UART communication to write debug information
  - **For all CPUs**
    - Renew IDT and Global Descriptor Table (GDT)
    - Reset CR3 register (*Page Table* pointer)
    - Setup the **Virtual Machine Control Structure (VMCS)** (one for each CPU)
- Finally, Jailhouse sends a **VMLAUNCH** instruction returning the control to Linux **that runs within a "cell" (VM) under Jailhouse!!!**

Non-root cell creation

- Command to **start a non-root cell**
  - **`jailhouse cell create <path/to/conf.cell>`**

- Jailhouse reads configuration binary into memory and sends the **`JAILHOUSE_CELL_CREATE`** command to the driver with an address of the loaded binary which is attached

- Jailhouse invokes **`jailhouse_cmd_cell_create()`** (driver/control.c) which copies the mentioned binary from the user-space memory to a kernel space and performs some checks for loaded cell description (signature, size)

- Jailhouse makes an image for a guest according to the taken configuration
  - Fills up the special **`struct cell`** defined in `driver/cell.h` with pointers at mapped memory for guest's regions and PCI devices

Non-root cell creation

- The driver leaves an information about the new cell in sysfs and **plugs requested CPUs out of the Linux (root cell)**
- It also ***unplugs*** **PCI devices from Linux** at this time
  - Jailhouse emulates PCI dummy driver to cut it out of Linux as far as real unplug could not be performed
  - Linux will reprogram the BARs and locate resources where we do not expect them.
- The driver issues the **`JAILHOUSE_HC_CELL_CREATE`** hypercall, which results in calling the **`cell_create()`** in hypervisor/control.c
  - Gives a command for all new cell's processors to suspend except the current one (which executes this code right now)

Non-root cell initialization

- The **`cell_init()`** function fills the **`cpu_set`** field in cell struct with values of I/O ports' bitmaps and calls a routine to save (already reallocated to guest) locations and handlers of the memory-mapped devices such as PCI, IOAPIC, and IOMMU.
- After checking that all CPUs are not owned by somebody else, the operation moves to **`arch_cell_create()`** (hypervisor/arch/x86/control.c) where begins the **Linux *shrinking***
  - **1-to-1 assignment idea**: if the root cell has something that initializing cell wants, then access for Linux cell will be denied, and the new cell gets it.
- After resources (e.g., I/O ports, PCIs, etc.) were reassigned, a (optional) **per-cell shared memory area** is created between the hypervisor and the cell
  - This area contains also information about **PM timer address**, the number of the **CPUs assigned**, information about the current **cell state** (Running, Running/Locked, Shutdown, Failed, Failed Communication)

Inmate loading

- To execute some inmate in the new cell, it is needed to move it to the cell's memory region

  ```
  jailhouse cell load <name-of-cell> <inmate.bin> -a <offset-in-guest>
  ```

- All inmates are treated as **raw binaries**. The size of this binary must be less or equal to the guest memory region where it will be loaded
- Mechanism of transfer the file into cell's memory is similar to previous cases
- The driver sends **`JAILHOUSE_HC_CELL_SET_LOADABLE`** to the hypervisor, and it remaps guest regions marked as loadable to the root cell address space
  - Jailhouse will log the message "`Cell <name-of-cell> can be loaded.`"

Non-root cell starting

- Starts the new cell and running the inmate
  `jailhouse cell start <name-of-cell>`

- The hypercall **`JAILHOUSE_HC_CELL_START`** is invoked and it causes hypervisor's **`cell_start()`** that performs the *unmapping* of all loadable regions from the root cell back to the guest (the non-root cell)

- The cell's state becomes **`JAILHOUSE_CELL_RUNNING`** and on each CPU of the cell is invoked **`arch_cpu_reset()`**
  - This sends fake Startup Inter-Processor Interrupt (SIPI) to each CPU in the cell
  - At the next **`VMEntry`**, guest instruction pointer will be set at specific address, and the inmate starts

Jailhouse Demo

- Jailhouse on **BananaPI (M1 board, bananian-16.04)** with **FreeRTOS** and Linux
- Use Jailhouse **Open Model** (Linux is root cell)
- **FreeRTOS** is a real time operation system for embedded systems
  - It is **widely used on ARM** based microprocessor boards
- The demo aims at getting both systems **run together** on a multicore ARM processor system
  - It allows to **combine Linux GPOS with a hard RTOS**
  - Both systems are **almost isolated** from each other by the underlying hypervisor

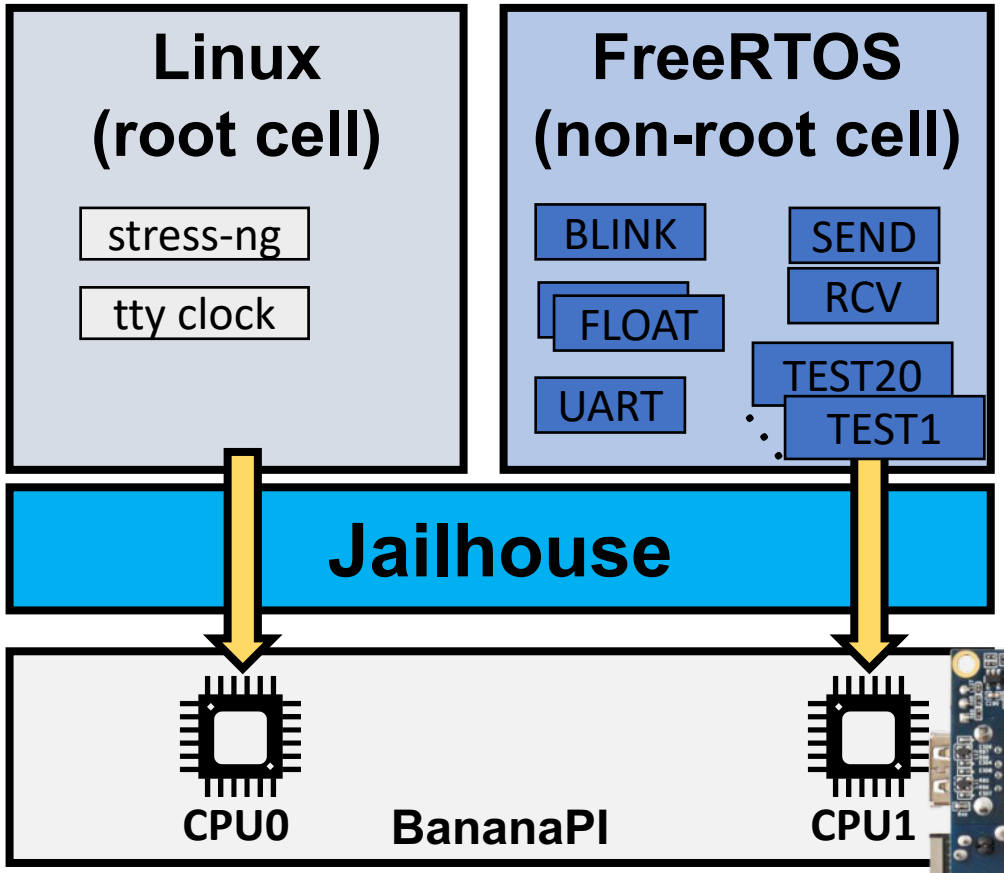
Jailhouse demo steps

1. **Install bananian on BananaPI board**
2. **Adjusting U-boot for kernel booting arguments**
3. **Cross Compiling Kernel for ARM on x86**
   - Apply patches and setup config
   - Compile kernel
4. **Installing the kernel**
5. **Cross Compiling Jailhouse(w/ FreeRTOS-cell) for ARM on x86 machine**
6. **Installing Jailhouse**
7. **Load Jailhouse and FreeRTOS cells**

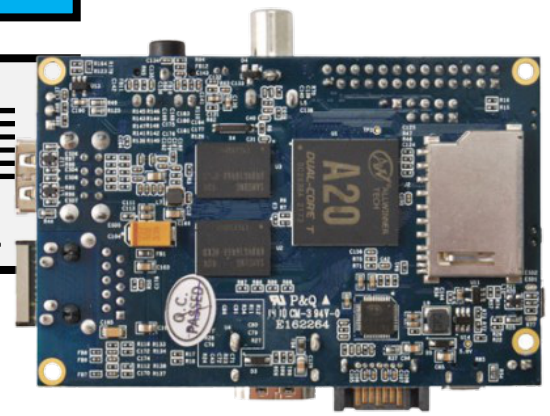
Root cell and FreeRTOS cell

- **RTOS: FreeRTOS**
  - **Scheduler**: Fixed Priority Preemptive Scheduling with Time Slicing (Round-robin)
  - **6 Tasks type**
    - **1 UART Task**: push logs to UART
    - **20 Test tasks**: print string to UART
    - **2 Floating point tasks**: compute floating point operations
    - **1 LED blink task:** call `pin_toggle` to write LED data registers
    - **1 Sender task**: task notification sender (`xTaskNotify`)
    - **1 Receiver task**: task notification receveir (`xTaskNotifyWait`)
- **GPOS: Linux**
  - **Scheduler:** SCHED_OTHER, standard Linux time-sharing scheduler (used when no real-time requirements needed)
  - **Tasks**
    - In general, whatever you want ☺
    - In this demo, we run a **digital clock** and **CPU stress test**

Demo testbed



- Allwinner A20 Dual-core 1.0GHz
- Mali-400
- 1 GB DDR3 RAM
- SATA interface
- Gigabit Ethernet



Root cell (Linux)

```
struct {
        struct jailhouse_system header;
        __u64 cpus[1];
        struct jailhouse_memory mem_regions[33];
        ...
} __attribute__((packed)) config = {
        .header = {
                ...
                .hypervisor_memory = {
                        .phys_start = 0x7c000000,
                        .size = 0x4000000,
                },
. . .
        .root_cell = {
                        .name = "Banana-Pi",
                        .cpu_set_size = sizeof(config.cpus),
                        .num_memory_regions = __ARRAY_SIZE(config.mem_regions),
                        .num_irqchips = __ARRAY_SIZE(config.irqchips),
                        .num_pci_devices = __ARRAY_SIZE(config.pci_devices),
                },
        .cpus = {
                0x3, //indicating CPU0 and CPU1 allocated to the root cell
        },
```

Root cell (Linux)

```
. . .
        .mem_regions = {
                /* SPI */ {
                        .phys_start = 0x01c05000,
                        .virt_start = 0x01c05000,
                        .size = 0x00001000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                 JAILHOUSE_MEM_IO,
                },
                /* PIO */ {
                        .phys_start = 0x01C20800,
                        .virt_start = 0x01C20800,
                        .size = 0xfc,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                 JAILHOUSE_MEM_IO,
                },
        }
. . .
```

Non-root cell (FreeRTOS)

```
struct {
        struct jailhouse_cell_desc cell;
        __u64 cpus[1];
        struct jailhouse_memory mem_regions[8];
        struct jailhouse_irqchip irqchips[1];
        struct jailhouse_pci_device pci_devices[1];
} __attribute__((packed)) config = {
        .cell = {
                . . .
                .cpu_set_size = sizeof(config.cpus),
                .num_memory_regions = __ARRAY_SIZE(config.mem_regions),
                .num_irqchips = __ARRAY_SIZE(config.irqchips),
                .num_pci_devices = __ARRAY_SIZE(config.pci_devices),
        },

        .cpus = {
                0x2, //CPU1 allocated to non-root cell
        },      }
```

Non-root cell (FreeRTOS)

```
.mem_regions = {
                . . .
                /* RAM */ {
                        . . .
                },
                /* UART 4-7 */ {
                        . . .
                },
                #define PIO_P7_REG (0x01c20800 + 7*0x24 + 0x0c)
                /* PIO port 7: blinking LED */ {
                        .phys_start = PIO_P7_REG,
                        .virt_start = PIO_P7_REG,
                        .size = 0x8,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
                                 JAILHOUSE_MEM_IO | JAILHOUSE_MEM_IO_32,
                },
        },
```

Run demo – Start Jailhouse and run cells w/ and w/ stress workload

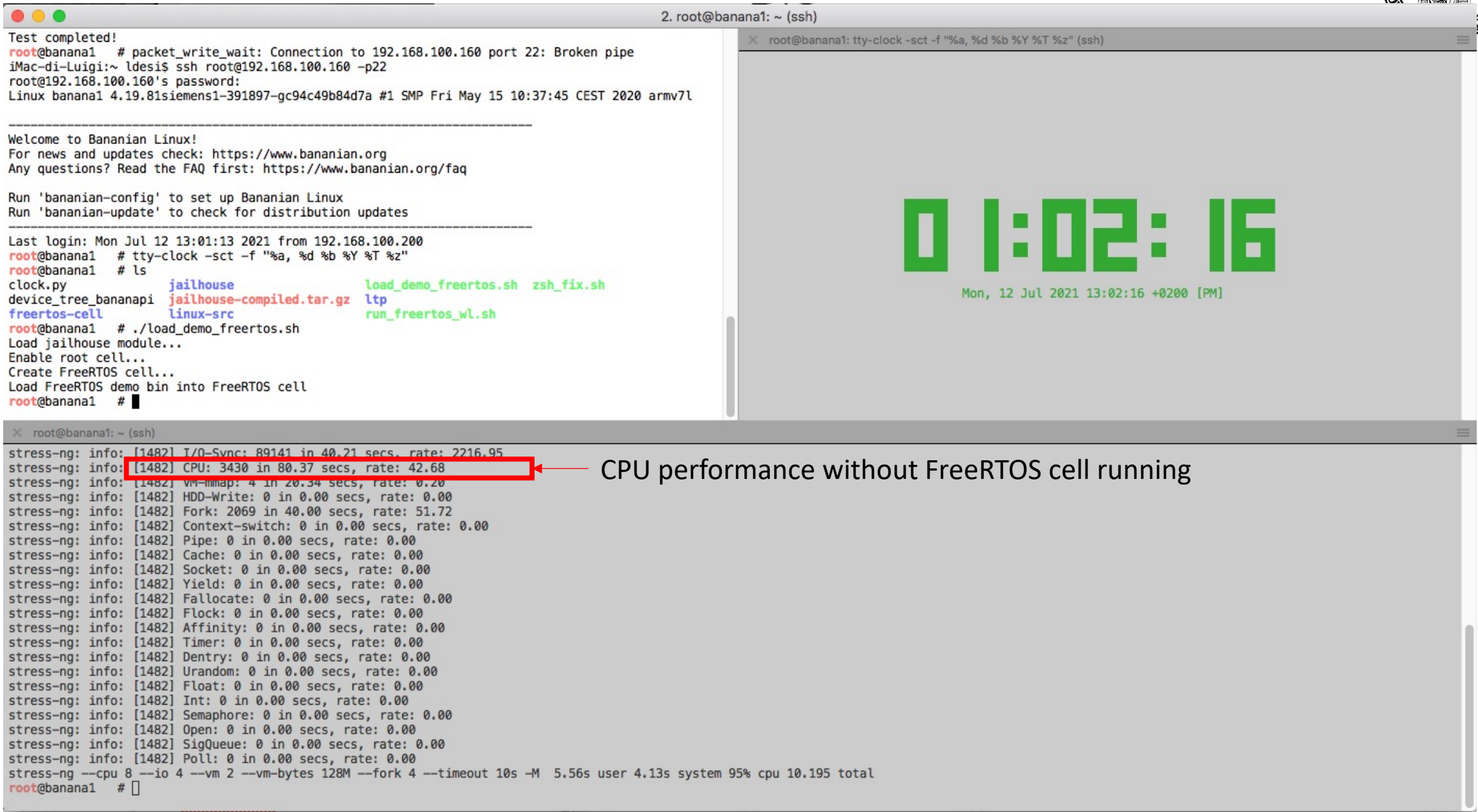
```
$ modprobe jailhouse

# If nothing goes wrong, you should see jailhouse using a `$ lsmod`
$ jailhouse enable ~/jailhouse/configs/arm/bananapi.cell
$ jailhouse cell create ~/jailhouse/configs/arm/bannapi-freertos-demo.cell
$ jailhouse cell load FreeRTOS ~/freertos-cell/freertos-demo.bin
$ jailhouse cell start FreeRTOS

# Run digital clock
$ tty-clock -sct -f "%a, %d %b %Y %T %z»

# Run stress test to assess temporal isolation between root and non-root cells
$ stress-ng --cpu 8 --io 4 --vm 2 --vm-bytes 128M --fork 4 --timeout 10s –M
```

Run demo – Turn off Jailhouse

#After making sure all things works well, turn off jailhouse using command below

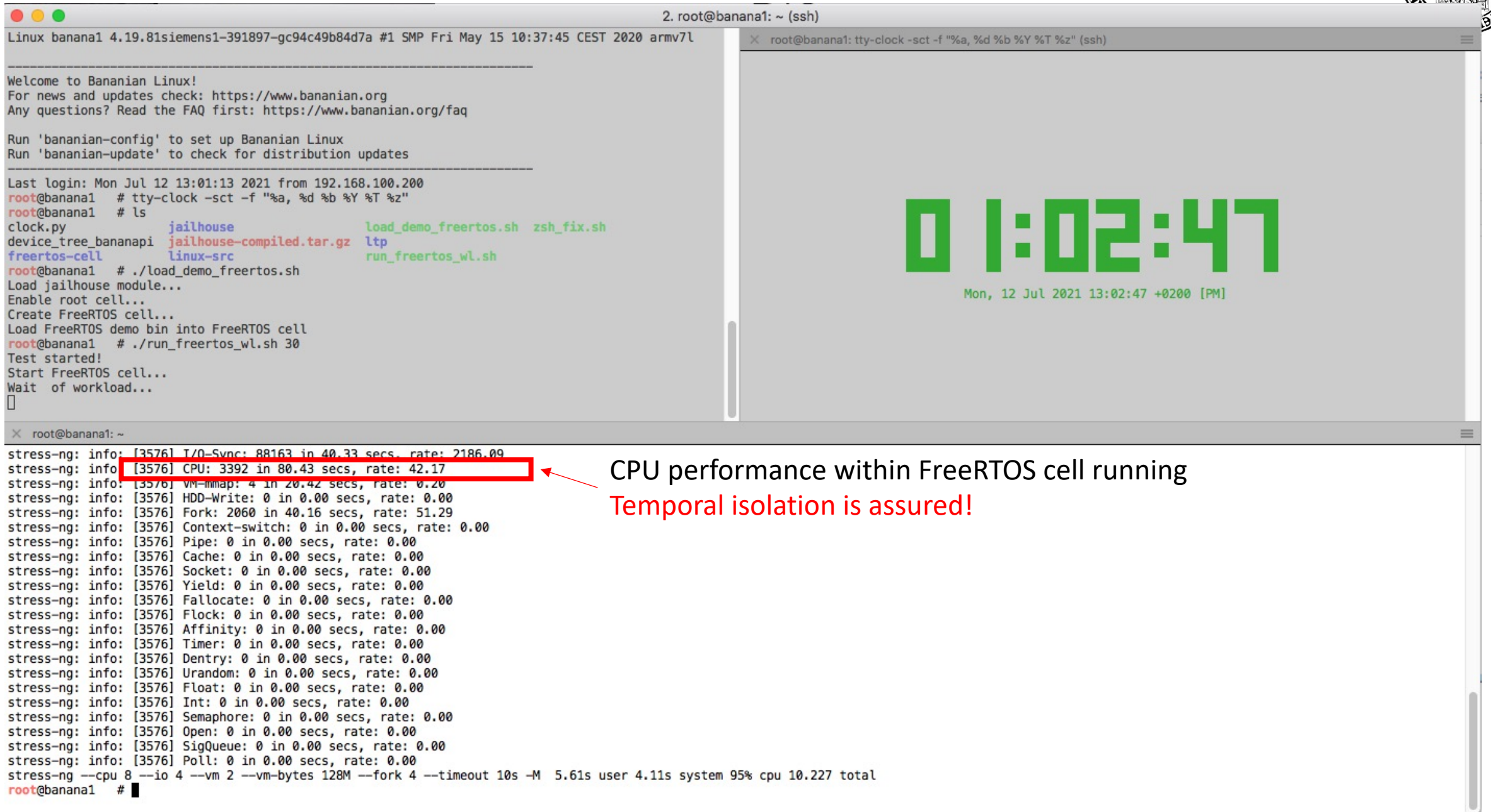
```
$ jailhouse cell shutdown FreeRTOS
$ jailhouse cell destroy FreeRTOS
$ jailhouse disable
$ rmmod jailhouse
```

Stress w/o FreeRTOS running



CPU performance without FreeRTOS cell running

Stress w/ FreeRTOS running



CPU performance within FreeRTOS cell running

Temporal isolation is assured!

x86/ARM on QEMU/KVM

- Use **jailhouse-images** repo: https://github.com/siemens/jailhouse-images
- The **host-side requirements**
  - Docker (tested with 19.03.5-ce)
  - QEMU >= 4.2
  - Kernel >= 4.4 with KVM support (for qemu-x86 image)
  - kvm_intel module loaded with parameter nested=1 (for qemu-x86 image on kernel < 4.20)
- To build a target image, just run **`build-images.sh`** and select one (or both) of the QEMU targets
- The generated image can then be executed using **`start-qemu.sh ARCHITECTURE`**
- **x86** (only works on Intel CPUs so far), **ARM64** and **ARM** are currently supported
  - On x86, make sure the kvm-intel module was loaded with nested=1 to enable nested VMX support

QEMU x86 Jailhouse Demo

Run Jailhouse-enabled image with 1Gb RAM, 4 vCPUs, serial redirected to file:

```
# /path/to/qemu-system-x86_64 \
        -drive file=/var/lib/libvirt/images/jailhouse_image.qcow2,format=qcow2,if=none,id=drive-ide0-0-0 \
        -device ide-hd,bus=ide.0,unit=0,drive=drive-ide0-0-0,id=ide0-0-0,bootindex=2 \
        -drive if=none,id=drive-ide0-0-1,readonly=on \
        -m 1G \
        -serial file:serial.txt \
        -netdev user,id=net \
        -cpu host,-kvm-asyncpf,-kvm-steal-time,-kvmclock \
        -smp 4 -enable-kvm -machine q35,kernel_irqchip=split \
        -serial vc \
        -device intel-iommu,intremap=on,x-buggy-eim=on \
        -device e1000e,addr=2.0,netdev=net \
        -device intel-hda,addr=1b.0 -device hda-duplex \
        -vga vmware
```

QEMU x86 Jailhouse Demo

- **apic-demo** will program the APIC timer interrupt to fire at 10 Hz, measuring the jitter against the PM timer and displaying the result on the console
- Run the following:

```
# jailhouse enable /path/to/qemu-x86.cell
# jailhouse cell create /path/to/apic-demo.cell
# jailhouse cell load apic-demo /path/to/apic-demo.bin
# jailhouse cell start apic-demo
```

QEMU x86 Jailhouse Demo

```
Initializing Jailhouse hypervisor v0.12 (289-gb6019359-dirty) on CPU 1
Code location: 0xffffffff0000050
Using x2APIC
Page pool usage after early setup: mem 47/974, remap 0/131072
Initializing processors:
 CPU 1... (APIC ID 1) OK
 CPU 0... (APIC ID 0) OK
 CPU 2... (APIC ID 2) OK
 CPU 3... (APIC ID 3) OK
Initializing unit: VT-d
DMAR unit @0xfed90000/0x1000
Reserving 24 interrupt(s) for device ff:00.0 at index 0
Initializing unit: IOAPIC
Initializing unit: Cache Allocation Technology
Initializing unit: PCI
Adding PCI device 00:01.0 to cell "QEMU-VM"
Adding PCI device 00:02.0 to cell "QEMU-VM"
Reserving 5 interrupt(s) for device 00:02.0 at index 24
Adding PCI device 00:1b.0 to cell "QEMU-VM"
Reserving 1 interrupt(s) for device 00:1b.0 at index 29
Adding PCI device 00:1f.0 to cell "QEMU-VM"
Adding PCI device 00:1f.2 to cell "QEMU-VM"
Reserving 1 interrupt(s) for device 00:1f.2 at index 30
Adding PCI device 00:1f.3 to cell "QEMU-VM"
Adding PCI device 00:1f.7 to cell "QEMU-VM"
Reserving 2 interrupt(s) for device 00:1f.7 at index 31
Adding virtual PCI device 00:0c.0 to cell "QEMU-VM"
Adding virtual PCI device 00:0d.0 to cell "QEMU-VM"
```

```
Page pool usage after late setup: mem 270/974, remap 65543/131072
Activating hypervisor
Created cell "apic-demo"
Page pool usage after cell creation: mem 285/974, remap 65543/131072
Cell "apic-demo" can be loaded
CPU 3 received SIPI, vector 100
Started cell "apic-demo"
Calibrated TSC frequency: 3591761.000 kHz
Calibrated APIC frequency: 3591761 kHz
Timer fired, jitter:   3657 ns, min:   3657 ns, max:   3657 ns
Timer fired, jitter:   4918 ns, min:   3657 ns, max:   4918 ns
Timer fired, jitter:   2947 ns, min:   2947 ns, max:   4918 ns
Timer fired, jitter:   2822 ns, min:   2822 ns, max:   4918 ns
Timer fired, jitter:   2840 ns, min:   2822 ns, max:   4918 ns
Timer fired, jitter:   3494 ns, min:   2822 ns, max:   4918 ns
Timer fired, jitter:   3175 ns, min:   2822 ns, max:   4918 ns
Timer fired, jitter:  60235 ns, min:   2822 ns, max:  60235 ns
Timer fired, jitter:   2939 ns, min:   2822 ns, max:  60235 ns
Timer fired, jitter:   2797 ns, min:   2797 ns, max:  60235 ns
Timer fired, jitter:   3013 ns, min:   2797 ns, max:  60235 ns
```