Full system simulation in gem5

What we will cover

- What is full system simulation?
- Basics of booting up a real system in gem5
- Creating disk images using Packer and QEMU
- Extending/modifying a gem5 disk image
- Using m5term to interact with a running system

What is Full System Simulation?

Full-system simulation is a type of simulation that emulates a complete computer system, including the CPU, memory, I/O devices, and system software like operating systems.

It allows for detailed analysis and debugging of hardware and software interactions.

Components Simulated:

- CPUs (multiple types and configurations)
- Memory hierarchy (caches, main memory)
- I/O devices (disk, network interfaces)
- Entire software stack (OS, drivers, applications)

Basics of Booting Up a Real System in gem5

Overview: gem5 can simulate the process of booting up a real system, providing insights into the behavior of the hardware and software during startup.

Steps Involved

1. **Setting Up the Simulation Environment**:
   - Choose the ISA (e.g., x86, ARM).
   - Configure the system components (CPU, memory, caches).
2. **Getting the correct resources such as kernel, bootloader, diskimages, etc.**
3. **Configuring the Boot Parameters**:
   - Set kernel command line parameters, if necessary.
4. **Running the Simulation**:
   - Start the simulation and monitor the boot process.

Let's run a full system simulation in gem5

The incomplete code already has a board built.

Let's run a full-system workload in gem5.

This workload is an Ubuntu 24.04 boot. It will throw three m5 exits at:

- Kernel Booted
- When `after_boot.sh` runs
- After run script runs

Choosing the Correct Full-System Script

Depending on your host machine's capabilities (whether KVM is available and which ISA you want to simulate),
choose the appropriate script from `materials/02-Using-gem5/07-full-system/`:

- **If your machine supports KVM and you want to simulate x86 ISA**:
  Choose -> `x86-fs-kvm-run.py`

- **If your machine supports KVM and you want to simulate ARM ISA**:
  Choose -> `arm-fs-kvm-run.py`

- **If your machine does not support KVM acceleration (or you are on Windows Home Edition)**:
  Choose -> `x86-fs-run.py` (x86 simulation without KVM, much slower)

How to check if KVM is supported

This should have already been checked back in the **gem5-tutorial chapter** when you created the container.
At that time, if your host supports KVM, you needed to add `--device /dev/kvm` to your `docker run` command.

You can run the following command on your Linux host:

```
grep -E -c '(vmx|svm)' /proc/cpuinfo
```

- If the output is greater than 0, your CPU supports virtualization (Intel VT-x = `vmx`, AMD-V = `svm`).
- You should also see `/dev/kvm` on your host if KVM is enabled in BIOS.

Obtain the workload and set exit event

Once you have selected the correct script, you also need to set the matching workload:

For `x86-fs-kvm-run.py` or `x86-fs-run.py` (x86 full system):

```
workload = obtain_resource("x86-ubuntu-24.04-boot-with-systemd", resource_version="1.0.0")
board.set_workload(workload)
```

For `arm-fs-kvm-run.py` (ARM full system):

```
workload = obtain_resource("arm-ubuntu-24.04-boot-with-systemd", resource_version="1.0.0")
board.set_workload(workload)
```

This ensures the correct kernel, bootloader, and disk image are loaded for the selected ISA.

Note on runtime with and without KVM:

- If you run **with KVM enabled**, booting `x86-ubuntu-24.04-boot-with-systemd` usually takes only a few seconds or minutes.
- If you run **without KVM**, booting the same workload may take **3+ hours** because every instruction of the OS boot is simulated in detail.

In this case, you can switch to the lighter workload:

```
workload = obtain_resource("x86-ubuntu-24.04-boot-no-systemd", resource_version="1.0.0")
board.set_workload(workload)
```

The **no-systemd** image removes the systemd service manager and other startup tasks.
It still boots into Ubuntu 24.04, but with a much simpler init process, making it **significantly faster** to simulate when hardware acceleration (KVM) is not available.

Runtime comparison

Workload:

- **x86-ubuntu-24.04-boot-with-systemd**
  - With KVM: ~1–2 minutes
  - Without KVM: 3+ hours
- **x86-ubuntu-24.04-boot-no-systemd**
  - With KVM: ~1 minute
  - Without KVM: ~5–10 minutes

Obtain the workload and set exit event (conti.)

Let's make the exit event handler and set it in our simulator's object.

```
def exit_event_handler():
    print("first exit event: Kernel booted")
    yield False
    print("second exit event: In after boot")
    yield False
    print("third exit event: After run script")
    yield True

simulator = Simulator(
    board=board,
    on_exit_event={
        ExitEvent.EXIT: exit_event_handler(),
    },
)
simulator.run()
```

Viewing the terminal/serial output with m5term

Before booting this workload, let's build the `m5term` application so we can connect to the running system.

```
cd /workspaces/2025/gem5/util/term
make
```

Now you have a binary `m5term`.

To make it easier to run from anywhere, add the path to your environment:

```
export PATH=$PATH:/workspaces/2025/gem5/util/term
```

You can also add this line to your ~/.bashrc so it will be loaded automatically every time you open a new shell:

```
echo 'export PATH=$PATH:/workspaces/2025/gem5/util/term' >> ~/.bashrc
source ~/.bashrc
```

Watch gem5's output

Now, let's run the workload and connect to the terminal of the disk image boot using `m5term`.

Run gem5 with:

Depending on your host and ISA support, use the appropriate command:

- **x86 with KVM**

```
gem5 x86-fs-kvm-run.py
```

- **x86 without KVM**

```
gem5 x86-fs-run.py
```

- **ARM with KVM**

```
gem5 arm-fs-kvm-run.py
```

Connect to the simulated terminal

In another terminal window inside the same container, run the following command to connect to the disk image boot's terminal:

```
m5term 3456
```

Here `3456` is the port number on which the terminal is running.
You will see this printed in the gem5 output.

If you run multiple gem5 instances, they will have sequential port numbers.
If you are running in a non-interactive environment, there will be no ports to connect to.

Expected output:

When the disk image successfully boots and you connect using `m5term`, you should see the Ubuntu login prompt over the serial console:

```
Welcome to Ubuntu 24.04 LTS!

(skipped)

Ubuntu 24.04 LTS gem5 ttyS0

gem5 login: gem5 (automatic login)

In after_boot.sh...
Interactive mode: false
Starting gem5 init... trying to read run script file via readf....
```

This output is what you will see inside the `m5term` terminal window, confirming that the guest operating system inside gem5 has finished booting and that you are connected to its console.

Creating your own disk images (Optional)

Note:
Before starting, make sure you have the **QEMU GUI backend** installed.
Packer relies on QEMU to boot and install the OS, and without a proper GUI backend the installation may fail.

Important:
This section is **optional**. You are **not required** to go through this process at least once — it is only provided for those who wish to try creating disk images manually.

Creating disk images using Packer and QEMU

To create a generic Ubuntu disk image that we can use in gem5, we will use:

- Packer: This will automate the disk image creation process.
- QEMU: We will use a QEMU plugin in Packer to actually create the disk image.
- Ubuntu autoinstall: We will use autoinstall to automate the Ubuntu install process.

gem5 resources already has code that can create a generic Ubuntu image using the aforementioned method.

- Path to code: `gem5-resources/src/x86-ubuntu`

Let's go through the important parts of the creation process.

Getting the ISO and the user-data file

As we are using Ubuntu autoinstall, we need a live server install ISO.

- This can be found online from the Ubuntu website: iso

We also need the user-data file that will tell Ubuntu autoinstall how to install Ubuntu.

- The user-data file on gem5-resources specifies all default options with a minimal server installation.

You can download the ISO from the Ubuntu old releases archive at:

```
wget https://old-releases.ubuntu.com/releases/22.04/ubuntu-22.04.2-live-server-amd64.iso
```

This archive contains older Ubuntu release images, including the 22.04.2 live server ISO required for the installation.

How to get our own user-data file

To get a user-data file from scratch, you need to install Ubuntu on a machine.

- Post-installation, we can retrieve the `autoinstall-user-data` from `/var/log/installer/autoinstall-user-data` after the system's first reboot.

You can install Ubuntu on your own VM and get the user-data file.

Using QEMU to get the user-data file

We can also use QEMU to install Ubuntu and get the aforementioned file.

- First, we need to create an empty disk image in QEMU with the command: `qemu-img create -f raw ubuntu-22.04.2.raw 10G`
- Then we use QEMU to boot the diskimage:

```
qemu-system-x86_64 -m 2G \
      -cdrom ubuntu-22.04.2-live-server-amd64.iso \
      -boot d -drive file=ubuntu-22.04.2.raw,format=raw \
      -enable-kvm -cpu host -smp 2 -net nic \
      -net user,hostfwd=tcp::2222-:22
```

After installing Ubuntu, we can use ssh to get the user-data file.

Special note for Windows + Docker users

If you are running inside a Windows Docker container without X11 forwarding (no GUI support), you cannot use the default graphical installer. Instead, use curses (text mode):

```
qemu-system-x86_64 -m 2G \
      -cdrom ubuntu-22.04.2-live-server-amd64.iso \
      -boot d -drive file=ubuntu-22.04.2.raw,format=raw \
      -enable-kvm -cpu host -smp 2 \
      -net nic -net user,hostfwd=tcp::2222-:22 \
      -display curses
```

1. When the GRUB menu appears, press `e` to edit the boot entry.
2. Find the line starting with `linux /casper/vmlinuz ...` and change it to:

```
linux /casper/vmlinuz console=ttyS0,115200n8 nomodeset ---
```

3. Press F10 or Ctrl+X to boot.

4. The installer will now run entirely in text mode, and you can proceed with the installation as usual.

Booting from the installed raw disk image

After the installation is complete, you can boot directly from the installed .raw disk image (no need to attach the ISO anymore):

```
qemu-system-x86_64 -m 2G \
  -drive file=ubuntu-22.04.2.raw,format=raw \
  -enable-kvm -cpu host -smp 2 \
  -net nic -net user,hostfwd=tcp::2222-:22
```

You can then log in using the username and password you set during installation.

Important parts of the Packer script

Let's go over the Packer file.

- **bootcommand**:

```
"e<wait>",
"<down><down><down>",
"<end><bs><bs><bs><bs><wait>",
"autoinstall  ds=nocloud-net\\;s=http://{{ .HTTPIP }}:{{ .HTTPPort }}/ ---<wait>",
"<f10><wait>"
```

  This boot command opens the GRUB menu to edit the boot command, then removes the `---` and adds the autoinstall command.

- **http_directory**: This directory points to the directory with the user-data file and an empty file named meta-data. These files are used to install Ubuntu.

Important parts of the Packer script (Conti.)

- **qemu_args**: We need to provide Packer with the QEMU arguments we will be using to boot the image.
  - For example, the QEMU command that the Packer script will use will be:

```
qemu-system-x86_64 -vnc 127.0.0.1:32 -m 8192M \
-device virtio-net,netdev=user.0 -cpu host \
-display none -boot c -smp 4 \
-drive file=<Path/to/image>,cache=writeback,discard=ignore,format=raw \
-machine type=pc,accel=kvm -netdev user,id=user.0,hostfwd=tcp::3873-:22
```

- **File provisioners**: These commands allow us to move files from the host machine to the QEMU image.
- **Shell provisioner**: This allows us to run bash scripts that can run the post installation commands.

Let's use the base Ubuntu image to create a disk image with the GAPBS benchmarks

Update the `materials/02-Using-gem5/07-full-system/x86-ubuntu-gapbs/x86-ubuntu.pkr.hcl` file.

The general structure of the Packer file would be the same but with a few key changes:

- We will now add an argument in the `source "qemu" "initialize"` block.
  - `disk_image = true` : This will let Packer know that we are using a base disk image and not an iso from which we will install Ubuntu.
- Remove the `http_directory = "http"` directory as we no longer need to use autoinstall.
- Change the `iso_checksum` and `iso_urls` to that of our base image.

  Let's get the base Ubuntu 24.04 image from gem5 resources and unzip it.

```
wget https://storage.googleapis.com/dist.gem5.org/dist/develop/images/x86/ubuntu-24-04/x86-ubuntu-24-04.gz
gzip -d x86-ubuntu-24-04.gz
```

`iso_checksum` is the `sha256sum` of the iso file that we are using. To get the `sha256sum` run the following in the linux terminal.

```
sha256sum ./x86-ubuntu-24-04.gz
```

- **Update the file and shell provisioners:** Let's remove the file provisioners as we dont need to transfer the files again.
- **Boot command:** As we are not installing Ubuntu, we can write the commands to login along with any other commands we need (e.g. setting up network or ssh). Let's update the boot command to login and enable network:

```
"<wait30>",
"gem5<enter><wait>",
"12345<enter><wait>",
"sudo mv /etc/netplan/50-cloud-init.yaml.bak /etc/netplan/50-cloud-init.yaml<enter><wait>",
"12345<enter><wait>",
"sudo netplan apply<enter><wait>",
"<wait>"
```

Changes to the post installation script

For this post installation script we need to get the dependencies and build the GAPBS benchmarks.

Add this to the `materials/02-Using-gem5/07-full-system/x86-ubuntu-gapbs/scripts/post-installation.sh` script

```
git clone https://github.com/sbeamer/gapbs
cd gapbs
make
```

Let's run the Packer script and use this disk image in gem5!

```
cd /workspaces/2025/materials/02-Using-gem5/07-full-system/x86-ubuntu-gapbs
./build.sh
```

Let's use our built disk image in gem5

Let's add the md5sum and the path to our `local JSON(materials/02-Using-gem5/07-full-system/completed/local-gapbs-resource.json)`.

Let's run the `gem5 GAPBS config(materials/02-Using-gem5/07-full-system/completed/x86-fs-gapbs-kvm-run.py)`.

```
GEM5_RESOURCE_JSON_APPEND=./completed/local-gapbs-resource.json gem5 x86-fs-gapbs-kvm-run.py
```

This script should run the bfs benchmark.

Let's see how we can access the terminal using m5term

- We are going to run the same `gem5 GAPBS config(materials/02-Using-gem5/07-full-system/x86-fs-gapbs-kvm-run.py)` but with a small change.

Let's change the last `yield True` to `yield False` so that the simulation doesn't exit and we can access the simulation.

```
def exit_event_handler():
    print("first exit event: Kernel booted")
    yield False
    print("second exit event: In after boot")
    yield False
    print("third exit event: After run script")
    yield False
```

Again, let's use m5term

Now let's connect to our simulation by using the `m5term` binary

```
m5term 3456
```