Virtual Machines

Pr. Olivier Gruber

Full-time Professor

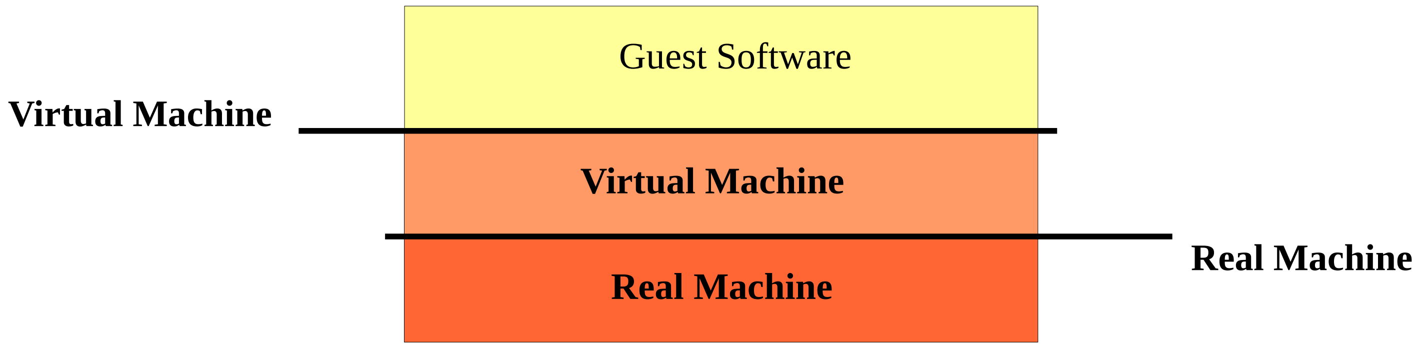
Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

Olivier.Gruber@imag.fr

Virtual Machine Basics

- ***Virtual Machines*** versus ***Real Machines***
  - *A virtual machine defines a machine (interface)*
  - *A virtual machine is a machine (implementation)*



Three Case Studies

- Palm-OS
  - Created in the late 90's
  - One of the first Personal Digital Assistant (PDA)
  - Evolved into smart phones
- Real-Time Operating System (RTOS)
  - Own the embedded system market
  - Absolutely not about performance, but about meeting deadlines
- UNIX-like OS

Case Study: PalmOS

- Case study: PalmOS
  - Created in the late 90's
  - One of the first Personal Digital Assistant (PDA)
- PalmOS specific features
  - Illustrative of the embedded software philosophy
  - Typically an event-oriented world
  - Essentially driven by GUI events
- Why study Palm-OS
  - Remind you than not everything is Linux
  - Remind you than not everything is a general-purpose desktop
  - Open your mind...

PalmOS – Overview

- Hardware target
  - Runs on a slow processor
  - Small amount of main memory (less than 1MB)
  - Persistent (battery-backed up), no disk or flash
- Essential use
  - Core applications
    - Email and calendar, no music player!
    - But many applications have been developed
  - Cradle
    - All data and applications synchronized with a PC
  - Must run 3-4 weeks on single AA batteries
    - Still much better than any of today's smart phones...
    - Even though late 90's hardware had little or no power management
    - And today's smart phones have the latest in power management

PalmOS – Rationales

- Could PalmOS follow a traditional OS design?
  - Footprint is way too big
    - Linux kernel does not fit in 1MB
    - A tad better with embedded linux kernels, years later
  - Processes and threads are too costly
    - Virtual memory and hardware protection are unnecessary
    - Applications are assumed to be bug free...
    - Applications can freely cooperate over shared data
  - File systems are inadequate
    - Can't have two copies of data and code (running image and files)
    - Can't pay the translation costs (from file to memory formats)
    - Does not provide fine-grain data replication

PalmOS – Design

- Central design points
  - Avoid dual storage of data (memory and file)
  - Avoid format translation (back and forth from files)
  - Keep it simple, help developers with replication
  - Permits application integration through shared database
- Introduced the concept of the PalmOS database
  - A database is memory-resident collection of C structs
  - A database provides search and scan operations
    - Manages in-memory C structs
  - Applications get pointers back to C structs
    - Direct manipulations of the C structs
  - Direct manipulations of the C structs in database
    - Database packing is available to fight fragmentation
    - Invalidates pointers to database structs

PalmOS – Design (cont...)

- Replication
  - Each database is replication-aware
    - Manages timestamp and dirty-bits
  - Device-level synchronization
    - PalmOS controls the two-way synchronization with the PC
- Conduits
  - Plugins for the replication engine
  - Translates PC data to Palm-suited data
- Example:
  - Adapting email to a small footprint device
    - Remove attachments or shorten long messages
  - Limit the number of messages per synchronization
    - Do not replicate all new emails
    - Could implement some LRU algorithm on emails to keep

PalmOS – Discussion

- Discussing the design
  - Applications are just event handlers
    - No threads and no process
    - Just a basic event loop for GUI and replication events
  - No saving of application state
    - There is no need, memory is persistent
    - Smaller application footprint (no save and load code)
    - Less overhead (no translation, no copy)
  - Instant-on property
    - No loading/saving of state
    - No loading of executable code
    - Just deliver *events* to the active application
  - Failures... no memory isolation...
    - Oh well... everything is synchronized, hence backed up
    - Still...

Case Study: RTOS

- Case study: Real-Time Operating Systems
  - RTOS dominate the embedded system market
  - Most RTOS today are proprietary, often expensive, with their own tool chains
- Why discuss the RTOS?
  - Remind you than not everything is Linux (again)
  - Remind you than not everything is a general-purpose desktop (again)
  - Illustrative of a **different** embedded software philosophy
  - A task-oriented world, with time deadlines

Important:

An RTOS is not about faster responses

It is about respecting deadlines

RTOS – Hard Real-Time

- Synchronous programming for safety critical systems
  - Software in planes, spacecrafts, or nuclear plants
  - Developed in specialized programming languages (Lustre, Esterel, etc.)
- Global Finite State Machine (FSM)
  - The FSM controls “reactions” written in C
    - Reactions are run to completion
    - Always under a Worst Completion Time (WCT)
  - Polled devices are input to the FSM
- Requires totally predictable hardware
  - Single core, in-order execution, no cache, no MMU, no IRQs
  - So that worst execution time can be computed statically

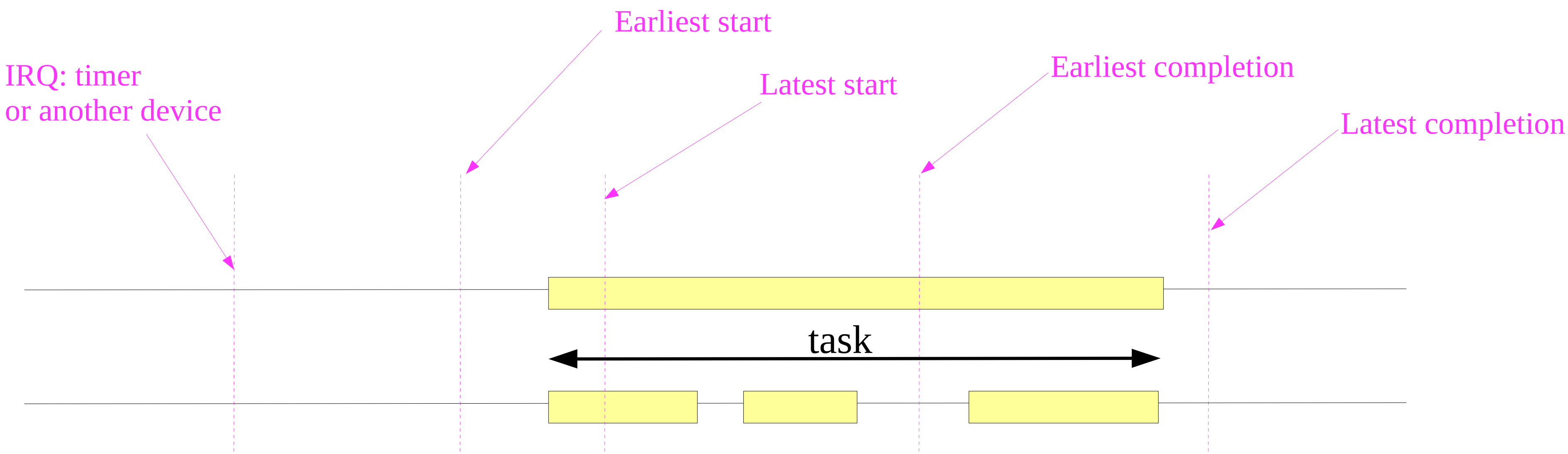
```
void scheduler(struct queue* rq) {
  ...
  for (;;) {
    poll_devices();
    fsm_step();
  }
}
```

A research challenge:

Predictable processors are a thing of the past, they are no longer produced
Modern processors have a lot of performance improvements
But the ability to statically compute WCT is drastically impacted

RTOS – Soft Real-Time

- Popular for embedded systems that are not for safety critical systems
  - Task-oriented design, with tasks running to completion or not
  - Leveraging event-oriented or thread-oriented programming

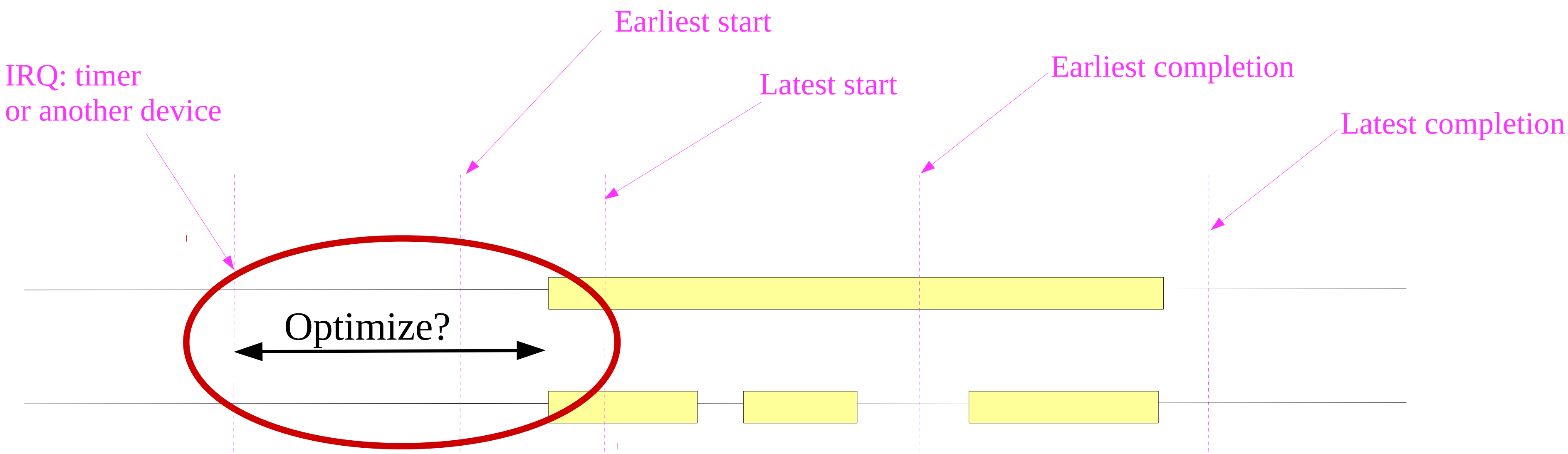


Tasks via events:
- **Could be a single event, running to completion**
- **Could be multiple events**

Tasks via threads:
- **Could be a thread, never blocking (I/O or synchronization)**
- **Could be a thread, allowed to block**

RTOS – Soft Real-Time

- Real-time is demanding on kernel design/implementation...



Reducing latency is difficult engineering

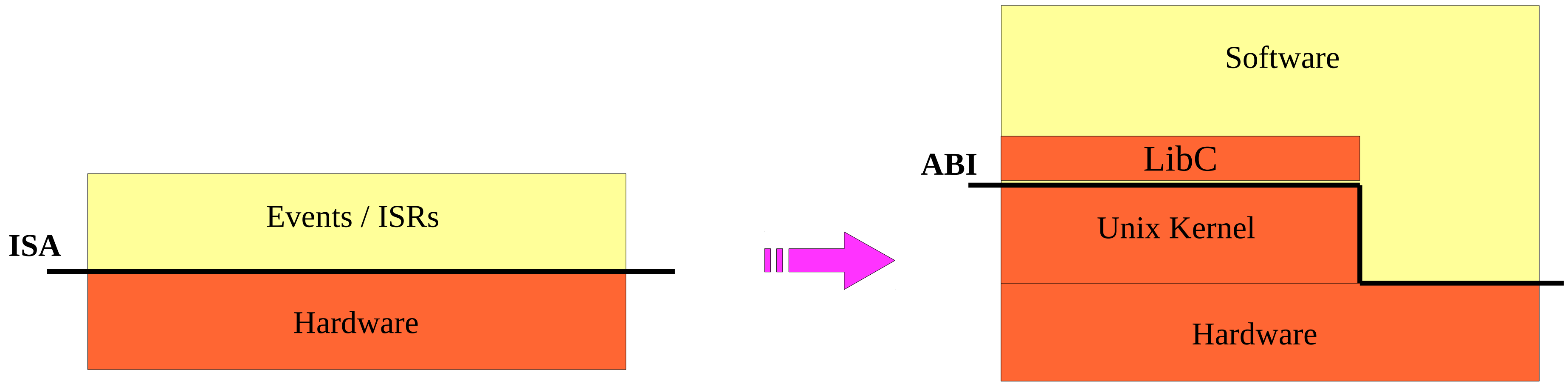
for both the kernel and the application developers

RTOS – Latency Scrutinized

- With events
  - IRQ handling has to wait for interrupts to be enable
    - Critical sections within your kernel will disable interrupts
  - The bottom will only execute after the current event finishes
    - With task priorities, you may have to wait until higher priority tasks complete
- With threads
  - IRQ handling has to wait for interrupts to be enable
    - Critical sections within your kernel will disable interrupts
  - Waiting until the current threads in kernel code can be preempted
    - This should be shorter than waiting for the current event to complete
  - Wait until lower priority threads release needed resources
    - This is a challenge for developers that must understand this requirement
    - Mastering synchronization becomes key to the overall correctness
  - Wait until higher priority tasks finish or block

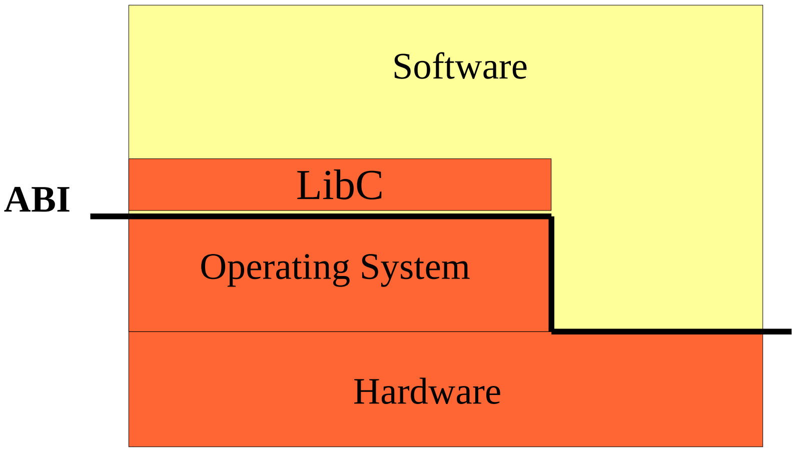
Case Study – Unix Family of Operating Systems

- A traditional view of an operating system
  - Goes all the way back to the 70s
  - Promoting a certain programming model (philosophy even)
- Why study the Unix family?
  - Look into the motivation for the key concepts
  - Discuss the evolution of those concepts



Unix Family of Operating Systems

- Early key concepts
  - A shared kernel and user-mode processes
  - Promoting C with the libC wrapping the ABI
    - Malloc and free for memory management
    - Streams (files and pipes)
    - Signals for traps and inter-process communication
  - Introduced device classes
    - All devices must fit into either a character or a block devices
    - With special files under /dev giving access to devices



UNIX – About devices

- Device classes
  - Character devices
    - Stream of "characters"
    - Essentially serial lines
    - Examples: keyboards and terminals
  - Block devices
    - Fixed-size blocks
    - Read/Write/Seek operations
    - Essentially tapes and disks
- Which programming model?
  - Hardware is asynchronous
  - Thus event-oriented programming would be adequate
  - But stream-based programming was pushed forward
  - Then event-oriented programming was also adopted

Characters are bytes in fact (uint8_t)
Remember back then, ASCII codes where 7bits
No other languages than english were supported
Java popularized the use of Unicode (1995)

Network cards did not exist
    Later forced into the block model
    Now recognized as a new device type
Video cards did not exist
    No concept in the kernel for 30 years
    Changed with DRI

UNIX – About streams

- Stream-based programming
  - Open/read/write/close
  - **Blocking operations**
  - One algorithm as one flow
- Streams for everything
  - Files as streams of bytes
  - Pipes as communication channels
  - Then sockets when networks appeared

```
void main(int argc, char* argv[]) {
  int fd;
  fd = open("toto");
  for (;;) {
    // read a record
    uint8_t bytes[10];
    int offset =0;
    int remaining =10;
    while (remaining) {
      int n = read(fd, bytes+offset,remaining);
      if (n==-1)
        return;
      offset += n;
      remaining -= n;
    }
    // process record
    ….
  }
}
```

UNIX – About events


```
struct read_event {
  struct event event;
  int fd;
  int offset, remaining;
  uint8_t *buffer;
#define STATE_INIT 0
#define STATE_READ 1
#define STATE_PROCESS 2
#define STATE_EOF 3
  uint8_t state;
};
void react(struct queue* rq, struct event* e) {
   struct read_event *evt = read_event_of(e);
   switch(evt→state) {
   case STATE_INIT:
     evt→fd = open("toto");
     evt→state = STATE_READ;
     event_push(rq,evt);
     break;
   case STATE_READ:
     read(evt))
      break;
   case STATE_PROCESS:
     process(rq,evt);
      break;
   }
}
```

Context Data

Finite State Machine

Non-blocking operation

```
void read(struct queue* rq,
                 struct read_event* e) {
   while (remaining) {
      int n = read(fd, e→buffer+offset,remaining);
      if (n==-1) {
         evt→state = STATE_EOF;
         return;
      }
      if (n==0) {
         event_push(rq,e);
         return;
      }
      offset += n;
      remaining -= n;
   }
   evt→state = STATE_PROCESS;
   event_push(rq,e);
}
```

UNIX – Blocking or non-blocking

- Stream-based vs Event-based programming
  - A religious debate that is still going on
    - Stream-based programming
      - Yielded thread-based programming when threads appeared for multi-core
    - Event-oriented programming
      - GUI remained event-oriented, single threaded
  - The debate is hotter than ever on the server side
    - High performance servers started advocating multi-threading
    - Recently, servers are going towards event-oriented programming (e.g. node.js)
  - Embedded systems
    - A mix of both, some advocate threads, others defend events...
  - Android
    - A clear mix & a clear separation of concerns
    - Event-oriented for the GUI – All events run to completion
    - Threads for workers – Using blocking calls is allowed

UNIX – About /dev

- Devices as special files under /dev
  - As special files (mknod, major, minor)
    - Open/Close/Read/Write and optional Seek
  - Major-minor identifies a kernel module
    - Usually a device driver, but it could be any module

→ Require file system support in the Kernel

```
$ ls -al /dev
C rw-r-----    1 root kmem    1,1    mem
C rw-rw-rw-    1 root root    1,3    null
C rw-rw-rw-    1 root root    1,5    zero

B rw-rw----    1 root disk    8,0    sda
B rw-rw----    1 root disk    8,1    sda1

C rw--w----    1 root tty     4,0    tty0
C rw-rw----    1 root tty     4,1    tty1

C rw-------    1 root root    254,0  rtc0

C rw-------    1 root root    10,239 uhid
C rw-------    1 root root    10,223 uinput
```

Also go have a look at...

/sys

/proc

Internal kernel data structures

reified through files

UNIX – About /dev

- Special files – the limitations of a file model
  - The need for ioctl beyond read/write/seek
    - The man page says
      - “The ioctl() function manipulates the underlying device parameters of special files”
    - But in reality
      - It is a generic “function call” to the underlying device driver
      - Usually passing a buffer, whose content is descibed by the request parameter

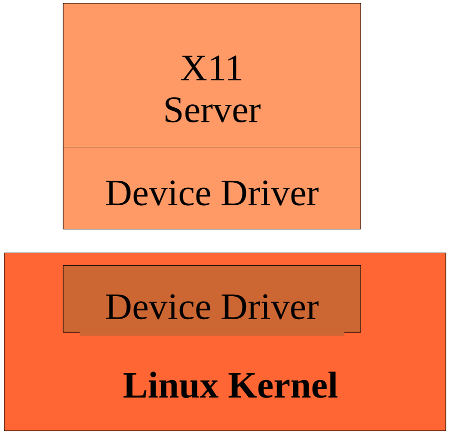
```
#include <sys/ioctl.h>
int ioctl(int desc, unsigned long request, …);
```

  - The need to memory map special files
    - Best example is the frame buffer of the video card

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int  munmap(void *addr, size_t length);
```

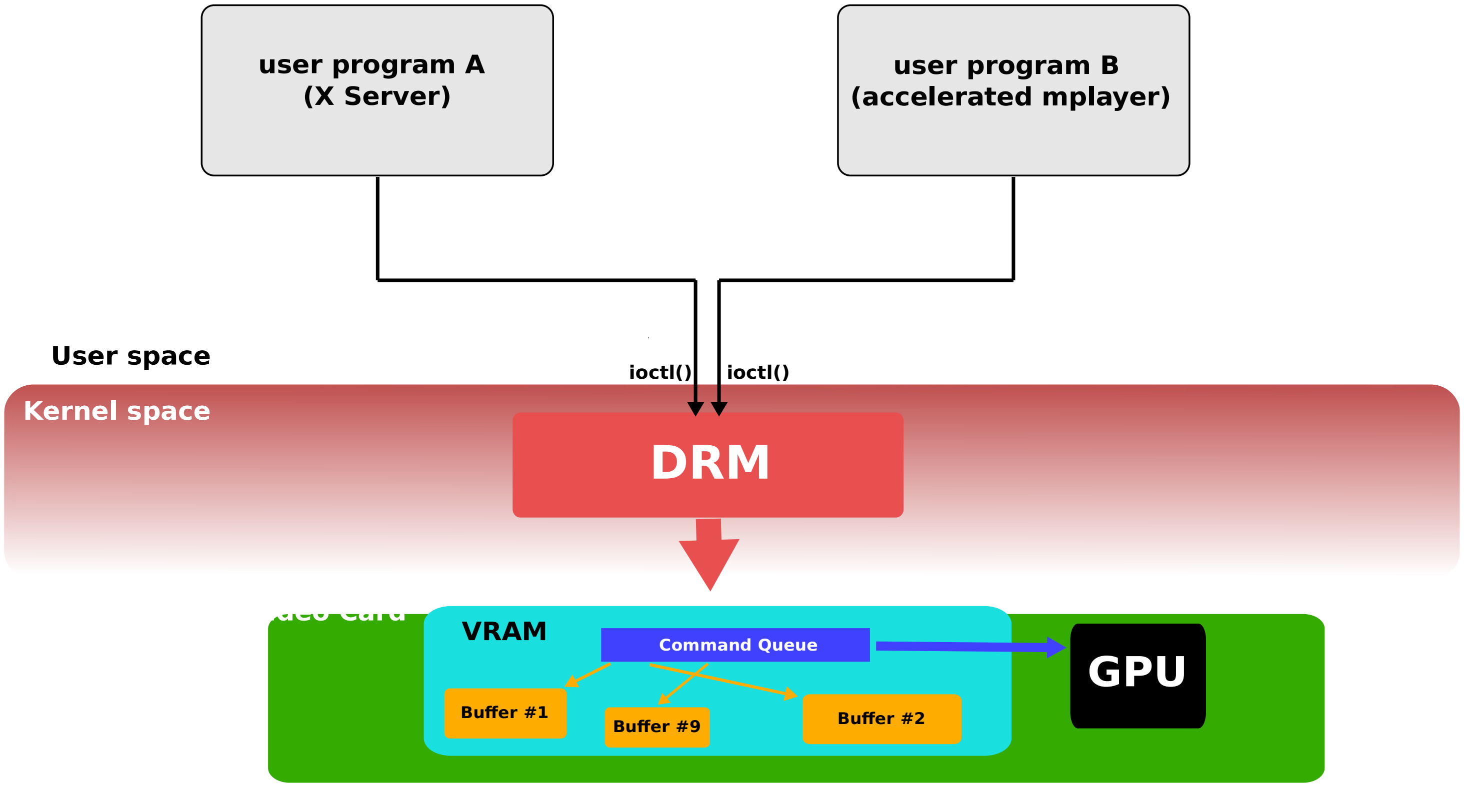
UNIX – About /dev

- An example – X11 Window Manager
  - Uses /dev to access devices
  - Because until very recently, the kernel had no concepts of a GUI
- Pointer device
  - Reads /dev/mouse to grab mouse events
  - Use ioctl to configure the mouse, if the mouse supports it
- Video device
  - Reads/writes to /dev/fb0 to display pixels onto the screen
  - Mmap the video buffer
  - Use ioctl to configure the video card
  - May use ioctl to access some accelerator functions
  - May use write to request accelerator functions and read for getting at the result
  - May also use the mmap memory with a circular buffer to send commands



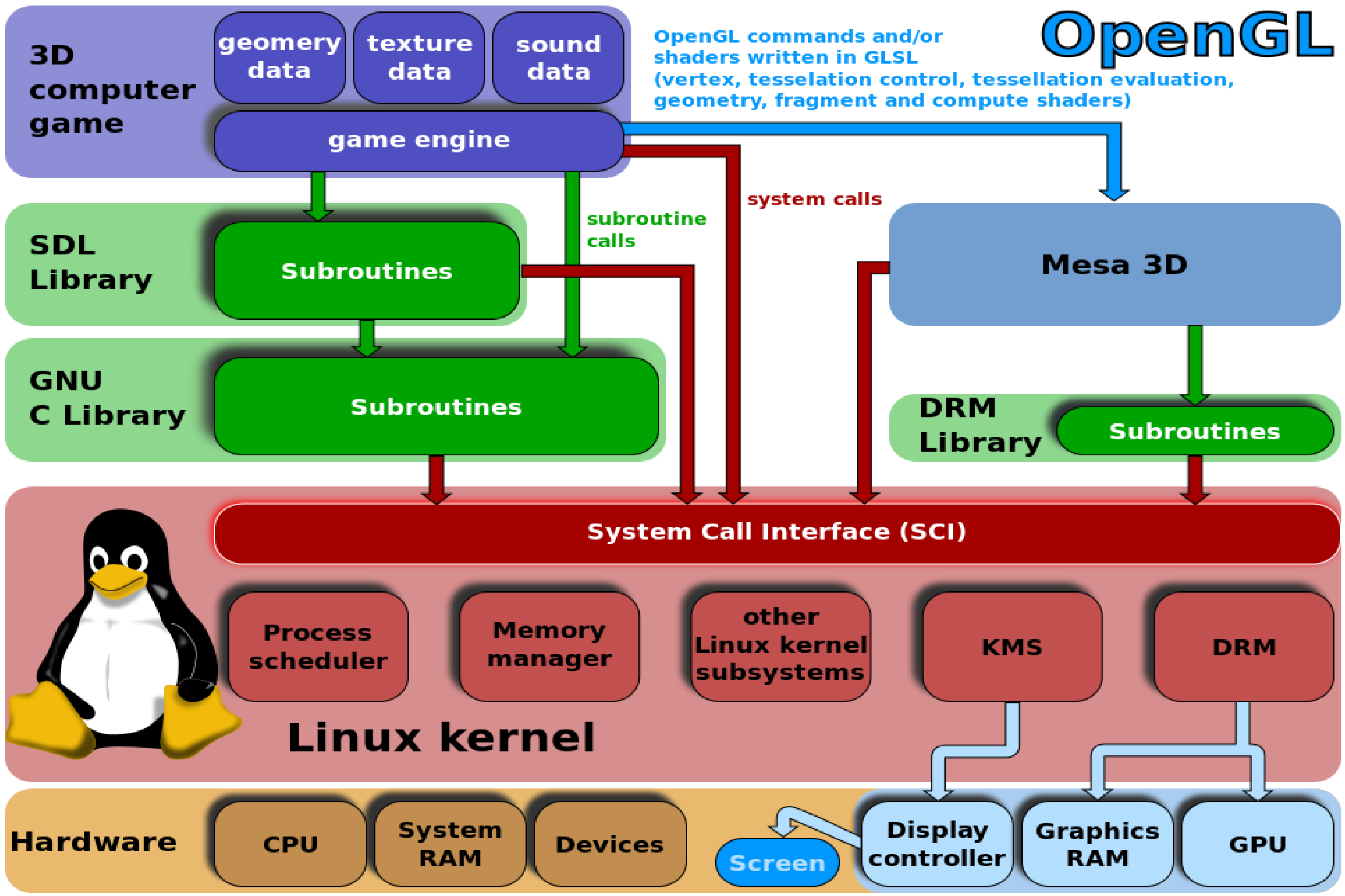
UNIX – Graphic Support

- Direct Rendering Infrastructure (DRI)
  - Direct Rendering Manager



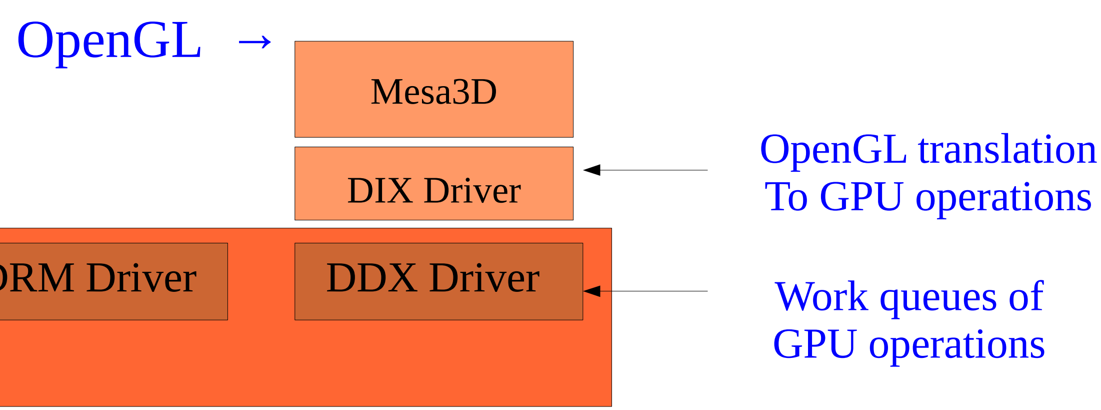
UNIX – Graphic Support

- Direct Rendering Infrastructure (DRI)



UNIX – Graphic Support

- Mesa3D
  - Open-source OpenGL
  - Simplified overview...



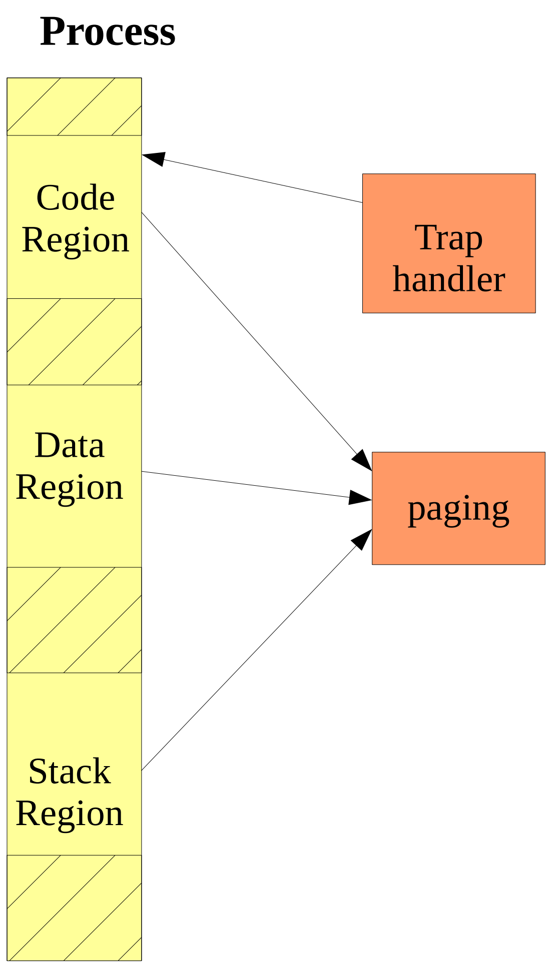
UNIX – About processes

- Basic idea
  - A process provides a virtual machine
  - So that a real machine can be shared by multiple applications
- Process rationales
  - Memory isolation (safety, separation of concerns)
  - Security (access rights)
  - Paging (virtual memory larger than physical)
  - Management concept (kill, resource accounting)

UNIX – About processes

- Processes as address spaces
  - A virtualized memory, isolated by hardware
  - Trap handler generates signals
- Mapped memory regions
  - Paging memory regions

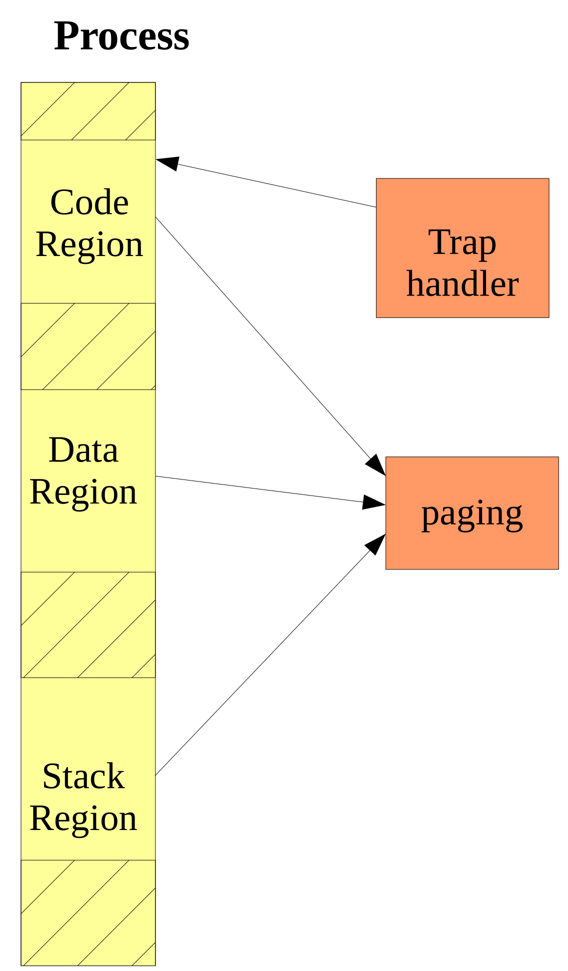
⇒ Paging requires file system support in the Kernel



UNIX – About processes

- Process management
  - Create by forking (duplicate process)
  - Kill by pid
  - Exec a file → requires loaders
- Loaders in the kernel
  - Such as Elf loader or shell script loader
  - Elf loader
    - Reads in code/data section in code/data regions
  - Scrip loader
    - Loads the interpreter in (like /bin/bash)
    - Pass the script file as an argument

→ Loaders require file system support in the Kernel



UNIX – About threads

- Before threads
  - Every “task” was a process
  - Processes are heavy weight to create/destroy
  - Context switch between processes is expensive
  - Blocking calls meant a process was carrying a single task
- Original motivation
  - Light-weight multi-tasking within a process
- Example:
  - A server with multiple clients
  - One socket per client
  - One thread per client

Preserves the programmin model with blocking operations on streams

UNIX – About threads (cont...)

- Multiple threads introduced the need for synchronization
  - Introduced mutexes and semaphores
  - Concurrent programming proved to be hard for most developers
  - Concurrency bugs are hard to find
  - Testing is much harder
- Multi-core evolution
  - Single core performance reached a peak early 2000
  - Multi-threading became also a performance quest
  - But beware, using threads is no guarantee of higher performance

So, do not confuse the two usages for threads

- light-weight multi tasking, preserving the blocking programming model

- searching for high performance computing on multi-core machines

UNIX – About threads (cont...)

- Multi-threading on multi-core
  - Synchronization has a cost in itself
    - Worst when using a syscall for each synchronization operation
    - User-level synchronization is faster, but strains the hardware (CAS-like typically)
  - Synchronization may induce convoy effects, drastically reducing parallelism
    - This happens on shared data structure
  - Context switch and cache efficiency
    - When context-switching between threads, the working set likely changes
  - Load balancing
    - Trade off between CPU usage and L1/L2 cache effectiveness

UNIX – About events

- Events are also supported
  - Almost all the ABI can be configured to be non-blocking
  - This allows to look at a thread as an event executor
  - It is possible to halt threads, waiting for events
  - With epoll for example, allows to wait on events on several sockets
- Graphical toolkits
  - Always promoted an event-oriented programming model
  - Some toolkits are single threaded, others permit the use of multiple threads

So, you have three programming models really, that can be combined

- light-weight multi tasking, preserving the blocking programming model

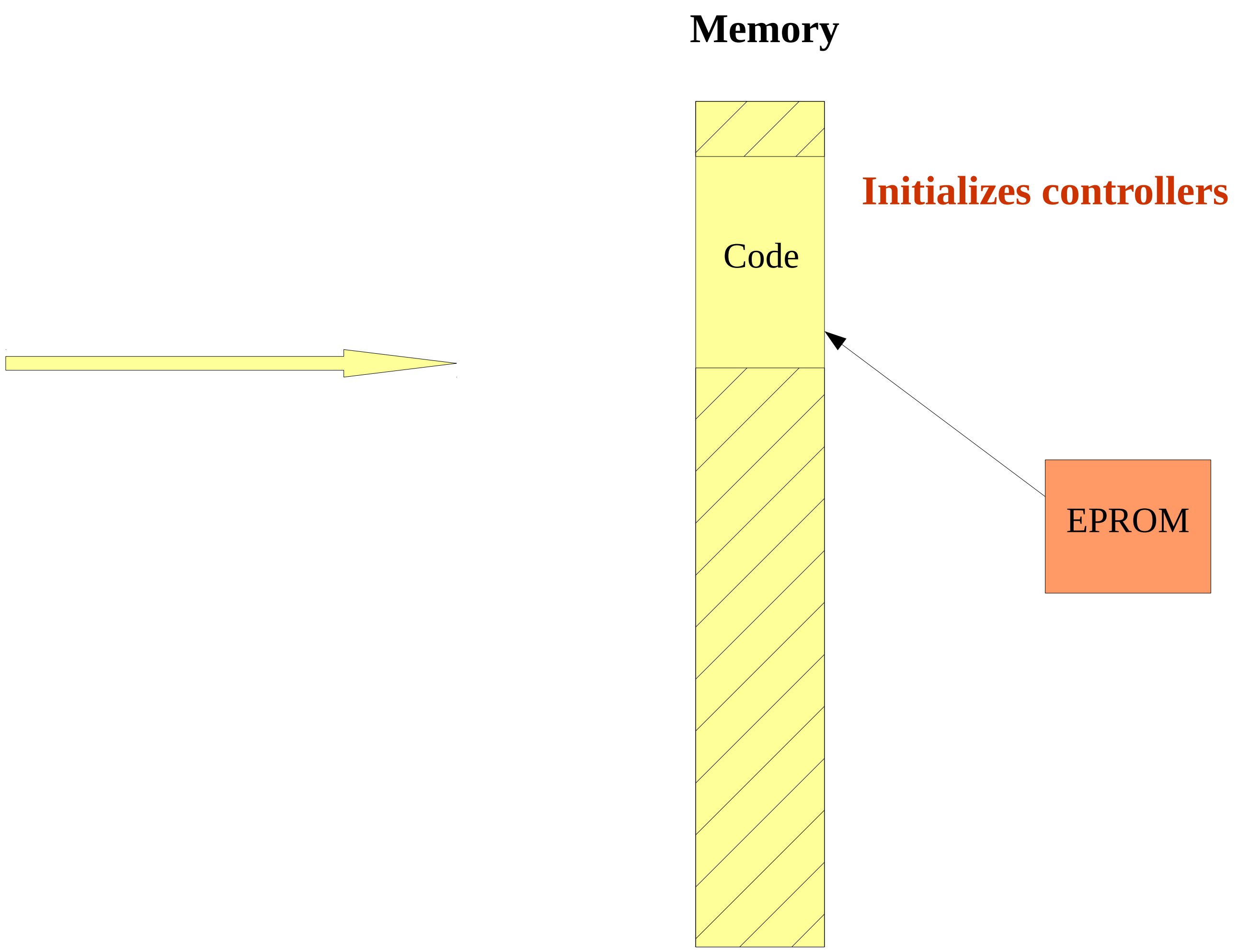
- event-oriented programming using one or more threads

- using threads for high performance computing on multi-core machines

Boot Process

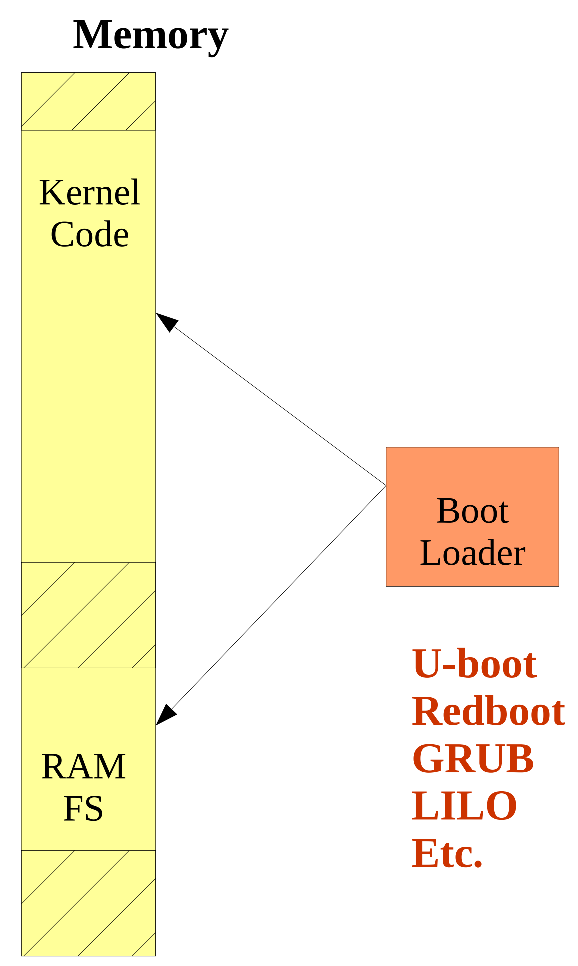
- Boot process
  - From the hardware wakeup
  - Up to your application
  - How does it happen?

- Major steps
  - Hardware initialization
  - Boot loader
  - Kernel initialization
  - Distribution initialization



Boot Process

- Major steps
  - Hardware initialization
  - Boot loader
  - Kernel initialization
  - Distribution initialization



Nota Bene:

The boot loader needs access to many devices

- disks

- cd-rom or dvd

- USB

So it needs device drivers

Boot Process

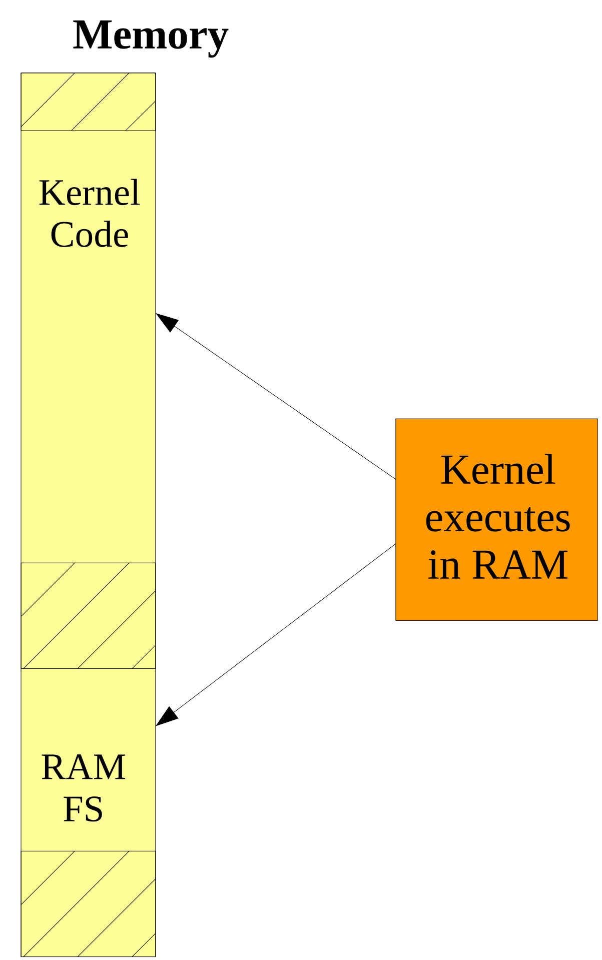
- Major steps
  - Hardware initialization
  - Boot loader
  - Kernel initialization
  - Distribution initialization

Kernel self-initialize

Mounts the RAM file system as root "/"

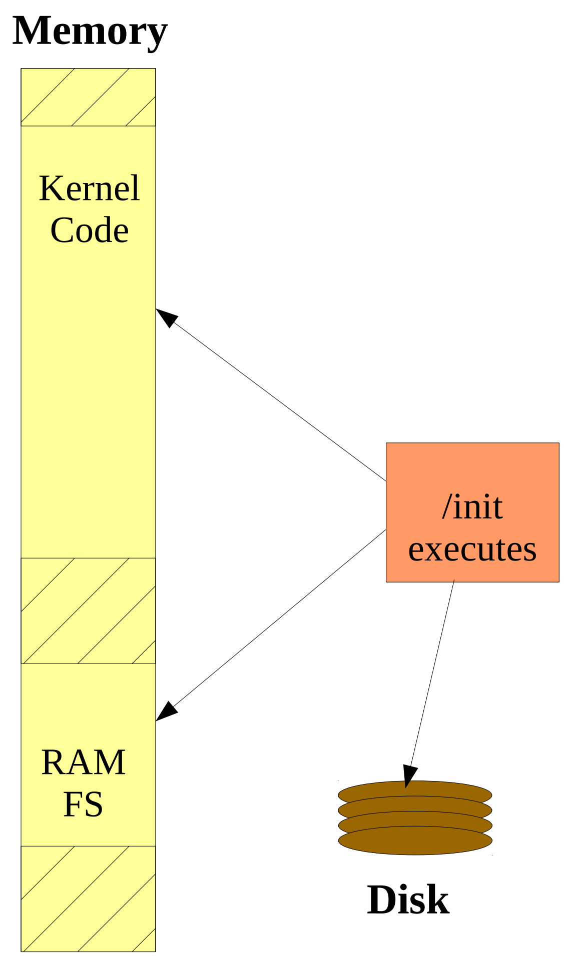
Loads device drivers from the RAMFS

Create the first process from /init



Boot Process

- Major steps
  - Hardware initialization
  - Boot loader
  - Kernel initialization
  - Distribution initialization



After that, it is a distribution initialization...

But most start with a shell /init

It mounts the real "/" from your disk

Then the root file system pivot happens

The rest of the initialization happens from your root partition

Food for Thoughts – Kernel vs Distribution

- Linux kernel vs Linux distribution
  - Two very different things
  - One kernel... (multiple versions of course)
  - Many distribution... with very different goals
  - Ubuntu, Debian, Raspbian, uc-linux, …
  - Promote the C programming language and shell scripting
- Google Android
  - Linux kernel
  - A few core libraries (webkit, gstreamer, etc.)
  - Promote the Java programming language
  - But a Java-based distribution with its own software concepts
  - Components and services **distributed across processes**

Food for Thoughts – OS vs DBMS

- Unix-like operating systems
  - Use private files or communicate through pipes/sockets (safe programming)
  - Single threaded and blocking libC (easy programming)
  - File system crashes meant loosing files and their content
- Opposite to DataBase Management Systems (DBMS)
  - Multi-threaded by essence, with shared data, protected through transactions
  - Strong query capabilities
  - Protected data, through failure recovery

You must reflect on these design differences!

Still true today?

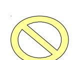
Absolutely...

Food for Thoughts – OS vs Other Platforms

- OSGi Platforms
  - Set-top boxes, ADSL boxes, factory-management gateways, etc.
  - A world of services and modules
  - Remote management – install/uninstall/start/stop
  - But **everything runs in a single** Java Virtual Machine
  - Runs on bare metal or on embedded operating system
  - Also runs on Linux/Windows/MacOS
- Eclipse Platform
  - OSGi platform
  - GUI integration
  - Resource integration

Food for Thoughts – OS vs Other Platforms

- Eclipse vs Linux
  - Both virtual machines, both application platforms
  - Both can run bare metal

✔	Memory Management	✔
✔	Threads and concurrency control	✔
✔	File system and sockets	✔
✔	Manage applications	✔
✔	Security	✔
	Memory Isolation	✔
✔	Window Manager	✔
✔	Web Integration	✔

Not in Oracle Java but available in Android... Was not available in Windows for many years...
Some have proposed memory isolation in Java and other high-level languages