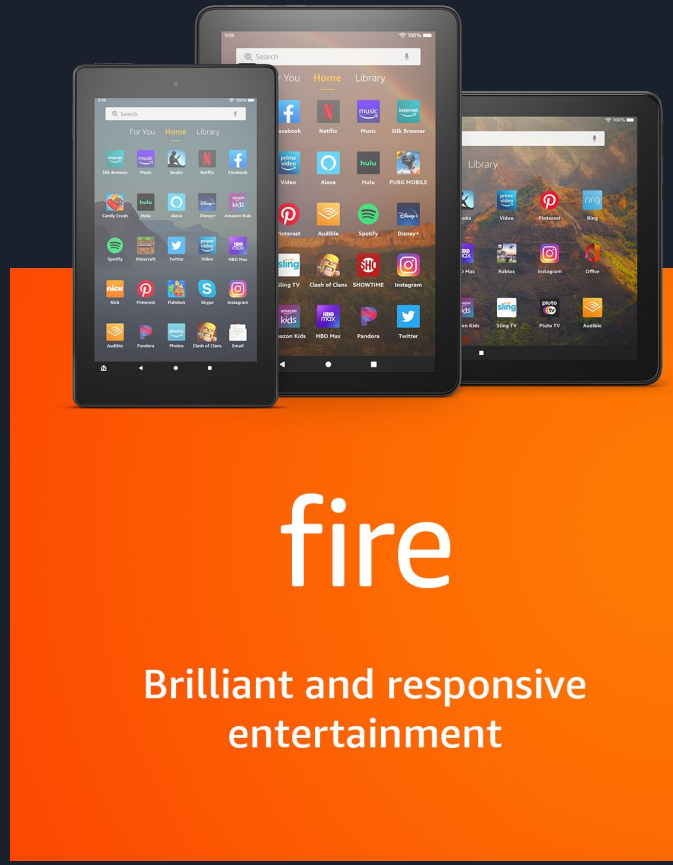
Fire OS

Amazon’s operating system for Kindle Fire

Kindle Fire History and Overview

- Kindle fire is Amazon's version of the iPad
- It provides access to movies, books, music, social media, and work-related content.

- **2004:** Amazon started selling e-books
- **2009:** Apple launched the iPad, which contained access to iTunes and iBooks stores
- **2011:** Jeff Bezos disclosed the "Otter project" and announced the launch of Kindle Fire



Kindle Fire Overview

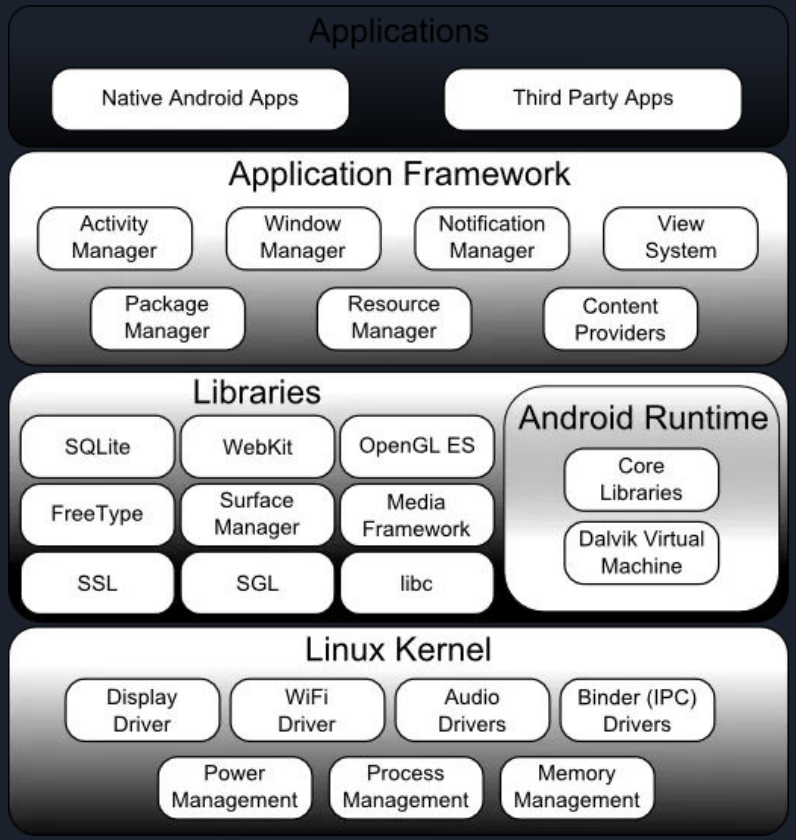
Hardware

- 7” multi-touch display with a 1024 x 600 pixel resolution capable of displaying 16 million colors at 169 pixels per inch
- 8GB of internal storage
- 2GB of this memory is reserved for the OS
- Texas Instruments OMAP 4430, a 1GHz dual-core processor

Software

- Fire OS
- OS is based on Google’s Android 2.3 (Gingerbread) mobile operating system
- Highly customized to match low price sales ($199) and reduced storage
- Customized in this case mostly means simplified

Diving into Fire OS Architecture



User applications

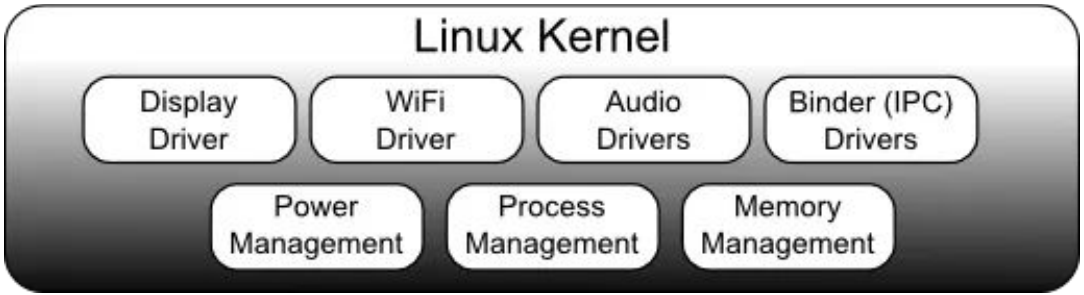
Applications manager

Libraries and runtime environment

Kernel

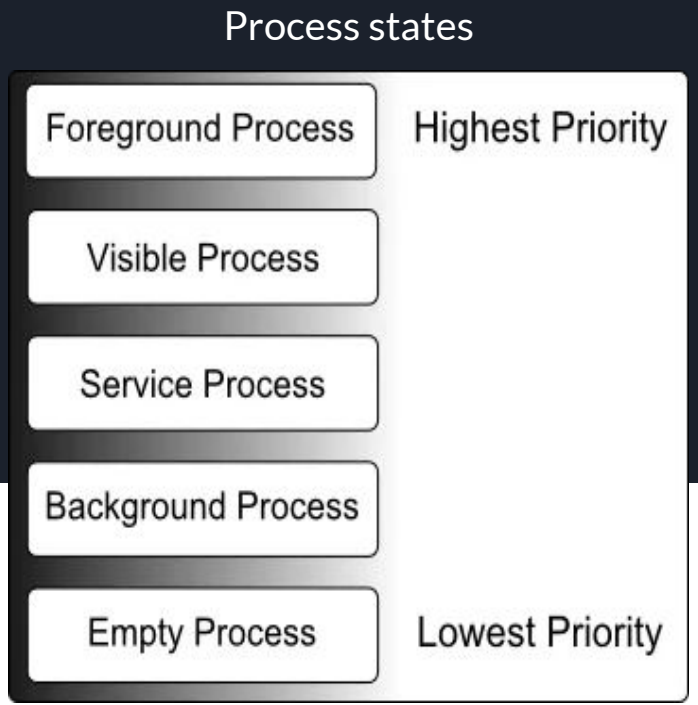
Kernel

- Based on Linux version 2.6
- Multitasking execution environment - dual core
- Android applications do not run as processes directly on the Linux kernel, instead they run on within its own instance of the Dalvik VM
- Why? Applications are essentially sandboxed and cannot interfere with the OS
- Also enforces level of abstraction so that applications aren't tied to specific hardware
- Fun fact: the Dalvik executable (.dex) format has a 50% smaller memory footprint than standard Java bytecode



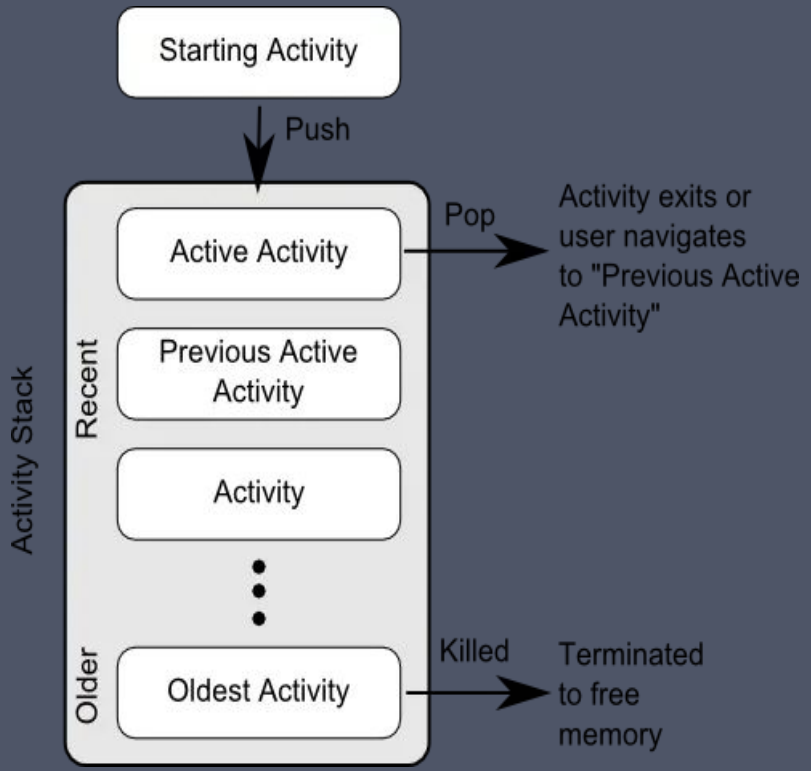
Resource Management

- Each running Android application is viewed by the OS as a **separate** process
- If resources on the device are reaching capacity, it will terminate processes to free up memory
- Importance hierarchy: considers both the **priority** and **state** of all running processes
- Processes are terminated from the **lowest priority** and working up the hierarchy until sufficient resources have been liberated



Activity Stack

- For each application that is running, the runtime system maintains an Activity Stack
- When an application is launched, the first of the application's activities to be started is placed onto the top of the stack
- When the active activity exits, it is popped off the stack and the activity located beneath it becomes the current active activity
- **4 activity states**: Running, Paused, Stopped, Killed



thread_create()

```
static int kthread(void *_create)
{
        /* Copy data: it's on kthread's stack */
        struct kthread_create_info *create = _create;
        int (*threadfn)(void *data) = create->threadfn;
        void *data = create->data;
        struct kthread self;
        int ret;

        self.should_stop = 0;
        init_completion(&self.exited);
        current->vfork_done = &self.exited;

        /* OK, tell user we're spawned, wait for stop or wakeup */
        __set_current_state(TASK_UNINTERRUPTIBLE);
        create->result = current;
        complete(&create->done);
        schedule();

        ret = -EINTR;
        if (!self.should_stop)
                ret = threadfn(data);

        /* we can't just return, we must preserve "self" on stack */
        do_exit(ret);
}
```

```
void
create(void (*func)())
{
  //look through all threads
  for(int i = 0; i < NTHREAD; i++){
    //found free thread
    if(thread[i].state == FREE){
      thread[i].ra = (uint64)(func);
      thread[i].sp = (uint64)(thread[i].stack + TSSIZE);
      thread[i].state = RUNNABLE;
      break;
    }
  }
}
```

```
struct task_struct *kthread_create(int (*threadfn)(void *data),
                                   void *data,
                                   const char namefmt[],
                                   ...)
{
        struct kthread_create_info create;

        create.threadfn = threadfn;
        create.data = data;
        init_completion(&create.done);

        spin_lock(&kthread_create_lock);
        list_add_tail(&create.list, &kthread_create_list);
        spin_unlock(&kthread_create_lock);

        wake_up_process(kthreadd_task);
        wait_for_completion(&create.done);

        if (!IS_ERR(create.result)) {
                struct sched_param param = { .sched_priority = 0 };
                va_list args;

                va_start(args, namefmt);
                vsnprintf(create.result->comm, sizeof(create.result->comm),
                          namefmt, args);
                va_end(args);
                /*
                 * root may have changed our (kthreadd's) priority or CPU mask.
                 * The kernel thread should not inherit these properties.
                 */
                sched_setscheduler_nocheck(create.result, SCHED_NORMAL, &param);
                set_cpus_allowed_ptr(create.result, cpu_all_mask);
        }
        return create.result;
}
```

```
/**
 * sys_sched_yield - yield the current processor to other threads.
 *
 * This function yields the current CPU to other tasks. If there are no
 * other threads running on this CPU then this function will return.
 */
SYSCALL_DEFINE0(sched_yield)
{
        struct rq *rq = this_rq_lock();

        schedstat_inc(rq, yld_count);
        current->sched_class->yield_task(rq);

        /*
         * Since we are going to call schedule() anyway, there's
         * no need to preempt or enable interrupts:
         */
        __release(rq->lock);
        spin_release(&rq->lock.dep_map, 1, _THIS_IP_);
        do_raw_spin_unlock(&rq->lock);
        preempt_enable_no_resched();

        schedule();

        return 0;
}
```

yield_thread()

```
void
yield(void)
{
  if(current->state != MONITOR){
    current->state = RUNNABLE;
  }
  schedule();
}
```

Potentially Interesting Stuff

```
/* Run through task list and migrate tasks from the dead cpu. */
static void migrate_live_tasks(int src_cpu)
{
        struct task_struct *p, *t;

        read_lock(&tasklist_lock);

        do_each_thread(t, p) {
                if (p == current)
                        continue;

                if (task_cpu(p) == src_cpu)
                        move_task_off_dead_cpu(src_cpu, p);
        } while_each_thread(t, p);

        read_unlock(&tasklist_lock);
}
```

```
/*
 * Schedules idle task to be the next runnable task on current CPU.
 * It does so by boosting its priority to highest possible.
 * Used by CPU offline code.
 */
void sched_idle_next(void)
{
        int this_cpu = smp_processor_id();
        struct rq *rq = cpu_rq(this_cpu);
        struct task_struct *p = rq->idle;
        unsigned long flags;

        /* cpu has to be offline */
        BUG_ON(cpu_online(this_cpu));

        /*
         * Strictly not necessary since rest of the CPUs are stopped by now
         * and interrupts disabled on the current cpu.
         */
        raw_spin_lock_irqsave(&rq->lock, flags);

        __setscheduler(rq, p, SCHED_FIFO, MAX_RT_PRIO-1);

        activate_task(rq, p, 0);

        raw_spin_unlock_irqrestore(&rq->lock, flags);
}
```

Thank you!

Questions?

```
/*
 * If it changed from the expected state, bail out now.
 */
if (unlikely(!ncsw))
        break;
```

Some quality comments from the smart programmers at Amazon:

```
/*
 * Ahh, all good. It wasn't running, and it wasn't
 * runnable, which means that it will never become
 * running in the future either. We're all done!
 */
break;
```

```
/*
 * Was it really running after all now that we
 * checked with the proper locks actually held?
 *
 * Oops. Go back and try again..
 */
if (unlikely(running)) {
        cpu_relax();
        continue;
}
```