



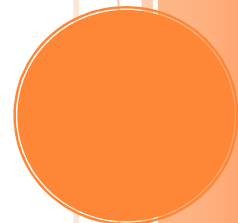
# OPERATING SYSTEM

*Lecturer: DHASON PADMAKUMAR*

***Student Name: LAPICHE Antonin***

***Student Number/ID: TP020975***

***Submission Date: 19/11/2009***



# OPERATING SYSTEM

## Table of content

|                                                |    |
|------------------------------------------------|----|
| Introduction.....                              | 2  |
| Process Control Management .....               | 3  |
| Processes.....                                 | 3  |
| States .....                                   | 3  |
| Scheduler and Group scheduling .....           | 4  |
| Deadlock Management .....                      | 6  |
| What is deadlock? .....                        | 6  |
| How to prevent from a deadlock? .....          | 7  |
| Memory Management .....                        | 9  |
| Virtual Memory Addressing .....                | 9  |
| Page Allocation .....                          | 10 |
| The buddy algorithm.....                       | 10 |
| Page Replacement Algorithm or “Swapping” ..... | 11 |
| The clock algorithm.....                       | 11 |
| Disk Scheduling Management .....               | 13 |
| Completely Fair Queuing Scheduler (CFQ).....   | 13 |
| Conclusion .....                               | 14 |
| References .....                               | 15 |

## INTRODUCTION<sup>1</sup>

For some time, **Google** has recognized the power of the mobile phone. In fact Google has tailored versions of its home page for different devices. It also makes mobile Gmail, Maps, and other programs available for free to users of a variety of phone models.

But like any good company, Google isn't content to rest on its laurels. Making a Web page that fits on a mobile phone's screen is a worthy task, but Google knew that it could do more. In 2005, it purchased **Android**, a Silicon Valley company that had been quietly working on creating software for the next generation of mobile phones: smart phones.

In 2007, Android was officially unveiled and available for free.

Android is a **Linux**-based **open source** platform for **smart phones**. The Android platform includes an **Operating System** based upon **Linux kernel (2.6)**, a **GUI**, a **Web browser** and many other **applications** considered key to a modern cellular handset. Android allows synchronization to a user's address book, calendar and other personal information management programs, though individual software makers will have to customize their offerings. Naturally, Google services like Calendar and Maps are built-in. Android allows users to browse the Internet more easily, integrate mapping services with local business listings and use many other software features traditionally associated with personal computers rather than mobile phones.

Android has also been implemented by users on a wide range of devices that it was not installed on by the manufacturer (Nokia, Dell, Asus, Motorola, etc). This is possible because Android is **open source**.

The **Android Software Development Kit (SDK)** is available for **free** download and works on a wide variety of platforms including Windows XP, Vista, Mac OS and Linux. It allows people to develop their own Android application in Java language.

Moreover Android can use **devices** like touch screens, video cameras, accelerometers, GPS and accelerated 3D graphics. It works with most media standards.

All this features shows at which point Android is more than the OS of a manufacturer for a phone but more likely a multi platform operating system.

---

<sup>1</sup> (Android (operating system))

## PROCESS CONTROL MANAGEMENT

Android is a Linux 2.6.29 (latest release) based operating system, so it has the same PCM as every Linux OS. The PCM is divided into three important parts:

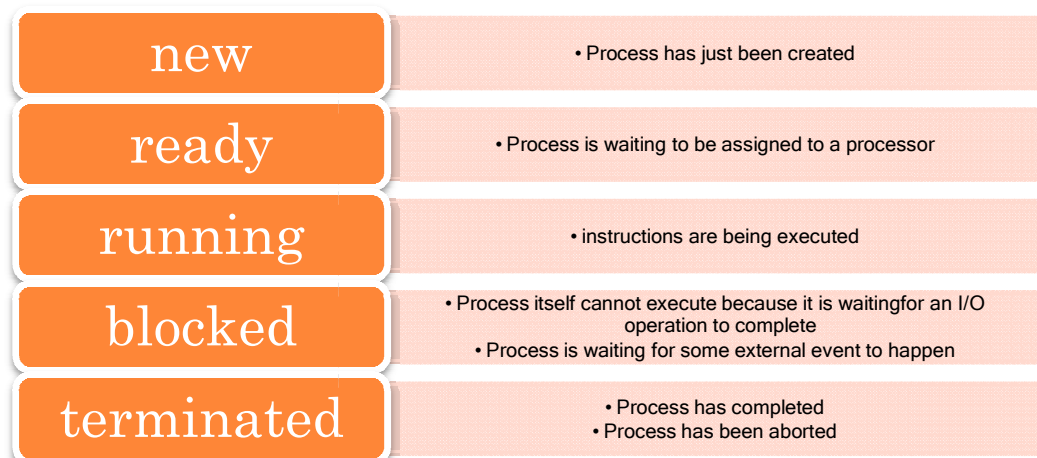
- Processes
- States of the processes
- Scheduling

### Processes

A process is a program in execution which progresses in a sequential manner. It requires resources like memory, files to complete its task and obviously CPU time .

### States

The best way to sum up the behavior of a process is to proceed by steps:



In order to switch between the “ready” and the “running” state, there is a special entity of the kernel called the “scheduler”. That’s what we are going to see in the next part.

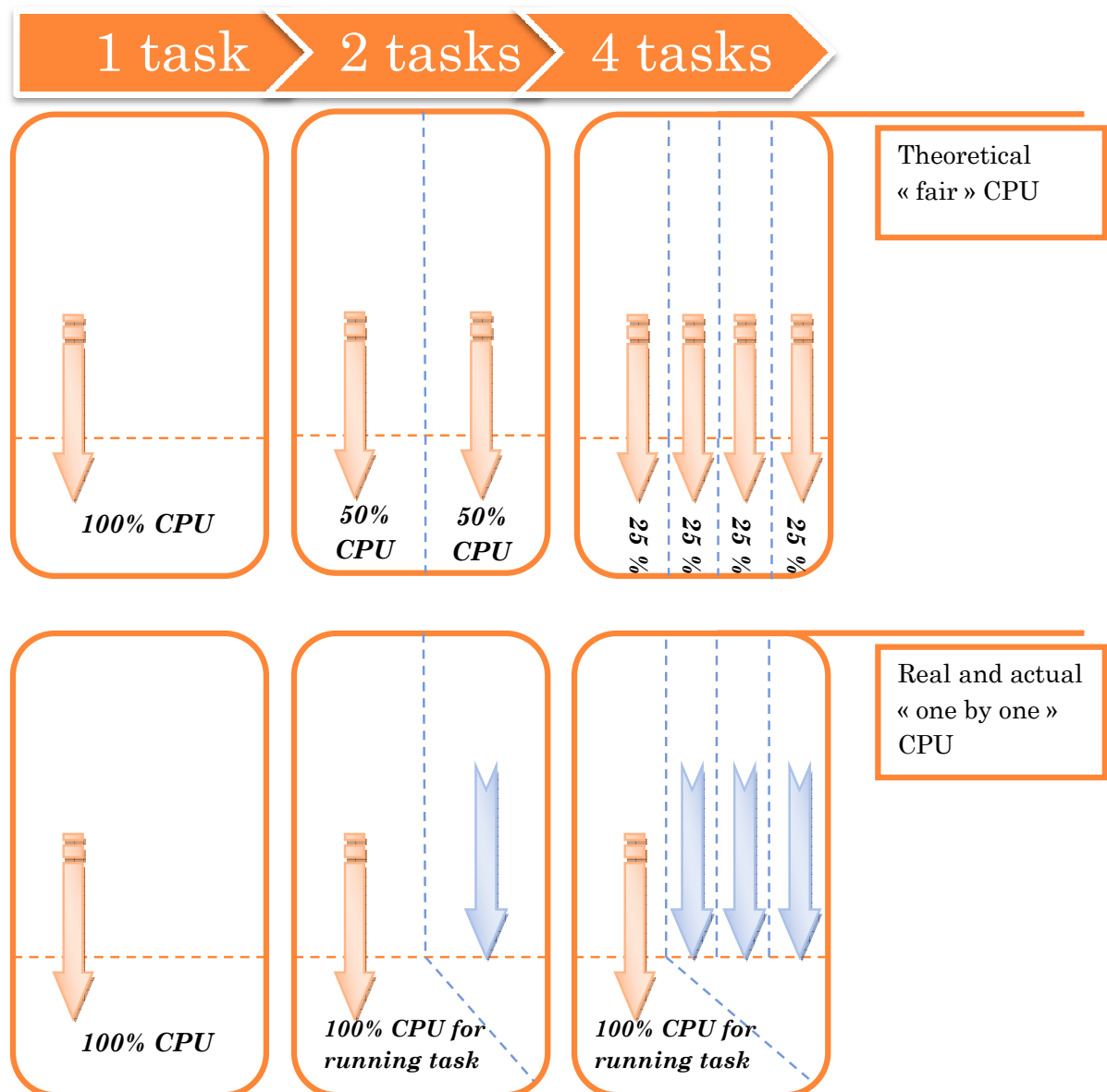
## Scheduler and Group scheduling

Linux is a multitasking operating system. To do multitask with less processor than tasks, you need to simulate several CPUs or switch between the different processes. The main goal of the scheduler is to do that. It decides when a process need to be switched from “ready” to “running” state (and reverse).

The scheduler in the latest kernel of Linux (after 2.6.23) is called the CFS:  
Completely Fair Scheduler.

The creator of this scheduler (Ingo Molnar) defines the CFS as an emulator of an “ideal, precise, multitasking CPU”.

It means that the CPU could run multiple processes at the same timesharing equally the power of the CPU.



It's obvious that the real CPU is totally unfair because processes are waiting while one is fully executed.

To avoid this unfairness, CFS<sup>2</sup> runs what we call a **fair clock**. The fair clock's rate of increase is calculated by dividing the total time by the total number of processes waiting. Calculating the clock's rate allows us to evaluate the amount of CPU time which would be used by the theoretical CPU. Now we can differentiate the processes.

The scheduler picks up the needier (in terms of CPU time) and puts it into the "running" state. Then the wait time decreases and the priority goes to another process.

Let me show a simple example with 3 processes: P1, P2 and P3.

The CPU burst time is randomly chosen.

The fair clock is like a dynamic quantum. It's calculated by:

$$FC = \frac{T}{n}$$

Where  $T$  is the sum of all the processes times (total time) and where  $n$  is the number of processes.

The priority is given to the longest Time (higher priority is 1).

| Process | Time(ns) | Priority | Fair Clock(ns) | T = 16 ns |
|---------|----------|----------|----------------|-----------|
| P1      | 10       | 1        | 5,333333333    | n = 3     |
| P2      | 4        | 2        |                |           |
| P3      | 2        | 3        |                |           |

We obtain the following Gant chart (for a better understanding, please see the spreadsheet just after with the detailed steps):

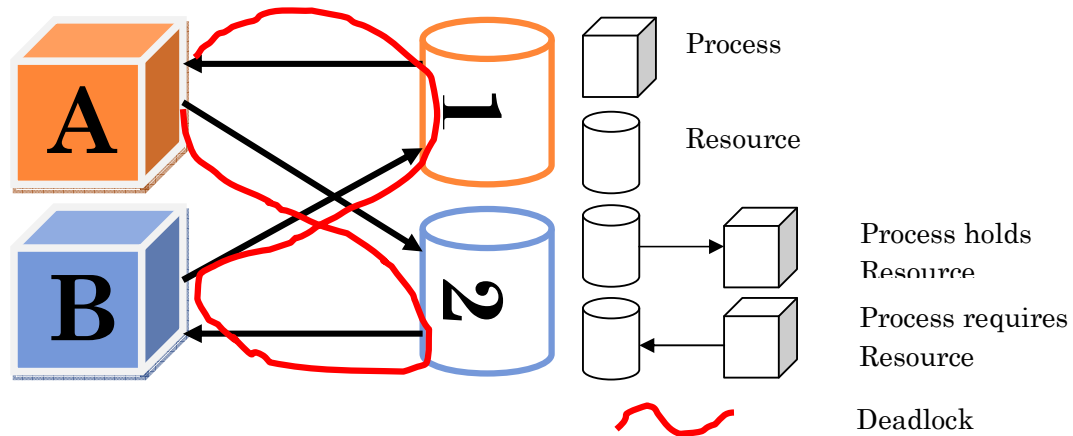
|      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |
|------|------|------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1 FC | 1 FC | 1 FC | 1 FC  | 1 FC  | 1 FC  | 1 FC  | 1 FC  | 1 FC  | 1 FC  | 1 FC  | 1 FC  | 1 FC  | 1 FC  | 1 FC  |       |
| ↔    | ↔    | ↔    | ↔     | ↔     | ↔     | ↔     | ↔     | ↔     | ↔     | ↔     | ↔     | ↔     | ↔     | ↔     |       |
| P1   | P1   | P2   | P3    | P2    | P1    | P2    | P3    | P1    | P1    | P2    | P3    | P1    | P3    | P3    |       |
| 0,00 | 5,33 | 8,89 | 11,26 | 12,84 | 13,89 | 14,60 | 15,06 | 15,38 | 15,58 | 15,72 | 15,82 | 15,88 | 15,92 | 15,95 | 16 ns |

<sup>2</sup> (Completely Fair Scheduler)

## DEADLOCK MANAGEMENT

### What is deadlock?

To well understand what a deadlock is, we are going to consider the following example:



Process A requires resource 1 while resource 1 is held by process B. In the same time, process B requires resource 2. The problem is that neither A or B can be executed! That's what we call Deadlock. The deadlock is permanent because none of the processes are triggered.

We notice that if the deadlock is true for 2 processes, it can also be for n processes.

Now, we know that a deadlock can occurs, but there are 4 conditions that need to be present:

- ☑ Hold and Wait: Processes already holding resources (or not) may require new resources or may require several resources.
- ☑ Circular Wait: Two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds.
- ☑ Mutual Exclusion: Resource can only be used by one process at a time.
- ☑ No preemption: Resource can't be released by any other thing that the holding process.<sup>3</sup>

<sup>3</sup> (Completely Fair Scheduler)

## How to prevent from a deadlock?

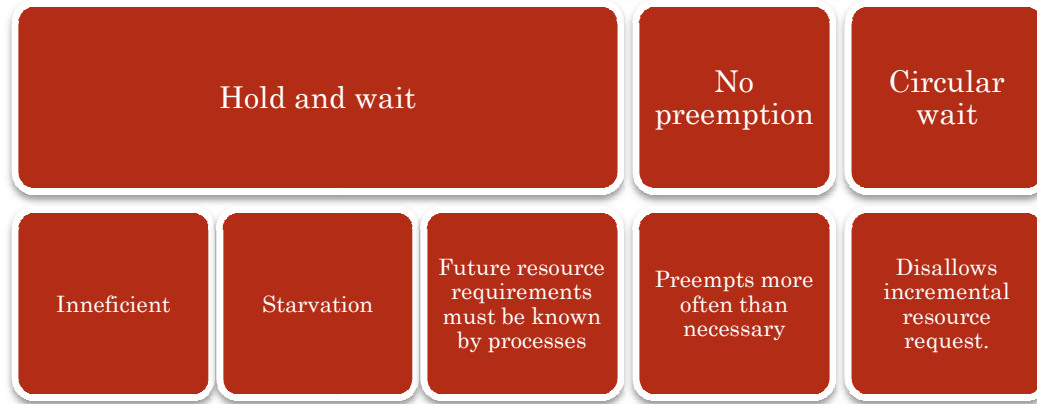
Linux 2.6 follows the deadlock prevention scheme. It means that the possibility of having a deadlock is excluded. Then you just have to avoid one of the 4 previous conditions.

- ☒ Hold and Wait:
  - Require a process to request and be allocated all its resources before it begins execution.
  - May lead to low resource utilization.
  - Starvation is a problem because a process may be held for a long time waiting for all its required resources.
  - If it needs several more resources, it needs to release all the currently held resources and then requests all of those it needs. That means that the the application needs to be aware of all of the required resources.
- ☒ Circular Wait:
  - Impose a total ordering on all resource types.
  - Require each process to request resources only in a strict increasing order.
  - Resources from the same resource type have to be requested together.
- ☒ Mutual Exclusion:
  - Can't really be disallowed because no process may have exclusive access to a resource.
  - If we need a resource, we really need it!
- ☒ No Preemption:
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - The state of preempted resources has to be saved and later restored. Not practical for many types of resources like printers.<sup>4</sup>

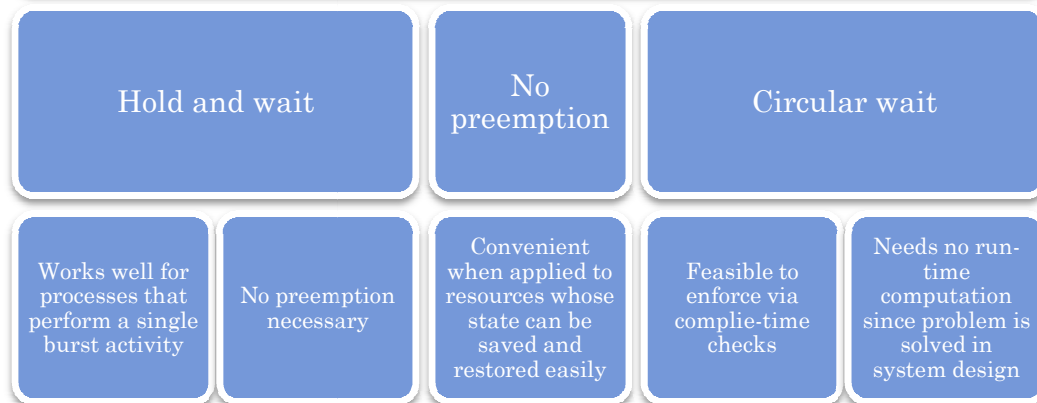
---

<sup>4</sup> (Completely Fair Scheduler)

## Disadvantages of prevention



## Advantages of prevention



As you see, the best scheme is to avoid circular wait by using ordering. That's what Linux uses.

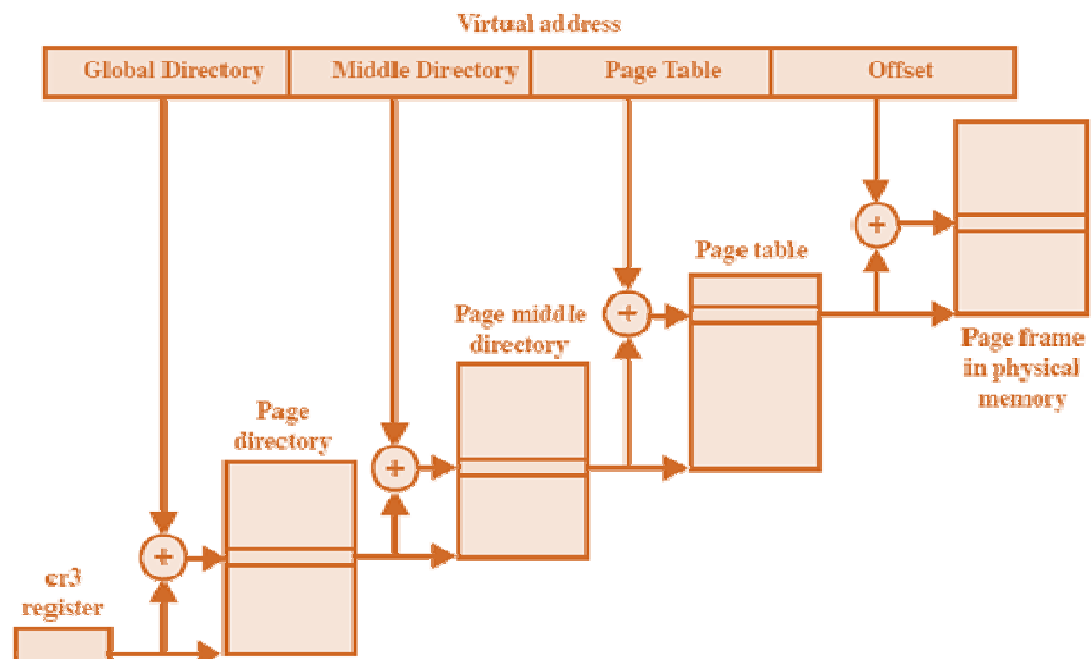
## MEMORY MANAGEMENT<sup>5</sup>

### Virtual Memory Addressing

Linux operating system uses paging memory partitioning technique. It uses a three-level page table structure:

- **Page directory:** An active process has a single page directory that is the size of one page. Each entry in the page directory points to one page of the page middle directory. The page directory must be in main memory for an active process.
- **Page middle directory:** The page middle directory may cover multiple pages. Each entry in the page middle directory points to one page in the page table.
- **Page table:** The page table may also cover multiple pages. Each page table entry refers to one virtual page of the process. To use this three-level page table structure, a virtual address in Linux is viewed as consisting of four fields. The leftmost (most significant) field is used as an index into the page directory. The next field serves as an index into the page middle directory. The third field serves as an index into the page table. The fourth field gives the offset within the selected page of memory.

We can sum up by the following diagram:



<sup>5</sup> (Stallings)

## Page Allocation<sup>6</sup>

To increase the efficiency of swapping, Linux defines a mechanism for dealing with contiguous blocks of pages mapped into contiguous blocks of page frames. For this purpose, the buddy system is used.

The kernel maintains a list of contiguous page frame groups of fixed size; a group may consist of 1, 2, 4, 8, 16, or 32 page frames. As pages are allocated and deallocated in main memory, the available groups are split and merged using the **buddy algorithm**.

The buddy algorithm

The buddy memory allocation technique allocates memory in powers of 2, i.e  $2^x$ , where x is an integer. U is the upper limit of powers of 2 and L the lower limit.

Let's have an example<sup>7</sup>:

Our processes are:

| Processes | Size in KB |
|-----------|------------|
| A         | 34         |
| B         | 66         |
| C         | 35         |
| D         | 67         |

Our memory is 2000KB. We fix L=6 and U=10.

The smallest block will be  $2^6=64\text{KB}$ . The largest will be  $2^{10}=1024\text{KB}$ .

|       |       |       |        |        |        |        |      |      |      |      |     |     |     |     |     |     |
|-------|-------|-------|--------|--------|--------|--------|------|------|------|------|-----|-----|-----|-----|-----|-----|
| time  | 64K   | 64K   | 64K    | 64K    | 64K    | 64K    | 64K  | 64K  | 64K  | 64K  | 64K | 64K | 64K | 64K | 64K | 64K |
| t = 0 | 1024K |       |        |        |        |        |      |      |      |      |     |     |     |     |     |     |
| t = 1 | A-64K | 64K   | 128K   |        | 256K   |        |      |      | 512K |      |     |     |     |     |     |     |
| t = 2 | A-64K | 64K   | B-128K |        | 256K   |        |      |      | 512K |      |     |     |     |     |     |     |
| t = 3 | A-64K | C-64K | B-128K |        | 256K   |        |      |      | 512K |      |     |     |     |     |     |     |
| t = 4 | A-64K | C-64K | B-128K |        | D-128K |        | 128K |      | 512K |      |     |     |     |     |     |     |
| t = 5 | A-64K | 64K   | B-128K |        | D-128K |        | 128K |      | 512K |      |     |     |     |     |     |     |
| t = 6 | 128K  |       |        | B-128K |        | D-128K |      | 128K |      | 512K |     |     |     |     |     |     |
| t = 7 | 256K  |       |        |        | D-128K |        | 128K |      | 512K |      |     |     |     |     |     |     |
| t = 8 | 1024K |       |        |        |        |        |      |      |      |      |     |     |     |     |     |     |

1. Program A requests a block of 64KB
2. Program B requests a block of 128KB
3. Program C requests a block of 64KB

<sup>6</sup> (Stallings)

<sup>7</sup> (Buddy memory allocation)

4. Program D requests a block of 128KB
5. Program C releases its memory
6. Program A releases its memory
7. Program B releases its memory
8. Program D releases its memory

We can write the buddy algorithm as below:

#### Function allocate()

1. *Look for a memory slot of a suitable size (the minimal  $2^k$  block that is larger or equal to that of the requested memory)*
  - 1.1. *If it is found, it is allocated to the program*
  - 1.2. *Else, it tries to make a suitable memory slot. The system does so by trying the following:*
    - 1.2.1. *Split a free memory slot larger than the requested memory size into half*
    - 1.2.2. *If the lower limit is reached, then allocate that amount of memory*
    - 1.2.3. *Go back to step 1 (look for a memory slot of a suitable size)*
    - 1.2.4. *Repeat this process until a suitable memory slot is found*

#### Function free()

1. *Free the block of memory*
2. *Look at the neighboring block - is it free too?*
3. *If it is, combine the two, and go back to step 2 and repeat this process until either the upper limit is reached (all memory is freed), or until a non-free neighbor block is encountered*

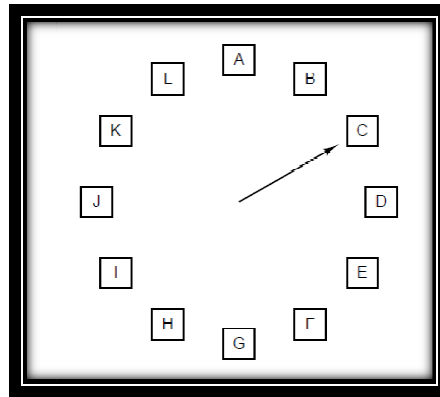
### Page Replacement Algorithm or “Swapping”

The Linux memory manager (or pager) determines which pages to keep in memory and which pages to replace when free memory becomes rare.

The Linux page replacement algorithm is based on the clock algorithm

The clock algorithm

You need to keep all the page frames on a circular list in the form of a clock. A hand points to the oldest page. All the page frames have a R bit which lets know who is the oldest page.



Let's have an example (A,B,C,D are already loaded in frames):

| Time        | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|---|---|---|---|---|---|---|---|---|---|----|
| Requests    |   | C | A | D | B | E | B | A | B | C | D  |
| 0           | A | A | A | A | A | E | E | E | E | E | D  |
| 1           | B | B | B | B | B | B | B | B | B | B | B  |
| 2           | C | C | C | C | C | C | C | A | A | A | A  |
| 3           | D | D | D | D | D | D | D | D | D | C | C  |
| Page faults | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 4  |

Here we can see the move of the clock's hand (in green):

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | 1 | E | 1 | E | 1 | E | 1 | E | 1 | E | 1 | D |
| 1 | B | 0 | B | 1 | B | 0 | B | 1 | B | 0 | B | 0 | B |
| 1 | C | 0 | C | 0 | C | 1 | A | 1 | A | 0 | A | 0 | A |
| 1 | D | 0 | D | 0 | D | 0 | D | 1 | C | 0 | C | 0 | C |

For the first page fault, A is 1 so we reset to 0 and we go to the next page. B is 1 so we reset to 0 and we go to the next page. C is 1 so we reset to 0 and we go to the next page. D is 1 so we reset to 0 and we go to the next page. A is 0, that means that it's the oldest process. We replace by E and we go to the next page, hand is on page number 1.

So we can write the clock algorithm as below:

#### Function clock\_algorithm ()

```

1. while (oldest page not found)
    1.1.1. if(used bit for current page = 0) then replace
           current page
    1.1.2. else reset used bit
    1.2. then advance clock pointer

```

The Linux scheme is a little bit different. The R bit is replaced with an 8-bit "age" variable. Each time that a page is accessed, the age variable is

incremented. In the background, Linux periodically sweeps through the global page pool and decrements the age variable for each page as it rotates through all the pages in main memory. A page with an age of 0 is an old and is the best candidate for replacement. The larger the value of age, the more frequently a page has been used in recent times and the less eligible it is for replacement. Thus, the Linux algorithm is a form of least frequently used policy.

## DISK SCHEDULING MANAGEMENT<sup>8</sup>

The 2.6 Linux Kernel includes selectable disk schedulers. They control the way the Kernel commits reads and writes to disks.

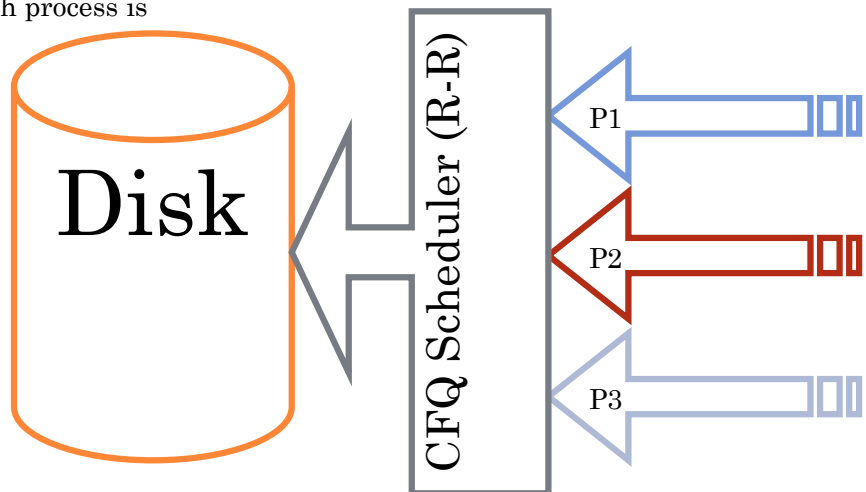
Without a scheduler, the kernel would just give service to the first request “in”. It can lead to massive Hard Disk thrashing: if one process was reading from one part of the disk, and one writing to another, the heads would have to seek back and forth across the disk for every operation. The scheduler’s main goal is to avoid that and to optimize disk access times.

To do this, the Android OS uses just one of the Linux schedulers: the CFQ disk scheduler.

### Completely Fair Queuing Scheduler (CFQ)<sup>9</sup>

In the CFQ algorithm, each process is associated its own queue, and each queue is assigned a timeslice.

The scheduler runs over each queue in a round-robin way. It gives services to the requests from the queue until timeslice is over or until there is no more requests.



If there are no more requests, the CFQ scheduler will wait for a brief moment in an inactive way (10ms by default) for a new request on the queue.

If in this short time, there is a request from the process in the active queue, the scheduler avoids searching for another process. Else, the scheduler moves on the next queue designed by the Round-Robin.

<sup>8</sup> (Linux I/O schedulers), (Linux I/O schedulers)

<sup>9</sup> (Completely Fair Queuing), (Love)

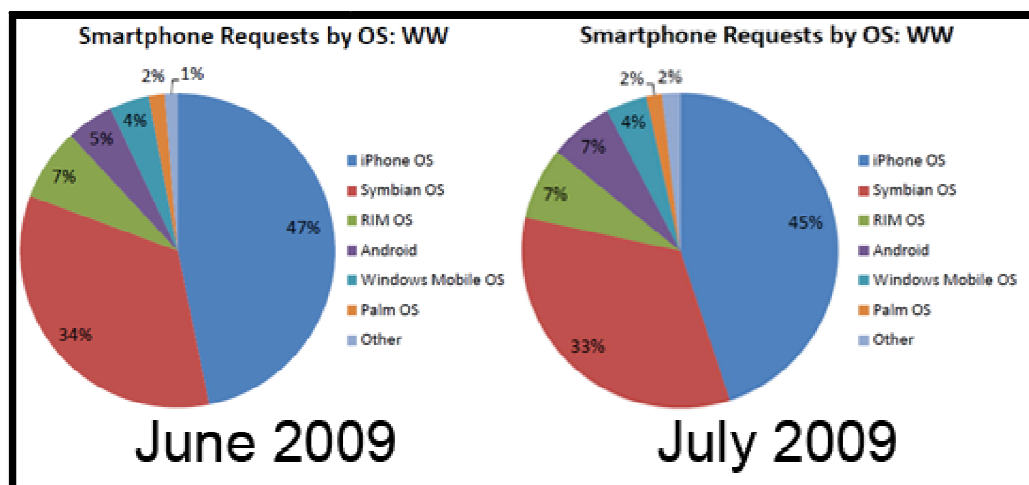
## CONCLUSION

As a conclusion, we can say that operating systems on mobile devices are part of the future.

Now you can install applications on smartphones and others smart portable devices.

Android is a big step on this direction because it provides a free platform for developers. Moreover, like Google open its source code, it allows people to increase the potential of this OS or just to modify for their own applications.

To really show you the progression of this brand new OS, just have a look of its progression in one month. Think about it in a year!



## REFERENCES

2009 <<http://www.devshed.com/c/a/BrainDump/Linux-IO-Schedulers/3/>>.

Android (operating system). 2009

<[http://en.wikipedia.org/wiki/Android\\_%28operating\\_system%29](http://en.wikipedia.org/wiki/Android_%28operating_system%29)>.

Buddy memory allocation. 2009

<[http://en.wikipedia.org/wiki/Buddy\\_memory\\_allocation](http://en.wikipedia.org/wiki/Buddy_memory_allocation)>.

CFS group scheduling. 2009 <<http://lwn.net/Articles/240474/>>.

Completely Fair Queuing. 2009 <<http://fr.wikipedia.org/wiki/CFQ>>.

Completely Fair Scheduler. 2009

<[http://fr.wikipedia.org/wiki/Completely\\_Fair\\_Scheduler](http://fr.wikipedia.org/wiki/Completely_Fair_Scheduler)>.

Completely Fair Scheduler. 2009 <<http://www.linuxjournal.com/article/10267>>.

Deadlock. 2009 <<http://en.wikipedia.org/wiki/Deadlock>>.

How to run Google Android in Virtual Box. 2009

<<http://www.taranfx.com/blog/how-to-run-google-android-in-virtualbox-vmware-on-netbooks>>.

Linux I/O schedulers. 2009 <<http://www.wlug.org.nz/LinuxIoScheduler>>.

Linux I/O schedulers. 2009 <<http://www.devshed.com/c/a/BrainDump/Linux-IO-Schedulers/3/>>.

Live-android - Project hosting on google code. 2009

<<http://code.google.com/p/live-android/>>.

Love, Robert. Linux System Programming. O'REILLY, 2007.

Sibsankar Haldar, Alex A. Aravind. Operating Systems. Pearson Education, 2009.

Stallings, William. Operating systems: internals and design principles. Prentice Hall, 2008.

What is Completely Fair Scheduler? 2009

<<http://immike.net/blog/2007/08/01/what-is-the-completely-fair-scheduler/>>.