

Lab Session 4

Module Information:

Module Name: Operating Systems

Module Code: UFCFWK-15-2

Module Tutor: Shancang Li, Ian Johnson, David Wyatt

Main tasks:

1. Download and build a local version of UWE's variant of Pintos OS
2. Use Git to fork a PintosOS
 - Use your GCC to build the Pintos OS
 - Use Qemu to run your newly built OS
3. Modify your version of Pintos OS to support
 - process termination messages
4. Use Git to push your changes to your remote repo

Download and build a local version of Pintos OS

The source code for Pintos OS is provided in Gitlab:

<https://gitlab.uwe.ac.uk/s23-li/Pintos>

Fork a version of Pintos into your own Gitlab workspace and then clone the resulting repo locally. Change into the directory and list its contents. You should see output similar to:

- "threads/" - Source code for the base kernel, which you will modify starting in this worksheet.
- "userprog/" - Source code for the user program loader, which you will modify in the assignment.
- "vm/" - An almost empty directory and not something we will consider during the course.
- "filesystem/" - Source code for a basic file system. You will use this file system in the assignment, but you will not modify it.
- "devices/" - Source code for I/O device interfacing: keyboard, timer, disk, etc. You will modify the timer implementation in project 1. Otherwise you should have no need to change this code.
- "lib/" - An implementation of a subset of the standard C library. The code in this directory is compiled into both the Pintos kernel and, for the assignment, user programs that run under it. In both kernel code and user programs, headers in this

directory can be included using the `#include <...>` notation. You should have little need to modify this code.

- "lib/kernel/" - Parts of the C library that are included only in the Pintos kernel. This also includes implementations of some data types that you are free to use in your kernel code: bitmaps, doubly linked lists, and hash tables. In the kernel, headers in this directory can be included using the `#include <...>` notation.
- "lib/user/" - Parts of the C library that are included only in Pintos user programs. In user programs, headers in this directory can be included using the `#include <...>` notation.
- "tests/" - Tests for each project. You can modify this code if it helps you test your submission, but we will replace it with the originals before we run the tests.
- "examples/" - Example user programs for use in the assignment.
- "misc/" and "utils/" - Contains programs for running Pintos, building filesystems, and other useful tools.

To build Pintos you must run `make` in different directories. First, `cd` into the **threads** directory and then issue the **make** command. This will create a **build** directory under **threads**, populating it with a **Makefile** and a few subdirectories. The kernel is built inside.

Unlike, the minimal OS of worksheet 2, Pintos is built with the standard Linux system tools, i.e. there is no need to use our previously compiled cross-compiler. Why does this work?

Pintos is no different from our minimal OS, it boots an x86 machine from scratch, making no dependencies to existing Linux libraries, but still uses its system tools to build. It does this by providing a complicated set of command line arguments to GCC and other tools. We could have done something similar for our minimal OS, but the drawback is that we must remember these different options, we must also be sure that they do not differ between Linux distros. The Pintos **make** system does just this.

Additionally, Pintos provides its own library for building its File system and boot loader, while in the minimal OS we used Grub's tools to build the ISO image and install the boot loader.

Once the build has completed, the following files are of particular interest in the **build** directory:

- "Makefile" - A copy of `pintos/src/Makefile.build`. It describes how to build the kernel.
- "kernel.o" - Object file for the entire kernel. This is the result of linking object files compiled from each individual kernel source file into a single object file.
- "kernel.bin" - Memory image of the kernel, that is, the exact bytes loaded into memory to run the Pintos kernel. This is just `kernel.o` with debug information stripped out, which saves a lot of space, which in turn keeps the kernel from bumping up against a 512 kB size limit imposed by the kernel loader's design.
- "loader.bin" - Memory image for the kernel loader, a small chunk of code written in assembly language that reads the kernel from disk into memory and starts it up. It is exactly 512 bytes long, a size fixed by the PC BIOS.

Running Pintos

The Pintos source base provides a use set of programs for working with the OS, including tools to build a basic filesystem and program to run Pintos in a simulator. The original Pintos distribution used the Bochs simulator, however, as per our minimal OS we will use QEMU, instead. In the simplest case, you can invoke **pintos** as

```
pintos argument...
```

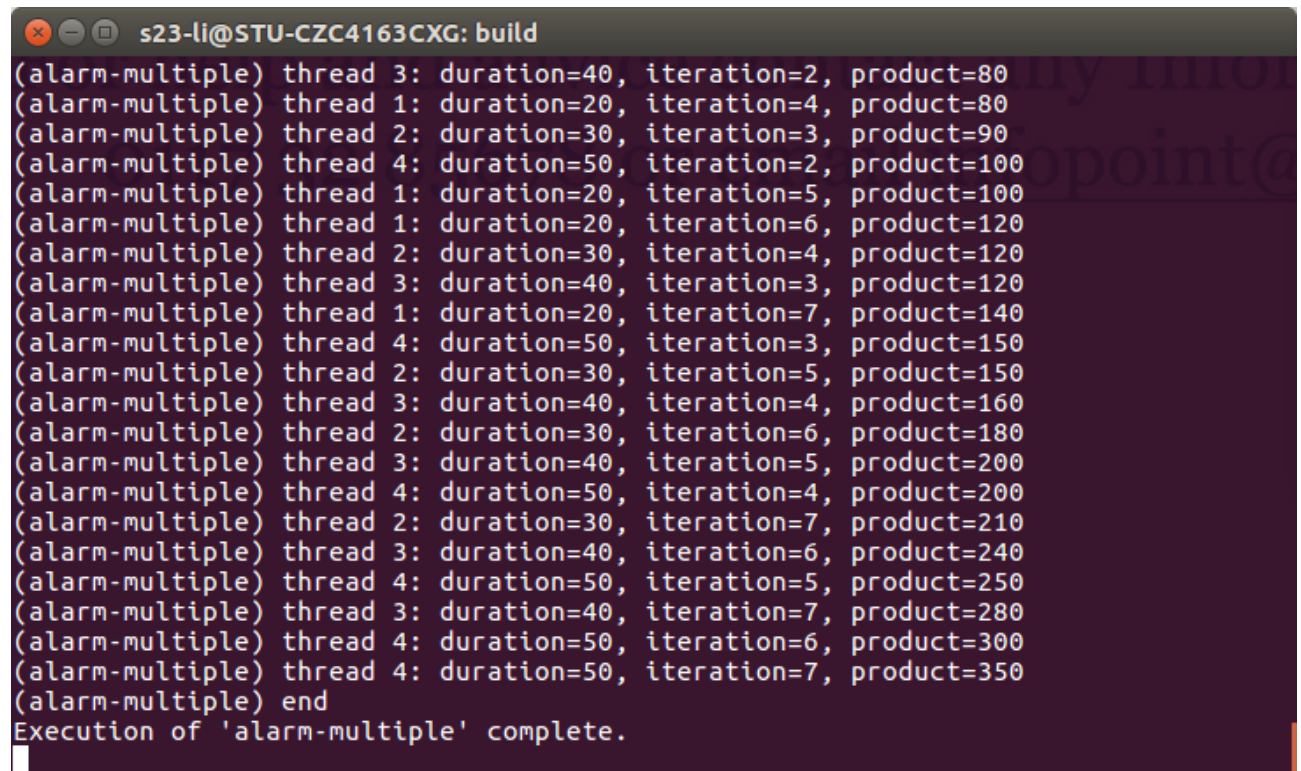
Each argument is passed to the Pintos kernel for it to act on.

The **pintos** program can be found in the **utils** directory, and you should add this to your local path.

Let's try it! Change into the **build** directory, then issue the command

```
pintos run alarm-multiple
```

which passed the arguments **run alarm-multiple** to the Pintos kernel. In these argument, **run** instructs the kernel to run a test and **alarm-multiple** is the test to run.



```
s23-li@STU-CZC4163CXG: build
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(alarm-multiple) thread 1: duration=20, iteration=6, product=120
(alarm-multiple) thread 2: duration=30, iteration=4, product=120
(alarm-multiple) thread 3: duration=40, iteration=3, product=120
(alarm-multiple) thread 1: duration=20, iteration=7, product=140
(alarm-multiple) thread 4: duration=50, iteration=3, product=150
(alarm-multiple) thread 2: duration=30, iteration=5, product=150
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
```

```

QEMU
(alarm-multiple) thread 0: duration=10, iteration=7, product=70
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(alarm-multiple) thread 1: duration=20, iteration=6, product=120
(alarm-multiple) thread 2: duration=30, iteration=4, product=120
(alarm-multiple) thread 3: duration=40, iteration=3, product=120
(alarm-multiple) thread 1: duration=20, iteration=7, product=140
(alarm-multiple) thread 4: duration=50, iteration=3, product=150
(alarm-multiple) thread 2: duration=30, iteration=5, product=150
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.

```

The Pintos kernel has comand and options other than run, you can see these by using the command line argument **-h**, e.g.

```
pintos -h
```

User Programs

For the remainder of worksheet 3 and also the group assignment we will study user programs. The original repo already supports loading and running user programs, but no I/O or interactivity is possible. In the worksheet and assignment you will enable programs to interact with the OS via system calls.

You will be working in the **userprog** directory for the duration, but you will also be interacting with many other parts of Pintos. The relevant parts are outlined below.

Overview

Up to now, all of the code you have run under Pintos has been part of the operating system kernel. This means, for example, that the alarm example from the previous section ran as part of the kernel, with full access to privileged parts of the system. Once we start running user programs on top of the operating system, this is no longer true. The remainder of worksheet and the assignment deals with the consequences.

We allow more than one process to run at a time. Each process has one thread (multithreaded processes are not supported). User programs are written under the illusion that they have the entire machine. This means that when you load and run multiple processes at a time, you must manage memory, scheduling, and other state correctly to maintain this illusion.

The alarms example compiled the code directly into your kernel, so we had to require certain specific function interfaces within the kernel. From now on, we will test your operating system by running user programs. This gives you much greater freedom. You must make sure that the user program interface meets the specifications described here, but given that constraint you are free to restructure or rewrite kernel code however you wish!

The easiest way to get an overview of the programming you will be doing is to simply go over each part you'll be working with. In **userprog**, you'll find a small number of files, but here is where the bulk of your work will be:

- "process.c" and "process.h" - Loads ELF binaries and starts processes.
- "pagedir.c" and "pagedir.h" - A simple manager for 80x86 hardware page tables. Although you probably won't want to modify this code for this project, you may want to call some of its functions for the group assignment. There will be an in lab presentation covering more details of this part.
- "syscall.c" and "syscall.h" - Whenever a user process wants to access some kernel functionality, it invokes a system call. This is a skeleton system call handler. Currently, it just prints a message and terminates the user process. In group assignment you will add code to do everything else needed by system calls.
- "exception.c" and "exception.h" - When a user process performs a privileged or prohibited operation, it traps into the kernel as an "exception" or "fault". These files handle exceptions. Currently all exceptions simply print a message and terminate the process. Some, but not all, solutions to group assignment require modifying **page_fault()** in this file.
- "gdt.c" and "gdt.h" - The 80x86 is a segmented architecture. The Global Descriptor Table (GDT) is a table that describes the segments in use. These files set up the GDT. You should not need to modify these files. You can read the code if you're interested in how the GDT works.
- "tss.c" and "tss.h" - The Task-State Segment (TSS) is used for 80x86 architectural task switching. Pintos uses the TSS only for switching stacks when a user process enters an interrupt handler, as does Linux. You should not need to modify these files. You can read the code if you're interested in how the TSS works.

Using the File System

You will need to interface to the file system code for this project, because user programs are loaded from the file system and many of the system calls you must implement deal with the file system. However, the focus of this project is not the file system, so Pintos provides a simple but complete file system in the **filesys** directory. You will want to look over the **filesys.h** and **file.h** interfaces to understand how to use the file system, and especially its many limitations.

There is no need to modify the file system code for the assignment, and so we recommend that you do not. Working on the file system is likely to distract you from this project's focus.

You need to be able to create a simulated disk with a file system partition. The `pintos-mkdisk` program provides this functionality. From the **userprog/build** directory, execute **`pintos-mkdisk filesys.dsk --filesys-size=2`**. This command creates a simulated disk named `filesys.dsk` that contains a 2 MB Pintos file system partition. Then format the file system partition by passing `-f -q` on the kernel's command line: `pintos -f -q`. The `-f` option causes the file system to be formatted, and `-q` causes Pintos to exit as soon as the format is done.

You'll need a way to copy files in and out of the simulated file system. The `pintos -p` ("**put**") and `-g` ("**get**") options do this. To copy file into the Pintos file system, use the command **`pintos -p file -- -q`**. (The `--` is needed because `-p` is for the `pintos` script, not for the simulated kernel.) To copy it to the Pintos file system under the name `newname`, add `-a newname`: **`pintos -p file -a newname -- -q`**. The commands for copying files out of a VM are similar, but substitute `-g` for `-p`.

Incidentally, these commands work by passing special commands `extract` and `append` on the kernel's command line and copying to and from a special simulated "scratch" partition. If you're very curious, you can look at the `pintos` script as well as **`filesys/fsutil.c`** to learn the implementation details.

Here's a summary of how to create a disk with a file system partition, format the file system, copy the **`echo`** program into the new disk, and then run `echo`, passing argument **`x`**. (Argument passing won't work until you implemented it as part of the assignment!) It assumes that you've already built the examples in `examples` and that the current directory is **userprog/build**:

```
pintos-mkdisk filesys.dsk --filesys-size=2
```



```
s23-li@STU-CZC4163CXG: build
dinc -I../.. -I../..lib -I../..lib/user -I. -Wall -W -Wstrict-prototypes -Wmis
sing-prototypes -Wsystem-headers -MMD -MF tests/filesys/base/child-syn-read.d
gcc -m32 -WL,--build-id=none -nostdlib -static -WL,-T,../..lib/user/user.lds t
ests/filesys/base/child-syn-read.o tests/lib.o tests/filesys/seq-test.o lib/user
/entry.o libc.a -o tests/filesys/base/child-syn-read
gcc -m32 -c ../..tests/filesys/base/child-syn-wrt.c -o tests/filesys/base/child
-syn-wrt.o -g -msoft-float -O -DBEN_MODS -std=gnu99 -fno-stack-protector -nostdi
nc -I../.. -I../..lib -I../..lib/user -I. -Wall -W -Wstrict-prototypes -Wmissi
ng-prototypes -Wsystem-headers -MMD -MF tests/filesys/base/child-syn-wrt.d
gcc -m32 -WL,--build-id=none -nostdlib -static -WL,-T,../..lib/user/user.lds t
ests/filesys/base/child-syn-wrt.o tests/lib.o tests/filesys/seq-test.o lib/user/
entry.o libc.a -o tests/filesys/base/child-syn-wrt
make[1]: Leaving directory '/home/netlab/s23-li/os/Pintos/userprog/build'
bash$ cd build/
bash$ ls
Makefile  filesys      kernel.o  libc.a      tests      userprog
devices   kernel.bin  lib       loader.bin  threads
bash$ pintos-mkdisk filesys.disk --filesys-size=2
pintos-mkdisk: command not found
bash$ ../..utils/pintos-mkdisk filesys.disk --filesys-size=2
bash$ ls
Makefile  filesys      kernel.bin  lib       loader.bin  threads
devices   filesys.disk kernel.o    libc.a    tests       userprog
bash$
```

you will see that the filesys.disk created.

Then run

```
pintos -f -q
```

```
s23-li@STU-CZC4163CXG: build
operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
PiLo hda1
Loading.....
Kernel command line: -f -q
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 97,280,000 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 182 sectors (91 kB), Pintos OS kernel (20)
Kernel PANIC at ../../filesystem/filesys.c:22 in filesys_init(): No file system device found, can't initialize file system.
Call stack: 0xc0027792.
The 'backtrace' program can make call stacks useful.
Read "Backtraces" in the "Debugging Tools" chapter
of the Pintos documentation for more information.
Timer: 63 ticks
Thread: 0 idle ticks, 63 kernel ticks, 0 user ticks
Console: 660 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
bash$
```

```
pintos -p ../../examples/echo -a echo -- -q
```



```
s23-li@STU-CZC4163CXG: build
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 389,120,000 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 182 sectors (91 kB), Pintos OS kernel (20)
hda2: 76 sectors (38 kB), Pintos scratch (22)
scratch: using hda2
Kernel PANIC at ../../filesystem/filesys.c:22 in filesys_init(): No file system dev
ice found, can't initialize file system.
Call stack: 0xc0027792.
The 'backtrace' program can make call stacks useful.
Read "Backtraces" in the "Debugging Tools" chapter
of the Pintos documentation for more information.
Timer: 66 ticks
Thread: 0 idle ticks, 66 kernel ticks, 0 user ticks
hda2 (scratch): 0 reads, 0 writes
Console: 766 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
bash$ pwd
/home/netlab/s23-li/os/Pintos/userprog/build
bash$
```

pintos -q run 'echo x'

```
s23-li@STU-CZC4163CXG: build
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 389,120,000 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 182 sectors (91 kB), Pintos OS kernel (20)
hda2: 76 sectors (38 kB), Pintos scratch (22)
scratch: using hda2
Kernel PANIC at ../../filesystem/filesys.c:22 in filesys_init(): No file system dev
ice found, can't initialize file system.
Call stack: 0xc0027792.
The 'backtrace' program can make call stacks useful.
Read "Backtraces" in the "Debugging Tools" chapter
of the Pintos documentation for more information.
Timer: 66 ticks
Thread: 0 idle ticks, 66 kernel ticks, 0 user ticks
hda2 (scratch): 0 reads, 0 writes
Console: 766 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
bash$ pwd
/home/netlab/s23-li/os/Pintos/userprog/build
bash$
```

```
s23-li@STU-CZC4163CXG: build
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
PiLo hda1
Loading.....
Kernel command line: -q run 'echo x'
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 419,020,800 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 182 sectors (91 kB), Pintos OS kernel (20)
Kernel PANIC at ../../filesystem/filesys.c:22 in filesys_init(): No file system dev
ice found, can't initialize file system.
Call stack: 0xc0027792.
The 'backtrace' program can make call stacks useful.
Read "Backtraces" in the "Debugging Tools" chapter
of the Pintos documentation for more information.
Timer: 63 ticks
Thread: 0 idle ticks, 63 kernel ticks, 0 user ticks
Console: 671 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
bash$ ../../utils/pintos -q run 'echo x'
```

You can delete a file from the Pintos file system using the `rm` file kernel action, e.g. **pintos -q rm file**. Also, **ls** lists the files in the file system and **cat file** prints a file's contents to the display.

How User Programs Work

Pintos can run normal C programs, as long as they fit into memory and use only the system calls you implement. Notably, `malloc()` cannot be implemented because none of the system calls required for this project allow for memory allocation. Pintos also can't run programs that use floating point operations, since the kernel doesn't save and restore the processor's floating-point unit when switching threads.

The `src/examples` directory contains a few sample user programs. The Makefile in this directory compiles the provided examples, and you can edit it to compile your own programs as well. Some of the example programs will only work once projects 3 or 4 have been implemented.

Pintos can load ELF executables with the loader provided for you in `userprog/process.c`. ELF is a file format used by Linux, Solaris, and many other operating systems for object files, shared libraries, and executables. You can actually use any compiler and linker that

output 80x86 ELF executables to produce programs for Pintos. (We've provided compilers and linkers that should do just fine.)

You should realize immediately that, until you copy a test program to the simulated file system, Pintos will be unable to do useful work. You won't be able to do interesting things until you copy a variety of programs to the file system. You might want to create a clean reference file system disk and copy that over whenever you trash your filesys.dsk beyond a useful state, which may happen occasionally while debugging.

Virtual Memory

This will be covered in labs in week 32 (13/02/2017) and then this page will be updated with more information.

For the interested reader you can read early on the original Stanford Pintos pages, these notes are based on then same original documentation. [Extra Pintos information at Stanford.](#)

Extending Pintos to add Process Termination Messages

Whenever a user process terminates, because it called `exit` or for any other reason, print the process's name and exit code, formatted as if printed by `printf ("%s: exit(%d)\n", ...);`. The name printed should be the full name passed to `process_execute()`, omitting command-line arguments. Do not print these messages when a kernel thread that is not a user process terminates, or when the halt system call is invoked. (note: as you will not be implementing the `haltsystem` call until the assignment, you can either skip that for now, or better still design the code so it will work for that case later on.)

Aside from this, don't print any other messages that Pintos as provided doesn't already print. You may find extra messages useful during debugging, but they are not for the tutors consumption!

Acknowledgements

As noted the Pintos OS was developed at the University of Stanford by:

- The Pintos core and this documentation were originally written by Ben Pfaff blp@cs.stanford.edu.
- Additional features were contributed by Anthony Romano chz@vt.edu.
- The GDB macros supplied with Pintos were written by Godmar Back gback@cs.vt.edu, and their documentation is adapted from his work.
- The original structure and form of Pintos was inspired by the Nachos instructional operating system from the University of California, Berkeley ([Christopher]).
- The Pintos projects and documentation originated with those designed for Nachos by current and former CS 140 teaching assistants at Stanford University, including at least Yu Ping, Greg Hutchins, Kelly Shaw, Paul Twohey, Sameer Qureshi, and John Rector.
- Example code for monitors (see section A.3.4 Monitors) is from classroom slides originally by Dawson Engler and updated by Mendel Rosenblum.

