Kernel Customizations for cv-Linux

- ***Ideally, I plan to customize the kernel on a PC, in these main phases***
  1. Generating a standard *initrd* -> *Done*
  2. Generating a customized *initrd* -> *Done*
  3. Building a standard Linux kernel, with minimal device drivers and modules
  4. Generating a relevant *initrd* for the standard kernel
  5. Building a kgdb batched Linux kernel, with minimal device drivers and modules
  6. Generating the relevant *initrd* for the batched kernel

- ***The assumptions***
  1. The BIOS points to the a disk partition. GRUB stage-1 is installed to the MBR of this disk. GRUB stage-1.5 and stage-2 are installed on disk according to GRUB installation program

  2. Since GRUB is file-system aware and it can be configured with entries of kernel images and associated *initrd* images, I depend on GRUB for testing the three phases of this experiment

- ***Generating a standard initrd***
  1. Using *mkinitramfs* appears to be associated with kernels starting from 2.6.12. It creates a gzipped cpio archive as the initrd image. To decompress the image, you MUST copy it with a .gz suffixed name, "gunzip" it, then "cpio -i –make-directories <" it

  2. For kernels below 2.6.12
     The tools are: the loop device and mkinitrd

  3. The *mkinitramfs* and other related tools work under the frame of *initramfs-tools*. See "*man initiramfs-tools*"

- ***Generating a customized initrd***
  1. I use initramfs-tools. According its *man* page, it employs a set of scripts to control the *mkinitramfs* process and the behavior of the resulting ram disk. My strategy, to create a customized *initrd,* is to modify these scripts and to let *mkinitramfs* does the rest

  2. At the initialization phased, GRUB loads the kernel and ram disk. Then, the kernel reaches a point with minimal system-wise initializations. E.g. the kernel can not recognize the keyboard layout – I knew this after booting with a custom *initrd*

  3. At this stage, the kernel searches the current root file system, i.e. the *initrd* file system, for an executable *init* to run. For the *initrd*, the *init* is a shell script, and its executable because "*/bin/sh"* exists is in the *initrd*

  4. According to "*man initramfs-tools*", on an running Debian, the *init* script is located in "*/usr/share/initramfs-tools/init*". Reading this file, it appears to add some folders the *initrd*. Among these folders is a */dev* folder that is populated with minimal system configurations, e.g. using a default keyboard layout and not that of the system. Then it mounts the real root file system, that passed as a the kernel "*root*" parameter in

"/boot/grub/menu.lst", to "*/root*". Finally, it starts the "*/root/sbin/init*", which is the actual Linux *init* process

5. According to many references, which I'll to add later, this process process does more sophisticated initializations. Among those is calling a "*pivot_root*" method, to switch / from initrd to the real root, which is "*/root*" at this phase. After I booted with the custom *initrd*, the "*pivot_root*" appears in "*/bin*", as part of *initrd*

6. For several times, I attempted to modify this file, *mkinitramfs* using it, adding the resulting *initrd* to */boot*, add a relevant entry to "*/boot/grub/menu.lst*", and booting. Finally, I decided to only modify the last line of the *init* script. Instead of running the *init* process, I called "*/bin/sh –login*". Linux booted successfully, an I got a shell to the customized *initrd*. The started shell has the advantage of being running without termination, unless it is explicitly exited. For the kernel, this shell is the *init*, i.e. the never terminating process. If I start any terminating process instead, or "*exit*"ed the shell, the kernel reaches that state of "*kernel panic: not syncing, attempted to kill init!*"

7. My next attempt was to disable mounting the real root from disk "*/root*". I commented out the related parts in the "*/usr/share/initramfs-tools/init*" script and the "*/usr/share/initramfs-tools/init-bottom/udev*" script to ignore the mounting the real root file system. Also, I removed the kernel root parameter from the corresponding boot entry in the "*/boot/grub/menu.lst*" file

- ***Building a standard Linux kernel, with minimal device drivers and modules***

1. Downloading the source code, decompressing it, configuring the kernel, an building it must be done with normal user rights, NOT root. Also, avoid doing any of these tasks under /usr/src, or any subdirectory of it

2. The kernel source file *Documentation/Changes* specifies the proper tools version to use with this kernel. The reference book [TO ADD] assumes using kernel 2.6.18, with the following tools versions

gcc compiler	- version 3.2 - the latest is not always recommended - gcc --version
binutils linker and object files manipulation tool, e.g. to view the contents of a library	- version 2.12 - ld -v
make	- version 3.79.1 - the latest stable is recommended - make –version
util-linux utilities, including mounting and creating partitions	- it uses the kernel and its version depends on the kernel version, must be upgraded with kernel upgrades - version 2.10 - the latest is recommended for new feature, e.g. bind mount

	- fdformat --version
module-init-tools for modules initialization	- it uses the kernel and its version depends on the kernel version, must be upgraded with kernel upgrades - version 0.9.10 - the latest is recommended for new feature, e.g. black listing modules from auto loading by udev - depmod --V
e2fsprogs for the kernel to work with ext2, 3, 4 file systems	- it uses the kernel and its version depends on the kernel version, must be upgraded with kernel upgrades - version 1.29 - the latest is recommended for new feature, e.g. working with ext3, 4 - tune2fs
udev for the kernel to managing the */dev* folder based on the contents of the */sys* folder, which change with each kernel release	- it is essential for the kernel work - version 081 - there is a latest verison that is recommended for each kernel version, because /sys changes - udevinfo -V
procps for process managing and monitoring	- it is essential for the kernel work - version 3.2.0 - ps –version
pcmciautils for using PCMCIA devices	- version 004 - the latest is recommended for new features, e.g. automatic driver loading when using new deices are found - pccardctl -V

3. I tired "*make defconfig*", it fails with complains around compilation, perhaps gcc version issues. After searching the web, it appears that I need to build gcc 3.2. Looking at GCC build requirements, my plan is below
   - I installed the etch GCC, as the ISO C90 compiler. I already have bash, gawk, binutils, gzip, make, and tar. For GMP, MPFR, and MPC, I'll add their sources as subdirectories to gcc code. They will be built with gcc. For PPL, GCJ, and ADA stuff, I will configure to build without them. They appear to be out of scope features. The required disabled configurations are: --without-ppl –without-cloog --disable-libgcj --enable-languages=c, c++ --disable-libada
   - While compiling GCC 2.3 using the Debian's GCC 2.4.1, I got complain messages that the used GCC is not a working compiler. I tested the compiler by compiling a simple "hello world" C program. It did not work because of missing header files. Finally, after searching, I realized that I had to install the Debian package libc6-dev to get the libc header files

- At this point I recalled that the original problem when I tried to use the kernel build system was associated with the used compiler. My first guess was because of using a newer GCC that the recommended in Documentation/Chages, as noted by [Add the ref]. But dsdsds

4. “make menuconfig” failed with some error message with the word “curses”. I fixed it by installing the Debian package libncurses5-dev

- ***After building the kernel, I wanted to prepare a proper terminal***

1. Gnome terminal emulator is an xterm, i.e. it is X based. X has its own way of initializing its keymap and other related stuff

2. What I am about is the Linux console and keyboard, i.e. /dev/console or /dev/tty

3. From the “Linux keyboard and console how-to”. Under console, there are two parts:
   - Linux kernel keyboard driver:
     - When a key is pressed, a scancode is generated. If the the driver is in scancode mode, it just forwards this code to the terminal driver. i.e. the application from the driver perspective
     - If the driver is in keycode mode, it maps the scancode to a keycode and just forwards the result to the terminal driver
     - It the driver is in unicode mode, it uses a key map to map the keycode to a char or string, then it forwards the result to the terminal driver
   - Terminal driver:
     - The current loaded terminal drivers by “less /proc/tty/drivers”
     - Each one of this has its loaded font map
     - The terminal driver maps the received char or string, from the kernel keyboard driver, to the proper font. The it draws it to the proper position on the screen

4. Related commands and tools
   - The kbd utilities are used to configure the keymap of the kernel keyboard driver and the fonts of the terminal driver
   - Broune shell programming defines the syntax for sh commands and sh scripts
   - The “tty” command show to which console the stdin is currently connected
   - By defualt, stdin is the keyboard and stdout is the screen
   - In sh, 1 is stdout, and 2 is stderr, 2<&1 redirects stderr to stdout
   - By default, the kernel boots to /dev/console. To switch to a more advanced terminal like /dev/tty1, we need to redirect stdin and stdout to /dev/tty1

5. What I did:
   - Adding kbd utilities to the ram disk
     - I got the latest available kbd source from http://www.kernel.org/pub/linux/utils/kbd/
     - To build it, I had to install Debian packages of bison and fles
     - To install kbd into a special folder, and not to my machine, I ran:
       - “./configure –prefix=/home/aergawy/kbd-inst”

- I made an initramfs-tools hook script to add the required kbd bin and lib files to the ram disk. I started the hook script from an already existing one in /usr/share/initramfs-tools/hooks and I put it in /etc/initramfs-tools/hooks
  When first running mkinitramfs, I got errors like “array_<nameofmyhook>: command not found”. I fixed it by removing '-' from the name of my hook file
- My hook script is /etc/initramfs-tools/hooks/kbdconsoleutils
  - # adding kbd
  - cp -r /home/aergawy/kbd-inst/bin/* "${DESTDIR}/bin/"
  - cp -r /home/aergawy/fi-tty-mappings/lib/* "${DESTDIR}/lib/"

  - # adding necessary binaries
  - copy_exec /usr/bin/less /bin/
  - copy_exec /usr/bin/setsid /bin/

  - # adding the default terminfo file, linux
  - mkdir -p "${DESTDIR}/lib/terminfo/l"
  - cp /lib/terminfo/l/linux "${DESTDIR}/lib/terminfo/l/"
- In the hook
  - The file linux in the terminfo database is the default that is set by the kernel. “ncurses” dependent applications, e.g. “less”, use terminfo files to know the special signals to send to terminal drivers to do special actions. e.g. scrolling. The kernel defines the name of the current terminfo file using the environment variable TERM, which is read by “ncurses” dependent applications. If the TERM referred file is not in place, i.e. /lib/terminfo, the applications complain about “Not fully functional terminal”. I took only the “linux” file from my current machine terminfo
  - I copied the whole set of kbd bin
  - I copied only a sub-set of the kbd lib, which I put in /home/aergawy/fi-tty-mappings/lib/kbd. This set includes fi-latin1.map, lat1-16.psfu, linux-keys-bare, linux-with-alt-and-altgr, qwerty-layout. I choose the first two files and then decided the rest based on trial and error, when they are reported as missing “include” files at boot time

① Setting the keymap and console font
- I modified the *init* script as follows
  - echo "Setting tty1 mode to UTF-8 ..."
  - kbd_mode -u -C /dev/tty1
  - echo "Loading fi key map ..."
  - loadkeys /lib/kbd/fi-latin1.map
  - echo "Setting font to tty1..."
  - setfont /lib/kbd/lat1-16.psfu -C /dev/tty1

① Getting process-control capable terminal
- In the the init script, from the BusyBox FAQ, I replaced exec sh –login, with
  - setsid sh -c 'exec sh </dev/tty1 >/dev/tty1 2>&1'

- ***To develop my applications utilizing new features in a customized kernel, I had to enable loading the root file system on disk***
  - At some point, the default *init* script searched for a root file system on the detected disks. It attempts to pivot from the ram disk to the hard disk, as specified in the *root* entry of the kernel command line, in the file /boot/grub/menu.lst, e.g. root=/dev/hda3
  - After the status I reached from the above work, when I booted with root-file system pivoting enabled, it failed with: “Waiting for root file system”. After a while, it reports: “Alert! /dev/hda3 doesn't exist. Dropping to shell!”
  - I suspect not detecting my hard disk, so I had to compare /dev on a running Debian to /dev that is created by the customized kernel. I discovered that *hda**x*** files are missing
  - Not targeting this to be part of the final customization. I started a trial-and-error overshoot solution. I need to get the bus where the disk is connected probed by the kernel.
  - Using kernel 2.6.26, I had to enable most of the SCSI and ATA support:
    - Generic/default IDE chipset support
    - PCI IDE chipset support
    - Generic PCI IDE chipset support
    - Leave all the configuatios under PCMCIA/CardBus support as default
  - You can see if ATA is detected via dmesg to show the kernel boot log. Also /sys/devices -> pci, isa, acpi.
  - ***TODO*** I need to check for more hardware and software tools to manually probe hardware, to discover the system
  - To disable starting X under such untested system, disable *gdm:*
    - *update-rc.d -f gdm remove*
    - *Generally, update-rc.d - <install/remove> System-V style init script links, e.g. update-rc.d -f gdm remove*
  - As an interesting experiment, I modified the default *init* script so that it mounts the sub-folder /root on the ram disk with the root file system on the disk. I did this by just disabling the last line in the file
  - All works fine, we could mount an pivot to the disk

- ***Targeting no GUI support by the final customization, I had to enable the frame-buffer support, i.e. “VGA Text Console”***
  - In the kernel configurations: Device drivers / Graphics support / Console display driver support
  - Enable the built-in “boot-up logo”. Seeing it at the boot-up is a form of testing
  - I enabled several “frame buffer drivers”, e.g. ATIMach64 and VESA2.0. Later on, I realized that VESA2.0 is a generic driver that works with almost all graphics hardware that runs on an Intel-compatible device
  - I enabled the option “support for frame buffer devices”, i.e. fb***x***
  - I enabled “video mode selection support”. It used to specify video modes in /boot/grub/menu.lst, i.e. the kernel boot command line. The parameter is vga=*<required mode>*
  - ***Note***: to query which video modes are available by the system, use vga=ask. While booting, this lists all available modes on the system to choose from
  - I tested the resulting *fbdev* using mplayer. I had to enable sound support in kernel