

Praktikum angewandte Systemsoftwaretechnik

Aufgabe 1

Alexander Würstlein, Moritz Strübe, Rainer Müller

Lehrstuhl Informatik 4

Oktober 2013

Rechnerzugang

- Die Rechnerarbeit findet in der *Manlobbi* (Raum: **0.058-113**) statt.
- Es sind spezielle Zugänge erforderlich
- Zusätzliche Software kann auf Anfrage problemfrei nachinstalliert werden.
- Es wird nur `/home` und `/proj` gesichert.

Rechnerumgebung

- Rechner: faui49man[1-12]
- Logins: nachher, dann 1x ssh faui49 (erstellt /home)
- /home mit Backup aber begrenzte Grösse und NFS
- lokal für grosse Dinge und wenns schnell gehen soll:
/srv/scratch/\$USER
- z.B. Kernelbauen...
- notfalls auch per NFS über /net/faui49man11/srv/scratch/...

Virtualisierungswerkzeug: KVM

- In PASST entwickeln und arbeiten wir mit Virtualisierungstechniken.
- Wir empfehlen qemu/KVM, Bearbeitung ist aber auch z.B. Virtualbox oder VMware möglich.
- Für KVM (Hardwaresupport) werden Schreibrechte auf `/dev/kvm` benötigt:

Zugriffsrechte bzw. ACLs auf `/dev/kvm` prüfen

```
>> ls -la /dev/kvm
```

```
crw-rw-rw-+ 1 root kvm 10, 332 2012-04-29 21:43 /dev/kvm
```

Virtuelle Festplatte vorbereiten

Virtuelle Festplatte anlegen:

```
>> dd if=/dev/zero of=p_passt.img \  
      bs=1 count=1 seek=8G  
>> du -sh p_passt.img  
4,0K p_passt.img
```

- Erstellt eine *virtuelle* Festplatte in der Datei `p_passt.img`
- Nicht allozierter Platz wird auch tatsächlich nicht belegt (*sparse file*)
- Mit `qemu-img(8)` können auch Abbilder in besseren Formaten (z.B. `qcow2`) angelegt werden.

Hilfscript beim Umgang mit KVM

```
boot.sh
```

```
#!/bin/sh
```

```
kvm -m 1024 -nodefaults -nographic \  
-echr 0x01 -serial mon:stdio \  
-serial tcp:localhost:'id -u',server,nowait,nodelay \  
-net nic,model=virtio -net user \  
-drive file=p_passt.img,if=virtio,cache=writeback \  
"$@"
```

- Häufig benutzte Optionen werden mit diesem Skript gekapselt; weitere Optionen können angehängt werden
- Mittels `-nographic` wird die Graphikkarte nicht simuliert, d.h. Interaktion mit System ist nur via serieller Konsole (unter Linux `/dev/ttyS0`) möglich!
- Es wird ein Netzwerk per NAT zur Verfügung gestellt. ICMP (also z.B. ping) funktioniert nicht.

Escape Key

Im KVM/qemu ist der Escape-key auf C-a aktiviert. Folgende Kommandos sind verfuegbar:

Qemu Escape Commands

```
C-a h    print this help
C-a x    exit emulator
C-a s    save disk data back to file (if -snapshot)
C-a t    toggle console timestamps
C-a b    send break (magic sysrq)
C-a c    switch between console and monitor
C-a C-a  sends C-a
```

Ein *emergency sync* (SysRq-s) kann damit so ausgelöst werden: C-a b s.

Minimales Installationsprogramm laden

Download Debian Installer (netinst)

```
host="http://ftp.fau.de/\ndebian/dists/wheezy/main/installer-amd64/\ncurrent/images/netboot/debian-installer/amd64/"\nwget $host/linux\nwget $host/initrd.gz
```

- Minimaler Debian Installer (nur wenige MiB gross)
- Besteht nur aus Kernel und Ramdisk mit kleinem Installationsprogramm
- Alles Weitere wird vom Debian Spiegel nachgeladen.
- Für die eigentliche Installation den Spiegel `ftp.fau.de` benutzen!

Installation im Textmodus

Starten mittels Hilfsscript

```
/proj/i4passt/boot.sh -kernel linux -initrd initrd.gz \  
    -append "console=ttyS0 priority=low"  
Loading Linux 2.6.32.38 ...  
Loading initial ramdisk ...  
[    0.000000] Initializing cgroup subsys cpuset  
[    0.000000] Initializing cgroup subsys cpu  
[    0.000000] Linux version 2.6.32.38 (root@fau48d)
```

Quellen vorbereiten

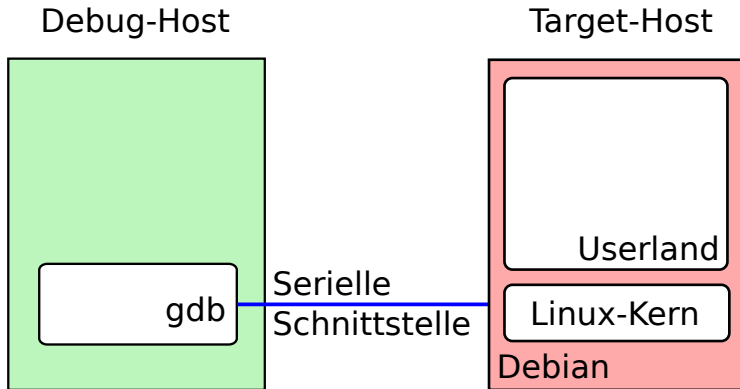
- Im Labornetz existiert ein Spiegel des Linux-Kernels:

Clonen der Linux Quellen

```
>> mkdir /srv/scratch/$USER/  
>> cd /srv/scratch/$USER/  
>> git clone /proj/i4passt/kernel/linux-stable.git  
git clone /proj/i4passt/kernel/linux-stable.git  
Cloning into linux-stable...  
done.
```

- Konfiguration von Linux mittels `make menuconfig` oder `make xconfig`
- Bauen mittels `make -j 4`

Logisches Rechnersetup



Wichtige Debug-Kernel-Optionen

- `CONFIG_DEBUG_INFO`
 - Übersetzt den Kernel mit Debuginformationen.
- `CONFIG_FRAME_POINTER`
 - Unterbindet das Wegoptimieren des Framepointers.
- `CONFIG_DEBUG_RODATA`
 - Erlaubt das Schreiben in das Text-Segment. Dies wird für Softwarebreakpoints benötigt.
- Module gegebenenfalls statisch binden.
 - Erspart das Erstellen der Initramfs (erleichtert möglicherweise das Laden des Kernels durch Qemu)
 - Module müssen nicht manuell in GDB geladen werden.
 - Module in der Kconfig ausschalten → alle Module sind statisch.

Booten des Kerns mittels Qemu

- Qemu implementiert eigenen Bootloader, so dass der Bootloader nicht unbedingt nötig ist.
- Über Kommandozeile werden die Bootparameter übergeben:
 - `-kernel` Pfad zu bzImage
 - `-append` Kernelparameter
 - `-initrd` Bei Bedarf: Pfad zur Initramfs
- Nützliche Kerneloptionen:
 - `kgdboc=ttyS1,115200`: KGDB konfigurieren
 - `kgdbwait`: Beim Booten auf eine GDB-Verbindung warten

Alternative: Installation des Kernels in die VM

- Auf dem Buildhost: `fakeroot make deb-pkg V=1 -j4`
- Entstehende `.deb` dateien per `scp` in die virtuelle Maschine kopieren, und mit `dpkg -i *.deb` installieren
- Geeignete Kernel-Boot-Optionen setzen!

Grub-Optionen in `/etc/default/grub`

```
GRUB_DEFAULT=0
GRUB_TIMEOUT=5
GRUB_CMDLINE_LINUX_DEFAULT="verbose"
GRUB_CMDLINE_LINUX="console=ttyS0 kgdboc=ttyS1,115200"
GRUB_TERMINAL=serial
GRUB_SERIAL_COMMAND="serial --unit=0 --speed=115200 --stop=1"
```

- Aktivieren der Änderungen: `update-grub`
- Neustarten: `reboot`

Debuggen mit dem GDB

- Programm muss mit Debugsymbolen (`-g`) übersetzt werden. In Linux gibt es hierfür eine Konfigurations-Option.
- Normalerweise (wie z.B. in Systemprogrammierung) werden *lokale* Anwendungen untersucht.
- In PASST: *remote debugging*.
- Unterbrechen funktioniert nicht via `kgdb`.

Der laufende Linux-Kernel kann so unterbrochen werden:

```
echo g >/proc/sysrq-trigger
```

Debuggen mit dem GDB

Aufruf und Verbindung zum entfernten Kern:

```
>> gdb vmlinux
```

```
[...]
```

```
Reading symbols from /build/morty/linux-2.6.38/vmlinux  
...done.
```

```
(gdb) target remote localhost:4444
```

```
Remote debugging using localhost:4444
```

```
kgdb_breakpoint (new_dbg_io_ops=<value optimized out>) at  
/build/morty/linux-2.6.38/kernel/debug/debug_core.c:960  
960 wmb(); /* Sync point after breakpoint */
```

```
(gdb)
```


Breakpoints

- Breakpoints:
 - `b [<Dateiname>:]<Funktionsname>`
 - `b <Dateiname>:<Zeilennummer>`
 - `b *<Adresse>`

Breakpoint im Systemcall `open`

```
b sys_open
```

- Fortfahren der Ausführung mit `c` (continue)
- Schrittweise Abarbeitung auf Ebene der Quellsprache mit
 - `s` (step: läuft in Funktionen hinein)
 - `n` (next: behandelt Funktionsaufrufe als einzelne Anweisung)
 - `finish` läuft bis zum Ende der aktuellen Funktion (nützlich wenn man versehentlich `s` statt `n` verwendet hat)
- Breakpoints anzeigen: `info breakpoints`
- Breakpoint löschen: `delete breakpoint`

Variablen, Stack

- Anzeigen von Variablen mit: `p expr`
 - `expr` ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
- Automatische Anzeige von Variablen bei jedem Programmstopp (Breakpoint, Step, ...): `display expr`
- Setzen von Variablenwerten mit `set <variablenname>=<wert>`
- Ausgabe des Funktionsaufruf-Stacks (backtrace): `bt`

Watchpoints

- Stoppt Ausführung bei Zugriff auf eine bestimmte Variable
- `watch expr`: Stoppt, wenn sich der Wert des C-Ausdrucks `expr` ändert
- `rwatch expr`: Stoppt, wenn `expr` gelesen wird
- `awatch expr`: Stopp bei jedem Zugriff (kombiniert `watch` und `rwatch`)
- Anzeigen und Löschen analog zu den Breakpoints