AKBP II
A Programmer’s Introduction to the JX System

Michael Golm

University of Erlangen
Department of Computer Science
(Operating Systems and Distributed Systems)
Martensstrasse 1
91058 Erlangen, Germany
golm@cs.fau.de

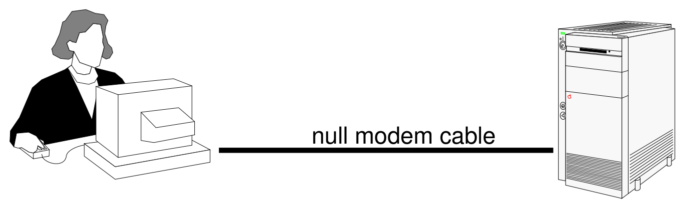
CVS

- Initial
  - tcsh: setenv CVSROOT /proj/akbp/repository
  - cvs co jx
- During the development cycle
  - cvs (-n) update -d -P -R
  - cvs commit

JX Directory Tree

```
<jx>
 ├── Makefile
 ├── boot.rc
 ├── jcore
 │    ├── Makefile
 │    └── *.[chS] Dateien des µkernels
 ├── libs
 │    ├── Makefile
 │    ├── COMPONENTS, ALLCOMPONENTS
 │    ├── <component XYZ>
 │    │    ├── META
 │    │    └── java & class Dateien
 │    └── <component ...>
 ├── emulation
 └── edomains
```


Hardware Configuration



Boot Configuration

- Edit configuration file **`boot.rc`**
  - Select configuration
  - Specify configurations:
    domain name, initial component, initial class, size of heap, args
- Example:

```
# Select initial component
[My SysConf]

# Konfigurationen
[My SysConf]
"Bar", "test.jll", "jx/test/Test", 8000000, "blubber"
"Foo", "murks.jll", "foo/bar/Main", 5000000, "bla", "blubb"

[Other Configuration]
"FS", "fs.jll", "fs/FileSystem", 8000000
"DB", "db.jll", "db/DataBase", 5000000
```


Boot Procedure

- Build μkernel **`jxcore`** and Java part **`code.zip`**:
  - Setup build environment: **`source /proj/akbp/setupenv.tcsh`**
  - list needed components in the COMPONENTS file

```
COMPONENTS = init
COMPONENTS += test
COMPONENTS += murks
```

  - invoke **`make`** in the top-level directory and wait
- Boot using GRUB

Monitor

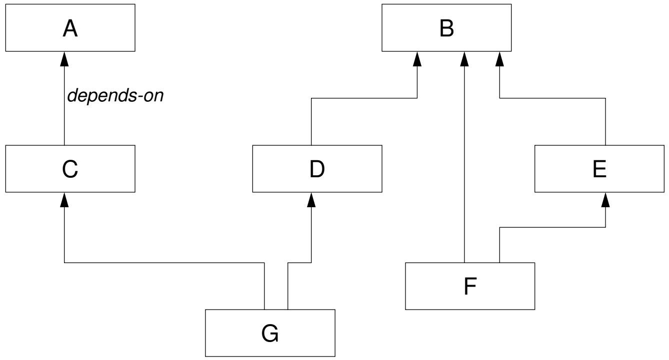
- Used to explore a running system
- System is halted when monitor is activated (interrupts are disabled)
- On the development workstation: start the terminal program
  - screen
    - autodetach off
    - screen -t jx 9 /dev/ttyS0 38400
  - debug/serial
  - kermit
  - ...
- Any key activates the monitor

Monitor: Commands

- help: display help
- domains: Show list of all domains
- domain <domainid>: display information about a specific domain
- threads: show all threads
- heap <domainid>: show heap usage of a specific domain
- runq: display runqueue
- ns: display names of kernel name server
- c: continue normal system operation
- q: quit system and reboot
- ...

Components

- components depend on other components
- dependency relation creates a non-cyclic dependency graph



Components

- Each component has its own directory in **`libs`**
- **`libs/`*`<componentname>`*`/META`**-File
  - NEEDLIBS = *Lists all required components*
  - SUBDIRS = *Lists all sub-directories*
  - LIBNAME = *Name of this component*
  - NUMBERENV = int
- Sub-directories = package name; contain Java source files
- Add component to **`libs/ALLCOMPONENTS`**-File and **`libs/COMPONENTS`**-File, for example **`COMPONENTS += my_com`**
- Create sub-directories and invoke **`make makefiles`** in the top-level dir!

Entry point into new domain

```
package xxx;

import jx.zero.*;

public class YYY {
    public static void init(Naming naming, String[] args) {
    }
}
```

- **`jx.zero`** contains interfaces to access DomainZero (µKernel)
- **`jx.zero.Naming`** is the name server portal
  - **`Portal lookup(String name)`**
  - **`void registerPortal(Portal portal, String name)`**

Initialize domain

- New domain needs to initialize **`Debug.out`**

```
import jx.zero.debug.*;
...

public static void init(Naming naming, String[] args) {
    DebugChannel dbg = (DebugChannel) naming.lookup("DebugChannel0");
    Debug.out = new DebugPrintStream(new DebugOutputStream(dbg));

    ...
}
```

Build a component

- invoke **`build`** in any sub-directory
  - compiles Java source into class files
  - creates a zip file of all class files (libs/*componentname*.zip)
  - translates the class files into machine code (x86) (libs/*componentname*.jll)
  - adds the jll file to **`code.zip`**

Portals

- Mechanism similar to Java-RMI
- 1. Portal interface

```
public interface MyService extends Portal {
    void myMethod();
}
```

- 2. Service implementation

```
public class MyServiceImpl implements MyService, Service {
    public void myMethod() {
        ...
    }
}
```


Naming

- Portals can be registered at a name service
- Every domain has at least one **`Naming`** portal

```
MyService svc = new MyServiceImpl();
naming.registerPortal(svc, "FooService");
```

- Another (or the same) domain can lookup the portal using the name service

```
MyService svc = (MyService) naming.lookup("Foo");
```

- Wait until portal appears at the name service

```
MyService svc = (MyService) LookupHelper.waitUntilPortalAvailable(naming, "Foo");
```


Naming

- The initial naming portal of a domain can also be obtained using:

```
Naming naming = InitialNaming.getInitialNaming();
```

Threads

- Threads can be created as usual in Java (Subclass of **`java.lang.Thread`** or implement **`java.lang.Runnable`**); requires component **`jdk1`**
- DomainZero (component zero) contains interface for thread creation

```
CPUState thread = cpuManager.createCPUState(new ThreadEntry() {
    public void run() {
        for(;;) {
            Debug.out.println(""+mem.get8());
            cpuManager.yield();
        }
    }});
cpuManager.start(thread);
```


Memory: allocation

- Management of large memory regions and sharing of data between domains is done using **`Memory`** objects
- Get **`Memory`** object from **`MemoryManager`**

```
MemoryManager memoryManager = (MemoryManager) naming.lookup("MemoryManager");
Memory mem = memoryManager.alloc(4096);
```


Memory: access

- Access using getter/setter methods

```
byte b = mem.get8(pos);
int i = mem.get32(pos);
mem.set8(pos, value);
```

- get32/set32 use little-endian encoding
- get32/set32 uses 32 bit index (**`get32(2)`** reads bytes 8..11 as little endian)
- getBigEndian32/setBigEndian32 use big-endian encoding

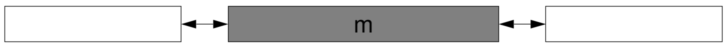
Memory: revocation

- Revoke access to memory using **`revoke()`**. In the following example mem2 is the only remaining reference to the memory.

```
Memory mem2 = mem.revoke();
```

Memory: splitting

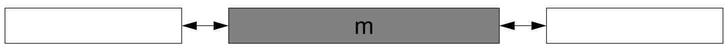
- Create subranges by splitting (split2()):



after split(m):

- Join splitted memory (joinPrevious(), joinNext())

after m = joinPrevious(m1) or m = joinNext(m0):



Memory: splitting

- Example:

```
Memory mem = memoryManager.alloc(100+s);
Memory[] arr = new Memory[2];
mem.split2(100, arr);

// arr[0] contains reference to first half (size == 100)
// arr[1] contains reference to second half (size == s)
```


ApplicationStarter

- Start normal Java applications using the **`main()`** method
- Example:

```
# Select initial component
[My SysConf]

# Konfigurationen
[My SysConf]
"Bar", "legacy_starter.jll", "jx/start/ApplicationStarter", 8000000, "jx/test/
Test", "test.jll", “blubber”
"Foo", "legacy_starter.jll", "jx/start/ApplicationStarter", 5000000, "murks.jll",
"foo/bar/Main", “bla”, “blubb”
```


MonolithicApplicationStarter

- Run all applications in one domain
- End argument list by **`null`**
- Example:

```
# Select initial component
[My SysConf]

# Konfigurationen
[My SysConf]
"FooBar", "legacy_starter.jll", "jx/start/MonolithicApplicationStarter", 8000000,
"jx/test/Test", "test.jll", “blubber”, null, "murks.jll", "foo/bar/Main", “bla”,
“blubb”, null
```

Visualization Tools: Thread Activity

- steps to create a thread activity diagram:
  - enable PROFILE_EVENT_THREADSWITCH in settings.makefile
  - compile & boot
  - use the debug/serial tool to drive the serial line
  - activate the monitor and type the command “rswitchesonly”
  - this starts a binary transmission of the thread switch log
- use the **`ta`** script to create a **`diagram.mif`** file from the binary log file **`rswitchesonly`**
  - **`ta switchesb rswitchesonly diagram.mif - - 1`**

Synchronization

- Java has a built-in recursive mutual exclusion lock (synchronized methods and synchronized blocks)
- Problems:
  - Is either slow or causes memory overhead
  - Can not be used in interrupt handlers (they are not allowed to block)
  - Is prone to deadlock

JX synchronization mechanisms

- jx.zero.AtomicVariable

```
public interface AtomicVariable extends Portal {
    void set(Object value);
    Object get();
    void atomicUpdateUnblock(Object value, CPUState state);
    void blockIfEqual(Object testValue);
    void blockIfNotEqual(Object testValue);
}
```

- Non-preemptive scheduling
- jx.zero.CAS

```
public interface CAS extends Portal {
    boolean casObject(Object obj, Object compareValue, Object setValue);
    boolean casInt(Object obj, int compareValue, int setValue);
    boolean casBoolean(Object obj, boolean compareValue, boolean setValue);
    boolean casShort(Object obj, short compareValue, short setValue);
    boolean casByte(Object obj, byte compareValue, byte setValue);
}
```

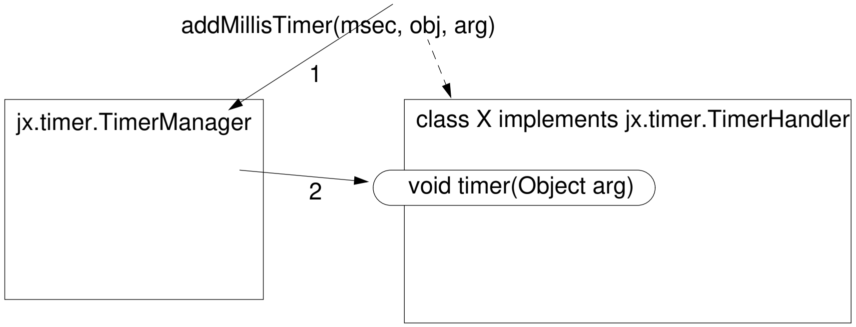

Timer

- TimerManager:

```
public interface TimerManager extends Portal {

    Timer addMillisTimer(int expiresFromNowInMillis, TimerHandler
handler, Object argument);

    Timer addMillisIntervalTimer(int expiresFromNowInMillis, int
intervalInMillis, TimerHandler handler, Object argument);

    Timer addTimer(int expiresFromNow, int interval, TimerHandler
handler, Object argument);

    boolean deleteTimer(TimerHandler t);

    int getTimeInMillis();
    int getTimeBaseInMicros();
    int getCurrentTime();
}
```

Timer

- components: **`timer`** (interfaces) and **`timer_pc`** (implementation)



Network

- Unix (Linux):
  - List all network interfaces: **`/sbin/ifconfig -a`**
  - Dump network packets: **`/usr/sbin/tcpdump -i eth0`**
    - **`-e`**: print linke-level header (ethernet)
    - **`-v`** & **`-vv`**: verbose and more verbose
    - **`-x`**: print packet in hex
    - use filter expressions
    - **`tcpdump -x -v -i eth0 ether src or dst 8:67:42:24:00:07`**
    - **`tcpdump -vv -R -e -x -p -nn -N -i eth0`**

Network

- **"PCIDomain", "pci_pc.jll", "jx/devices/pci/PCIGod", "jx/scheduler/HLRRobin", 300000**
- **"NicNetDomain", "legacy_starter.jll", "jx/start/MonolithicApplicationStarter", 20000000, "jx/net/StartNetDevice", "net_devices.jll", "NIC", "eth0", null, "jx/net/protocols/StartNetworkProtocols", "net_protocols.jll", "NIC", "NET", null**
- Example use:
  name: name of the network portal (“NET”)
  srcPort: local port number
  dstPort: remote port number

```
NetInit net = (NetInit)LookupHelper.waitUntilPortalAvailable(naming, name);

UDPSender u = net.getUDPSender(srcPort, new IPAddress("192.168.34.2"), dstPort);
Memory buf = net.getUDPBuffer(50);

u.send(buf);
```

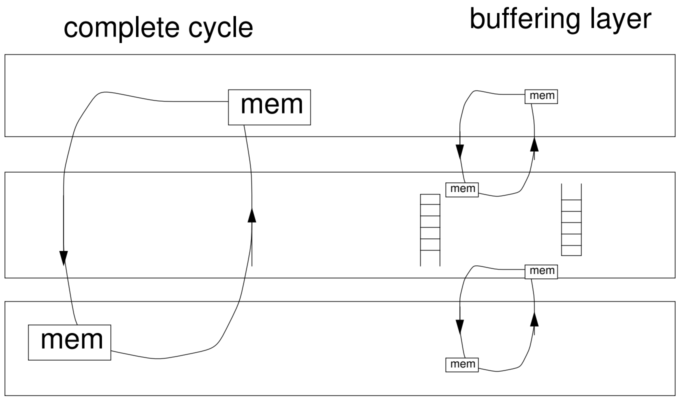

Network: IP

- Send IP packets:
  name: name of the network portal (“NET”)

```
NetInit net = (NetInit)LookupHelper.waitUntilPortalAvailable(naming, name);

UDPSender u = net.getIPSender(new IPAddress("192.168.34.2"));
Memory buf = net.getIPBuffer(50);

u.send(buf);
```

Buffer Management

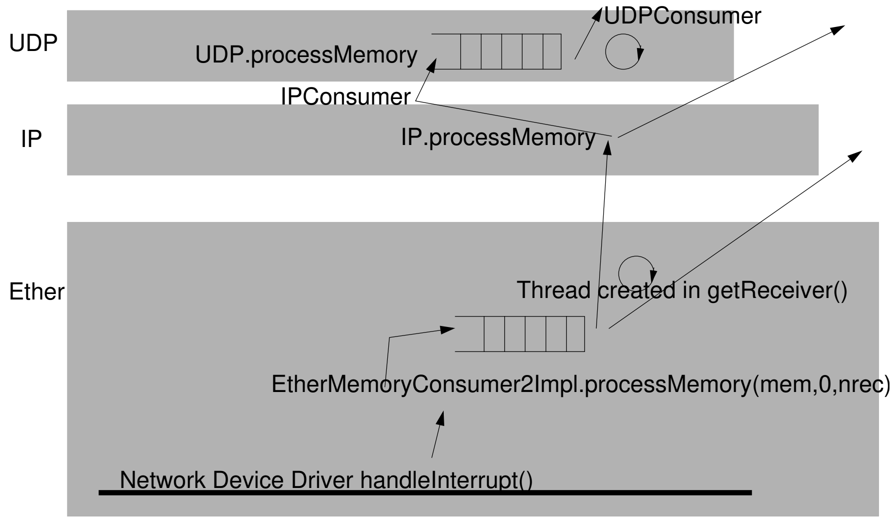
- Buffers are cycled through the stack
- Each layer has an initial pool of buffers that are used to receive packets



Network

- Modern NICs have on-device buffer. They detect network collisions and automatically resend the packet.
- Packets can get lost within the network.
- Packets can be received in arbitrary order.

Network Stack: Receive



Network Stack: Transmit