Copyright Notice

• These slides are distributed under the Creative Commons Attribution 3.0 License

• You are free:
  • to share—to copy, distribute and transmit the work
  • to remix—to adapt the work

• under the following conditions:
  • Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows: “Courtesy of Mark P. Jones, Portland State University”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
Introducing “pork”

• pork = the “Portland Oregon Research Kernel”
• An implementation of (a subset of) L4 X.2
• Similar API to Pistachio, but specific to IA32 platform
• Written around the start of 2007
• “I have almost all the pieces that I need to build an L4 kernel … perhaps I should try putting them together?”
• Built using the techniques we have seen so far in this course …

Performance Benchmarking: Pingpong, Pistachio, and Pork
The pingpong benchmark

• A small L4 benchmark from the Karlsruhe Pistachio distribution, written in C++

• A single ipc call transfers contents of n message registers (MRs) between threads

• create two threads, “ping” & “pong”:
  for n = 0, 4, 8, …, 60:
    for 128K times:
      send n MRs from “ping” to “pong”
      send n MRs from “pong” to “ping”
      measure cycles & time per ipc call

• Cycles measured using rdtsc, time measured using interrupts

Expected Performance Model

\[ t = A + Bn \]

where A = system call overhead
B = cost per word
Test Platform

- Dell Mini 9 netbook (1.6GHz Atom N270 CPU)
- Booting via grub from a flashdrive

Pistachio “Output”
Pork “Output”

The Portland EA Kernel (pork), February 2007

<table>
<thead>
<tr>
<th>ping pong</th>
<th>pistachio Inter-AS IPC</th>
<th>pork Inter-AS IPC</th>
<th>Ratio, pork/pistachio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cycles</td>
<td>microseonds</td>
<td>cycles</td>
</tr>
<tr>
<td>0</td>
<td>1340.67</td>
<td>0.77</td>
<td>1519.59</td>
</tr>
<tr>
<td>4</td>
<td>1595.58</td>
<td>0.81</td>
<td>1530.14</td>
</tr>
<tr>
<td>8</td>
<td>1301.64</td>
<td>0.81</td>
<td>1556.71</td>
</tr>
<tr>
<td>12</td>
<td>1306.29</td>
<td>0.81</td>
<td>1579.67</td>
</tr>
<tr>
<td>16</td>
<td>1317.96</td>
<td>0.82</td>
<td>1607.34</td>
</tr>
<tr>
<td>20</td>
<td>1335.16</td>
<td>0.83</td>
<td>1634.98</td>
</tr>
<tr>
<td>24</td>
<td>1333.26</td>
<td>0.83</td>
<td>1664.64</td>
</tr>
<tr>
<td>28</td>
<td>1342.28</td>
<td>0.84</td>
<td>1687.47</td>
</tr>
<tr>
<td>32</td>
<td>1350.34</td>
<td>0.84</td>
<td>1702.89</td>
</tr>
<tr>
<td>36</td>
<td>1358.46</td>
<td>0.85</td>
<td>1721.46</td>
</tr>
<tr>
<td>40</td>
<td>1362.08</td>
<td>0.85</td>
<td>1745.56</td>
</tr>
<tr>
<td>44</td>
<td>1374.64</td>
<td>0.86</td>
<td>1787.86</td>
</tr>
<tr>
<td>48</td>
<td>1382.88</td>
<td>0.86</td>
<td>1804.48</td>
</tr>
<tr>
<td>52</td>
<td>1390.88</td>
<td>0.87</td>
<td>1818.78</td>
</tr>
<tr>
<td>56</td>
<td>1398.02</td>
<td>0.87</td>
<td>1842.79</td>
</tr>
<tr>
<td>60</td>
<td>1406.13</td>
<td>0.88</td>
<td>1875.66</td>
</tr>
</tbody>
</table>

Transcribed Data (Inter-AS)

Inter-AS = “ping” and “pong” in different address spaces
Cycles (Inter-AS)

\[
\text{pistachio} = 1274.66 + 2.27n \quad \text{(least squares)}
\]

\[
\text{pork} = 1512.57 + 6n
\]

Microseconds (Inter-AS)
Pork : Pistachio  (Inter-AS)

Transcribed Data (Intra-AS)

Intra-AS = “ping” and “pong” in same address space
Cycles (Intra-AS)

\[
pistachio = 756.54 + 2.21n \quad \text{(least squares)}
\]

\[
pork = 1073.54 + 6.11n
\]

Microseconds (Intra-AS)
Pork : Pistachio  (Intra-AS)

Estimating Clock Frequency

<table>
<thead>
<tr>
<th>cycles/microsecond</th>
<th>pistachio</th>
<th>pork</th>
</tr>
</thead>
<tbody>
<tr>
<td>1611.26</td>
<td>1599.57</td>
<td></td>
</tr>
<tr>
<td>1597.01</td>
<td>1610.67</td>
<td></td>
</tr>
<tr>
<td>1606.96</td>
<td>1572.43</td>
<td></td>
</tr>
<tr>
<td>1612.70</td>
<td>1595.63</td>
<td></td>
</tr>
<tr>
<td>1607.27</td>
<td>1575.82</td>
<td></td>
</tr>
<tr>
<td>1596.58</td>
<td>1602.92</td>
<td></td>
</tr>
<tr>
<td>1606.34</td>
<td>1632.00</td>
<td></td>
</tr>
<tr>
<td>1597.95</td>
<td>1654.38</td>
<td></td>
</tr>
<tr>
<td>1607.55</td>
<td>1606.50</td>
<td></td>
</tr>
<tr>
<td>1598.19</td>
<td>1624.02</td>
<td></td>
</tr>
<tr>
<td>1602.45</td>
<td>1586.87</td>
<td></td>
</tr>
<tr>
<td>1598.42</td>
<td>1568.30</td>
<td></td>
</tr>
<tr>
<td>1607.91</td>
<td>1582.81</td>
<td></td>
</tr>
<tr>
<td>1598.71</td>
<td>1595.42</td>
<td></td>
</tr>
<tr>
<td>1606.92</td>
<td>1616.48</td>
<td></td>
</tr>
<tr>
<td>1597.88</td>
<td>1589.54</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycles/microsecond</th>
<th>pistachio</th>
<th>pork</th>
</tr>
</thead>
<tbody>
<tr>
<td>1620.42</td>
<td>1586.34</td>
<td></td>
</tr>
<tr>
<td>1614.04</td>
<td>1614.56</td>
<td></td>
</tr>
<tr>
<td>1621.85</td>
<td>1549.38</td>
<td></td>
</tr>
<tr>
<td>1612.33</td>
<td>1588.88</td>
<td></td>
</tr>
<tr>
<td>1623.78</td>
<td>1627.76</td>
<td></td>
</tr>
<tr>
<td>1612.24</td>
<td>1570.04</td>
<td></td>
</tr>
<tr>
<td>1623.70</td>
<td>1604.93</td>
<td></td>
</tr>
<tr>
<td>1612.82</td>
<td>1641.04</td>
<td></td>
</tr>
<tr>
<td>1621.96</td>
<td>1588.99</td>
<td></td>
</tr>
<tr>
<td>1612.87</td>
<td>1619.00</td>
<td></td>
</tr>
<tr>
<td>1621.87</td>
<td>1589.63</td>
<td></td>
</tr>
<tr>
<td>1614.89</td>
<td>1618.59</td>
<td></td>
</tr>
<tr>
<td>1621.83</td>
<td>1566.71</td>
<td></td>
</tr>
<tr>
<td>1613.11</td>
<td>1599.37</td>
<td></td>
</tr>
<tr>
<td>1621.70</td>
<td>1555.62</td>
<td></td>
</tr>
<tr>
<td>1613.42</td>
<td>1581.96</td>
<td></td>
</tr>
</tbody>
</table>

Pretty consistent with 1.6GHz processor frequency, but estimates from pork are typically a little lower than those for Pistachio
Summary

- IPC in Pork is slower than Pistachio (17-65%)
- Overhead for crossing address spaces is higher in Pork than Pistachio (65% vs 35%)

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pork/Pistachio (Inter-AS)</td>
<td>1.17 – 1.35</td>
</tr>
<tr>
<td>Pork/Pistachio (Intra-AS)</td>
<td>1.42 – 1.65</td>
</tr>
<tr>
<td>Inter-AS/Intra-AS (Pork)</td>
<td>1.58 – 1.70</td>
</tr>
<tr>
<td>Inter-AS/Intra-AS (Pistachio)</td>
<td>1.30 – 1.40</td>
</tr>
</tbody>
</table>

Performance Tuning Opportunities?

- Are there opportunities for performance-tuning Pork to reduce the gap?

- Inter-AS: 
  \[
  \text{pistachio} = 1274.66 + 2.27n \\
  \text{pork} = 1512.57 + 6n
  \]  
  (least squares)

- Intra-AS: 
  \[
  \text{pistachio} = 756.54 + 2.21n \\
  \text{pork} = 1073.54 + 6.11n
  \]  
  (least squares)

- Example: Pork takes ~6 cycles to transfer a machine word, where Pistachio uses around ~2
Transfer Message in pork

Source:
for (i=1; i<=u; i++) {
    rutcb->mr[i] = sutcb->mr[i];
}

Machine Code:
209:  ba 01 00 00 00
20e: 8b 84 97 00 01 00 00
215: 89 84 91 00 01 00 00
21c: 83 c2 01
21f: 39 d3
221: 73 eb

Transfer Message in Pistachio

Source:
INLINE void tcb_t::copy_mrs(tcb_t * dest, word_t start, word_t count) {
    ASSERT(start + count <= IPC_NUM_MR);
    ASSERT(count > 0);
    word_t dummy;
    #if defined(CONFIG_X86_SMALL_SPACES)
    asm volatile ("mov %0, %%es" : : "r" (X86_KDS));
    #endif
    /* use optimized IA32 copy loop -- uses complete cacheline transfers */
    __asm__ __volatile__ ("cld\n"
                          "rep movsl (%0), (%1)\n" : /* output */
                          "=g"(dummy), "=D"(dummy), "=c"(dummy)
                          : /* input */
                          "c"(count), "S"(&get_utcb()->mr[start]),
                          "D"(&dest->get_utcb()->mr[start]));
    #if defined(CONFIG_X86_SMALL_SPACES)
    asm volatile ("mov %0, %%es" : : "r" (X86_UDS));
    #endif
}
Transfer Message in Pistachio

Machine Code:

```assembly
b15:  31 c9           xor    %ecx,%ecx   initialization
b17:  8b 73 0c        mov    0xc(%ebx),%esi
b1a:  8b 7d 0c        mov    0xc(%ebp),%edi
b1d:  88 d1           mov    %dl,%cl
b1f:  81 c6 04 01 00 00 add    $0x104,%esi
b25:  81 c7 04 01 00 00 add    $0x104,%edi
b2b:  fc              cld
b2c:  f3 a5           rep movsl %ds:(%esi), %es:(%edi)  loop
```

Reflections

- In this case, the performance differences between pork and Pistachio can be understood and (likely) addressed
- Could be handled by a compiler intrinsic (looks like a function, but treated specially by the compiler)
- Familiar in C (memcpy)
- How easily can other performance gaps be closed?
- Other opportunities for intrinsics? Special handling for fast paths? Algorithmic tweaks? Refined choice of data structures? etc.
Implementing pork

Introducing “pork”

- pork = the “Portland Oregon Research Kernel”
- An implementation of (a subset of) L4 X.2
- Similar API to Pistachio, but specific to IA32 platform
- Written around the start of 2007
- “I have almost all the pieces that I need to build an L4 kernel … perhaps I should try putting them together?”
- Built using the techniques we have seen so far in this course …
- … let’s take a tour!
boot.S should look very familiar …

```assembly
.global entry

entry: cli # Turn off interrupts

#----------------------------------------
# Create initial page directory:
#----------------------------------------

#----------------------------------------
# Turn on paging/protected mode execution:
#----------------------------------------

#----------------------------------------
# Initialize GDT:
#----------------------------------------

#----------------------------------------
# Initialize IDT:
#----------------------------------------

#----------------------------------------
# Initialize PIC:
#----------------------------------------

... jmp init # Jump off into kernel, no return!

#----------------------------------------
# Halt processor: Also used as code for the idle thread.
#----------------------------------------

.global halt

halt: hlt

jmp halt

#----------------------------------------
# Data areas:
#----------------------------------------

.data
...```
Exception handlers

# Descriptors and handlers for exceptions: ------------------------
intr 0, divideError
intr 1, debug
intr 2, nmiInterrupt
intr 3, breakpoint
intr 4, overflow

intr 5, boundRangeExceeded
intr 6, invalidOpcode
intr 7, deviceNotAvailable
intr 8, doubleFault, err=HWERR
intr 9, coprocessorSegmentOverrun

intr 10, invalidTSS, err=HWERR
intr 11, segmentNotPresent, err=HWERR
intr 12, stackSegmentFault, err=HWERR
intr 13, generalProtection, err=HWERR
intr 14, pageFault, err=HWERR

// Slot 15 is Intel Reserved
intr 16, floatingPointError
intr 17, alignmentCheck, err=HWERR
intr 18, machineCheck
intr 19, simdFloatingPointException

// Slots 20-31 are Intel Reserved

Hardware interrupt handlers

# Add descriptors for hardware irqs: ----------------------------
.equ IRQ_BASE, 0x20  # lowest hw irq number

.irq num, 0x21,0x22,0x23, 0x24,0x25,0x26,0x27, 0x28,0x29,0x2a,0x2b, 0x2c,0x2d,0x2e,0x2f
.intr \num, service=hardwareIRQ, err=(\num-IRQ_BASE)
.endr

intr 0x20, timerInterrupt
System call entry points

# Add descriptors for system calls: ---------------------------------------
# These are the only idt entries that we will allow to be called
# from user mode without generating a general protection fault,
# so they will be tagged with dpl=3.
intr  INT_THREADCONTROL, threadControl,   err=NOERR, dpl=3
intr  INT_SPACECONTROL,  spaceControl,    err=NOERR, dpl=3
intr  INT_IPC,           ipc,              err=NOERR, dpl=3
intr  INT_EXCHANGREGS,  exchangeRegisters, err=NOERR, dpl=3
intr  INT_SCHEDULE,      schedule,         err=NOERR, dpl=3
intr  INT_THREADSWITCH,  threadSwitch,     err=NOERR, dpl=3
intr  INT_UNMAP,         unmap,            err=NOERR, dpl=3
intr  INT_PROCESSCONTROL, processorControl, err=NOERR, dpl=3
intr  INT_MEMCONTROL,    memoryControl,    err=NOERR, dpl=3
intr  INT_SYSTEMCLOCK,   systemClock,      err=NOERR, dpl=3

Overall kernel structure
ENTRY invalidOpcode() {
    byte* eip = (byte*)current->context.iret.eip;
    if (eip[0]==0xf0 && eip[1]==0x90) { // Check for LOCK NOP instruction
        current->context.iret.eip += 2; // found => KernelInterface syscall
        KernelInterface_SetBaseAddress = kipStart(current->space);
        KernelInterface_SetAPIVersion = API_VERSION;
        KernelInterface_SetAPIFlags = API_FLAGS;
        KernelInterface_SetKernelId = KERNEL_ID;
        resume();
    }
    handleException(6);
}
**What's in the KIP?**

<table>
<thead>
<tr>
<th>~</th>
<th>Schedule SC</th>
<th>ThreadSwitch SC</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExchangeRegisters SC</td>
<td>Unmap SC</td>
<td>LPC SC</td>
<td>IPC SC</td>
</tr>
<tr>
<td>MemoryControl_pSC</td>
<td>ProcessorControl_pSC</td>
<td>ThreadControl_pSC</td>
<td>SpaceControl_pSC</td>
</tr>
<tr>
<td>ProcessorInfo</td>
<td>PageInfo</td>
<td>ThreadInfo</td>
<td>ClockInfo</td>
</tr>
<tr>
<td>ProcDescPtr</td>
<td>BootInfo</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>KipAreaInfo</td>
<td>UtcbInfo</td>
<td>VirtualRegInfo</td>
<td>~</td>
</tr>
<tr>
<td>~</td>
<td>~</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>~</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>MemoryInfo</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>~</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>~</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>KernDescPtr</td>
<td>API Flags</td>
<td>API Version</td>
<td>0(0/32)</td>
</tr>
<tr>
<td>~</td>
<td></td>
<td>~</td>
<td></td>
</tr>
</tbody>
</table>

kip.S

```assembly
.data
.align (1<<PAGESIZE)
.global Kip, KipEnd
Kip:  .byte 'L', '4', 230, 'K'
     .long API_VERSION, API_FLAGS, (KernelDesc - Kip)

.global Sigma0Server, Sigma1Server, RootServer
Kdebug: .long 0, 0, 0, 0  # Kernel debugger information
Sigma0Server: .long 0, 0, 0, 0  # Sigma0 information
Sigma1Server: .long 0, 0, 0, 0  # Sigma1 information
RootServer: .long 0, 0, 0, 0  # Rootserver information
     .long RESERVED

.global MemoryInfo
.macro memoryInfo offset, number
     .long ((\offset<16) | \number)
.endm
MemoryInfo: memoryInfo offset=(MemDesc-Kip), number=0

KdebugConfig: .long 0, 0
     .long RESERVED, RESERVED, RESERVED, RESERVED
     .long RESERVED, RESERVED, RESERVED, RESERVED
     .long RESERVED, RESERVED, RESERVED, RESERVED
     .long RESERVED, RESERVED, RESERVED, RESERVED
     .long RESERVED

VirtRegInfo: .long NUMMRS-1  # virtual register information
```

...
Onetime macros

KernelDesc: .long KERNEL_ID  # Kernel Descriptor

.macro kernelGenDate day, month, year
.long (\year-2000)<<9 | (\month<<5) | \day
.endm
kernelGenDate day=4, month=2, year=2007

.macro kernelVer ver, subver, subsubver
.long (((\ver<<8) | \subver)<<16) | \subsubver
.endm
kernelVer ver=1, subver=2, subsubver=0

Kernel entry points

SystemCalls: .long (spaceControlEntry - Kip)
.long (threadControlEntry - Kip)
.long (ipcEntry - Kip)
...
.long (exchangeRegistersEntry - Kip)
.long (threadSwitchEntry - Kip)
...

#-- Privileged system call entry points: ------------------------
.align 128
spaceControlEntry: int $INT_SPACECONTROL
ret
threadControlEntry: int $INT_THREADCONTROL
ret ...

#-- System call entry points: -------------------------------
ipcEntry: int $INT_IPC
ret

threadSwitchEntry: int $INT_THREADSWITCH
ret ...

Thread Ids

• User programs can reference other threads using thread ids

```c
#define isGlobal(tid)  (mask(tid, VERSIONBITS) & 0 (mod 64) != 0 (mod 64))
#define threadId(t,v)  ((t<<VERSIONBITS)|v)
#define anythread      (-1 (32/64)
#define anylocalthread  (1 (26/56) 000000)

typedef unsigned ThreadId;  // Global thread id
#define nilthread       0
#define anythread       (-1)
#define anylocalthread  ((-1)<<6)
#define threadId(t,v)   ((t<<VERSIONBITS)|v)
#define threadNo(tid)   mask((tid)>>VERSIONBITS, THREADBITS)
#define isGlobal(tid)   (mask(tid,6))
```

Thread Ids

Local Thread Id

- User-thread numbers can be freely allocated within the interval 0 , SystemBase
- Global thread IDs have a version field whose content can be freely set by those threads that can create and delete threads.
- The microkernel checks version fields whenever a thread is accessed through its global thread ID. However, the se-
- Note that any thread has a global thread ID, and two wild cards. The thread ID 000000 matches all threads that reside in the same address space.

Global Thread Id

- Local thread IDs identify threads within the same address space. They are identified by the 6 lowermost bits being 0.
- Global thread IDs are unique through the entire system. They identify threads independently of the address space in which they are used. Local thread IDs identify threads within the same address space. They are identified by the 6 lowermost bits being 0.
- Global thread IDs are unique through the entire system. They identify threads independently of the address space in which they are used. Local thread IDs identify threads within the same address space. They are identified by the 6 lowermost bits being 0.

Thread Ids

- User programs can reference other threads using thread ids

```c
#define isGlobal(tid)  (mask(tid, VERSIONBITS) & 0 (mod 64) != 0 (mod 64))
#define threadId(t,v)  ((t<<VERSIONBITS)|v)
#define anythread      (-1 (32/64)
#define anylocalthread  (1 (26/56) 000000)

typedef unsigned ThreadId;  // Global thread id
#define nilthread       0
#define anythread       (-1)
#define anylocalthread  ((-1)<<6)
#define threadId(t,v)   ((t<<VERSIONBITS)|v)
#define threadNo(tid)   mask((tid)>>VERSIONBITS, THREADBITS)
#define isGlobal(tid)   (mask(tid,6))
```
Flexpages (fpages)

- A generalized form of “page” that can vary in size:

  \[ fpage (b, 2^s) \]

  \[
  \frac{b}{2^{10}} \quad (22/54) \quad s \quad \begin{array}{c}
  0 \quad r \quad w \quad x
  \end{array}
  \]

- Includes both 4KB pages and 4MB superpages as special cases

- Also includes special cases to represent the full address space (complete) and the empty address space (nilpage):

  \[
  \text{complete} \quad 0 \quad (22/54) \quad s = 1 \quad 0 \quad r \quad w \quad x
  \]

  \[
  \text{nilpage} \quad 0 \quad (32/64)
  \]

- Can be represented, in practice, using collections of 4KB and 4MB pages
Example

The first 128KB of an address space can be represented by:

<table>
<thead>
<tr>
<th>1 x 128KB</th>
<th>128K</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 x 64KB</td>
<td>64K</td>
</tr>
<tr>
<td>4 x 32KB</td>
<td>32K</td>
</tr>
<tr>
<td>8 x 16KB</td>
<td>16K</td>
</tr>
<tr>
<td>16 x 8KB</td>
<td>8K</td>
</tr>
<tr>
<td>32 x 4KB</td>
<td>4K</td>
</tr>
</tbody>
</table>

If two flexpages overlap, then one includes the other.

Flexpage implementation

```c
/*---------------------------------------------*/
/* The Flexpage datatype: */
/*---------------------------------------------*/
typedef unsigned Fpage;

static inline Fpage fpage(unsigned base, unsigned size) {
    return align(base, size) | (size<<4);
}

static inline Fpage completeFpage() {
    // [0::Bit 22 | 1::Bit 6 |0|r|w|x]
    return (1<<4);
}

extern unsigned fpsize[];
// initialized to 0 -> 0, 1 -> 32, 2 -> 0, ..., 11 -> 0,
// 12 -> 12, 13 -> 13, ..., 32 -> 32, 33 -> 0, ...
extern unsigned fpmask[];
// initialized to 0 -> 0, 1 -> ~0, 2 -> 0, ..., 11 -> 0,
// 12 -> 0xfff, 13 -> 0x1fff, ..., 32 -> 0xffffffff, 33 -> 0, ...

static inline unsigned fpageMask(Fpage fp)  {
    return fpmask[(fp>>4)&0x3f]; }
static inline unsigned fpageSize(Fpage fp)  {
    return fpsize[(fp>>4)&0x3f]; }
static inline bool isComplete(Fpage fp) {
    return ~fpageMask(fp) == 0; }
static inline bool isNilpage(Fpage fp) {
    return fpageMask(fp) == 0; }
static inline unsigned fpageStart(Fpage fp) {
    return fp & ~fpageMask(fp); }
static inline unsigned fpageEnd(Fpage fp) {
    return fp | fpageMask(fp); }
```
Initialization of fpsize and fpmask arrays

```c
void initSpaces() {
    // Basic consistency checks:
    ASSERT((unsigned)Kip % PAGESIZE == 0, "KIP alignment error");
    ASSERT((KipEnd - Kip) <= (1 << KIPAREASIZE), "KIP size error");
    ASSERT(KIPAREASIZE <= PAGESIZE, "KIP area size error");
    ASSERT(UTCBSIZE <= PAGESIZE, "UTCB area size error");

    // Initialize fpage mask and size arrays.
    unsigned i;
    for (i = 0; i < 64; i++) {
        fpsize[i] = fpmask[i] = 0;
    }
    unsigned k = 0xfff;
    for (i = 12; i <= 32; i++) {
        fpsize[i] = i;
        fpmask[i] = k;
        k = (k << 1) | 1;
    }
    fpsize[1] = 32;
    fpmask[1] = ~0;
    ...
}
```

Memory Management
Kernel Memory Allocator

- `void initMemory(void);`
  The kernel reserves a pool of 4K pages as part of the initialization process.

- `void* allocPage1(void);`
  Allocates a single page from the kernel pool

- `void freePage(void* p);`
  Returns a single page to the kernel pool

- `bool availPages(unsigned n);`
  Checks to see if there are (at least) n free pages

- Around ~150 lines of code, most in `initMemory()`
- No automatic GC in pork ...

Why `alloc1()`?

- A function `f` that requires the allocation of up to `N` pages (but never more) has a name of the form `fN`

- A function that calls `fN()` will either:
  - Call `availPages(N)` beforehand
  - Have a name of the form `gM`, where `M` is `N` plus the number of additional pages that `gM` might require ...

- Goal: minimize number of checks for free pages
  - Reduce code size
  - Improve performance
  - Fewer places to write error handling code
Alas, this could fail!

- Consider the following function:
  ```c
  void g1() {    // 1 suffix because this function
                   // allocates a page
    f();
    void* p = allocPage1();
    ...
  }
  ```

- But now suppose `f()` takes the form:
  ```c
  void f() {
    if (availPages(1)) { ... allocPage1(); ... }
  }
  ```

- Pork still uses this naming convention, but relies on “disciplined use”
- Maybe a type system could do better … ?
Thread control blocks (TCBs)

```c
struct TCB {
    ThreadId tid;       // this thread's id and version number
    byte status;       // thread status
    byte prio;         // thread priority
    byte padding;      // for gc of TCBs in kernel memory
    struct UTCB* utcb; // pointer to this thread's utcb
    unsigned vutcb;    // virtual address of utcb
    struct TCB* sendqueue; // list of threads waiting to send
    struct TCB* receiver; // pointer to owner of sendqueue
    struct TCB* prev;
    struct TCB* next;
    struct Space* space; // pointer to this thread's addr space
    unsigned faultCode; // exception number or page fault addr
    struct Context context; // context of user level process
    ThreadId scheduler; // scheduling parameters
    unsigned timeslice;
    unsigned timeleft;
    unsigned quantleft;
};
```

```c
typedef struct TCB TCBTable[32]
```

```c
Thread control blocks (TCBs)

```c
typedef TCBTable* tcbDir[4096]
```

```c
struct TCB* existsTCB(unsigned threadNo) {
    TCBTable* tab = tcbDir[threadNo>>TCBDIRBITS];
    if (tab) {
        struct TCB* tcb = ((struct TCB*)tab) + mask(threadNo, TCBDIRBITS);
        if (tcb->space) {
            return tcb;
        }
    }
    return 0;
}
```

```c
struct TCB* findTCB(ThreadId tid) {
    struct TCB* tcb = existsTCB(threadNo(tid));
    return (tcb && tcb->tid==tid) ? tcb : 0;
}
```
Thread control blocks (TCBs)

Allocating and initializing TCBs

```c
struct TCB* allocTCB(ThreadId tid, struct Space* space, ThreadId scheduler) {
    unsigned threadNo = threadNo(tid);
    TCBTable* tab = tcbDir[threadNo>>TCBDIRBITS];
    if (!tab) {
        tab = tcbDir[threadNo>>TCBDIRBITS] = (TCBTable*)allocPage1();
    }
    ++tab[0]->count; // Count an additional TCB in this page
    struct TCB* tcb = ((struct TCB*)tab) + mask(threadNo, TCBDIRBITS);
    tcb->tid = tid;
    tcb->status = Halted;
    tcb->space = space;
    tcb->utcb = 0;
    tcb->vutcb = 0xffffffff;
    tcb->sendqueue = 0;
    tcb->next = tcb;
    tcb->prev = tcb; // Default is unspecified
    tcb->prio = 128;
    tcb->scheduler = scheduler;
    tcb->timeslice =
    tcb->timeleft = 10000; // Default timeslice is 10ms
    tcb->quantleft = 0; // Default quantum is infinite
    initUserContext(&(tcb->context));
    enterSpace(space); // Register the thread in this space
    return tcb;
}
```
Thread Control Blocks (TCBs)

### Thread Id
- `ThreadId`
- `struct TCB* runqueue[256]`

### Scheduling data structures: runqueue
- Doubly-linked list of runnable threads with priority p
- Doubly-linked list of runnable threads with priority q
Scheduling data structures: runqueue

Doubly-linked list of blocked threads waiting to communicate with C

Switching to a new thread (w/o debugging)

static void inline switchTo(struct TCB* tcb) {
    struct Context* ctxt = &(tcb->context);
    current = tcb; // Change current thread
    *utcbptr = tcb->vutcb // Change UTCB address
        + (unsigned)&((struct UTCB*)0)->mr[0]);
    esp0 = (unsigned)(ctxt + 1); // Change esp0
    switchSpace(tcb->space); // Change address space
    returnToContext(ctxt);
}

...
Scheduling data structures: prioset

```c
/* Select a new thread to execute. We pick the next runnable thread with *
* the highest priority. */
void reschedule() {
    switchTo(holder = priosetSize ? runqueue[prioset[0]] : idleTCB);
}
```

Address Spaces
Address space layout

0 3GB 4GB

user space

kernel space

KIP

UTCB area

Kernel Information Page
(mapped in to every address space)

User Thread Control Block

One UTCB for each (possible) thread in the address space

Representing address spaces

```c
struct Space {
    unsigned pdir;  // Physical address of page directory
    struct Mapping* mem;  // Memory map
    Fpage kipArea;  // Location of kernel interface page
    Fpage utcbArea;  // Location of UCTBs
    unsigned count;  // Count of threads in this space
    unsigned active;  // Count of active threads in this space
    unsigned loaded;  // 1 => already loaded in cr3
};
```

```c
void enterSpace(struct Space* space) {
    space->count++;  // increment reference count;
}
```

```c
void configureSpace(struct Space* space, Fpage kipArea, Fpage utcbArea) {
    ASSERT(!activeSpace(space), "configuring active space");
    space->kipArea = kipArea;
    space->utcbArea = utcbArea;
}
```
A typical system call

ENTRY spaceControl() {
    if (!privileged(current->space)) { /* check for privileged thread */
        retError(SpaceControl_Result, NO_PRIVILEGE);
    } else {
        struct TCB* dest = findTCB(SpaceControl_SpaceSpecifier);
        if (!dest) {
            retError(SpaceControl_Result, INVALID_SPACE);
        } else if (!activeSpace(dest->space)) { /* ignore if active threads */
            Fpage kipArea = SpaceControl_KipArea;
            Fpage utcbArea = SpaceControl_UtcbArea;
            unsigned kipEnd, utcbEnd;
            if (isNilpage(utcbArea) /* validate utcb area */
                || fpageSize(utcbArea)<MIN_UTCBAREASIZE
                || (utcbEnd=fpageEnd(utcbArea))>=KERNEL_SPACE) {
                retError(SpaceControl_Result, INVALID_UTCB);
            } else if (isNilpage(kipArea) /* validate KIP area */
                || fpageSize(kipArea)!=KIPAREASIZE
                || (kipEnd=fpageEnd(kipArea))>=KERNEL_SPACE
                || (kipEnd>=fpageStart(utcbArea) && utcbEnd>=fpageStart(kipArea))) {
                retError(SpaceControl_Result, INVALID_KIPAREA);
            } else {
                configureSpace(dest->space, kipArea, utcbArea);
            }
        }
        SpaceControl_Result = 1;
        SpaceControl_Control = 0; /* control parameter is not used */
        resume();
    }
}

Spaces and mappings

Thread Id:

<table>
<thead>
<tr>
<th>Thread Id</th>
<th>tableidx,2</th>
<th>idx,8</th>
<th>version,12</th>
</tr>
</thead>
</table>

typedef struct TCB TCBTable[32]

typedef struct TCB TCBTable[4096]

tcbDir[4096]

struct TCB* runqueue[256]

struct Space

struct Mapping

id
queue data
scheduling params
Context
Representing mappings

```c
struct Mapping {
    struct Space*   space;   // Which address space is this in?
    struct Mapping* next;
    struct Mapping* prev;
    unsigned        level;
    Fpage           vfp;     // Virtual fpage
    unsigned        phys;    // Physical address
    struct Mapping* left;
    struct Mapping* right;
};
```

• A binary search tree of memory regions within a single address space

• A mapping data base that documents the way that memory regions have been mapped between address spaces

Small Objects

• Pork uses only two “small” object types (≤32 bytes):
  • Address space descriptors (Space)
  • Mapping descriptors (Mapping)

• Kernel allocates/frees pages to store small objects (each page can store up to 127 objects)

• Pages with free slots are linked together
Thread status

/*----------------------------------------------*
 * Thread status:
 * A byte field in each TCB specifies the current status of that thread:
 * +----+----+----+---------+
 * | b6 | b5 | b4 | ipctype |
 * +----+----+----+---------+
 * b3-b0: ipctype (4 bits)
 * b4: 1=>halted, or halt requested (i.e., will halt after IPC)
 * b5: 1=>blocked waiting to send an ipc of the specified type
 * b6: 1=>blocked waiting to receive an ipc of the specified type
 * A zero status byte indicates that the thread is Runnable.
 *----------------------------------------------*/

#define Runnable    0
#define Halted       0x10
#define Sending(type)    (0x20 | (type))
#define Receiving(type)  (0x40 | (type))

typedef enum {
    MRs, PageFault, Exception, Interrupt, Preempt, Startup
} IPCType;

static inline IPCType ipctype(struct TCB* tcb) {
    return (IPCType)(tcb->status & 0xf);
}
The ipc system call

/*---------------------------------------------*
 * The "IPC" System Call:
 *---------------------------------------------*/
ENTRY ipc() {

ThreadId to = IPC_GetTo; // Send Phase
if (to!=nilthread) {
  if (!sendPhase(MRs, current, to)) {
    reschedule();
  }
}

ThreadId fromSpec = IPC_GetFromSpec(current); // Receive Phase
if (fromSpec!=nilthread) {
  current->utcb->mr[0] = 0;
  recvPhase(MRs, current, fromSpec);
}
reschedule();
}

The send phase (Part I)

static bool sendPhase(IPCType sendtype, struct TCB* send, ThreadId recvId) {

  // Find the receiver TCB: -----------------------------------------------
  struct TCB* recv;
  if (recvId==anythread      ||
      recvId==anylocalthread ||
      !(recv=findTCB(recvId))) {
    sendError(sendtype, send, NonExistingPartner);
    return 0;
  }

  // Determine whether we can send the message immediately: ---------------
  if (isReceiving(recv)) {
    IPCType recvtype = ipcType(recv);
    ThreadId srcId   = recvFromSpec(recvtype, recv);
    if (((srcId==send->tid) ||
         (srcId==anythread) ||
         (srcId==anylocalthread && send->space==recv->space)) {
      // Destination is blocked and ready to receive from send:
      IPCError err = transferMessage(sendtype, send, recvtype, recv);
      if (err==NoError) {
        resumeThread(recv);
        return 1;
      } else {
        sendError(sendtype, send, err);
        recvError(recvtype, recv, err);
        return 0;
      }
    }
  }
  ...

The send phase (Part 2)

...  
// Destination is not ready to receive a message, so try to block: ------
if (sendCanBlock(sendtype, send)) {
  if (send->status==Runnable) {
    removeRunnable(send);
  }
  send->status = Sending(sendtype) | (Halted & send->status);
  send->receiver = recv;
  recv->sendqueue = insertTCB(recv->sendqueue, send);
} else {
  sendError(sendtype, send, NoPartner);
}
return 0;
}

Transferring messages

static IPCErr transferMessage(IPCType sendtype, struct TCB* send, 
IPCType recvtype, struct TCB* recv) {
  if (recvtype==MRs) {
    // Send to MRs (Destination is user ipc)
    ...
    switch (sendtype) {
      case MRS : ... // Send between sets of message registers
      case PageFault : ... // Send pagefault message to pager
      case Exception : ... // Send message to an exception handler
      case Interrupt : ... // Send message to an interrupt handler
    }
  } else if (sendtype==MRs) {
    // Receive from MRs (Source is user ipc)
    ...
    switch (recvtype) {
      case PageFault : ... // Receive a response from a pager
      case Exception : ... // Receive a response from an exception handler
      case Interrupt : ... // Receive a response from an interrupt handler
      case Startup   : ... // Receive startup message from thread's pager
    }
    return Protocol; // Protocol error: incompatible types/format
  }
Regular IPC:

\[
\begin{align*}
\text{struct UTCB* } & \text{ rutcb = recv->utcb; } \\
\text{struct UTCB* } & \text{ sutcb = send->utcb; } \\
\text{unsigned } & \text{ u = mask(sutcb->mr[0], 6); } & \text{// untyped items} \\
\text{unsigned } & \text{ t = mask(sutcb->mr[0]>>6, 6); } & \text{// typed items} \\
\text{if } & \text{ ((u+t)\geq\text{NUMMRS}) } \text{ || (t&1)) } \{ \\
& \text{return MessageOverflow; } \text{MsgTag [MRs]} \\
\} & \text{else } \{ \\
& \text{unsigned } i; \\
& \text{rutcb->mr[0] = MsgTag(sutcb->mr[0]>>16, 0, t, u); } \\
& \text{for (i=1; } i\leq u; i++) \{ \\
& \quad \text{rutcb->mr[i] = sutcb->mr[i]; } \\
& \} \\
& \text{if (t>0) } \{ \\
& \quad \text{Fpage acc = rutcb->acceptor; } \\
& \quad \text{do } \{ \\
& \quad \quad \text{IPCErr err = transferTyped(send, recv, acc,} \\
& \quad \quad \quad \text{rutcb->mr[i] = sutcb->mr[i],} \\
& \quad \quad \quad \text{rutcb->mr[i+1] = sutcb->mr[i+1]);} \\
& \quad \quad \text{if (err\neq\text{NoError}) } \{ \\
& \quad \quad \quad \text{return err; } \\
& \quad \quad \} \\
& \quad \quad i += 2; \\
& \quad \} \text{while } ((t-2)>0); \\
& \} \\
& \text{return NoError; } \\
\}
\end{align*}
\]

Example: IPCs from hardware interrupts

ENTRY hardwareIRQ() {
    \text{unsigned } n = \text{current->context.iwrite.error;}
    \text{maskAckIRQ(n); } & \text{// Mask and acknowledge the interrupt with the PIC} \\
\text{struct TCB* } & \text{ irqTCB = existsTCB(n); } \\
\text{if } & \text{ (irqTCB->status==\text{Halted} } \&\& \text{ irqTCB->vutcb!=nullthread) } \{ \\
& \text{if } \text{(sendPhase(Interrupt, irqTCB, irqTCB->vutcb)) } \{ \\
& \quad \text{irqTCB->status = Receiving(Interrupt) } | \text{ Halted; } \\
& \} \\
& \text{reschedule(); } & \text{// allow the user level handler to begin ...} \\
\}
Interrupt handler protocol

• When a hardware interrupt occurs, the kernel sends an IPC message from the interrupt thread to its pager with the tag:

\[ \text{Interrupt} \implies \text{MRs} \]

**From Interrupt Thread**

\[
\begin{array}{cccccc}
-1_{(12/44)} & 0_{(4)} & 0_{(4)} & t = 0_{(6)} & u = 0_{(6)} & \text{MR} 0
\end{array}
\]

```c
\text{case Interrupt :} \quad // \text{Send message to an interrupt handler}
\text{rutcb->mr[0] = MsgTag((-1)<<4, 0, 0, 0);}
\text{return NoError;}
```

• When the pager has finished handling the error, it sends an IPC message back to the interrupt thread to reenable the corresponding interrupt

**To Interrupt Thread**

\[
\begin{array}{cccccc}
0_{(16/48)} & 0_{(4)} & t = 0_{(6)} & u = 0_{(6)} & \text{MR} 0
\end{array}
\]
Example: IPCs from page faults

```c
ENTRY pageFault() {
    asm("movl %%cr2, %0\n" : "=r"(current->faultCode));
    if (current->space==sigma0Space && sigma0map(current->faultCode)) {
        printf("sigma0 case succeeded!\n");
    } else {
        ThreadId pagerId = current->utcb->pager;
        if (pagerId==nilthread) {
            haltThread(current);
        } else if (sendPhase(PageFault, current, pagerId)) {
            removeRunnable(current); // Block current if message already delivered
            current->status = Receiving(PageFault);
        }
    }
    refreshSpace();
    reschedule();
}
```

Page fault protocol

- When a thread triggers a page fault, the kernel translates that event into an IPC to the thread's pager:

  **To Pager**

<table>
<thead>
<tr>
<th>faulting user-level IP (32/64)</th>
<th>MR 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>fault address (32/64)</td>
<td>MR 1</td>
</tr>
<tr>
<td>0−2 (12/44)</td>
<td>MR 0</td>
</tr>
<tr>
<td>0 r w x</td>
<td>t = 0 (6)</td>
</tr>
<tr>
<td>0 (4)</td>
<td>u = 2 (6)</td>
</tr>
</tbody>
</table>

- The pager can respond by sending back a reply with a new mapping … that also restarts the faulting thread:

  **From Pager**

<table>
<thead>
<tr>
<th>MapItem / GrantItem</th>
<th>MR 1,2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (16/48)</td>
<td>MR 0</td>
</tr>
<tr>
<td>0 (4)</td>
<td>t = 2 (6)</td>
</tr>
<tr>
<td>t = 2 (6)</td>
<td>u = 0 (6)</td>
</tr>
</tbody>
</table>
Page fault protocol

- When a thread triggers a page fault, the kernel translates that event into an IPC to the thread’s pager:

```
To Pager

<table>
<thead>
<tr>
<th>MR 2</th>
<th>faulting user-level IP (32/64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR 1</td>
<td>fault address (32/64)</td>
</tr>
<tr>
<td>MR 0</td>
<td>-2 (12/44)</td>
</tr>
<tr>
<td></td>
<td>0 r w x 0 (4) t = 0 (6) u = 2 (6)</td>
</tr>
</tbody>
</table>
```

```c
case PageFault : { // Send pagefault message to pager
    unsigned rwx = (send->context.iret.error & 2) ? 2 : 4;
    rutcb->mr[0] = MsgTag((-2)<<4|rwx, 0, 0, 2);
    rutcb->mr[1] = send->faultCode;
    rutcb->mr[2] = send->context.iret.eip;
} return NoError;
```

Page fault protocol

- The pager can respond by sending back a reply with a new mapping … that also restarts the faulting thread:

```
From Pager

<table>
<thead>
<tr>
<th>MR 1,2</th>
<th>MapItem / GrantItem</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR 0</td>
<td>0 (16/48)</td>
</tr>
<tr>
<td></td>
<td>0 (4)</td>
</tr>
<tr>
<td></td>
<td>t = 2 (6)</td>
</tr>
<tr>
<td></td>
<td>u = 0 (6)</td>
</tr>
</tbody>
</table>
```

```c
case PageFault : { // Receive a response from a pager
    if (mask(sutcb->mr[0],12)==MsgTag(0, 0, 2, 0)) {
        return transferTyped(send, recv, completeFpage(), sutcb->mr[1], sutcb->mr[2]);
    }
    break;

return NoError;
```
Time to poke around … !