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”

Transcribed Data (Inter-AS)

<table>
<thead>
<tr>
<th>ping pong</th>
<th>pistachio Inter-AS IPC</th>
<th>cycles</th>
<th>microseconds</th>
<th>pork Inter-AS IPC</th>
<th>cycles</th>
<th>microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1240.67</td>
<td>0.77</td>
<td>1519.59</td>
<td>0.95</td>
<td>1.22</td>
<td>1.23</td>
</tr>
<tr>
<td>4</td>
<td>1293.58</td>
<td>0.81</td>
<td>1530.14</td>
<td>0.95</td>
<td>1.22</td>
<td>1.23</td>
</tr>
<tr>
<td>8</td>
<td>1301.64</td>
<td>0.81</td>
<td>1556.71</td>
<td>0.99</td>
<td>1.25</td>
<td>1.28</td>
</tr>
<tr>
<td>12</td>
<td>1306.29</td>
<td>0.81</td>
<td>1579.67</td>
<td>0.99</td>
<td>1.25</td>
<td>1.28</td>
</tr>
<tr>
<td>16</td>
<td>1317.96</td>
<td>0.81</td>
<td>1607.34</td>
<td>1.02</td>
<td>1.27</td>
<td>1.29</td>
</tr>
<tr>
<td>20</td>
<td>1325.16</td>
<td>0.83</td>
<td>1634.98</td>
<td>1.02</td>
<td>1.27</td>
<td>1.29</td>
</tr>
<tr>
<td>24</td>
<td>1333.26</td>
<td>0.83</td>
<td>1664.64</td>
<td>1.02</td>
<td>1.27</td>
<td>1.29</td>
</tr>
<tr>
<td>28</td>
<td>1342.28</td>
<td>0.84</td>
<td>1687.47</td>
<td>1.02</td>
<td>1.27</td>
<td>1.29</td>
</tr>
<tr>
<td>32</td>
<td>1350.34</td>
<td>0.84</td>
<td>1702.89</td>
<td>1.06</td>
<td>1.28</td>
<td>1.30</td>
</tr>
<tr>
<td>36</td>
<td>1358.46</td>
<td>0.85</td>
<td>1721.46</td>
<td>1.06</td>
<td>1.28</td>
<td>1.30</td>
</tr>
<tr>
<td>40</td>
<td>1362.08</td>
<td>0.85</td>
<td>1745.56</td>
<td>1.10</td>
<td>1.30</td>
<td>1.33</td>
</tr>
<tr>
<td>44</td>
<td>1374.64</td>
<td>0.86</td>
<td>1787.86</td>
<td>1.14</td>
<td>1.32</td>
<td>1.34</td>
</tr>
<tr>
<td>48</td>
<td>1382.80</td>
<td>0.86</td>
<td>1804.40</td>
<td>1.14</td>
<td>1.32</td>
<td>1.34</td>
</tr>
<tr>
<td>52</td>
<td>1390.88</td>
<td>0.87</td>
<td>1818.78</td>
<td>1.14</td>
<td>1.32</td>
<td>1.33</td>
</tr>
<tr>
<td>56</td>
<td>1398.02</td>
<td>0.87</td>
<td>1842.79</td>
<td>1.14</td>
<td>1.32</td>
<td>1.33</td>
</tr>
<tr>
<td>60</td>
<td>1406.13</td>
<td>0.88</td>
<td>1875.66</td>
<td>1.18</td>
<td>1.34</td>
<td>1.34</td>
</tr>
</tbody>
</table>

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

Pork “Output”

Cycles (Inter-AS)

Microseconds (Inter-AS)

- Pistachio = 1274.66 + 2.27n (least squares)
- Pork = 1512.57 + 6n
Pork : Pistachio (Inter-AS)

Transcribed Data (Intra-AS)

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

Cycles (Intra-AS)

Microseconds (Intra-AS)

pistachio = 756.54 + 2.21n (least squares)

pork = 1073.54 + 6.11n

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/Pork (Pistachio)</td>
<td>1.58 - 1.70</td>
</tr>
<tr>
<td>Inter-AS/Pistachio (Pork)</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:
  - \(\text{pistachio} = 1274.66 + 2.27n\) (least squares)
  - \(\text{pork} = 1512.57 + 6n\)
- Intra-AS:
  - \(\text{pistachio} = 756.54 + 2.21n\) (least squares)
  - \(\text{pork} = 1073.54 + 6.11n\)
- Example: Pork takes ~6 cycles to transfer a machine word, where Pistachio uses around ~2

Transfer Message in Pork

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

Machine Code:
```
209: ba 01 00 00 00          mov    $0x1,%edx
20e: 8b 84 97 00 01 00 00    mov    0x100(%edi,%edx,4),%eax
215: 89 84 91 00 01 00 00    mov    %eax,0x100(%ecx,%edx,4)
21c: 83 c2 01                add    $0x1,%edx
21f: 39 d3                   cmp    %edx,%ebx
221: 73 eb                   jae    20e
```

Transfer Message in Pistachio

Source:
```c
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
    __asm__ __volatile__ ("cld
                          "rep movsl (%0), (%1)
                          : /* output */
                          =S"(dummy), =D"(dummy), =c"(count),
                          "c"(count), "D"(&get_utcb()->mr[start]),
                          "S"(&dest->get_utcb()->mr[start]));
    #endif
}
```

Machine Code:
```
b15: 31 c9                   xor    %ecx,%ecx
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)
```

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

boot.S should look very familiar …

```
.globl entry
entry: cli  # Turn off interrupts

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

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

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

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

jmp init  # Jump off into kernel, no return!

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

.globl halt
halt: hlt

jmp halt

# Data areas:

```

Exception handlers

```
# Descriptors and handlers for exceptions: ------------------------
intr 0, divideError
intr 1, debug
intr 2, nmiInterrupt
intr 3, breakpoint
intr 4, overflow
intr 5, boundsRangeExceeded
intr 6, invalidOpCode
intr 7, deviceNotAvailable
intr 8, doubleFault, err=HEEE
intr 9, coprocessorSegmentOverrun
intr 10, invalidTSS, err=HEEE
intr 11, segmentNotPresent, err=HEEE
intr 12, stackSegmentFault, err=HEEE
intr 13, generalProtection, err=HEEE
intr 14, pageFault, err=HEEE

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

// Slots 20-31 are Intel Reserved
```

Hardware interrupt handlers

```
# Add descriptors for hardware irqs: -----------------------------
.squ 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

```c
intr INT_THREADCONTROL, threadControl, err=MOERE, dpl=3
intr INT_SPACECONTROL, spaceControl, err=MOERE, dpl=3
intr INT_PIC, icp, err=MOERE, dpl=3
intr INT_EXCHANGERSIG, exchangeRegisters, err=MOERE, dpl=3
intr INT_THREADSWITCH, threadSwitch, err=MOERE, dpl=3
intr INT_UNMAP, unmap, err=MOERE, dpl=3
intr INT_PROCCONTROL, processorControl, err=MOERE, dpl=3
intr INT_MEMCONTROL, memoryControl, err=MOERE, dpl=3
intr INT_SYSTEMCLOCK, systemClock, err=MOERE, dpl=3
```

Overall kernel structure

### An example exception handler

```c
ENTRY invalidOpcode() {
    byte* eip = (byte*)current->context.iret.eip;
    if (eip[0]==0x00 && eip[1]==0x00) { // Check for LOCK NOP instruction
        if (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>KernelDesc</th>
<th>ThreadDesc</th>
<th>Reserves</th>
<th>PAGE_SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcessorName</td>
<td>ProcessorVersion</td>
<td>ProcessorFlags</td>
<td>PAGE_SIZE</td>
</tr>
<tr>
<td>ThreadDesc</td>
<td>ThreadDesc</td>
<td>ThreadDescFlags</td>
<td>PAGE_SIZE</td>
</tr>
<tr>
<td>RootServer</td>
<td>RootServer</td>
<td>RootServerFlags</td>
<td>PAGE_SIZE</td>
</tr>
<tr>
<td>Sigma1Server</td>
<td>Sigma1Server</td>
<td>Sigma1ServerFlags</td>
<td>PAGE_SIZE</td>
</tr>
<tr>
<td>KipEnd</td>
<td></td>
<td></td>
<td>PAGE_SIZE</td>
</tr>
</tbody>
</table>

### kip.S

```asm
.data
.align (1<<PAGESIZE)
Kip:
    .set Kip, KipEnd
    .byte 'K', 'I', 'P', 'K', 'E', 'P'
    .long API_VERSION, API_FLAGS, (KernelDesc - Kip)

.global SigmaServer, SigmaServer, RootServer
KdebugConfig:
    .long 0, 0, 0
    .long 0, 0, 0
    .long 0, 0, 0
    .long 0, 0, 0
    .global MemoryInfo, MemoryInfo
    MemoryInfo offset=(KernelDesc-Kip), number=0

MemoryInfo Offset: offset
```

### The KIP

![Kernel Interface Page Diagram](image)
Onetime macros

```
// Kernel Descriptor
.long KERNEL_ID     # Kernel Descriptor

// Various date-related macros
.macro kernelGenDate day, month, year
    .long  (\year-2000)<<8 | (\month<<5) | \day
.endm

kernelGenDate day=1, 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 - Xip]
    .long [threadControlEntry - Xip]
    .long [ipcEntry - Xip]
    ...
    .long [exchangeRegistersEntry - Xip]
    .long [threadSwitchEntry - Xip]
    ...

    #-- Privileged system call entry points: --------------------------
    align 128
    spaceControlEntry: int ret
        $INT_SPACECONTROL
    threadControlEntry: int ret
        $INT_THREADCONTROL
    ...
    #-- System call entry points: -------------------------------
    ipcEntry: int ret
        $INT_IPC
    threadSwitchEntry: int ret
        $INT_THREADSWITCH
    ...
```

Thread Ids

```
* User programs can reference other threads using thread ids

```

```
/* Thread Ids: */
#define nithread 0
#define anythread -1
#define anylocalthread -1
#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:

```
```

```
* 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):

```
```
Example

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

| 1 x 128KB | 128K |
| 2 x 64KB | 64K |
| 4 x 32KB | 32K |
| 8 x 16KB | 16K |
| 16 x 8KB | 8K |
| 32 x 4KB | 4K |

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(void) {
    // [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(mask((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 gN, where N is N plus the number of additional pages that gN 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)

Scheduling data structures: runqueue

Scheduling data structures: prioset

Address Spaces
**Address space layout**

0 3GB 4GB

- **User space**
- **Kernel space**

**Kernel Information Page**

(mapped in to every address space)

**UTCB area**

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 UTCBs
    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++;
}
```

```c
void configureSpace(struct Space* space, Fpage kipArea, Fpage utcbArea) {
    space->kipArea  = kipArea;
    space->utcbArea = utcbArea;
}
```

---

**A typical system call**

```c
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
            configureSpace(dest->space, kipArea, utcbArea);
    }
}
```

---

**Spaces and 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

---

**Representing mappings**

```c
struct Mapping {
    struct Space* space; // Which address space 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
**Page Directories and Page Tables**

**User TCBs (UTCBs)**

**Thread status**

The `iptype` field in each TCB specifies the current status of that thread:

<table>
<thead>
<tr>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>ipctype</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>halt</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>blocked</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>blocked</td>
</tr>
</tbody>
</table>

A zero status byte indicates that the thread is Runnable.

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

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

```c
static inline IPCType ipctype(struct TCB* tcb) {
    return (IPCType)(tcb->status & 0xf);
}
```

```c
The send phase (Part I)
```

```c
static bool sendPhase(IPCType sendtype, struct TCB* send, ThreadId recvid) {
    // Find the receiver TCB: -----------------------------------------------
    struct TCB* recv = IPC_GetFromSpec(send, recvid);
    if (recv==nilthread) {
        sendError(sendtype, send, NoExistingPartner);
        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)) {
            IPCErr err = transferMessage(sendtype, send, recvtype, recv);
            if (err==NoError) {
                resumeThread(recv);
                return 1;
            } else {
                sendError(sendtype, send, err);
                recvError(recvtype, recv, err);
                return 0;
            }
        } else {
            sendError(sendtype, send, err);
            recvError(recvtype, recv, err);
            return 0;
        }
    }
    return 0;
}
```
Interrupt handler protocol

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

  From Interrupt Thread

  ![Interrupt From Thread Diagram]

  case Interrupt : // Send message to an interrupt handler
  rutc->mr[0] = MsgTag((-1)<<4, 0, 0, 0);
  return NoError;

Example: IPCs from hardware interrupts

ENTRY hardwareIRQ() {
  unsigned n = current->context.ioret.error;
  maskAckIRQ(n); // Mask and acknowledge the interrupt with the PIC
  struct TCB* irqTCB = existsTCB(n);

  if (irqTCB->status==Waited && irqTCB->vutcb->nilthread) {
    if (sendPhase(Interrupt, irqTCB, irqTCB->vutcb)) {
      irqTCB->status = Receiving(Interrupt); // Waite;
    }
  
  reschedule(); // allow the user level handler to begin ...
}

Interrupt handler protocol

- 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

![Interrupt To Thread Diagram]
Example: IPCs from page faults

```c
ENTRY pageFault() {
    asm{"movl $0, r8r2", t0\[
if (current->space==sigma0Space && sigma0map(current->faultCode)) {
    printf("sigma0 case succeeded\n");
} else {
    ThreadId pagerId = current->utcb->pager;
    if (pagerId=simthread) {
        htlThread(current);
    } else if (sendPhase(PageFault, current, pagerId)) {
        reschedule() = true;
        refreshSpace();
    }
    refreshSpace();
    reschedule();
}
}
```