11001010
0011010
101111
01010

CS 410/510

Languages & Low-Level Programming

Mark P Jones
Portland State University

Spring 2016

Week 6: L4 Implementation

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 …

```
        .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

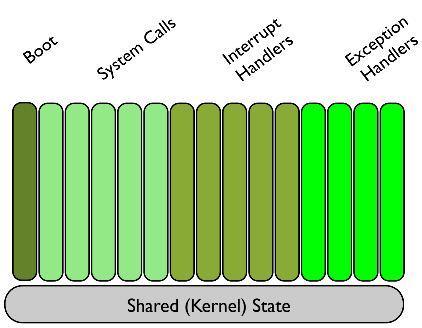
```
        # Add descriptors for hardware irqs: ------------------------------
        .equ    IRQ_BASE,    0x20        # lowest hw irq number

        .irp    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_EXCHANGEREGS,  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_PROCCONTROL,   processorControl,  err=NOERR, dpl=3
intr    INT_MEMCONTROL,    memoryControl,     err=NOERR, dpl=3
intr    INT_SYSTEMCLOCK,   systemClock,       err=NOERR, dpl=3
```

7

Overall kernel structure



8

An example exception handler

```
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);
}
```

9

The KIP

What's in the KIP?

~	Schedule *SC*	ThreadSwitch *SC*	*Reserved*	+F0 / +1E0
ExchangeRegisters *SC*	Unmap *SC*	Lipc *SC*	Ipc *SC*	+E0 / +1C0
MemoryControl *pSC*	ProcessorControl *pSC*	ThreadControl *pSC*	SpaceControl *pSC*	+D0 / +1A0
ProcessorInfo	PageInfo	ThreadInfo	ClockInfo	+C0 / +180
ProcDescPtr	BootInfo	~		+B0 / +160
KipAreaInfo	UtcbInfo	VirtualRegInfo	~	+A0 / +140
~				+90 / +120
~				+80 / +100
~				+70 / +E0
~				+60 / +C0
~		MemoryInfo	~	+50 / +A0
~				+40 / +80
~				+30 / +60
~				+20 / +40
~				+10 / +20
KernDescPtr	API Flags	API Version	$0_{(0/32)}$ 'K' 230 '4' 'L'	+0
+C / +18	+8 / +10	+4 / +8	+0	

kip.S

```
                .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

Thread Ids

- User programs can reference other threads using thread ids

global thread ID	thread no $_{(18/32)}$	version$_{(14/32)}$ $\neq 0 \pmod{64}$
global interrupt ID	intr no $_{(18/32)}$	1 $_{(14/32)}$
local thread ID	local id/64 $_{(26/58)}$	000000
nilthread	0 $_{(32/64)}$	
anythread	-1 $_{(32/64)}$	
anylocalthread	-1 $_{(26/58)}$	000000

```
/*-----------------------------------------------------------------------
 * Thread Ids:
 *---------------------------------------------------------------------*/
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

Flexpages (fpages)

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

fpage $(b, 2^s)$	$b/2^{10}$ $_{(22/54)}$	s $_{(6)}$	$0\,r\,w\,x$

- 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):

complete	0 $_{(22/54)}$	$s = 1$ $_{(6)}$	$0\,r\,w\,x$
nilpage	0 $_{(32/64)}$		

- Can be represented, in practice, using collections of 4KB and 4MB pages
- If two flexpages overlap, then one includes the other

Flexpage implementation

```
/*-------------------------------------------------------------------------
 * 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

```
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 `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:

```
void g1() {    // 1 suffix because this function
               // allocates a page
   f();
   void* p = allocPage1();
   ...
 }
```

- But now suppose f() takes the form:

```
void f() {
  if (availPages(1)) { … allocPage1(); … }
}
```

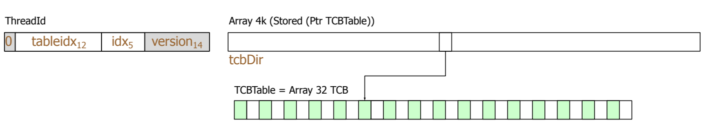
- Pork still uses this naming convention, but relies on "disciplined use"
- Maybe a type system could do better … ?

Thread Control Blocks

Thread control blocks (TCBs)

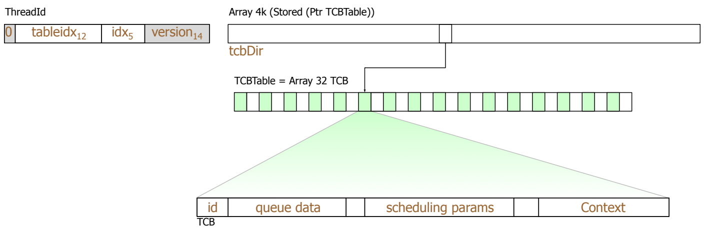
```
struct TCB {
  ThreadId       tid;           // this thread's id and version number
  byte           status;        // thread status
  byte           prio;          // thread priority
  byte           padding;
  byte           count;         // for gc of TCBs in kernel memory
  struct UTCB*   utcb;          // 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;
};
```


Thread control blocks (TCBs)




```
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;
}

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

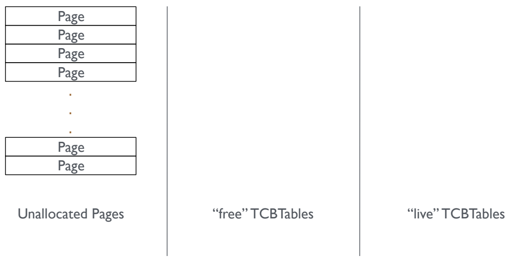
```
struct TCB* allocTCB1(ThreadId tid, struct Space* space, ThreadId scheduler) {
  unsigned    threadNo = threadNo(tid);
  TCBTable*   tab      = tcbDir[threadNo>>TCBDIRBITS];
  if (!tab) {
    tab = tcbDir[threadNo>>TCBDIRBITS] = (TCBTable*)allocPage1();
  }
  ++tab[0]->count;  // Count an additional 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;
  tcb->prio       = 128;       // Default is unspecified
  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;
}
```


Allocating/Freeing TCBs

- Primitives: `allocTCB1()` and `destroyTCB()`
- Every `TCB` contains a `count` field
- Reference count `TCBtables` using the `count` field for the first `TCB` in the table
- New `TCBTables` obtained via `allocPage1()`
- Empty `TCBTables` recycled using `freePage()`
- Lazy initialization (page already zeroed)

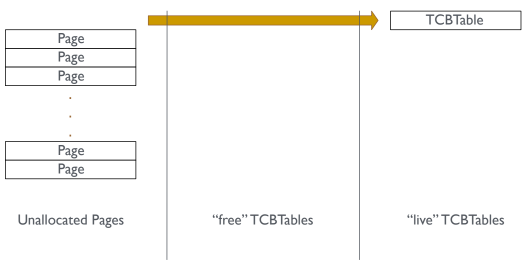
Type Safety?

- To maintain type safety if we were doing this in Habit:



Type Safety?

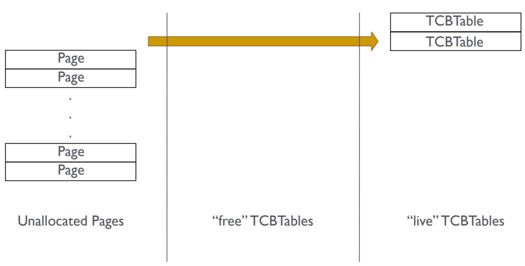
- To maintain type safety if we were doing this in Habit:



- must be fully initialized at allocation time …

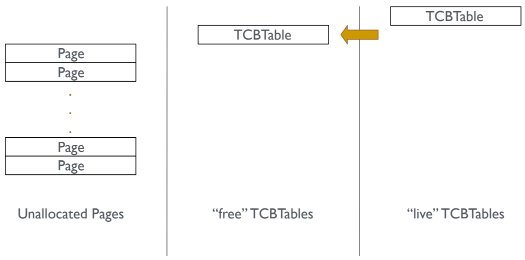
Type Safety?

- To maintain type safety if we were doing this in Habit:



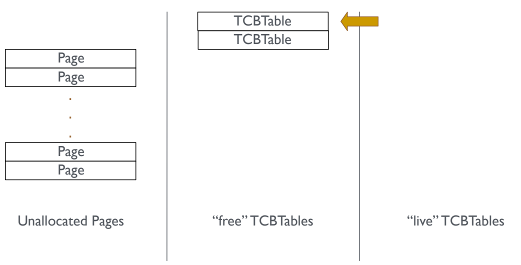
Type Safety?

- To maintain type safety if we were doing this in Habit:



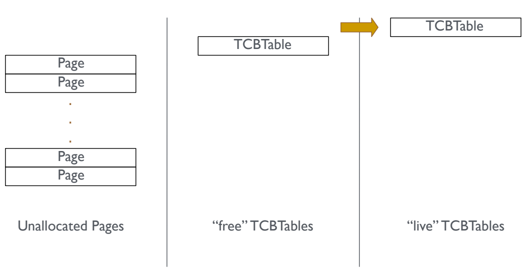
Type Safety?

- To maintain type safety if we were doing this in Habit:



Type Safety?

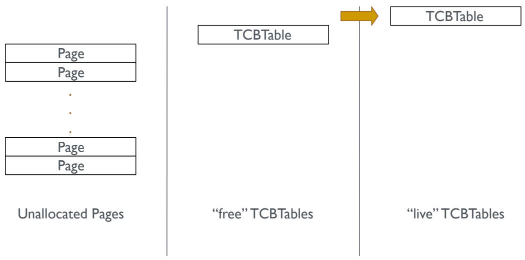
- To maintain type safety if we were doing this in Habit:



- Pages are never truly freed …

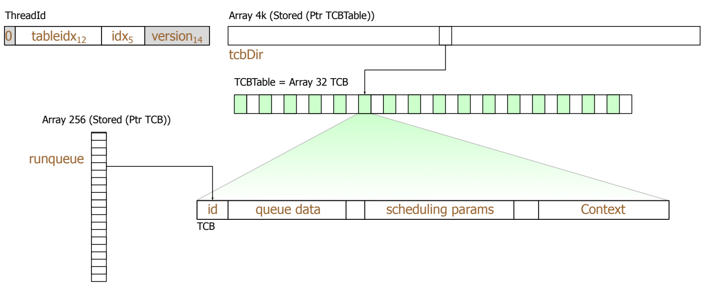
Type Safety?

- To maintain type safety if we were doing this in Habit:

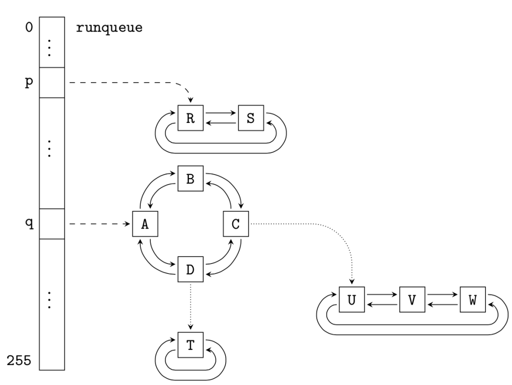


- … unless we have garbage collection … ?

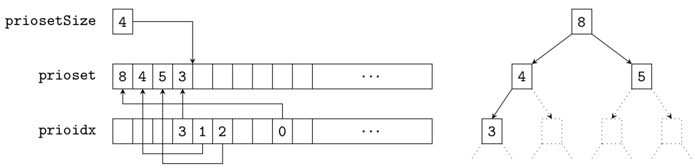
Thread Control Blocks (TCBs)



Scheduling data structures: runqueue



Scheduling data structures: prioset




```
/*-------------------------------------------------------------------------
 * 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);
}
```


Switching to a new thread

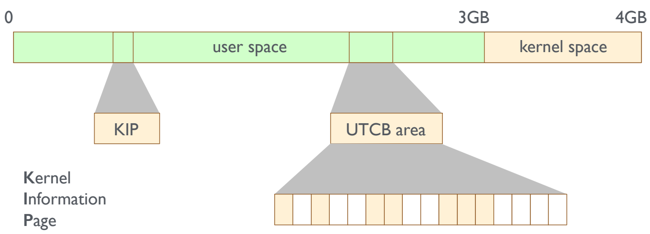
```
static void inline switchTo(struct TCB* tcb) {
  struct Context* ctxt = &(tcb->context);
DEBUG(printf("Switching to thread %x (tcb=%x)\n", tcb->tid, tcb);)
  current  = tcb;                    // Change current thread
  *utcbptr = tcb->vutcb              // Change UTCB address
             + (unsigned)&(((struct UTCB*)0)->mr[0]);
DEBUG(printf("set utcbptr to %x\n", *utcbptr);)
  esp0     = (unsigned)(ctxt + 1); // Change esp0
DEBUG(extern void showSpace(struct Space* space);)
DEBUG(showSpace(tcb->space);)
  switchSpace(tcb->space);          // Change address space
DEBUG(printf("switched space, context is at %x\n\n", ctxt);)
  returnToContext(ctxt);
}
```


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);
}

...

void switchSpace(struct Space* space) {
  if (space->pdir) {                 // No switch for kernel/inactive threads
    if (currentSpace!=space) {
      currentSpace = space;
      setPdir(currentSpace->pdir);
      currentSpace->loaded = 1;
    } else {
      refreshSpace();
    }
  }
}
```

Address Spaces

Address space layout



Kernel
Information
Page

(mapped in to every address space)

User
Thread
Control
Block

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

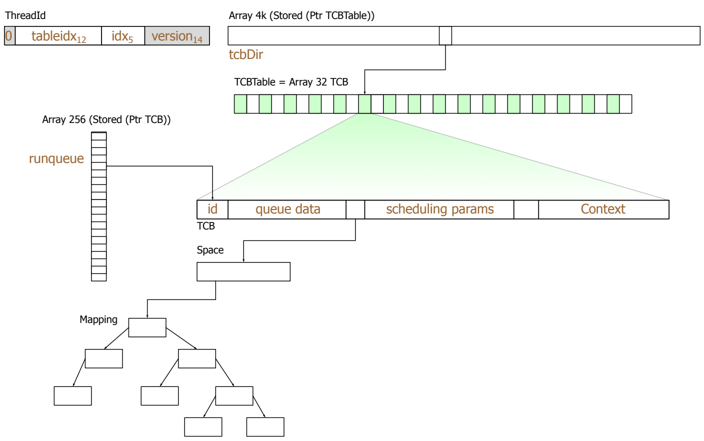
Representing address spaces

```
struct Space {                  // Structure known only in this module
  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
};

...

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

...

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



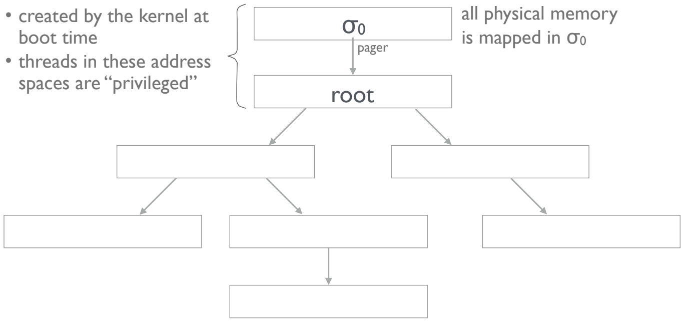
Representing mappings

```
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

The "recursive address space model"

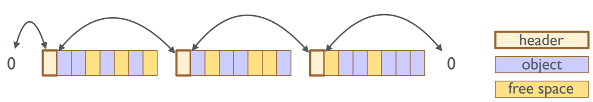
- created by the kernel at boot time
- threads in these address spaces are "privileged"



- In a dynamic system, we need the ability to revoke a previous mappings … this will get interesting …

Small Objects:

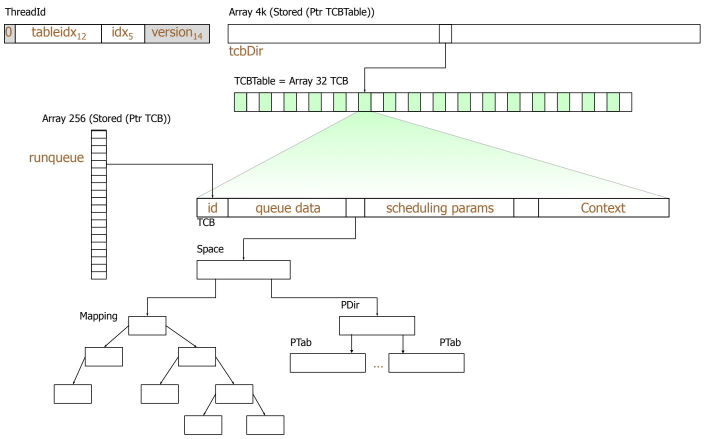
- Pork uses only two "small" object types (≤32 bytes):
  - Address space descriptors (Space)
  - Mapping descriptors (Mapping)
- Kernel allocates/frees pages to store small objects (each page can store up to 127 objects)
- Pages with free slots are linked together



Small Objects, continued:

- Issues:
  - Fragmentation: Compacting GC is probably feasible but possibly expensive
  - Denial of service/interference between spaces: Could have a separate pool of object pages for each address space …
- Costs?
  - Current implementation for allocating/freeing small objects takes ~70 lines, including header structure declaration
- For memory safety: separate pools of Space and Mapping objects …

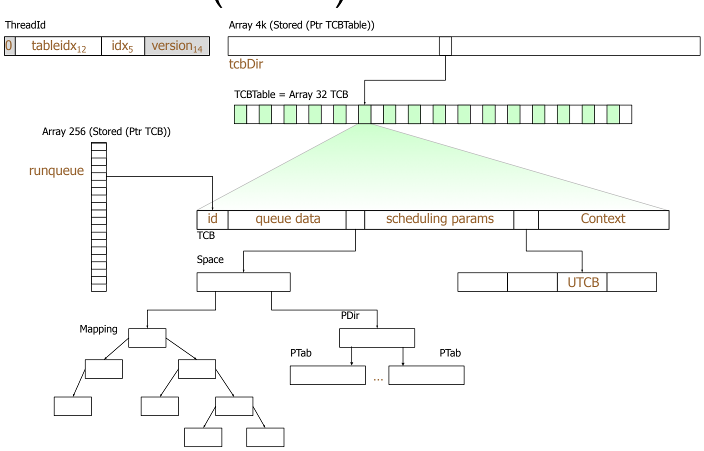
Page Dirs and Page Tables:



Page Dirs and Page Tables:

- Page directories and page tables use full pages
- Aligned pointers take fewer than 32 bits; how can this be tracked in the type system?
- Physical rather than virtual addresses
- Allocation and deallocation controlled via reference counts in Space structures

User TCBs (UTCBs):



User TCBs (UTCBs):

- Mapped into kernel space, also fully accessible in user space
- Just bits: No pointers, references, indexes
- Could be recycled into the pool of kernel pages
- Currently freed only when all active threads in a space have terminated

IPC

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 1)

```
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:
      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;
      }
    }
  }
  ...
```


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:

MRs ⟹ MRs

MsgTag **[MR$_0$]**

label $_{(16/48)}$	flags $_{(4)}$	t $_{(6)}$	u $_{(6)}$

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


Example: IPCs from hardware interrupts

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

  // An irq thread may be Halted, Sending (i.e., waiting for the user level
  // handler to receive notice of the interrupt), or Receiving (i.e., waiting
  // for the user level handler to finish processing the interrupt and send
  // an acknowledgement).  In theory, an interrupt can only occur if it is
  // unmasked at the PIC, and that, in turn, should only be possible when
  // (1) the corresponding irq thread is Halted; and (2) the "pager" for
  // the irq thread (stored in the vutcb field) is set to a non-nilthread id.

  if (irqTCB->status==Halted && irqTCB->vutcb!=nilthread) {
    if (sendPhase(Interrupt, irqTCB, irqTCB->vutcb)) {
      irqTCB->status = Receiving(Interrupt) | Halted;
    }
  }
  reschedule(); // 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:

From Interrupt Thread

$-1_{(12/44)}$	$0_{(4)}$	$0_{(4)}$	$t = 0_{(6)}$	$u = 0_{(6)}$	MR_0

- 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

$0_{(16/48)}$	$0_{(4)}$	$t = 0_{(6)}$	$u = 0_{(6)}$	MR_0

Interrupt handler protocol

`Interrupt` $\Longrightarrow$ `MRs`

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

From Interrupt Thread

$-1_{(12/44)}$	$0_{(4)}$	$0_{(4)}$	$t = 0_{(6)}$	$u = 0_{(6)}$	MR_0

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


Interrupt handler protocol

`MRs` $\Longrightarrow$ `Interrupt`

```
case Interrupt :   // Receive a response from an interrupt handler
  if (mask(sutcb->mr[0],12)==0) {
    ASSERT(mask(recv->tid, VERSIONBITS)==1, "Wrong irq version");
    ASSERT(threadNo(recv->tid) < NUMIRQs,   "IRQ out of range");
    enableIRQ(threadNo(recv->tid));   // Reenable interrupt
    return NoError;
  }
  break;
```

- 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

$0_{(16/48)}$	$0_{(4)}$	$t = 0_{(6)}$	$u = 0_{(6)}$	MR_0

Example: IPCs from page faults

```
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

faulting user-level IP $_{(32/64)}$					MR $_2$
fault address $_{(32/64)}$					MR $_1$
$-2_{\ (12/44)}$	$0\,r\,w\,x$	$0_{\ (4)}$	$t=0_{\ (6)}$	$u=2_{\ (6)}$	MR $_0$

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

From Pager

MapItem / GrantItem				MR $_{1,2}$
$0_{\ (16/48)}$	$0_{\ (4)}$	$t=2_{\ (6)}$	$u=0_{\ (6)}$	MR $_0$

Page fault protocol

`PageFault` $\Longrightarrow$ `MRs`

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

To Pager

faulting user-level IP $_{(32/64)}$					MR $_2$
fault address $_{(32/64)}$					MR $_1$
$-2_{\ (12/44)}$	$0\,r\,w\,x$	$0_{\ (4)}$	$t=0_{\ (6)}$	$u=2_{\ (6)}$	MR $_0$

```
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

MRs $\Longrightarrow$ PageFault

```
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;
```

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

From Pager

MapItem / GrantItem				$\text{MR}_{1,2}$
$0_{(16/48)}$	$0_{(4)}$	$t = 2_{(6)}$	$u = 0_{(6)}$	MR_0

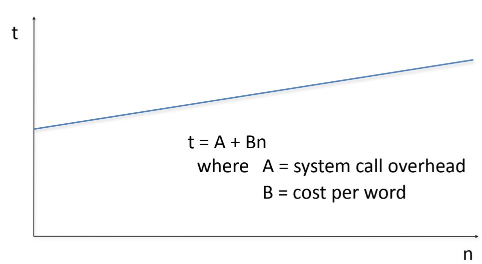
Let’s poke around … !

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

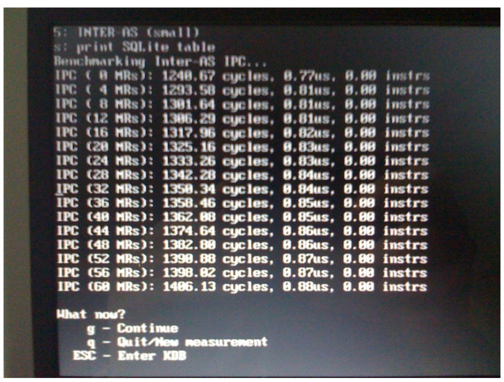
Expected Performance Model



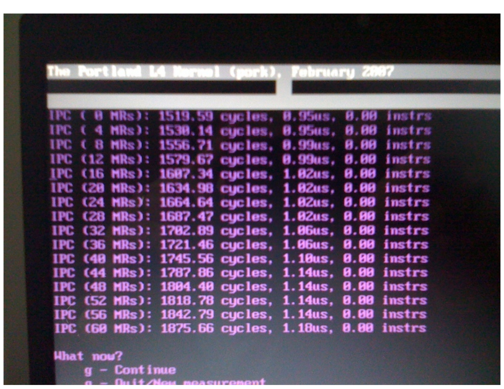
Test Platform

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

Pistachio “Output”



Pork “Output”

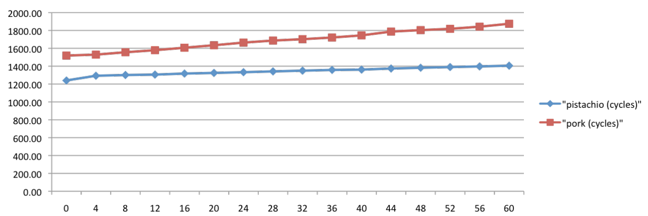


Transcribed Data (Inter-AS)

ping pong	pistachio Inter-AS IPC		pork Inter-AS IPC		Ratio, pork/pistachio	
#MRs	cycles	microseconds	cycles	microseconds	cycles	microseconds
0	1240.67	0.77	1519.59	0.95	1.22	1.23
4	1293.58	0.81	1530.14	0.95	1.18	1.17
8	1301.64	0.81	1556.71	0.99	1.20	1.22
12	1306.29	0.81	1579.67	0.99	1.21	1.22
16	1317.96	0.82	1607.34	1.02	1.22	1.24
20	1325.16	0.83	1634.98	1.02	1.23	1.23
24	1333.26	0.83	1664.64	1.02	1.25	1.23
28	1342.28	0.84	1687.47	1.02	1.26	1.21
32	1350.34	0.84	1702.89	1.06	1.26	1.26
36	1358.46	0.85	1721.46	1.06	1.27	1.25
40	1362.08	0.85	1745.56	1.10	1.28	1.29
44	1374.64	0.86	1787.86	1.14	1.30	1.33
48	1382.80	0.86	1804.40	1.14	1.30	1.33
52	1390.88	0.87	1818.78	1.14	1.31	1.31
56	1398.02	0.87	1842.79	1.14	1.32	1.31
60	1406.13	0.88	1875.66	1.18	1.33	1.34

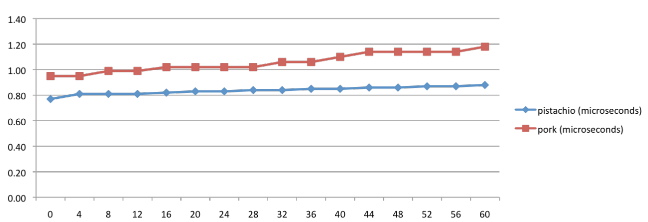
Inter-AS = “ping” and “pong” in different address spaces

Cycles (Inter-AS)

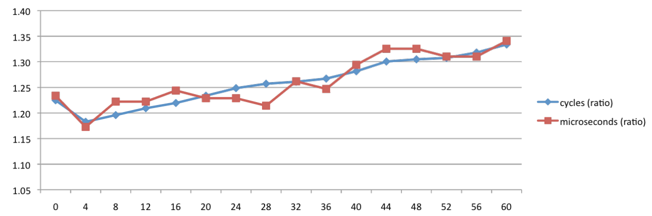


pistachio = 1274.66 + 2.27n (least squares)
pork = 1512.57 + 6n

Microseconds (Inter-AS)



Pork : Pistachio (Inter-AS)

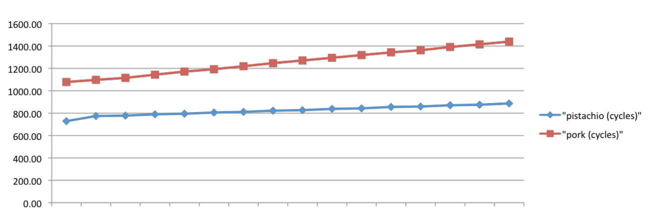


Transcribed Data (Intra-AS)

ping pong	pistachio Intra-AS IPC		pork Intra-AS IPC		Ratio, pork/pistachio	
#MRs	cycles	microseconds	cycles	microseconds	cycles	microseconds
0	729.19	0.45	1078.71	0.68	1.48	1.51
4	774.74	0.48	1097.90	0.68	1.42	1.42
8	778.49	0.48	1115.55	0.72	1.43	1.50
12	790.04	0.49	1143.99	0.72	1.45	1.47
16	795.65	0.49	1171.99	0.72	1.47	1.47
20	806.12	0.50	1193.23	0.76	1.48	1.52
24	811.85	0.50	1219.75	0.76	1.50	1.52
28	822.54	0.51	1247.19	0.76	1.52	1.49
32	827.20	0.51	1271.19	0.80	1.54	1.57
36	838.69	0.52	1295.20	0.80	1.54	1.54
40	843.37	0.52	1319.39	0.83	1.56	1.60
44	855.89	0.53	1343.43	0.83	1.57	1.57
48	859.57	0.53	1363.04	0.87	1.59	1.64
52	871.08	0.54	1391.45	0.87	1.60	1.61
56	875.72	0.54	1415.61	0.91	1.62	1.69
60	887.38	0.55	1439.58	0.91	1.62	1.65

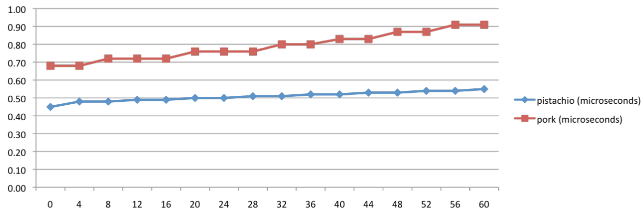
Intra-AS = “ping” and “pong” in same address space

Cycles (Intra-AS)

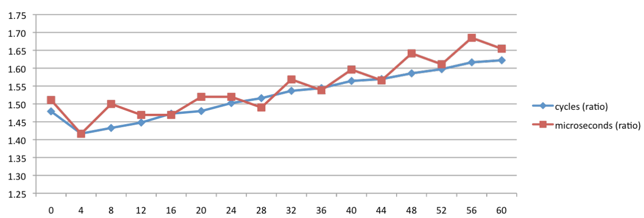


pistachio = 756.54 + 2.21n (least squares)
pork = 1073.54 + 6.11n

Microseconds (Intra-AS)



Pork : Pistachio (Intra-AS)



Estimating Clock Frequency

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

Inter-AS

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

Intra-AS

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

Summary

Comparison	Range
Pork/Pistachio (Inter-AS)	1.17 – 1.35
Pork/Pistachio (Intra-AS)	1.42 – 1.65
Inter-AS/Intra-AS (Pork)	1.58 – 1.70
Inter-AS/Intra-AS (Pistachio)	1.30 – 1.40

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

Performance Tuning Opportunities?

- Are there opportunities for performance-tuning pork to reduce the gap?
- Inter-AS: pistachio = 1274.66 + 2.27n (least squares)
  pork = 1512.57 + 6n
- Intra-AS: pistachio = 756.54 + 2.21n (least squares)
  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:

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

- Machine Code:

```
209: ba 01 00 00 00              mov   $0x1,%edx                  initialization

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
                                                                  loop
```

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 */
        "=S"(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:

initialization

```
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)
```

loop

Reflections

- In this case, the performance differences between pork and pistachio can be understood and 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.