CS 410/510

Languages & Low-Level Programming

Mark P Jones
Portland State University

Fall 2018

Week 5: Case Study - The L4 Microkernel

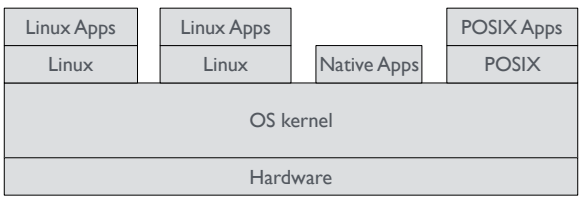
Copyright Notice

From ad-hoc to generic

- So far, we've been building bare-metal applications in an ad-hoc manner
- … which would be reasonable in a custom embedded system
- …. but what if we want a more generic, reusable foundation for building and deploying computer systems?
- (also known as an "operating system" 😀)
- Let's take a look at L4 as an initial case study …

Why L4?

Context …



In the "Programatica" project, we were looking to build a OS kernel with very high assurance of separation between domains

Approaches to Kernel Design

- In a **monolithic kernel**, all OS code runs in kernel mode
  - improves performance; reduces reliability
- A **microkernel** design aims to minimize the amount of code that runs in kernel mode (the "*trusted computing base*" or TCB) and implement as much functionality as it can in "*user level servers*"
  - A microkernel must abstract physical memory, CPU (threads), and interrupts/exceptions
  - A microkernel must also provide (efficient) mechanisms for communication and synchronization
  - A microkernel should be "policy free"

Microkernel design: L4

- L4 is a "second generation" µ-kernel design, originally designed by Jochen Liedtke
- Designed to show that µ-kernel based systems are usable in practice with good performance
- Minimalist philosophy: If it can be implemented outside the kernel, it doesn't belong inside

Why pick L4?

- L4 is industrially and technically **relevant**
  - Multiple working implementations (Pistachio, Fiasco, OKL4, etc…)
  - Multiple supported architectures (ia32, arm, powerpc, mips, sparc, …)
  - Already used in a variety of domains, including real-time, security, virtual machines & monitors, etc…
  - Open Kernel Labs spin-off from NICTA & UNSW
  - Commercial use by Qualcomm and others …

Why pick L4?

- L4 is industrially and technically **relevant**
- L4 is small enough to be **tractable**
  - Original implementation ~ 12K executable
  - Recent/portable/flexible implementations ~ 10-20 KLOC C++
  - Much easier to implement that a full POSIX OS, for example!

Why pick L4?

- L4 is industrially and technically **relevant**
- L4 is small enough to be **tractable**
- L4 is real enough to be **interesting**
  - For example, we can run multiple, separated instances of Linux (specifically: L4Linux, Wombat) on top of an L4 µ-kernel
  - Use somebody else's POSIX layer rather than build our own!
  - Detailed specification documents are available

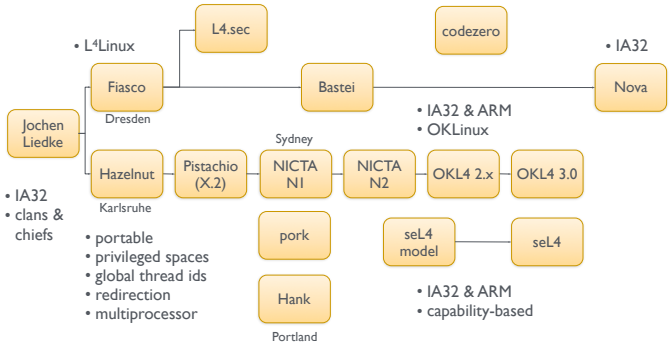
Why pick L4?

- L4 is industrially and technically **relevant**
- L4 is small enough to be **tractable**
- L4 is real enough to be **interesting**
- L4 is a good **representative** of the target domain and a good tool for exposing core research challenges
  - Threads, address spaces, IPC, preemption, interrupts, etc… are core µ-kernel concepts, regardless of API details
  - It should be possible to retarget to a different API or µ-kernel design

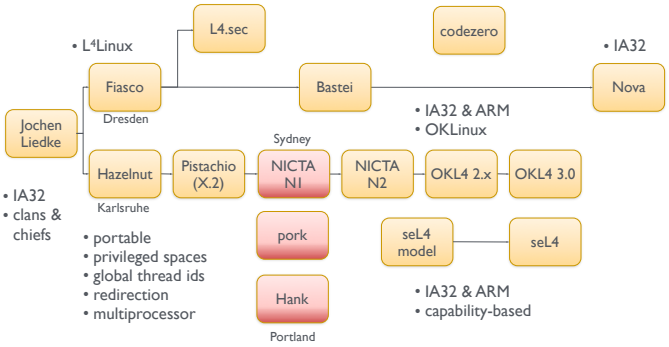
Why pick L4?

- L4 is industrially and technically **relevant**
- L4 is small enough to be **tractable**
- L4 is real enough to be **interesting**
- L4 is a good **representative** of the target domain and a good tool for exposing core research challenges
- L4 is **"not invented here"**
  - We're not in the business of OS design and implementation
  - Leverage the insights and expertise of the OS community so that we can focus on our own research goals
  - A credibility boost, showing that our methods apply to other people's problems (we can't change the OS design to make our lives easier …)

Evolution of L4

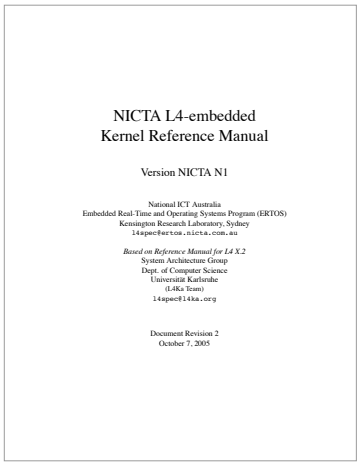


Evolution of L4 - Case Study 1



NICTA N1

- For concreteness, this presentation will be based (mostly) on the NICTA N1 version of the L4 spec
- Available in reference section of D2L course content
- (primary reference for pork)
- Lots of diagrams of bitdata and memory area structures
- … implications for language design?



NICTA L4-embedded
Kernel Reference Manual

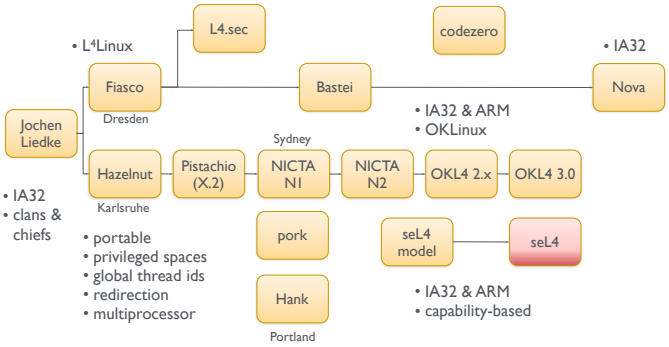
Version NICTA N1

National ICT Australia
Embedded Real-Time and Operating Systems Program (ERTOS)
Kensington Research Laboratory, Sydney
l4spec@ertos.nicta.com.au

Based on Reference Manual for L4 X.2
System Architecture Group
Dept. of Computer Science
Universität Karlsruhe
(L4Ka Team)
l4spec@l4ka.org

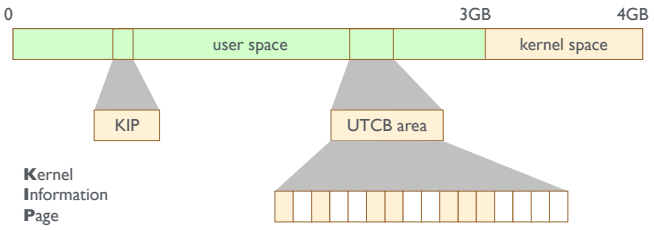
Document Revision 2
October 7, 2005

Evolution of L4 - Case Study 2



Address Space Layout

Userspace perspective



Kernel
Information
Page

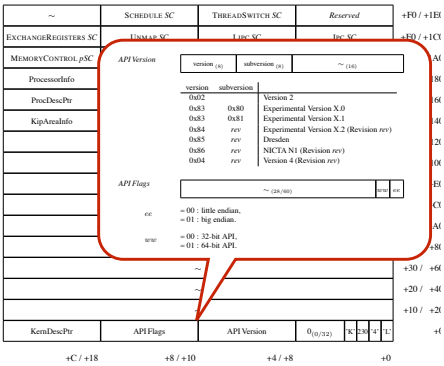
(mapped in to every address space)

User
Thread
Control
Block

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

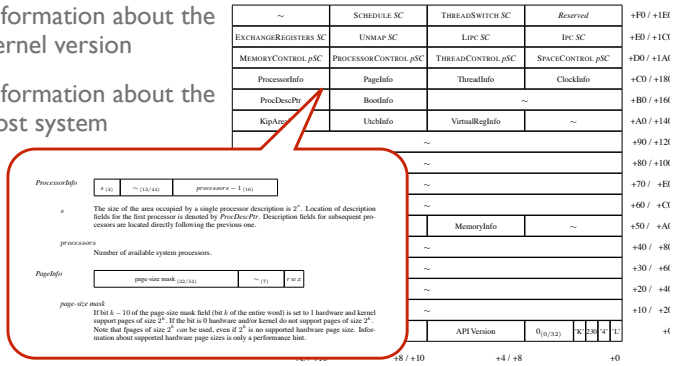
What's in the KIP?

- Information about the kernel version



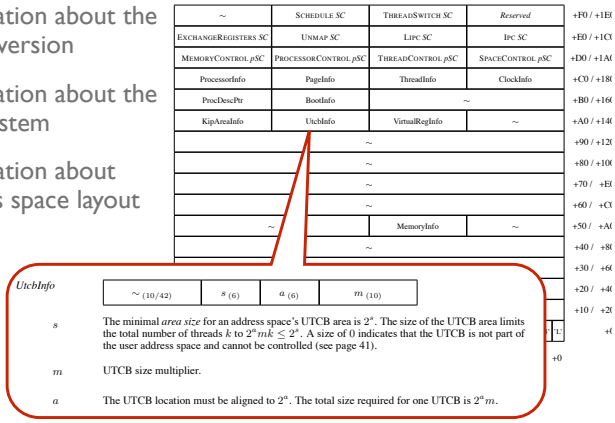
What's in the KIP?

- Information about the kernel version
- Information about the host system



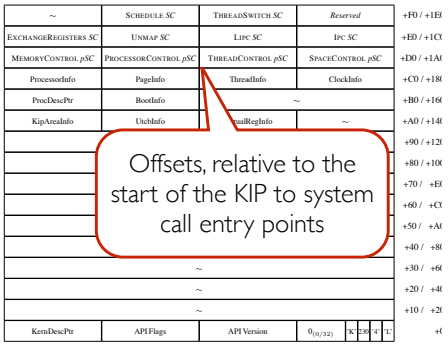
What's in the KIP?

- Information about the kernel version
- Information about the host system
- Information about address space layout



What's in the KIP?

- Information about the kernel version
- Information about the host system
- Information about address space layout
- System call entry points
- So how can a user process find the KIP address?



How to find the KIP

- Option 1: Design protocol
  - User code assumes a predetermined KIP address
- Option 2: "Slow system call" … a "virtual" instruction
  - User code executes the illegal instruction LOCK NOP
  - This triggers an illegal opcode exception, which enters the kernel
  - The kernel checks for this exception, loads the kip address in to the context registers, and returns to user mode

–	EAX	– **KernelInterface** →	EAX	*base address*
–	ECX		ECX	*API Version*
–	EDX		EDX	*API Flags*
–	ESI	lock: nop	ESI	*Kernel ID*
–	EDI		EDI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	≡

What are the gaps for?

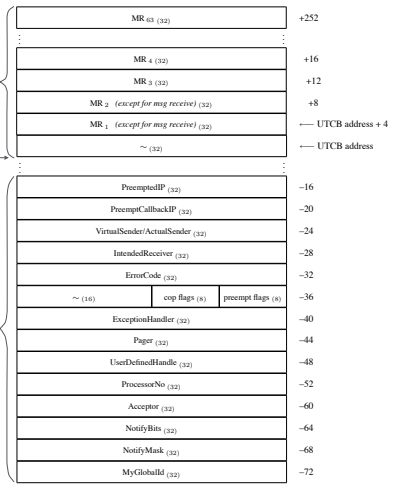
~	Schedule SC	ThreadSwitch SC	*Reserved*	+F0 / +1E0
ExchangeRegisters SC	Unmap SC	Lipc SC	Ipc SC	+E0 / +1C0
MemoryControl pSC	ProcessorControl pSC	ThreadControl pSC	SpaceControl pSC	+D0 / +1A0
ProcessorInfo	PageInfo	ThreadInfo	ClockInfo	+C0 / +180
ProcDescPtr	BootInfo	~		+B0 / +160
KipAreaInfo	UtcbInfo	VirtualRegInfo	~	+A0 / +140
~				+90 / +120
~				+80 / +100
~				+70 / +E0
~				+60 / +C0
Kdebug.config1	~ Kdebug.config0	**MemoryInfo**	~	**+50 / +A0**
root server.high	root server.low	~ root server.IP	root server.SP	**+40 / +80**
σ_1.high	σ_1.low	~ σ_1.IP	σ_1.SP	**+30 / +60**
σ_0.high	σ_0.low	~ σ_0.IP	σ_0.SP	**+20 / +40**
Kdebug.high	Kdebug.low	~ Kdebug.entry	Kdebug.init	**+10 / +20**
KernDescPtr	API Flags	API Version	0 (0/32) 'K' 230 4 'L'	+0
+C / +18	+8 / +10	+4 / +8	+0	

What's in the UTCB area?

- Every user thread has a User Thread Control Block (UTCB), which is a block of memory that the thread uses for communication with the kernel.
- The UTCB contains:
  - Message registers (MRs)
  - Thread control registers (TCRs)
- All UTCBs for a given address space are grouped in a single block called the UTCB area
- Example: If UTCBs are 512 bytes long, then an address space with a 4KB UTCB area can support at most 8 threads

UTCB Layout (IA32)

- 64 Message "registers" named MR_0, MR_1, …, MR_{63}
- Miscellaneous other fields:
  - ErrorCode
  - ExceptionHandler
  - Pager
  - Acceptor
  - …
- UTCB address points to the middle of the UTCB

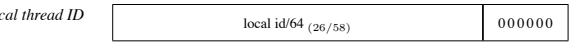


Trust, and UTCBs

- User processes can read and write whatever values they like in the UTCB (and in the UTCBs of other threads in the same address space)
- Protected thread parameters (e.g., priority) must be stored in a separate TCB data structure that is only accessible to the kernel
- Any data that is read from the UTCB cannot be trusted and must be validated by the kernel, as necessary, before use
- Mappings for the UTCB area must be created by the kernel (otherwise user space code could cause the kernel to page fault by reading from an unmapped UTCB)

UTCB addresses and local thread ids

- Every UTCB must be 64-byte aligned, so the lower 6 bits in any UTCB address will be zero
- Within a given address space, UTCB addresses are used as local thread ids:



- Other thread ids must have a nonzero value in their least significant 6 bits

How to find the UTCB

- Option 1: Design Protocol
  - User code assumes a predetermined UTCB address
- Option 2: The UTCB pointer
  - At boot time, the kernel creates a 4 byte, read only segment in the GDT for a specific kernelspace address and loads a corresponding segment selector in `%gs`
  - The kernel stores the UTCB address of the current thread in that location
  - User code can read the UTCB address from `%gs:0`

Configuring an address space

- The addresses of the KIP and the UTCB can be set when a new address space is created:
- First, create a new thread in a new address space (we'll see how this is done soon)
- Now use the (privileged) SpaceControl system call:

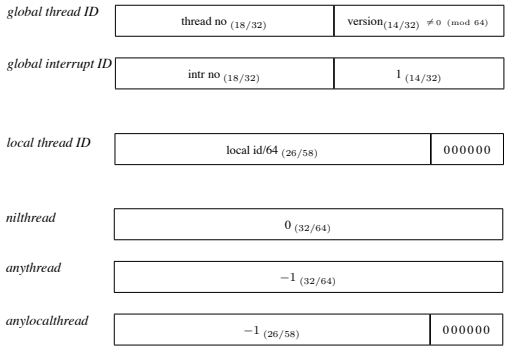
SpaceSpecifier	EAX	− **Space Control** →	EAX	*result*
control	ECX		ECX	*control*
KernelInterfacePageArea	EDX		EDX	~
UtcbArea	ESI	call *SpaceControl*	ESI	~
–	EDI		EDI	~
–	EBX		EBX	~
–	EBP		EBP	~
–	ESP		ESP	≡

- Threads cannot be activated (made runnable) until the associated address space has been configured in this way

Threads

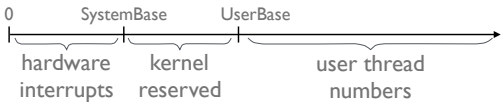
Thread Ids

- User programs can reference other threads using thread ids



Thread numbers

- Every thread number falls in to one of three ranges:



- The SystemBase and UserBase values are defined in the KIP
- Key insight: L4 translates hardware interrupts in to messages from (special) threads

Global ids bad …

- The reliance on global ids is one of the weaknesses of the original L4 design
  - Any thread can reference any other thread by using its global id
  - Any thread can interfere with another thread (e.g., a denial of service attack) by using its global id
  - Even if thread ids are not officially published, they can still be guessed or faked
- We could avoid these problems if there were a way to ensure that any thread only had the **capability** to access a specific set of authorized threads …

ThreadControl

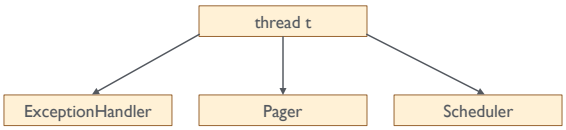
- New threads are created using the (privileged) ThreadControl system call:

dest	EAX	– **Thread Control** →	EAX	*result*
Pager	ECX		ECX	∼
Scheduler	EDX		EDX	∼
SpaceSpecifier	ESI	call *ThreadControl*	ESI	∼
UtcbLocation	EDI		EDI	∼
–	EBX		EBX	∼
–	EBP		EBP	∼
–	ESP		ESP	≡

- If dest does not exist then the new thread is created in the same address space as SpaceSpecifier
  - If SpaceSpecifier=dest, then a new address space is created
  - The UTCBLocation must be within the UTCB area
- If dest exists and SpaceSpecifier is nilthread, then the thread is deleted

Exception handlers, pagers, and schedulers

- Every thread has three associated threads



- The **exception handler** is responsible for dealing with any exceptions that t generates (specified in UTCB)
- The **pager** is responsible for dealing with any page faults that t generates (specified in UTCB)
- The **scheduler** is responsible for setting the priority and timeslice for t (hidden inside kernel TCB)

Schedule

- If s is the scheduler thread for t, then s can set t's scheduling parameters using the Schedule system call:

dest	EAX	– **Schedule** →	EAX	*result*
prio	ECX		ECX	∼
processor control	EDX		EDX	∼
preemption control	ESI	call *Schedule*	ESI	∼
ts len	EDI		EDI	*rem ts*
total quantum	EBX		EBX	*rem total*
–	EBP		EBP	∼
–	ESP		ESP	≡

- The specified priority cannot be higher than the scheduler's own priority
- ts is the timeslice: how long does the thread run before the kernel will switch to another thread
- quantum specifies a limit on the total time that a thread can run before it is suspended

ThreadSwitch

- A thread can give up any remaining part of its timeslice to another thread using the ThreadSwitch system call:

dest	EAX	– **ThreadSwitch** →	EAX	≡
–	ECX		ECX	≡
–	EDX		EDX	≡
–	ESI	call *ThreadSwitch*	ESI	≡
–	EDI		EDI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	≡

- If dest is nilthread, then the caller still yields the CPU and the kernel determines which thread will run next …

ExchangeRegisters

- A thread can read or write parameters of another thread using the ExchangeRegisters system call:

dest	EAX	– **Exchange Registers** →	EAX	*result*
control	ECX		ECX	*control*
SP	EDX		EDX	*SP*
IP	ESI	call *ExchangeRegisters*	ESI	*IP*
FLAGS	EDI		EDI	*FLAGS*
UserDefinedHandle	EBX		EBX	*UserDefinedHandle*
pager	EBP		EBP	*pager*
–	ESP		ESP	≡

- ExchangeRegisters is not "privileged" … but the destination thread must be in the same address space as the caller
- The exact effects of an ExchangeRegisters call are specified by a bit map in the control word:

control

from (18/32)	0 (3/19)	*r d h p u f i s S R H*

IPC

IPC - Interprocess Communication

- IPC is a fundamental system call for communication between threads in L4
- A typical use of IPC proceeds as follows:
  - Load the message registers in the UTCB with a message to send
  - Invoke the IPC system call, which has two phases:
    - Send the message register values to a specified thread
    - Receive new message register values from a thread
  - Resume thread that initiated the IPC

Why combine send and receive phases?

- The combination of send and receive phases in a single system call:
  - requires only one system call instead of separate send and receive system calls
  - accomplishes both send and receive actions with only a single transition in to kernel mode
  - matches common communication idioms:
    - RPC: Send a request to a thread and wait for its reply
    - Server: Send response to a previous request and then wait for a new request to arrive

Synchronization and blocking

- Communication between threads requires a sender and a receiver
  - If either party is not ready, then the communication blocks
- Some versions of L4 allow an IPC call to specify timeout periods, after which a blocked IPC call will be aborted.
  - In practice, it is hard to come up with a good methodology for picking sensible timeout values
- Other versions of L4 support only two possible timeout options: 0 (non blocking) and ∞ (blocking)

Specifics

to	EAX	− **Ipc** →	EAX	*from*
−	ECX		ECX	∼
FromSpecifier	EDX		EDX	∼
MR_0	ESI	call *Ipc*	ESI	MR_0
UTCB	EDI		EDI	≡
−	EBX		EBX	MR_1
−	EBP		EBP	MR_2
−	ESP		ESP	≡

- Some message registers passed in CPU registers
- "to" can be nilthread, if there is no send phase
- "FromSpecified" can be:
  - nilthread, if there is no receive phase
  - anythread, if it is a server that will accept requests from any other thread

Message tags

- The value in MR_0 provides a message tag that describes the structure of the message in the remaining message registers:

MsgTag **[MR0]**

label (16/48)	flags (4)	t (6)	u (6)

- label can be used to send/receive a 16 bit data value
- u is the number of untyped words (uninterpreted 32 bit word values) sent in message registers
- t is the number of typed-item words (MapItem, GrantItem; we'll talk about these soon …)

Example: Interrupt handlers

- 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

Example: Thread start

- When a new thread is constructed, it waits for a message from its pager before starting:

From Pager

Initial SP (32/64)				MR 2
Initial IP (32/64)				MR 1
0 (16/48)	0 (4)	$t = 0$ (6)	$u = 2$ (6)	MR 0

- When a newly created thread receives a message of this form, the kernel loads the specified `esp` and `eip` values from the message in to the thread's context and marks the thread as being runnable …

Example: Exception handling

- When a thread generates an exception, the kernel sends a message to the associated exception handler

EAX (32)					MR 12
ECX (32)					MR 11
EDX (32)					MR 10
EBX (32)					MR 9
ESP (32)					MR 8
EBP (32)					MR 7
ESI (32)					MR 6
EDI (32)					MR 5
ErrorCode (32)					MR 4
ExceptionNo (32)					MR 3
EFLAGS (32)					MR 2
EIP (32)					MR 1
$-4/-5$ (12/44)	0 (4)	0 (4)	$t = 0$ (6)	$u = 12$ (6)	MR 0

- If it chooses to resume the thread that generated the exception, it responds with a message of essentially the same format (possibly having updated registers in the process)

Address Space Management

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

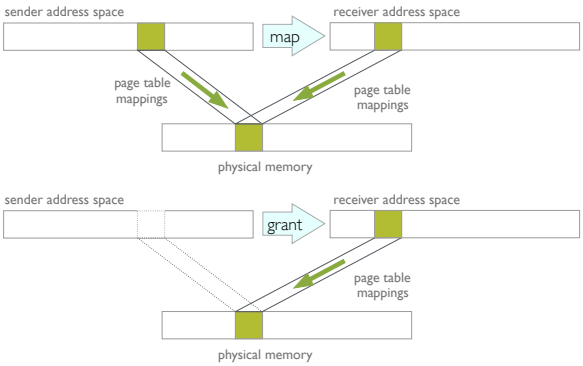
nilpage

0 (32/64)

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

Mapping and granting

- Address spaces in L4 are constructed by mapping or granting regions of memory between address spaces



MapItems and GrantItems

- A MapItem specifies a region of memory in the sender's address space that will be mapped in to the receiver's address space

snd fpage (28/60)	0 r w x	MR_{i+1}
snd base / 1024 (22/54)	0 (6) 1000	MR_i

- A GrantItem specifies a region of memory that will be removed from the sender's address space and added to the receiver's address space

snd fpage (28/60)	0 r w x	MR_{i+1}
snd base / 1024 (22/54)	0 (6) 1010	MR_i

- Base values are used for mapping between fpages of different sizes; we will mostly ignore them for now

Typed items in IPC messages

- An IPC message can contain multiple "typed items" (either MapItem or GrantItem values), that will create mappings in the receiver based on mappings in the sender
- The receiver sets an "acceptor" fpage in its UTCB to specify where newly received mappings should be received
- To receive anywhere, set the acceptor to "complete"
- To receive nowhere, set the acceptor to "nilpage"

Page faults

- 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

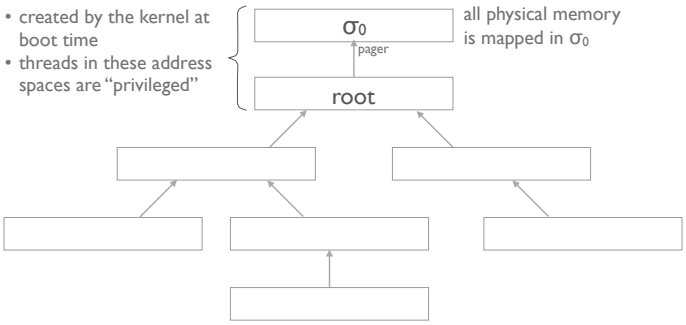
- The pager can respond by sending back a reply with a new mapping … that also restarts the faulting thread:

From Pager

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

The "recursive address space model"

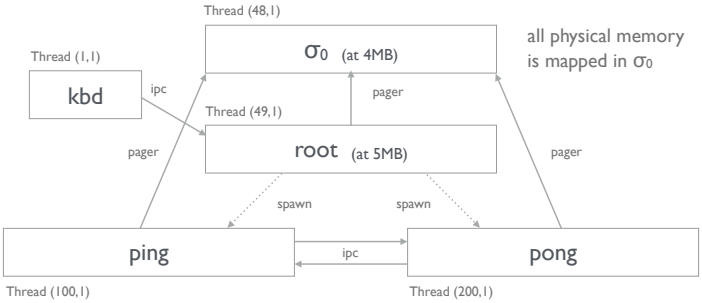
- created by the kernel at boot time
- threads in these address spaces are "privileged"



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

Let's look at an example …

A demo using "pork"



Initialization code in root.c:

```
printf("This is a root server!\n");
showKIP();

ping = L4_GlobalId(100,1);
pong = L4_GlobalId(200,1);

//startPing();
spawn("ping", ping,     // Name & thread id
      0, ping,          // utcbNo & space spec
      L4_Myself(),      // Scheduler
      L4_Pager(),       // Pager
      (L4_Word_t)ping_thread,                  // eip
      ((L4_Word_t)pingstack) + PINGSTACKSIZE); // esp

//startPong();
spawn("pong", pong,     // Name & thread id
      0, pong,          // utcbNo & space spec
      L4_Myself(),      // Scheduler
      L4_Pager(),       // Pager
      (L4_Word_t)pong_thread,                  // eip
      ((L4_Word_t)pongstack) + PONGSTACKSIZE); // esp

//keyboard listener
L4_ThreadId_t keyId  = L4_GlobalId(1, 1);  // Keyboard on IRQ1
L4_ThreadId_t rootId = L4_MyGlobalId();    // My id

printf("keyboard id = %x, my id = %x\n", keyId, rootId);
printf("associate produces %x\n",
       L4_AssociateInterrupt(keyId, rootId));
```


Event loop code in root.c:

```
L4_MsgTag_t tag = L4_Receive(keyId);
for (;;) {
  printf("received msg (tag=%x) from %x\n", tag, keyId);
  if (L4_IpcSucceeded(tag)   &&
      L4_UntypedWords(tag)==0 &&
      L4_TypedWords(tag)  ==0) {
    printf("Scancode = 0x%x\n", inb(0x60));
    L4_LoadMR(0, 0);   // tag: Empty message, ping back to interrupt thread
    tag = L4_Call(keyId);
    printf("root's Call completed ...\n");
  } else {
    printf("Ignoring message/failure, trying again ...\n");
    tag = L4_Receive(keyId);
  }
}
printf("This message won't appear!\n");
```