L4 Programming Introduction
Fundamental Concepts

- **Address Spaces**
  - Unit of protection, resource management.
- **Threads**
  - Execution abstraction and provide unique identifiers.
- **Time**
  - Actual abstraction varies between API versions of L4.

- **Communication: IPC**
  - Synchronous
  - Identification: thread ids
- **AS construction: mapping**
  - Flexpages
    - Architecture independent page abstraction
L4 System Calls

- KernelInterface
- ThreadControl
- ExchangeRegisters
- IPC
- ThreadSwitch
- Schedule
- MapControl
- SpaceControl
- ProcessorControl
- CacheControl

- 10 System Calls
  - N2 API – other API version vary
- 6-8 Kernel defined protocols
Root Task

- First task started at boot time
- Can perform privileged system calls
- Controls access to resources managed by privileged systems calls
  - ThreadControl, SpaceControl, ProcessorControl, MemoryControl
  - Thread allocation, memory attributes, processor modes, etc.
Kernel Information Page

• Kernel memory object located in the address space of a task.
  – Placed on address space creation
    • Location is dictated by `SpaceControl` system call

• Contains information about kernel and hardware
  – Page sizes supported
  – API version
  – Physical memory layout.
  – Syscall Addresses
<table>
<thead>
<tr>
<th>Address</th>
<th>Schedule SC</th>
<th>ThreadSwitch SC</th>
<th>Reserved</th>
<th>ExchangeRegisters SC</th>
<th>LIPC SC</th>
<th>IPC SC</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcessorInfo</td>
<td>PageInfo</td>
<td>ThreadInfo</td>
<td>ClockInfo</td>
<td>ProcessorControl pSC</td>
<td>SPACECONTROL pSC</td>
<td>MAPCONTROL pSC</td>
<td>Reserved</td>
</tr>
<tr>
<td>ProcDescPtr</td>
<td>BootInfo</td>
<td></td>
<td></td>
<td>+C0 / +180</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KipAreaInfo</td>
<td>UtcbInfo</td>
<td>VirtualRegInfo</td>
<td></td>
<td>+90 / +120</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+80 / +100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+70 / +E0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+60 / +C0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MemoryInfo</td>
<td></td>
<td>+50 / +A0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KernDescPtr</td>
<td>APIFlags</td>
<td>API Version</td>
<td>0_{(0/32)}</td>
<td>+C / +18</td>
<td>+8 / +10</td>
<td>+4 / +8</td>
<td>+0</td>
</tr>
<tr>
<td>KernDescPtr</td>
<td></td>
<td></td>
<td></td>
<td>+40 / +80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KernDescPtr</td>
<td></td>
<td></td>
<td></td>
<td>+30 / +60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KernDescPtr</td>
<td></td>
<td></td>
<td></td>
<td>+20 / +40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KernDescPtr</td>
<td></td>
<td></td>
<td></td>
<td>+10 / +20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
KernelInterface()

• System call provided to locate the kernel information page
• In ‘C’ api

```c
void * L4_KernelInterface (L4_Word_t *ApiVersion,
                           L4_Word_t *ApiFlags,
                           L4_Word_t *KernelId)
```
Traditional Thread

- Abstraction and unit of execution
- Consists of
  - Registers
    - Current variables
    - Status
  - Instruction Pointer
    - Next instruction to execute
  - Stack
    - Execution history of yet unreturned procedures
    - One stack frame per procedure invocation
L4 thread = trad. thread +

- A set of *virtual register*
- A priority and a timeslice
- A unique thread identifier
- An address space
- L4 provides a fixed overall number of threads in the system
  - Root task responsible for creating/deleting threads and assigning them to address spaces.
  - System, User and “Hardware” threads
Virtual Registers

- Per-thread “registers” defined by the microkernel
- Are realised via real machine registers or via memory locations
  - Realisation depended on architecture and ABI
- Two basic types (was three)
  - Thread Control Registers (TCRs)
    • Used to share information about threads between the kernel and user level
  - Message Registers (MRs)
    • Used to send messages between threads. Contains the message (or description of it, e.g. region of memory)
  - Buffer Registers (BRs)
    • Used to specify where messages (other than MRs themselves) are received
Thread Control Blocks (TCBs)

- State of a thread is stored in its TCB
- Some sensitive state can only be modified via a controlled interface (system calls)
  - e.g. address space associated with the thread
  - e.g. kernel stack pointer
  - sensitive state stored in Kernel TCB (KTCB)
- Some state can be freely visible and modifiable by user-level applications without compromising the system
  - kept in user-level TCB (UTCB)
  - Include virtual registers not bound to real register
  - *must only be modified via the provided library function!*
    - No consistency guarantees otherwise
    - Library function scope limited to current thread
  - Many fields only modified as a side effect of some operations (IPC)
KTCBs and UTCBs

Efficient and secure access to state traditionally kept in an in-kernel TCB

Traditional TCB

User-level TCB

Kernel TCB
# User-Level TCB (ARM)

<table>
<thead>
<tr>
<th>PreemptedIP (32)</th>
<th>+52</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreemptCallbackIP (32)</td>
<td>+48</td>
</tr>
<tr>
<td>Virtual/Sender/Actual/Sender (32)</td>
<td>+44</td>
</tr>
<tr>
<td>IntendedReceiver (32)</td>
<td>+40</td>
</tr>
<tr>
<td>ErrorCode (32)</td>
<td>+36</td>
</tr>
<tr>
<td>ProcessorNo (32)</td>
<td>+32</td>
</tr>
<tr>
<td>NotifyBits (32)</td>
<td>+28</td>
</tr>
<tr>
<td>NotifyMask (32)</td>
<td>+24</td>
</tr>
<tr>
<td>Acceptor (32)</td>
<td>+20</td>
</tr>
<tr>
<td>~ (16)</td>
<td>+16</td>
</tr>
<tr>
<td>cop flags (8)</td>
<td>+12</td>
</tr>
<tr>
<td>preempt flags (8)</td>
<td>+8</td>
</tr>
<tr>
<td>ExceptionHandler (32)</td>
<td>+4</td>
</tr>
<tr>
<td>Pager (32)</td>
<td></td>
</tr>
<tr>
<td>UserDefinedHandle (32)</td>
<td></td>
</tr>
<tr>
<td>MyGloballd (32)</td>
<td></td>
</tr>
<tr>
<td>+316</td>
<td></td>
</tr>
</tbody>
</table>

```
MR_0 (32)  r3
MR_1 (32)  r4
MR_2 (32)  r5
MR_3 (32)  r6
MR_4 (32)  r7
MR_5 (32)  r8
```

---

**UTCB address** (32)

---

**UTCB address** + 88

---

0xFFF000FF0
UTCB Programming

• Only modified via provided programming interface
  – Don’t access it directly

• You can mostly ignore its contents
  – Most stuff is set/read in the context of other actions (e.g. IPC)
  – Not needed for project
    • E.g. processor number
Thread Identifiers

• Global Identifiers
  – Identify a thread uniquely within the system
  – Defined by the root task at thread creation
    • according to some policy
    • Note: version[5..0] != 0

• Local Identifiers
  – Identify a thread within an address space
    • Only unique and useable within an address space
    • Used for some optimisations
    • Typically the address of the thread’s UTCB.

• Note: V4 local thread IDs removed
ThreadControl()

- Create, destroy, and modify threads
- Determines thread attributes:
  - Global thread identifier
  - address space
  - The thread permitted to control scheduling parameters
    - This is known as target threads “scheduler”
    - Note: the “scheduler” does not perform CPU scheduling
  - The page fault handler “pager”
  - Location of the UTCB within the address spaces allotted UTCB area (See SpaceControl later)
    - Note: ARM, the UTCB address is defined by the kernel, not ThreadControl()
ThreadControl()

- Threads can be created _active_ or _inactive_.
  - Thread is active iff it has a pager
  - Create of inactive threads is used to
    - Create and manipulate new address spaces
    - Allocate new threads to an existing address spaces
  - Inactive threads can be activated in one of two ways
    - by privileged thread using ThreadControl()
    - By a local thread (same address space) using ExchangeRegisters()

```
L4_Word_t L4_ThreadControl (L4_ThreadId_t dest,  
  L4_ThreadID_t SpaceSpecifier,  
  L4_ThreadId_t Scheduler,  
  L4_ThreadId_t Pager,  
  void * UtcbLocation)

ARM: utcb must be zero!
```
Task

• L4 does not define the concept of a “task”
• We use it informally to mean:
  – An address space
    • UTCB area
    • Kernel interface page
    • Redirector
  – Set of threads inside that address space
    • Global thread ID
    • UTCB location
    • IP, SP
    • Pager
    • Scheduler
    • Exception handler
  – Code, data, stack(s) mapped into address space
Steps in Creating a New “Task”

1. Create an inactive thread in a new address space (AS)
   - An AS is referred to via the ID of one of its threads

   \[
   \text{L4\_ThreadControl} \; (\text{task}, \; /* \text{new tid} */ \\
   \text{task}, \; /* \text{new space identifier} */ \\
   \text{me}, \; /* \text{scheduler of new thread} */ \\
   \text{L4\_nilthread}, \; /* \text{pager = nil, inactive,} \\
   \text{(void *) -1); /* NOP Utcb location */}
   \]

   ...creates a new thread in an otherwise empty address space
Steps in Creating a New “Task”

2. Define location of KIP and UTCB area in new address space

```c
L4_SpaceControl (task, /* new TID */
    0, /* control */
    kip_area, /* where KIP is mapped */
    utcb_area, /* location of UTCB array */.
L4_anythread, /* no redirector */
    &control); /* leave alone */
```
Steps in Creating a New “Task”

3. Specify the utcb location and assign a pager to the new thread to activate it.

   L4_ThreadControl (task, task, me,
   pager, /* new pager */
   (void *) utcb_base); /* utcb location */

   This results in the thread immediately waiting for an IPC containing the IP and SP of the new thread.
Steps in Creating a New “Task”

4. Send an IPC to the new thread with the IP and SP in the first two words of the message.

This results in the thread starting at the received IP with the SP set as received.
Adding Threads To a Task

• Use `ThreadControl()` to assign new threads to AS.

  ```c
  L4_ThreadControl (newtid, /* new thread id */
                     ExistingId, /* address space identifier */
                     me, /* scheduler of new thread */
                     pager,
                     (void *) utcb_base);
  ```

• Can also create new thread inactive
  – Task can then manage new threads itself
  – …using `ExchangeRegisters()`
Practical Considerations

• Above sequence for creating tasks and threads is cumbersome
  – Price paid for leaving policy out of kernel
  – Any shortcuts imply policy
• A system built on top of L4 will inherently define policies
  – Can define and implement library interface for task and thread creation
  – Incorporate system policy
  – See sos_task_new() for an example
• Actual apps would not use raw L4 system calls, but
  – Use libraries
  – Use IDL compilers
Manipulating threads within an Address Space

• So far can
  – Create a new address space with a single thread
  – Assign new threads to an existing address space

• ExchangeRegisters
  – Used to activate or modify an existing thread within an address space.
Exchange Registers

Thread ID
Instr. Ptr
Stack Ptr
Pager
UserHandle

Code

Data

Stack

Old User Handle
Old Instr. Ptr
Old Stack Ptr
Old Pager

Thread Execution Path
ExchangeRegisters()

L4_Thread_id_t L4_ExchangeRegisters (L4_Thread_id_t dest,
L4_Word_t control,
L4_Word_t sp,
L4_Word_t ip,
L4_Word_t flags,
L4_Word_t UserDefHandle,
L4_Thread_id_t pager,
L4_Word_t *old_control,
L4_Word_t *old_sp,
L4_Word_t *old_ip,
L4_Word_t *old_flags,
L4_Word_t *old_UserDefHandle,
L4_Thread_id_t *old_pager)
**ExchangeRegisters()**

- CPSR bits affected by flags

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>V</th>
<th>Q</th>
<th>~ (21)</th>
<th>T</th>
<th>~ (5)</th>
</tr>
</thead>
</table>

- N = negative
- Z = zero
- C = carry
- V = overflow
Threads

• Note the microkernel only manages (preserves) the user-level IP and SP
  – (and registers if preempted)
• The following is managed by user-level applications *(This means you)*
  – User stack location, allocation, size, deallocation
  – Thread ID allocation, deallocation
  – Entry point
  – UTCB slot allocation
    • KIP specifies UTCB requirements
Be CAREFUL!!!!

- Stack corruption is a common problem
- Stack corruption is very difficult to
  - diagnose
  - debug
Communication
IPC Overview

• Single system call that implements synchronous unbuffered IPC
  – Has a send and receive component
  – Either can be omitted
• Receive operation can
  – specify a specific thread from which to receive ("closed receive")
  – specify willingness to receive from any thread ("open wait")
    • can be any thread in system or any local thread (same AS)
• Results in five different logical operations:
  • Send() send a message to a specified thread
  • Receive() “closed” receive from a specific sender
  • Wait() “open” receive from any sender
  • Call() send and wait for reply
    – typical client RPC operation
  • Reply_and_Wait() send to a specific thread and wait for any new message
    – typical server operation
Thread Identifiers

- **Global Identifiers**
  - Thread IDs
  - Interrupt IDs
  - Special IDs
    - Nil thread
    - Any thread

<table>
<thead>
<tr>
<th>Thread No ((32))</th>
<th>Version ((32))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt No ((32))</td>
<td>1 ((32))</td>
</tr>
<tr>
<td></td>
<td>(0_{(64)})</td>
</tr>
<tr>
<td></td>
<td>(-1_{(64)})</td>
</tr>
</tbody>
</table>
In `<l4/types.h>`

```c
#define L4_nilthread ((L4_Threadld_t) { raw : 0UL})
#define L4_anythread ((L4_Threadld_t) { raw : ~0UL})
#define L4_anylocalthread ((L4_Threadld_t) { local : { X : {L4_SHUFFLE2(0, ~0UL)>>>>}}})
```
IPC Registers

• Message registers
  – virtual registers
    • not necessarily hardware registers
    • part of thread state
    • on ARM: 6 physical registers, rest in UTCB
  – actual number is system-configuration parameter
    • at least 8, no more than 64
  – contents form message
    • first is message tag, defining message size (etc)
    • rest untyped words, not (normally) interpreted by kernel
    • kernel protocols define semantics in some cases

• Simple IPC just copies data from sender’s to receiver’s MRs!
  – this case is highly optimised in the kernel (“fast path”)
  – Note: no page faults possible during transfer (registers don’t fault!)
Message Register IPC

Thread A

MR63
MR13
MR12
MR11
MR10
MR9
MR8
MR7
MR6
MR5
MR4
MR3
MR2
MR1
MR0

Thread B

MR63
MR13
MR12
MR11
MR10
MR9
MR8
MR7
MR6
MR5
MR4
MR3
MR2
MR1
MR0

Message transferred from one thread’s MRs to the other thread’s MRs

Guaranteed to not cause page faults
Overview of IPC operations

• L4_Ipc system call performs all IPC operations (both sending and receiving)

• Helper functions for frequent operations (see <l4/ipc.h>)
  – L4_Send
    • Send a message to a thread (blocking)
  – L4_Receive
    • Receive a message from a specified thread
  – L4_Wait
    • Receive a message from any sender
  – L4_ReplyWait
    • Send a response to a thread and wait for the next message
  – L4_Call
    • Send a message to a particular thread and wait for it to respond
      (usual RPC operation)
**MR\_0**

<table>
<thead>
<tr>
<th>label(_{48})</th>
<th>srnp(_{4})</th>
<th>t(_{6})</th>
<th>u(_{6})</th>
</tr>
</thead>
</table>

- Specifies message content
- u: number of words in message (excluding MR0)
- p: specifies *propagation*
  - allows sending a message on behalf of another thread
  - specified by *virtual sender* in UTCB
  - receiver gets from kernel virtual, rather than real sender ID
  - restricted for security (essentially allowed for local threads)
- n: specifies *asynchronous notification* operation (later)
- r: blocking receive
  - if unset, fail immediately if no pending message
- s: blocking send
  - if unset, fail immediately if receiver not waiting
- label: user-defined (e.g., opcode)
Example: Sending 4 untyped words

• Only 5 MRs transferred
  – Note: On ARM, 6 MRs are transferred in CPU registers
    • Fast
    • The rest (if used) are copied from and to memory

Note: u, s, r set implicitly by L4_MsgAppendWord()

Note: Ideally we would use an IDL compiler instead of generating message manually

<table>
<thead>
<tr>
<th>MR5</th>
</tr>
</thead>
<tbody>
<tr>
<td>word 4</td>
</tr>
<tr>
<td>word 3</td>
</tr>
<tr>
<td>word 2</td>
</tr>
<tr>
<td>word 1</td>
</tr>
<tr>
<td>label</td>
</tr>
</tbody>
</table>
Example IPC code

L4_Msg_t msg;
L4_MsgTag_t tag;

L4_MsgClear(&msg);
L4_MsgAppendWord(&msg, word1);
L4_MsgAppendWord(&msg, word2);
L4_MsgAppendWord(&msg, word3);
L4_MsgAppendWord(&msg, word4);
L4_MsgLoad(&msg);

tag = L4_Send(tid);
IPC result $\text{MR}_0$

- **MsgTag** $[\text{MR}_0]$
  - $u$ untyped words received ($u = 0$, send only IPC)
  - $t$ typed words sent/received ($t = 0$, send only IPC)

- **Flags** EXrp
  - E: error occurred (send or receive), see ErrorCode TCR for details
  - X: received cross processor IPC (ignore)
  - r: received redirected IPC (later)
  - p: received propagated IPC (check ActualSender in UTCB)

<table>
<thead>
<tr>
<th>label</th>
<th>flags</th>
<th>t</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>(48)</td>
<td>(4)</td>
<td>(6)</td>
<td>(6)</td>
</tr>
</tbody>
</table>
IPC

- to / from
- FromSpecifier
- $MR_0$
### IPC

- **Send**

  - dest
  - nilthread
  - MR₀

  ![Diagram of IPC communication](image)
IPC

- Receive from dest

- nilthread to / from
- dest FromSpecifier
- Timeouts
- MR₀
IPC

- Wait
  - Receive from *anyone*

- nilthread to / from
- anythread FromSpecifier
- MR₀

me
IPC

• Call

- dest to / from
- dest FromSpecifier
- MR₀

me → dest
IPC

- ReplyWait
  
  - dest
  - anythread
  - to / from
  - FromSpecifier
  - MR₀
Obsolete IPC features

• Currently, StringItems are not supported on the MIPS-64 and ARM version of L4Ka::Pistachio

• Alternatives
  – Break long messages into many short messages
  – Share memory

• Map and Grant “typed” items in IPC not supported
  – used to send page mapping via IPC.
  – use MapControl() system call instead

• Timeout on IPC
  – limit blocking time
  – limited use in practice
  – replace by send/receive block bits (s,r).
Drivers at User Level

- **IO ports**: part of the user address space
- **interrupts**: messages from “hardware” threads
  - Acknowledge hardware interrupt via replying to interrupt message

\[
\text{INTR} \quad \Rightarrow \quad \text{Driver} \quad \Rightarrow \quad \text{User} \quad \Rightarrow \quad \text{Device}
\]

\[= \text{ipc}\]
- Interrupts: messages from "hardware" threads
- Acknowledge hardware interrupt via replying to interrupt message

The interrupt message is sent to the hardware thread’s pager

= ipc
Interrupt Handlers

• Typical setup: interrupt handler is bottom-half device driver

• Interrupt handling:
  1. interrupt is triggered, hardware disables interrupt and invokes kernel
  2. kernel masks interrupt, enables interrupts and sends message to handler
  3. handler receives message, identifies interrupt cause, replies to kernel
  4. kernel acknowledges interrupt
  5. handler queues request to top-half driver, sends notification to top half, waits for next interrupt
Interrupt Association

- Association is done via the privileged thread (root task) using `ThreadControl`.

- To associate a thread to an interrupt
  - Set the pager of the hardware thread ID to the thread ID of the interrupt handler

- To disassociate the thread from an interrupt
  - Set the pager of the hardware thread ID to the hardware thread ID itself
• Forfeits the caller’s remaining time slice
ThreadSwitch()

- Forfeits the caller’s remaining time slice
  - Can donate remaining time slice to specific thread
    ➔ that thread will execute to the end of the time slice on the donor’s priority
ThreadSwitch()

- Forfeits the caller’s remaining time slice
  - Can donate remaining time slice to specific thread
    ➔ that tread will execute to the end of the time slice on the
donor’s priority

**Note:** This is what the manual says.
In the present implementation, the donation is only valid to
the next timer tick (10ms on ARM)!
**ThreadSwitch()**

- Forfeits the caller’s remaining time slice
  - Can donate remaining time slice to specific thread
    ➔ that thread will execute to the end of the time slice on the donor’s priority
  - **Note:** This is what the manual says.
    In the present implementation, the donation is only valid to the next timer tick (10ms on ARM)!
  - If no recipient specified (or recipient is not runnable)
    ➔ normal “yield” operation
    ➔ kernel invokes scheduler
    ➔ caller might receive a new time slice immediately
ThreadSwitch()

- Forfeits the caller’s remaining time slice
  - Can donate remaining time slice to specific thread
    ➔ that thread will execute to the end of the time slice on the donor’s priority
  
  Note: This is what the manual says.
  In the present implementation, the donation is only valid to the next timer tick (10ms on ARM)!

- If no recipient specified (or recipient is not runnable)
  ➔ normal “yield” operation
  ➔ kernel invokes scheduler
  ➔ caller might receive a new time slice immediately

- Directed donation can be used for
  ➔ explicit scheduling of threads
  ➔ implementing wait-free locks
  ➔ ...
SYSTEM CALLS

✔ KernelInterface
✔ ThreadControl
✔ ExchangeRegisters
✔ IPC
✔ ThreadSwitch

→ Schedule
  • MapControl
  • SpaceControl
  • ProcessorControl
  • CacheControl
L4 Scheduling

- L4 uses 256 hard priorities (0–255)
- Within each priority schedules threads round-robin
L4 Scheduling

- L4 uses 256 hard priorities (0–255)
- Within each priority schedules threads round-robin
- Scheduler is invoked when
  - the current thread is preempted
  - the current thread yields
L4 Scheduling

- L4 uses 256 hard priorities (0–255)
- Within each priority schedules threads round-robin
- Scheduler is invoked when
  - the current thread is preempted
  - the current thread yields
- The scheduler is not normally invoked when a thread blocks:
  - if destination thread is runnable, the kernel will switch to it
  - called direct process switch
L4 Scheduling

• L4 uses 256 hard priorities (0–255)

• Within each priority schedules threads round-robin

• Scheduler is invoked when
  ➔ the current thread is preempted
  ➔ the current thread yields

• The scheduler is **not** normally invoked when a thread blocks:
  ➔ if destination thread is runnable, the kernel will switch to it
    ➔ called *direct process switch*
  ➔ scheduler only invoked if destination is blocked too
  ➔ if both threads are runnable after IPC, the higher-prio one will run
    ❌ presently implementation doesn’t always observe prios correctly!
L4 Scheduling

- L4 uses 256 hard priorities (0–255)
- Within each priority schedules threads round-robin
- Scheduler is invoked when
  - the current thread is preempted
  - the current thread yields
- The scheduler is **not** normally invoked when a thread blocks:
  - if destination thread is runnable, the kernel will switch to it
    - called *direct process switch*
  - scheduler only invoked if destination is blocked too
  - if both threads are runnable after IPC, the higher-prio one will run
    - presently implementation doesn’t always observe prios correctly!
- This makes (expensive) scheduler invocation infrequent
Each thread has:

- a **priority**, determines whether it is scheduled
- a **time slice length**, determines, once scheduled, when it will be preempted.
- a **total quantum**
Each thread has:

- a **priority**, determines whether it is scheduled
- a **time slice length**, determines, once scheduled, when it will be preempted.
- a **total quantum**

When scheduled, the thread gets a new time slice

- the time slice is subtracted from the thread’s total quantum
- when total quantum is exhausted, the thread’s scheduler is notified
Total Quantum and Preemption IPC

- Each thread has:
  - a priority, determines whether it is scheduled
  - a time slice length, determines, once scheduled, when it will be preempted.
  - a total quantum

- When scheduled, the thread gets a new time slice
  - the time slice is subtracted from the thread’s total quantum
  - when total quantum is exhausted, the thread’s scheduler is notified

- When the time slice is exhausted, the thread is preempted
  - preemption-control flags in the UTCB can defer preemption
    - unless there is a runnable thread of higher than the sensitive priority
    - for up to a specified maximum delay
      - exceeding this causes an IPC to the exception handler
      - can be used to implement lock-free synchronisation
The Schedule() syscall does **not** invoke a scheduler!

Nor does it actually schedule any threads.
• The `Schedule()` syscall does **not** invoke a scheduler!

• Nor does it actually schedule any threads.

• `Schedule()` manipulates a thread’s scheduling parameters:
  
  – The caller must be registered as the destination’s scheduler
  ➔ set via `ThreadControl()`
The `Schedule()` syscall does **not** invoke a scheduler!

Nor does it actually schedule any threads.

`Schedule()` manipulates a thread’s scheduling parameters:

- The caller must be registered as the destination’s scheduler
  
  - set via `ThreadControl()`

- can change
  
  - priority
  
  - time slice length
  
  - total quantum
  
  - sensitive priority
  
  - processor number
    
    - only relevant for SMP
    
    - kernel will not transparently migrate threads between CPUs
SYSTEM CALLS

✔ KernelInterface
✔ ThreadControl
✔ ExchangeRegisters
✔ IPC
✔ ThreadSwitch
✔ Schedule
➜ MapControl
  • SpaceControl
  • ProcessorControl
  • CacheControl
Address spaces are created empty

Need to be explicitly populated with page mappings
  ➔ kernel does not map pages automatically (except KIP, UTCB)
Address Spaces

- Address spaces are created empty
- Need to be explicitly populated with page mappings
  - kernel does not map pages automatically (except KIP, UTCB)
- Normally AS populated by pager on demand
  - thread runs, faults on unmapped pages, pager creates mapping
- Can also be done pro-actively
  - Eg OS server can pre-map contents of executable
Address Spaces

- Address spaces are created empty
- Need to be explicitly populated with page mappings
  - kernel does not map pages automatically (except KIP, UTCB)
- Normally AS populated by pager on demand
  - thread runs, faults on unmapped pages, pager creates mapping
- Can also be done pro-actively
  - Eg OS server can pre-map contents of executable
- Address space is a second-class abstraction
  - there are no unique identifiers for address spaces
  - an AS is identified via one of its threads (syscall TID argument)
MapControl()

- Creates (maps) or destroys (unmaps) page mappings
- Privileged system call (only available to root task)
**MapControl()**

- Creates (maps) or destroys (unmaps) page mappings
- Privileged system call (only available to root task)

```c
L4_Word_t L4_MapControl (L4_ThreadId_t dest,
                          L4_Word_t control)
```

- **dest:** denominates target address space
- **control:** determines operation of syscall

<table>
<thead>
<tr>
<th>m</th>
<th>r</th>
<th>0 (24)</th>
<th>n (6)</th>
</tr>
</thead>
</table>

- **r:** read operation — returns (pre-syscall) mapping info
  - eg reference bits where hardware-maintained (x86)
- **m:** modify operation — changes mappings
- **n:** number of *map items* used to describe mappings
  - map items are contained in message registers MR₀ · · · MR₂ⁿ⁻₁
**SPECIFYING MAPPINGS: FPPAGES**

- A *flexpage* or *fpage* is used to specify mapping objects
  - generalisation of a hardware page
  - similar properties:
    - size is power-of-two multiple of base hardware page size
    - aligned to its size
  - fpage of size $2^s$ is specified as
    - $2^s$ is specified as $\text{base}/1024^{s}$
      
    | base/1024 | $s$ \((6)\) | $\sim$ \((4)\) |
    |------------|-------------|----------------|

- special fpages:
  - full AS
    | 0 | 0x3f | $\sim$ \((4)\) |
  - nil page
    | 0 | 0 \((6)\) | 0 \((4)\) |

- On ARM, $s \geq 12$
### Map Item

- Specifies a mapping to be created in destination AS

<table>
<thead>
<tr>
<th>fpage ( (28) )</th>
<th>0rwx</th>
</tr>
</thead>
<tbody>
<tr>
<td>phys adr/1024 ( (26) )</td>
<td>attr ( (6) )</td>
</tr>
</tbody>
</table>

- *fpage*: specifies where mapping is to occur in destination AS
- *phys adr*: base of physical frame(s) to be mapped
- *attr*: memory attributes (e.g., cached/uncached)
- *rwx*: permissions
- Specifies a mapping to be created in destination AS

<table>
<thead>
<tr>
<th>fpage (28)</th>
<th>0rwx</th>
</tr>
</thead>
<tbody>
<tr>
<td>phys adr/1024 (26)</td>
<td>attr (6)</td>
</tr>
</tbody>
</table>

- `fpage`: specifies where mapping is to occur in destination AS
- `phys adr`: base of physical frame(s) to be mapped
  - Note: shifted 4 bits to support 64MB of physical AS
- `attr`: memory attributes (e.g., cached/uncached)
- `rwx`: permissions
**Map Item**

- Specifies a mapping to be created in destination AS

<table>
<thead>
<tr>
<th>fpage (28)</th>
<th>0rwx</th>
</tr>
</thead>
<tbody>
<tr>
<td>phys adr/1024 (26)</td>
<td>attr (6)</td>
</tr>
</tbody>
</table>

- **fpage**: specifies where mapping is to occur in destination AS
- **phys adr**: base of physical frame(s) to be mapped
  - Note: shifted 4 bits to support 64MB of physical AS
- **attr**: memory attributes (e.g., cached/uncached)
- **rwx**: permissions
  - access rights in destination address space
  - can be used to change (up/downgrade) rights
    - (only if mapping is replaced by an otherwise identical one)
  - removing all rights removes the mapping (unmap operation)
Page Fault Handling

- Address-spaces are populated in response to page faults
- Page faults are converted into IPC messages
PAGE FAULT HANDLING

- Address-spaces are populated in response to page faults
- Page faults are converted into IPC messages:
  ① app triggers page fault
Page Fault Handling

- Address-spaces are populated in response to page faults
- Page faults are converted into IPC messages:
  1. App triggers page fault
  2. Kernel exception handler generates IPC from faulter to pager
Page Fault Handling

- Address-spaces are populated in response to page faults
- Page faults are converted into IPC messages:
  1. app triggers page fault
  2. kernel exception handler generates IPC from faulter to pager
  3. pager establishes mapping
     - calls MapControl() (if privileged) otherwise asks root task to do it
  4. pager replies to page-fault IPC
  5. kernel intercepts message, discards
  6. kernel restarts faulting thread

![Diagram of page fault handling](attachment:page_fault_handling_diagram.png)
### Page Fault Message

- Format of kernel-generated page fault message

<table>
<thead>
<tr>
<th>Fault IP</th>
<th>MR₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault address</td>
<td>MR₁</td>
</tr>
<tr>
<td>-2</td>
<td>MR₀</td>
</tr>
</tbody>
</table>

-2: 0rwx 0 (4) 0 (6) 2
**PAGE FAULT MESSAGE**

- Format of kernel-generated page fault message

<table>
<thead>
<tr>
<th>Fault IP</th>
<th>MR₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fault address</th>
<th>MR₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>0rwx</td>
<td></td>
</tr>
<tr>
<td>0 (4)</td>
<td></td>
</tr>
<tr>
<td>0 (6)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

- Eg. page fault at address 0x2002: Kernel sends

<table>
<thead>
<tr>
<th>Fault IP</th>
<th>MR₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x2002</th>
<th>MR₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>0rwx</td>
<td></td>
</tr>
<tr>
<td>0 (4)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
### Page Fault Message

- **Format of kernel-generated page fault message**

<table>
<thead>
<tr>
<th>Fault IP</th>
<th>MR₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault address</td>
<td>MR₁</td>
</tr>
<tr>
<td>-2</td>
<td>0rwx</td>
</tr>
<tr>
<td>MR₀</td>
<td></td>
</tr>
</tbody>
</table>

- **Eg. page fault at address 0x2002: Kernel sends**

<table>
<thead>
<tr>
<th>Fault IP</th>
<th>MR₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2002</td>
<td>MR₁</td>
</tr>
<tr>
<td>-2</td>
<td>0rwx</td>
</tr>
<tr>
<td>MR₀</td>
<td></td>
</tr>
</tbody>
</table>

- **Obviously, application can manufacture same message**
  - pager cannot tell the difference
  - not a problem, as application could achieve the same by forcing a fault
• E.g., pager handles write page fault at 0x2002
  - map item for map 4kB page at PA 0xc0000:

```
<table>
<thead>
<tr>
<th>phys adr</th>
<th>fpage size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8</td>
<td>12</td>
</tr>
<tr>
<td>0x300</td>
<td>0</td>
</tr>
</tbody>
</table>
```

  - note: *phys adr* must be aligned to *fpage* size
Pager Action

• E.g., pager handles write page fault at 0x2002
  - map item for map 4kB page at PA 0xc0000:
    
    |      |      |      |
    |------|------|------|
    | 0x8  |  12  |   0  |
    |      |      |      |
    | 0x300|      |   0  |

  - note: phys adr must be aligned to fpage size

• After establishing mapping, pager replies to page-fault message
  ➔ content of message completely ignored
  ➔ only servers for synchronisation: informing kernel that faultter can be restarted
  ➔ if pager did not establish a suitable mapping, client will trigger same fault again
SYSTEM CALLS

✔ KernelInterface
✔ ThreadControl
✔ ExchangeRegisters
✔ IPC
✔ ThreadSwitch
✔ Schedule
✔ MapControl
➜ SpaceControl
• ProcessorControl
• CacheControl
**SpaceControl()**

- Controls layout of new address spaces
  - KIP location (not on ARM)
  - UTCB area location (not on ARM)
SpaceControl()

- Controls layout of new address spaces
  - KIP location (not on ARM)
  - UTCB area location (not on ARM)

- Controls setting of *redirector*
  - used to limit communication
    - for information flow control
  - if set to a valid thread, IPC from the AS can only be sent:
    - locally (within AS)
    - to the redirector’s address space
  - any other message is instead delivered to the redirector
  - **Note:** not heavily tested in present version
• Controls layout of new address spaces
  ➔ KIP location (not on ARM)
  ➔ UTCB area location (not on ARM)

• Controls setting of *redirector*
  - used to limit communication
    ➔ for information flow control
  - if set to a valid thread, IPC from the AS can only be sent:
    ➔ locally (within AS)
    ➔ to the redirector’s address space
  - any other message is instead delivered to the redirector
  - **Note:** not heavily tested in present version
    ➔ your chance to pick up bonus points 😊
SPACECONTROL()

- Controls layout of new address spaces
  - KIP location (not on ARM)
  - UTCB area location (not on ARM)

- Controls setting of `redirector`
  - used to limit communication
    - for information flow control
  - if set to a valid thread, IPC from the AS can only be sent:
    - locally (within AS)
    - to the redirector’s address space
  - any other message is instead delivered to the redirector

  - Note: not heavily tested in present version
    - your chance to pick up bonus points

- On ARM `control` used to set PID register (later)
System Calls

- ✔ KernelInterface
- ✔ ThreadControl
- ✔ ExchangeRegisters
- ✔ IPC
- ✔ ThreadSwitch
- ✔ Schedule
  - • MapControl
  - • SpaceControl
  - ➜ ProcessorControl
  - • CacheControl
ProcessorControl()

- Sets processor core voltage and frequency (where supported)
  ➔ used for power management
- Privileged system call
System Calls

- ✔ KernelInterface
- ✔ ThreadControl
- ✔ ExchangeRegisters
- ✔ IPC
- ✔ ThreadSwitch
- ✔ Schedule
- ✔ MapControl
- ✔ SpaceControl
- ✔ ProcessorControl
- ➜ CacheControl
CacheControl()

- Used to flush caches or lock cache lines as per arguments
  ➔ target cache (I/D, L1/L2, ...)
  ➔ kind of operation (flush/lock/unlock)
  ➔ address range to flush from cache

- Privileged system call
  ➔ sort-of... Some functions can be called from anywhere (Hack!)
SYSTEM CALLS

✓ KernelInterface
✓ ThreadControl
✓ ExchangeRegisters
✓ IPC
✓ ThreadSwitch
✓ Schedule
✓ MapControl
✓ SpaceControl
✓ ProcessorControl
✓ CacheControl

That’s it!
L4 Protocols

✔ Page fault
  → already covered

✔ Thread start
  → already covered

✔ Interrupt
  → already covered

→ Preemption

• Exception

• Asynchronous notification
Preemption Protocol

- Each thread has three scheduling attributes:
  - priority
  - time slice length
  - total quantum

- Kernel schedules runnable threads according to their priority
  - round-robin between threads of equal priority
Preemption Protocol

- Each thread has three scheduling attributes:
  - priority
  - time slice length
  - total quantum

- Kernel schedules runnable threads according to their priority
  - round-robin between threads of equal priority

- When thread is scheduled
  - it is given fresh time slice
  - the time slice is deducted from its total quantum
Preemption Protocol

- Each thread has three scheduling attributes:
  - priority
  - time slice length
  - total quantum

- Kernel schedules runnable threads according to their priority
  - round-robin between threads of equal priority

- When thread is scheduled
  - it is given fresh time slice
  - the time slice is deducted from its total quantum

- When total quantum is exhausted, the kernel sends a message on behalf of the preempted thread to its scheduler
  - scheduler can provide new quantum (using Schedule())
  - not heavily tested

- Format of preemption message:

  -3 0 0 0 0 0 MR₀
L4 Protocols

✔ Page fault
✔ Thread start
✔ Interrupt
✔ Preemption
→ Exception
  • Asynchronous notification
Exception Protocol

• Other exceptions (invalid instruction, division by zero...) result in a kernel-generated IPC to thread’s exception handler
**Exception Protocol**

- Other exceptions (invalid instruction, division by zero...) result in a kernel-generated IPC to thread’s *exception handler*

- Exception IPC
  - kernel sends (partial) thread state
    
    | exception word\(_{k-1}\) | MR\(_{k+1}\) |
    |-------------------------|-------------|
    | ...                     | ...         |
    
    | exception word\(_{0}\) | MR\(_{2}\)  |
    
    | exception IP            | MR\(_{1}\)  |
    
    | label                   | MR\(_{0}\)  |
    | 0                       | 0           |
    | 0                       | 0           |
    | 0                       | k           |

- label:
  - -4: standard exceptions, architecture independent
  - -5: architecture-specific exception
Exception Protocol

- Other exceptions (invalid instruction, division by zero...) result in a kernel-generated IPC to thread’s exception handler

- Exception IPC
  - kernel sends (partial) thread state
    - [Diagram showing exception word, MR, and exception IP with labels: 0, 0, 0, k]
  - label:
    - -4: standard exceptions, architecture independent
    - -5: architecture-specific exception

- Exception handler may reply with modified thread state
**Exception Handling**

- Possible responses of exception handler:
  - **retry**: reply with unchanged state
    - possibly after removing cause
    - possibly changing other parts of state (registers)
  - **continue**: reply with IP+=4 (assuming 4-byte instructions)
  - **emulation**: compute desired result,
    reply with appropriate register value and IP+=4
  - **handler**: reply with IP of local exception handler code
    to be executed by the thread itself
  - **ignore**: will block the thread indefinitely
  - **kill**: use ExchangeRegisters() (if local) or ThreadControl() to restart or kill thread
L4 Protocols

- Page fault
- Thread start
- Interrupt
- Preemption
- Exception
- Asynchronous notification
Very restricted form of asynchronous IPC:

- delivered without blocking sender
- delivered immediately, directly to receiver’s AS
ASYNCHRONOUS NOTIFICATION

- Very restricted form of asynchronous IPC:
  - delivered without blocking sender
  - delivered immediately, directly to receiver’s AS
  - message consists of a bit mask OR-ed to receiver’s bitfield
    receiver.NotifyBits | = sender.MR₁
**Asynchronous notification**

- Very restricted form of asynchronous IPC:
  - delivered without blocking sender
  - delivered immediately, directly to receiver’s AS
  - message consists of a bit mask OR-ed to receiver’s bitfield
    receiver.NotifyBits |\= sender.MR1
  - no effect if receiver’s bits already set
  - receiver can prevent asynchronous notification
    by setting a flag in its UTCB
Asynchronous Notification

- Very restricted form of asynchronous IPC:
  - delivered without blocking sender
  - delivered immediately, directly to receiver’s AS
  - message consists of a bit mask OR-ed to receiver’s bitfield
    receiver.NotifyBits | = sender.MR1
  - no effect if receiver’s bits already set
  - receiver can prevent asynchronous notification
    by setting a flag in its UTCB

- Two ways to receive asynchronous notifications:

  synchronously  by a form of blocking IPC wait
  - receiver specifies mask of notification bits to wait for
  - on notification, kernel manufactures a message in a defined format

  asynchronously  by checking NotifyBits in UTCB
  - but remember it’s asynchronous and can change at any time!
L4 Protocols

✔ Page fault
✔ Thread start
✔ Interrupt
✔ Preemption
✔ Exception
✔ Asynchronous notification