L4 Programming

COMP9242 2005/S2 Week 2

RECAP: L4 ABSTRACTIONS AND MECHANISMS

THREE BASIC ABSTRACTIONS:

- Address spaces
- Threads
- Time

TWO BASIC MECHANISMS:

- Inter-process communication (IPC)
- Mapping

System Calls

- KernelInterface
- ThreadControl
- ExchangeRegisters
- IPC
- ThreadSwitch
- Schedule
- SystemClock
- Unmap
- SpaceControl
- ProcessorControl
- MemoryControl

KERNEL INTERFACE PAGE (KIP)

- Kernel memory object
  - ➜ mapped into address space (AS) via **`KernelInterface()`** syscall
  - ➜ location defined at AS creation time by **`SpaceControl()`**

KERNEL INTERFACE PAGE (KIP)

- Kernel memory object
  - ➜ mapped into address space (AS) via **`KernelInterface()`** syscall
  - ➜ location defined at AS creation time by **`SpaceControl()`**
- Contains information about kernel and hardware
  - ➜ kernel version
  - ➜ supported features (page sizes)
  - ➜ physical memory layout
  - ➜ system call addresses

KERNEL INTERFACE PAGE (KIP)

- Kernel memory object
  - ➜ mapped into address space (AS) via **`KernelInterface()`** syscall
  - ➜ location defined at AS creation time by **`SpaceControl()`**
- Contains information about kernel and hardware
  - ➜ kernel version
  - ➜ supported features (page sizes)
  - ➜ physical memory layout
  - ➜ system call addresses
- C language API

```
void* L4_KernelInterface (L4_Word_t *ApiVersion,
                          L4_Word_t *ApiFlags,
                          L4_Word_t *KernelId)
```

System Calls

- KernelInterface
- ➜ ThreadControl
- ExchangeRegisters
- IPC
- ThreadSwitch
- Schedule
- SystemClock
- Unmap
- SpaceControl
- ProcessorControl
- MemoryControl

THREADS

- Traditional thread:
  - ⋆ execution abstraction
  - ⋆ consists of:
    - ➜ registers (GP and status registers)
    - ➜ stack

THREADS

- Traditional thread:
  - ⋆ execution abstraction
  - ⋆ consists of:
    - ➜ registers (GP and status registers)
    - ➜ stack
- L4 thread also has:
  - ➜ *virtual registers*
  - ➜ scheduling priority and time slice
  - ➜ unique thread-ID
  - ➜ address space

THREADS

- Traditional thread:
  - ⋆ execution abstraction
  - ⋆ consists of:
    - ➜ registers (GP and status registers)
    - ➜ stack
- L4 thread also has:
  - ➜ *virtual registers*
  - ➜ scheduling priority and time slice
  - ➜ unique thread-ID
  - ➜ address space
- L4 provides for a fixed overall number of threads
  - ➜ system, user and “hardware” threads
  - ➜ user threads created/deleted/allocated by *root task*

Virtual Registers

- Kernel-defined, user-visible thread state

Virtual Registers

- Kernel-defined, user-visible thread state
- Implemented as physical machine registers or memory locations
  - ➜ depends on architecture and ABI

Virtual Registers

- Kernel-defined, user-visible thread state
- Implemented as physical machine registers or memory locations
  - ➜ depends on architecture and ABI
- Three types
  - ⋆ *thread control registers* (TCRs)
    - ➜ for sharing info between kernel and user

Virtual Registers

- Kernel-defined, user-visible thread state
- Implemented as physical machine registers or memory locations
  - ➜ depends on architecture and ABI
- Three types
  - ⋆ *thread control registers* (TCRs)
    - ➜ for sharing info between kernel and user
  - ⋆ *Message Registers* (MRs)
    - ➜ contain the message passed in an IPC operation
    - ➜ ... or descriptors pointing to message

VIRTUAL REGISTERS

- Kernel-defined, user-visible thread state
- Implemented as physical machine registers or memory locations
  - ➜ depends on architecture and ABI
- Three types
  - ⋆ *thread control registers* (TCRs)
    - ➜ for sharing info between kernel and user
  - ⋆ *Message Registers* (MRs)
    - ➜ contain the message passed in an IPC operation
    - ➜ ... or descriptors pointing to message
  - ⋆ *Buffer Registers* (BRs)
    - ➜ specify receive buffers for IPC messages

Thread Control Block (TCB)

- Contains thread state

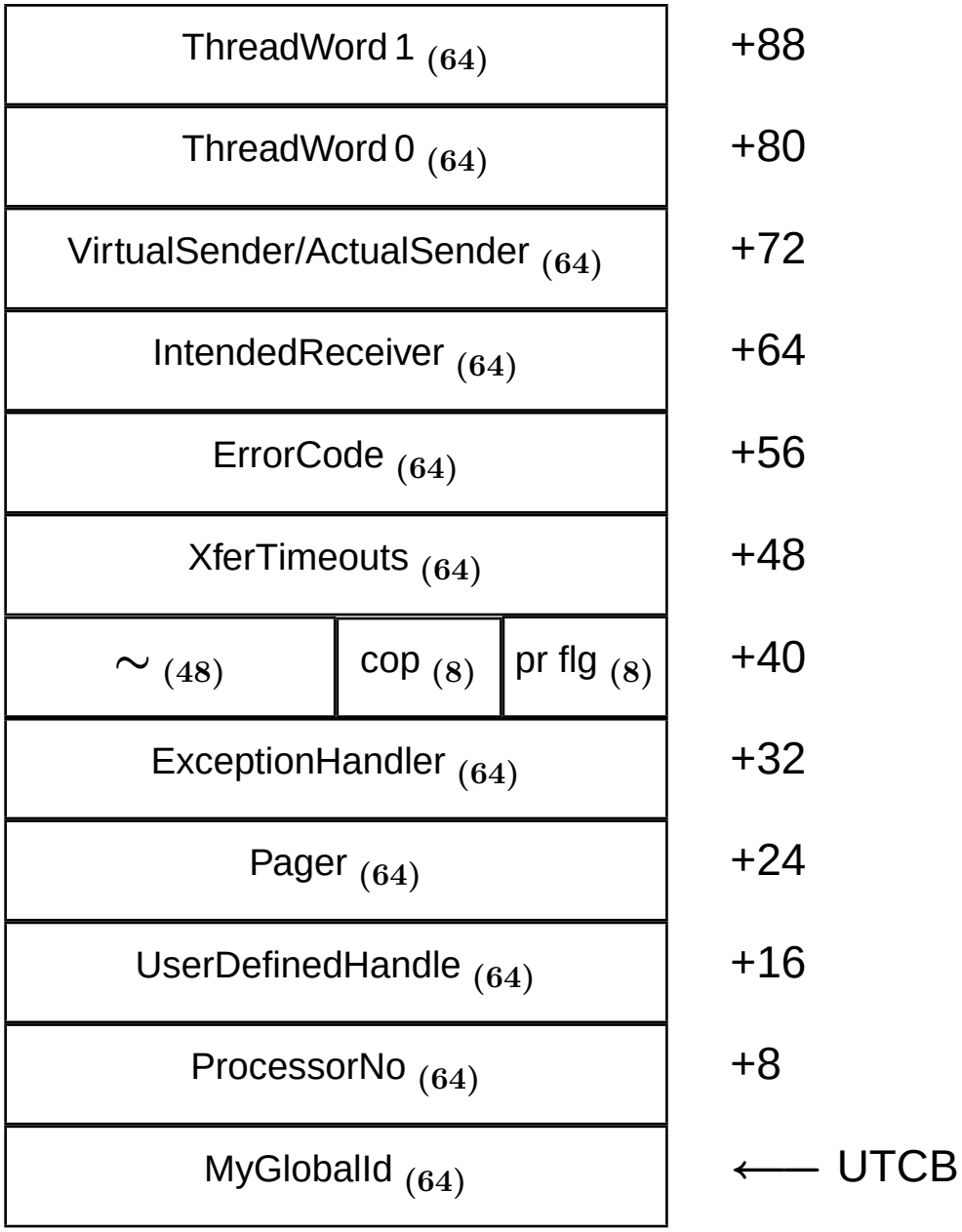
Thread Control Block (TCB)

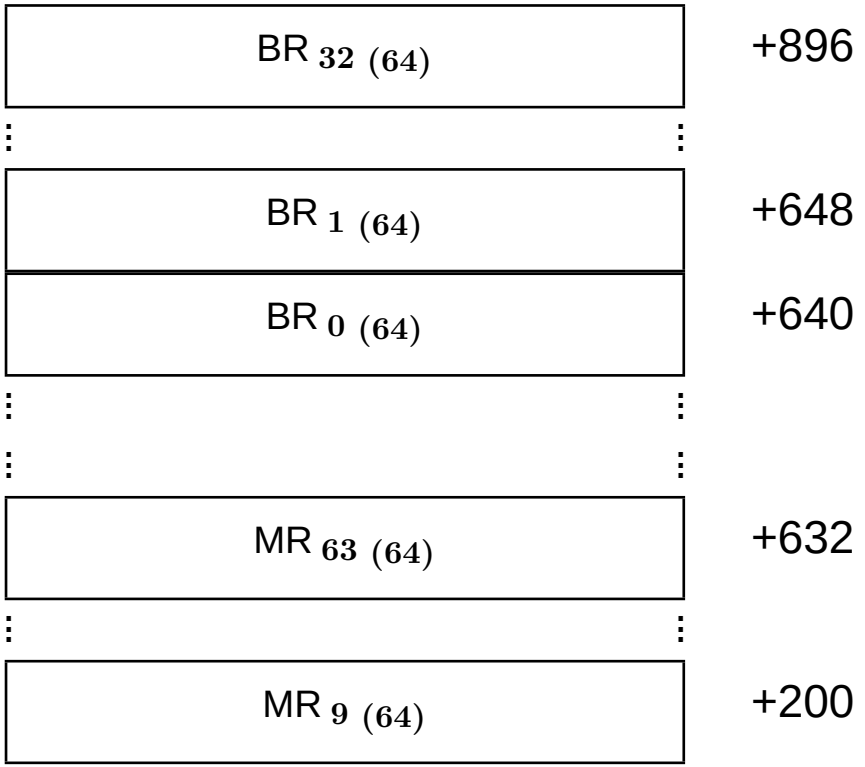
- Contains thread state
  - ⋆ kernel-controlled state, must only be modified by syscalls
  - ⋆ state that can be exposed to user w/o compromising security

Thread Control Block (TCB)

- Contains thread state
  - ⋆ kernel-controlled state, must only be modified by syscalls
    - ➜ kept in kernel TCB (KTCB)
  - ⋆ state that can be exposed to user w/o compromising security
    - ➜ kept in *user-level TCB* (UTCB)
    - ➜ includes virtual registers (as far as not bound to real registers)
    - ➜ *must only be modified via the provided library functions!*
      No consistency guarantees otherwise
    - ➜ many fields only modified as side effect of some operations (IPC)

User-Level TCB (MIPS-64)





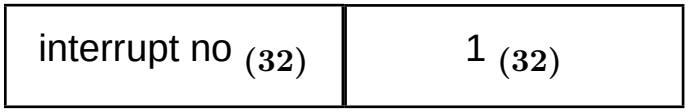
MR $_0$...MR $_8$ are in registers **v1**, **s0**...**s7**

THREAD IDENTIFIERS

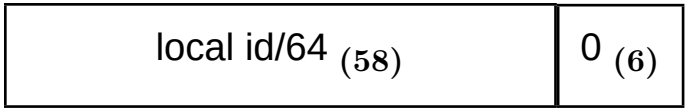
- Global IDs
  - ➜ uniquely identify a thread system-wide
  - ➜ defined by root task at thread creation
    - ➜ ... according to some policy
    - ➜ Note: $\text{version}_{[5..0]} \neq 0$
- Local IDs
  - ➜ identify a thread within its address space
    - ➜ can only be used within AS
    - ➜ supports some optimisations
    - ➜ typically identical to the thread's UTCB address
- Can translate one into the other

Global Thread ID

Global Interrupt ID



Local Thread ID



ThreadControl()

- Create, destroy, modify threads
  - ➜ privileged system call (can only be performed by root task)
- Determines thread attributes
  - ➜ global thread ID
  - ➜ address space
  - ➜ thread permitted to control scheduling parameter
    - ➜ this is known as the target thread's *scheduler*
    - ➜ note: the "scheduler" thread doesn't actually perform CPU scheduling!
  - ➜ page fault hander ("*pager*")
  - ➜ location of thread's UTCB within the UTCB area of the thread's AS
    - ➜ implicitly defines the local thread ID (= UTCB array index)

ThreadControl()

- Can create threads *active* or *inactive*
  - ⋆ thread is active iff it has a pager
  - ⋆ creation of inactive threads is used to
    - ➜ create and manipulate new address spaces
    - ➜ allocate new threads to existing address spaces

ThreadControl()

- Can create threads *active* or *inactive*
  - ⋆ thread is active iff it has a pager
  - ⋆ creation of inactive threads is used to
    - ➜ create and manipulate new address spaces
    - ➜ allocate new threads to existing address spaces
  - ⋆ inactive threads can be activated in one of two ways
    - ➜ by a 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 space,
                            L4_ThreadId_t scheduler,
                            L4_ThreadId_t pager,
                            void          *utcb)
```

TASK

- L4 does not define a concept of a “task”

TASK

- L4 does not define a concept of a “task”
- We use it informally meaning:
  - ⋆ an address space
    - ➜ UTCB area
    - ➜ kernel interface page
    - ➜ redirector
  - ⋆ thread(s) inside that address space
    - ➜ global thread ID
    - ➜ UTCB location (= local thread ID)
    - ➜ IP, SP
    - ➜ pager
    - ➜ scheduler
    - ➜ exception handler
  - ⋆ code, data, stack(s) mapped into address space

Creating a Task

1. Create inactive thread in a new address space
   - Note: L4 does not (presently) support first-class names for AS!
   - An AS is referred to via the ID of one of its threads

```
L4_ThreadId_t task =  according to policy;
L4_ThreadId_t me   = L4_Myself();
L4_ThreadControl (task,              /* new TID */
                  task,              /* new address space */
                  me,                /* scheduler of new thread */
                  L4_nilthread,      /* pager, nil=inactive */
                  (void*)-1);        /* no utcb yet */
```

   ... creates a new thread in an otherwise empty address space

Creating a Task...

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

```
L4_SpaceControl (task,                  /* new TID */
                 0,                     /* control */
                 kip_fpage,             /* where KIP is mapped */
                 utcb_fpage,            /* location of UTCB array */
                 L4_anythread,          /* no redirector */
                 &control);
```

Creating a Task...

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

```
L4_SpaceControl (task,                  /* new TID */
                 0,                     /* control */
                 kip_fpage,             /* where KIP is mapped */
                 utcb_fpage,            /* location of UTCB array */
                 L4_anythread,          /* no redirector */
                 &control);
```

3. Define UTCB address of new thread

```
utcb_base = utcb_adr + offset;
L4_ThreadControl (task, task, me,
                  pager,  /* new pager */
                  (void*) utcb_base);
```

Thread will now wait for an IPC containing IP and SP.

Creating a Task...

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

```
L4_SpaceControl (task,              /* new TID */
                 0,                 /* control */
                 kip_fpage,         /* where KIP is mapped */
                 utcb_fpage,        /* location of UTCB array */
                 L4_anythread,      /* no redirector */
                 &control);
```

3. Define UTCB address of new thread

```
utcb_base = utcb_adr + offset;
L4_ThreadControl (task, task, me,
                  pager,  /* new pager */
                  (void*) utcb_base);
```

Thread will now wait for an IPC containing IP and SP.

4. Send IPC to thread containing IP, SP in MR_1, RM_2

➜ thread will then start fetching instructions from IP

Adding Threads to a Task

- Use ThreadControl() to add new threads to AS

```
L4_ThreadId_t tid =  according to policy;
utcb_base =  ...;
L4_ThreadControl (tid, task, me,
                  pager, (void*) utcb_base);
```

ADDING THREADS TO A TASK

- Use ThreadControl() to add new threads to AS

```
L4_ThreadId_t tid = according to policy;
utcb_base = ...;
L4_ThreadControl (tid, task, me,
                  pager, (void*) utcb_base);
```

- Can create new threads inactive instead
  - ➜ task can then manage new threads itself
  - ➜ ... using **`ExchangeRegisters()`**

- Note: Maximum number of threads defined at address-space creation time
  - ➜ via the size of the UTCB area
  - ➜ size and alignment conditions of UTCBs are defined in KIP

Practical Considerations

- Above sequence for creating tasks and threads is cumbersome
  - ➜ price to be paid for leaving policy out of kernel
  - ➜ any shortcuts imply policy

PRACTICAL CONSIDERATIONS

- Above sequence for creating tasks and threads is cumbersome
  - ➜ price to be 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 interfaces for task and thread creation
  - ➜ incorporating system policy
- Actual apps would not use raw L4 system calls, but
  - ➜ use libraries
  - ➜ use IDL compiler (Magpie)

System Calls

- KernelInterface
- ThreadControl
- ExchangeRegisters
- IPC
- ThreadSwitch
- Schedule
- SystemClock
- Unmap
- SpaceControl
- ProcessorControl
- MemoryControl

EXCHANGEREGISTERS()

- Reads, and optionally modifies, kernel-maintained thread state

```
L4_ThreadId_t
L4_ExchangeRegisters (L4_ThreadId_t  dest,
                      L4_Word_t      control,
                      L4_Word_t      sp,
                      L4_Word_t      ip,
                      L4_Word_t      flags,
                      L4_Word_t      usr_handle,
                      L4_ThreadId_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_usr_handle,
                      L4_ThreadId_t *old_pager)
```

EXCHANGEREGISTERS()

- Reads, and optionally modifies, kernel-maintained thread state

```
L4_ThreadId_t
L4_ExchangeRegisters (L4_ThreadId_t  dest,
                      L4_Word_t      control,
                      L4_Word_t      sp,
                      L4_Word_t      ip,
                      L4_Word_t      flags,
                      L4_Word_t      usr_handle,
                      L4_ThreadId_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_usr_handle,
                      L4_ThreadId_t *old_pager)
```

  - ⋆ setting pager activates inactive thread

EXCHANGEREGISTERS()

- Reads, and optionally modifies, kernel-maintained thread state

```
L4_ThreadId_t
L4_ExchangeRegisters (L4_ThreadId_t  dest,
                      L4_Word_t      control,
                      L4_Word_t      sp,
                      L4_Word_t      ip,
                      L4_Word_t      flags,
                      L4_Word_t      usr_handle,
                      L4_ThreadId_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_usr_handle,
                      L4_ThreadId_t *old_pager)
```

  - ⋆ setting pager activates inactive thread
  - ⋆ **`usr_handle`** is an arbitrary user-defined value
    - ➜ can be used to implement thread-local storage
  - ⋆ **`flags`** allows setting processor status bits

ExchangeRegisters()

Status-register bits affected by flags (MIPS)

Bit	Name	Effect
31	XX	enable MIPS V4 instructions (R4700 ignored)
27	RP	reduced power mode (R4700 ignored)
26	FR	enable floating-point registers 16...31
25	RE	reverse endianness in user mode
23	PX	enable 64-bit instructions in user mode
5	UX	enable 64-bit addressing in user mode

Threads and Stacks

- Kernel does not allocate or manage stacks in any way
  - ➜ only preserves IP, SP on context switch
- User level (servers) must manage
  - ➜ stack location, allocation, size
  - ➜ entry point address
  - ➜ thread ID allocation, deallocation
  - ➜ UTCB slot allocation, deallocation
    - ➜ KIP specifies UTCB space requirements and alignment conditions
- Beware of stack overflow!

System Calls

- KernelInterface
- ThreadControl
- ExchangeRegisters
- ➜ IPC
- ThreadSwitch
- Schedule
- SystemClock
- Unmap
- SpaceControl
- ProcessorControl
- MemoryControl

IPC Overview

- Single IPC syscall incorporates a send and a receive phase
  - ➜ 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 the system, or any local thread (same AS)

IPC Overview

- Single IPC syscall incorporates a send and a receive phase
  - ➜ 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 the system, or any local thread (same AS)
- Results in five different logical operations
  - ⋆ **Send()**: send msg to specified thread
  - ⋆ **Receive()**: receive msg from specified thread
  - ⋆ **Wait()**: receive msg from any thread
  - ⋆ **Call()**: send msg to specified thread and wait for reply
    - ➜ typical client operation
  - ⋆ **Reply&Wait()**: send msg to specified thread and wait for any message
    - ➜ typical server operation

IPC Registers

- Message registers
- Buffer registers

IPC Registers

- Message registers
  - ★ 64 virtual registers
    - ➜ on MIPS 9 physical registers, 55 in UTCB
  - ★ contents form message
    - ➜ untyped words
    - ➜ “typed items”
      - ➜ MapItem
      - ➜ GrantItem
      - ➜ StringItem
- Buffer registers

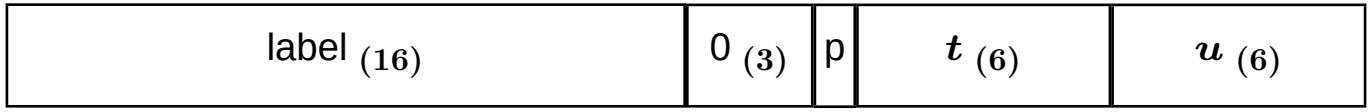
IPC Registers

- Message registers
  - ★ 64 virtual registers
    - ➜ on MIPS 9 physical registers, 55 in UTCB
  - ★ contents form message
    - ➜ untyped words
    - ➜ “typed items”
      - ➜ MapItem
      - ➜ GrantItem
      - ➜ StringItem
- Buffer registers
  - ★ 34 virtual registers
    - ➜ all in UTCB
  - ★ specify map/grant/strings
    - ➜ willingness to receive them
    - ➜ destinations for them

IPC Registers

- Message registers
  - ⋆ 64 virtual registers
    - ➜ on MIPS 9 physical registers, 55 in UTCB
  - ⋆ contents form message
    - ➜ untyped words
    - ➜ “typed items”
      - ➜ MapItem
      - ➜ GrantItem
      - ➜ StringItem
- Buffer registers
  - ⋆ 34 virtual registers
    - ➜ all in UTCB
  - ⋆ specify map/grant/strings
    - ➜ willingness to receive them
    - ➜ destinations for them
- 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 Tag MR_0



- Specifies message content

  u number of untyped words in message

  t number of words holding typed items

  label user-defined (e.g., opcode)

  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)

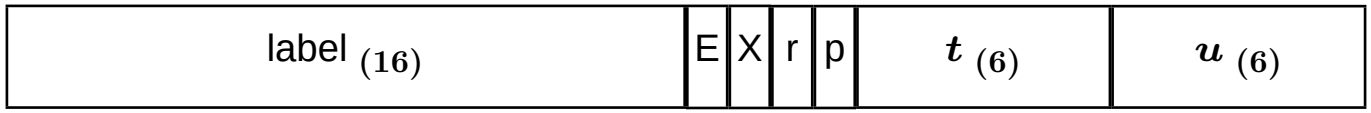
EXAMPLE: SENDING 4 UNTYPED WORDS


```
L4Msg_t msg;
L4MsgTag_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);
```

Delivers MR_0, ..., MR_4 to thread **`tid`**

Note: Should use IDL compiler rather then doing this manually!

IPC Result MR_0



- Specifies details of received message

  u number of untyped words received

  t number of typed words received

  E error occurred, check ErrorCode in UTCB

  X message came from another CPU

  r message was redirected (later)

  p sender used propagation, check **`ActualSender`** in UTCB

Typed Items: StringItems

- Support sending of out-of-line data
  - ➜ buffers pointed to by StringItems in message registers

TYPED ITEMS: STRINGITEMS

- Support sending of out-of-line data
  - ➜ buffers pointed to by StringItems in message registers
- **DO NOT USE!**
  - ➜ stings have a number of distasteful side effects on the kernel
  - ➜ we are in the process of removing them from the API
  - ➜ just to be sure, they aren't implemented on MIPS ☺

IPC Timeouts

- Remember: All L4 IPC is synchronous, i.e. *blocking*
- Timeouts are used to limit blocking time

IPC Timeouts

- Remember: All L4 IPC is synchronous, i.e. *blocking*
- Timeouts are used to limit blocking time
- Two types of timeouts:
  - ⋆ send/receive timeouts limit blocking to commencement of transfer
    - ➜ SndTimeout from invoking system to start of send
    - ➜ RcvTimeout from end of send (or invocation if no send) to start of receive
  - ⋆ transfer timeouts (SndXfer, RcvXfer) limit duration of transfer
    - ➜ only relevant for IPC with StringItems (due to possible page faults)!
    - ➜ ignore

IPC TIMEOUTS

- Remember: All L4 IPC is synchronous, i.e. *blocking*
- Timeouts are used to limit blocking time
- Two types of timeouts:
  - ⋆ send/receive timeouts limit blocking to commencement of transfer
    - ➜ SndTimeout from invoking system to start of send
    - ➜ RcvTimeout from end of send (or invocation if no send) to start of receive
  - ⋆ transfer timeouts (SndXfer, RcvXfer) limit duration of transfer
    - ➜ only relevant for IPC with StringItems (due to possible page faults)!
    - ➜ ignore
- Timeout specification kept in UTCB

IPC Timeouts

- Timeout specifications can be
  - ⋆ relative time period (to time of IPC syscall invocation)
  - ⋆ absolute time point

IPC TIMEOUTS

- Timeout specifications can be
  - ⋆ relative time period (to time of IPC syscall invocation)

0	$1_{(5)}$	$0_{(10)}$	0
0	$e_{(5)}$	$m_{(10)}$	$m \times 2^e \mu$sec (1mus ... 610h)
$0_{(16)}$			∞

    ➜ use convenience function **`TimePeriod(`**μ**`s)`** to build timeout values
  - ⋆ absolute time point

IPC TIMEOUTS

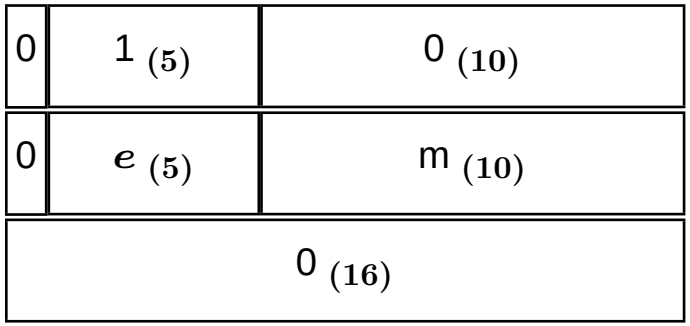
- Timeout specifications can be
  - ★ relative time period (to time of IPC syscall invocation)

0	$1_{(5)}$	$0_{(10)}$	0
0	$e_{(5)}$	$m_{(10)}$	$m \times 2^e \mu$sec (1mus ... 610h)
$0_{(16)}$			∞

    - ➜ use convenience function **`TimePeriod(`**μ**`s)`** to build timeout values
  - ★ absolute time point
    - ➜ allow specification of a future clock value
    - ➜ accuracy decreasing into the future
    - ➜ use convenience functions to build time point values
    - ➜ check L4 Reference Manual for details

IPC TIMEOUTS

- Timeout specifications can be
  - ⋆ relative time period (to time of IPC syscall invocation)



0

$m \times 2^e \mu$sec (1mus ... 610h)

∞

    - ➜ use convenience function **`TimePeriod(`**μ**`s)`** to build timeout values
  - ⋆ absolute time point
    - ➜ allow specification of a future clock value
    - ➜ accuracy decreasing into the future
    - ➜ use convenience functions to build time point values
    - ➜ check L4 Reference Manual for details
- Timeouts also used for timed sleep
  - ➜ receive with timeout from non-existing thread

INTERRUPTS

- Modelled as IPC messages sent by virtual hardware threads
  - ➜ received by interrupt handler thread registered for that interrupt
  - ➜ empty (MR_0=0) reply to interrupt thread acknowledges interrupt
- Interrupt handler association is via ThreadControl()
  - ➜ set the hardware thread's *pager* to the handler thread
  - ➜ disassociate by setting the pager to the hardware thread's own ID

System Calls

- KernelInterface
- ThreadControl
- ExchangeRegisters
- IPC
- ➜ ThreadSwitch
- Schedule
- SystemClock
- Unmap
- SpaceControl
- ProcessorControl
- MemoryControl

ThreadSwitch()

- Forfeits the caller’s remaining time slice

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

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
  - ⋆ 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 tread will execute to the end of the time slice on the donor's priority
  - ⋆ 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
- SystemClock
- Unmap
- SpaceControl
- ProcessorControl
- MemoryControl

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 *lazy scheduling*

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 *lazy scheduling*
  - ➜ scheduler only invoked if destination is blocked too
  - ➜ if both threads are runnable after IPC, the higher-prio one will run
    - ➜ presently not always done 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 *lazy scheduling*
  - ➜ scheduler only invoked if destination is blocked too
  - ➜ if both threads are runnable after IPC, the higher-prio one will run
    - ➜ presently not always done correctly
- This makes (expensive) scheduler invocation infrequent

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*

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

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

SCHEDULE()

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

SCHEDULE()

- 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()`**

SCHEDULE()

- 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
- ➜ SystemClock
- Unmap
- SpaceControl
- ProcessorControl
- MemoryControl

SystemClock

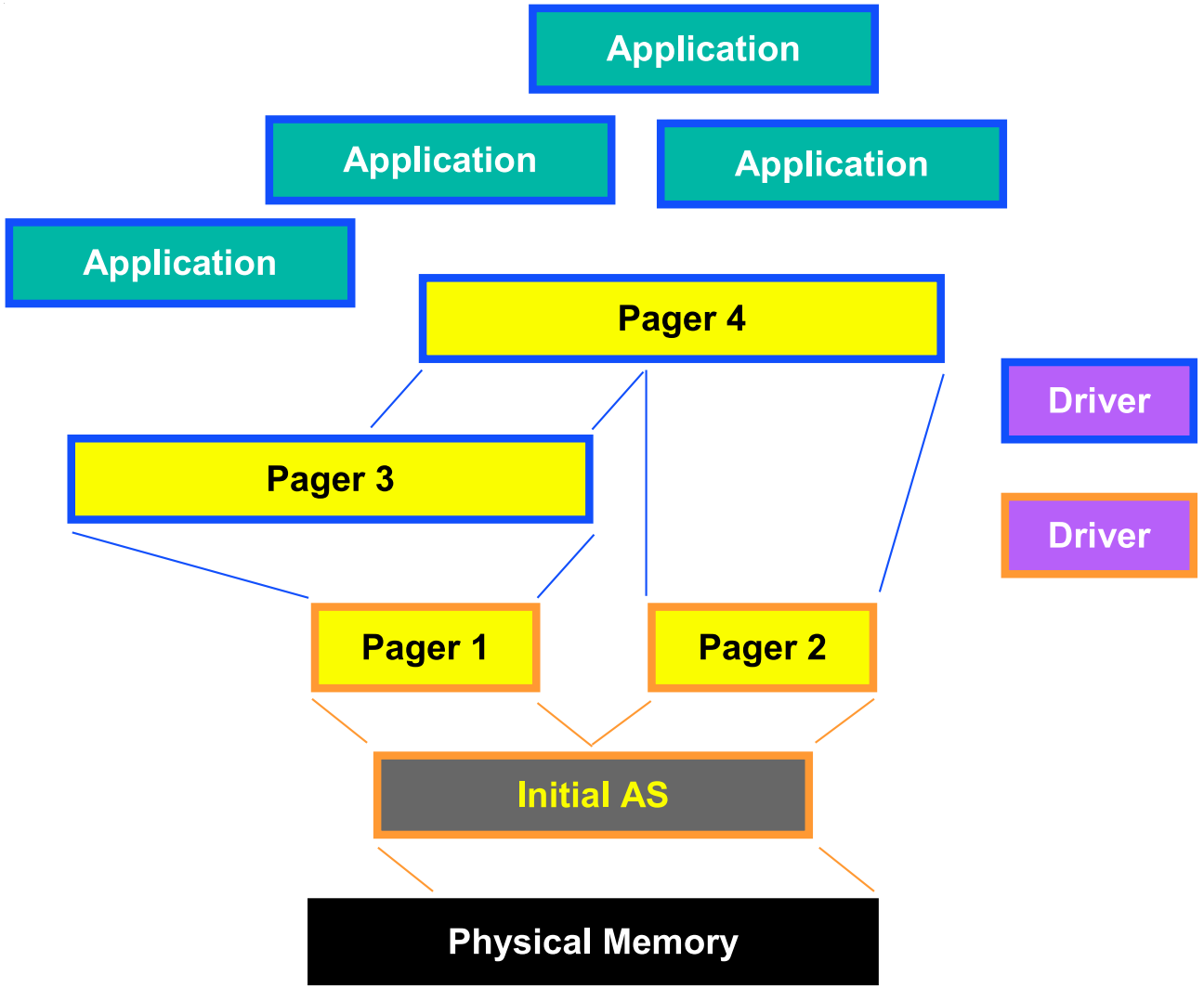
- Returns current system clock value
  - ➜ 64-bit value counting microseconds
- Usually not a real system call

System Calls

- KernelInterface
- ThreadControl
- ExchangeRegisters
- IPC
- ThreadSwitch
- Schedule
- SystemClock
- ➜ Unmap (and mapping in general)
- SpaceControl
- ProcessorControl
- MemoryControl

Address Spaces

Remember, address spaces in L4 are constructed recursively

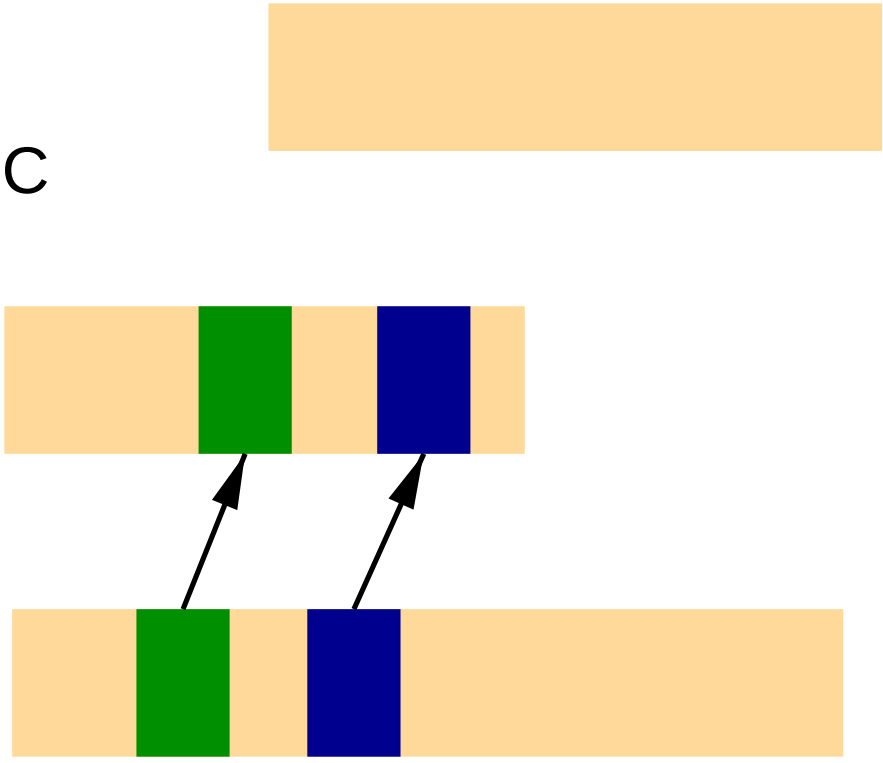


Address-Space Operations

- 3 basic operations for address-space construction
  - ➜ map
  - ➜ grant
  - ➜ flush/unmap
- Map, grant are performed during IPC
  - ➜ MapItem, GrantItem
- Unmap is a separate system call

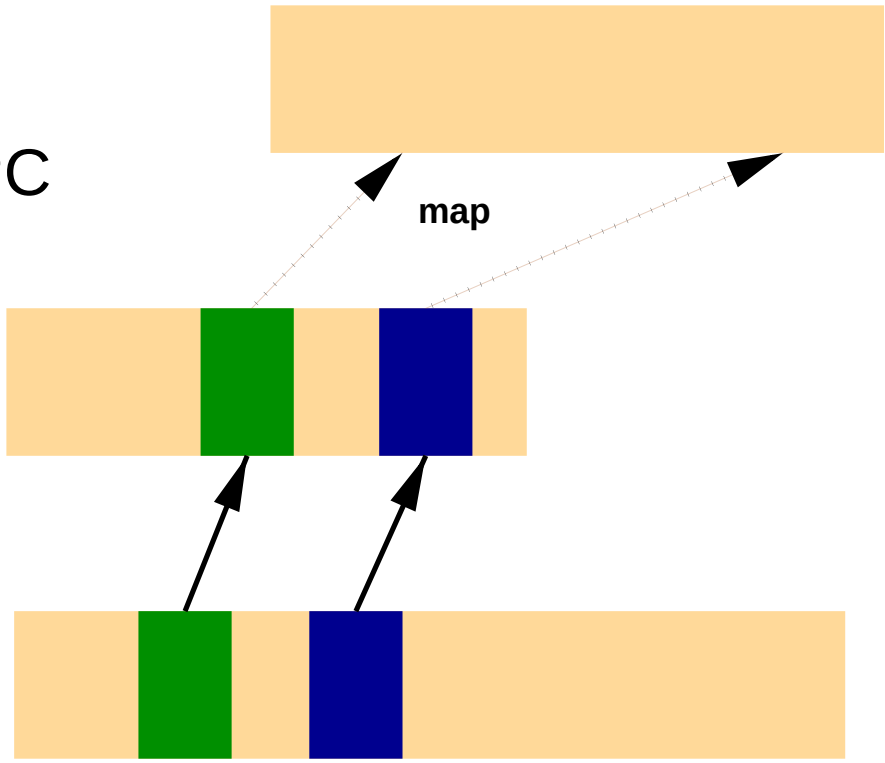
Address-Space Operations

- 3 basic operations for address-space construction
  - ➜ map
  - ➜ grant
  - ➜ flush/unmap
- Map, grant are performed during IPC
  - ➜ MapItem, GrantItem
- Unmap is a separate system call



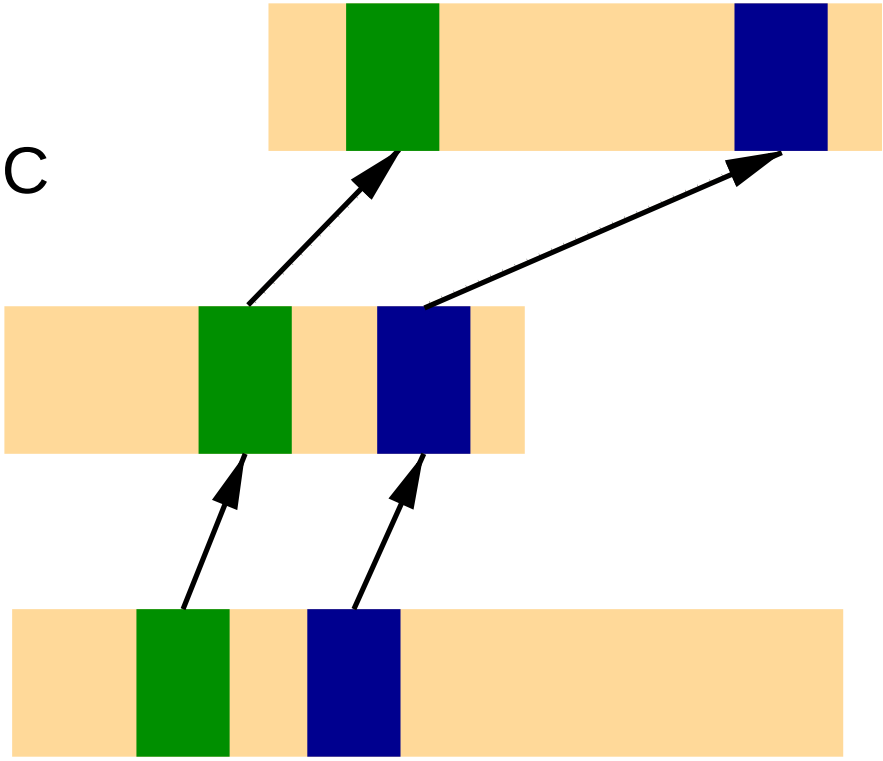
Address-Space Operations

- 3 basic operations for address-space construction
  - ➜ map
  - ➜ grant
  - ➜ flush/unmap
- Map, grant are performed during IPC
  - ➜ MapItem, GrantItem
- Unmap is a separate system call



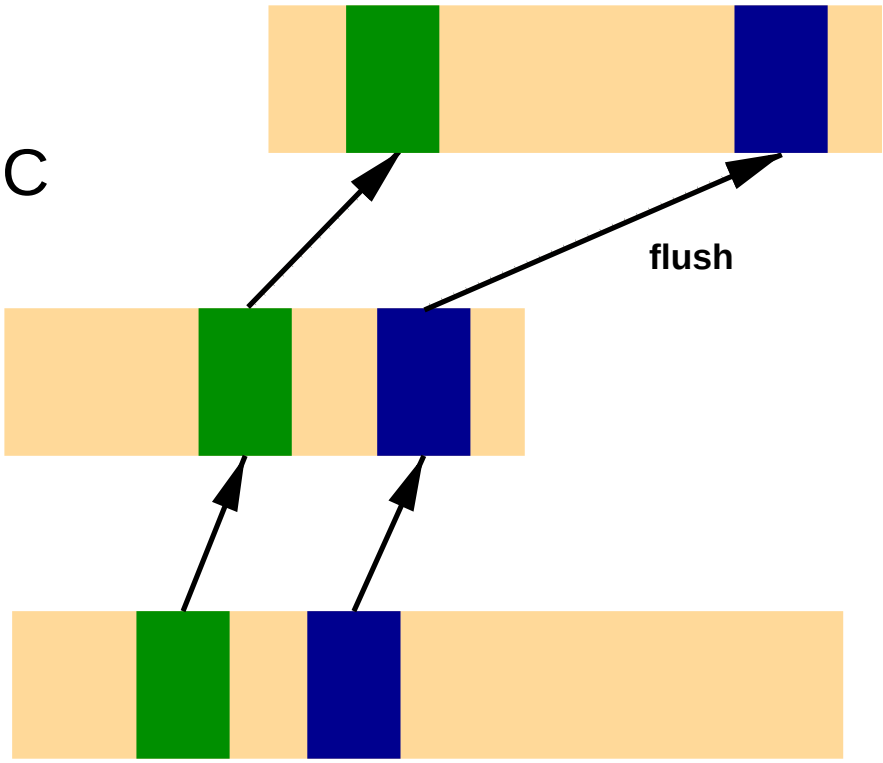
Address-Space Operations

- 3 basic operations for address-space construction
  - ➜ map
  - ➜ grant
  - ➜ flush/unmap
- Map, grant are performed during IPC
  - ➜ MapItem, GrantItem
- Unmap is a separate system call



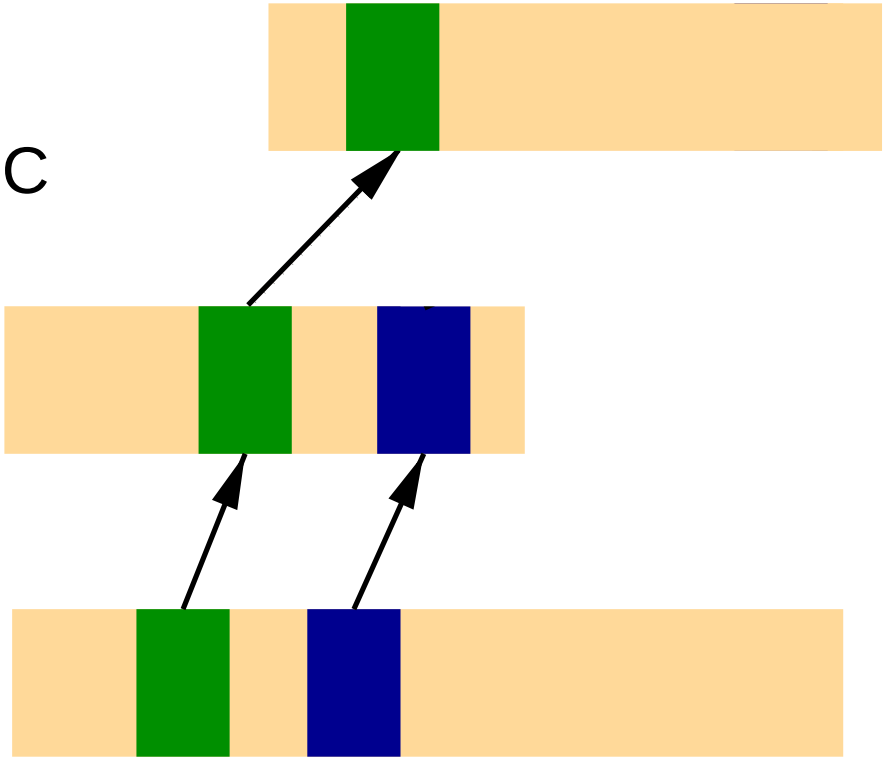
Address-Space Operations

- 3 basic operations for address-space construction
  - ➜ map
  - ➜ grant
  - ➜ flush/unmap
- Map, grant are performed during IPC
  - ➜ MapItem, GrantItem
- Unmap is a separate system call



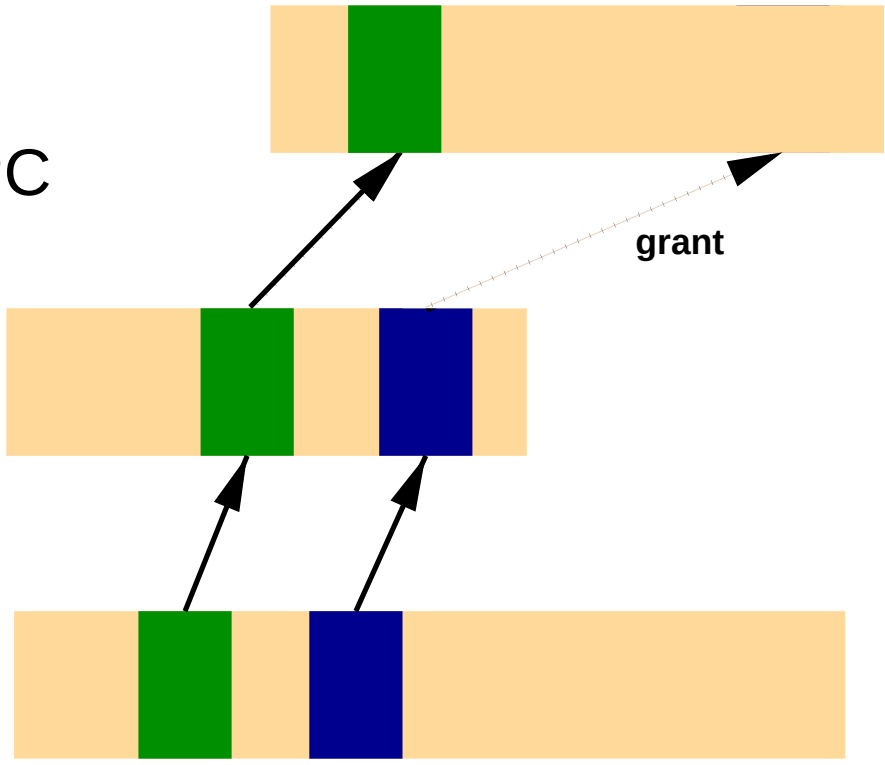
Address-Space Operations

- 3 basic operations for address-space construction
  - ➜ map
  - ➜ grant
  - ➜ flush/unmap
- Map, grant are performed during IPC
  - ➜ MapItem, GrantItem
- Unmap is a separate system call



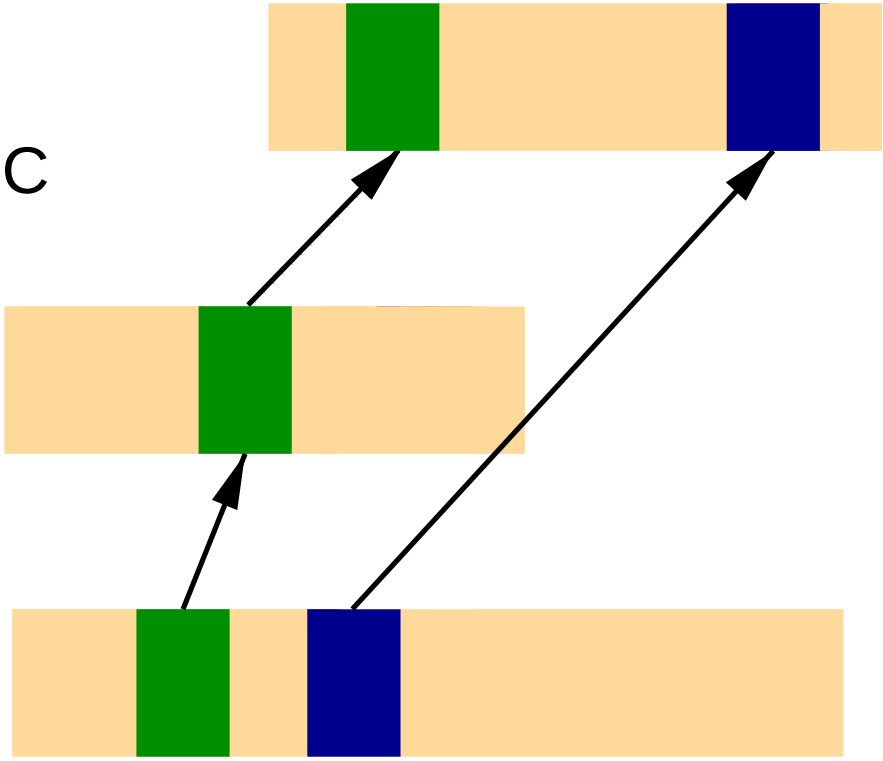
Address-Space Operations

- 3 basic operations for address-space construction
  - ➜ map
  - ➜ grant
  - ➜ flush/unmap
- Map, grant are performed during IPC
  - ➜ MapItem, GrantItem
- Unmap is a separate system call



Address-Space Operations

- 3 basic operations for address-space construction
  - ➜ map
  - ➜ grant
  - ➜ flush/unmap
- Map, grant are performed during IPC
  - ➜ MapItem, GrantItem
- Unmap is a separate system call



Page Fault Handling

- Page faults are converted into IPC messages

Page Fault Handling

- Page faults are converted into IPC messages:

① app triggers page fault

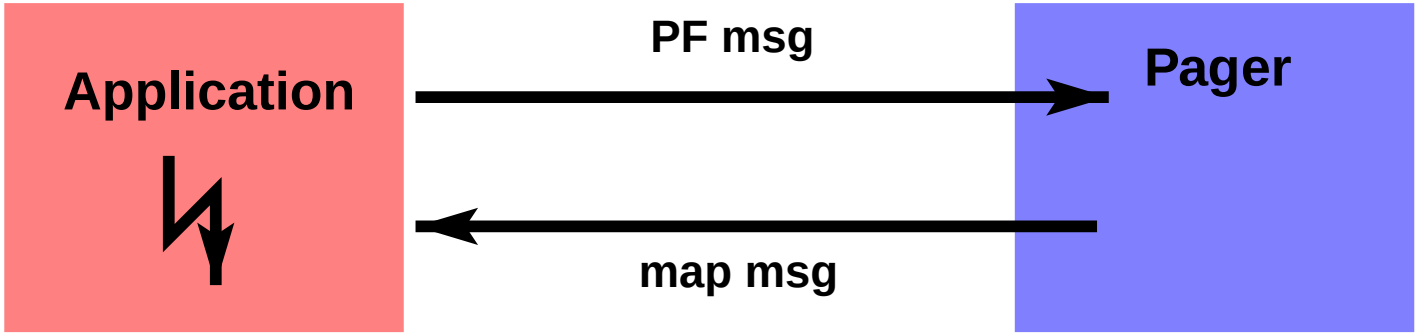
Page Fault Handling

- 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

- Page faults are converted into IPC messages:

① app triggers page fault
② kernel exception handler generates IPC from faulter to pager
③ pager responds with mapping message
④ kernel intercepts message, establishes mapping
⑤ kernel restarts faulting thread



- How are mappings specified?

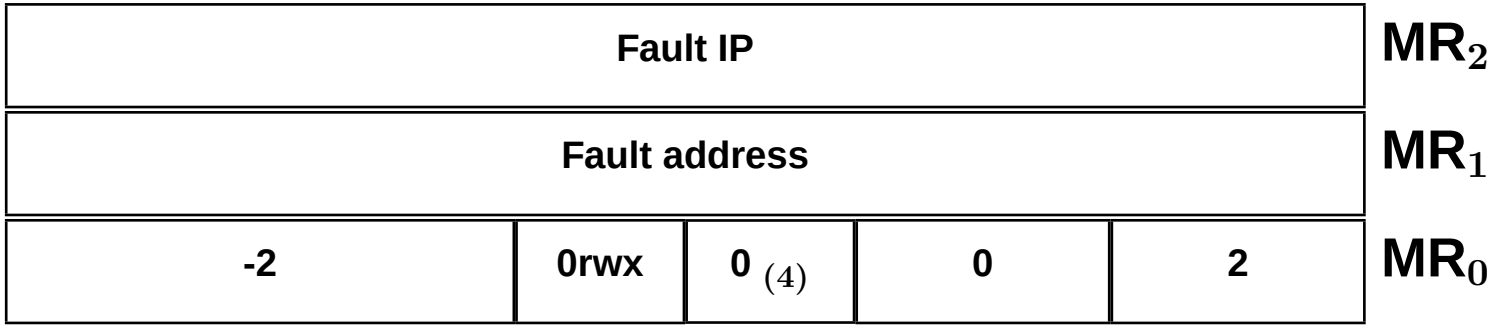
Page Fault Message

- Format of kernel-generated page fault message

Fault IP					MR_2
Fault address					MR_1
-2	0rwx	0 (4)	0	2	MR_0

PAGE FAULT MESSAGE

- Format of kernel-generated page fault message



- Obviously, application can manufacture same message
  - ➜ pager cannot tell the difference
  - ➜ not a problem, as application could achieve the same by forcing a fault

Specifying Mappings: Fpages

- 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

base/1024	s (6)	∼ (4)

  - ⋆ special fpages:

0	**0x3f**	∼ (4)	**full AS**
0	**0** (6)	**0** (4)	**nil page**

  - ⋆ On MIPS, $s \geq 12$

Buffer Register BR_0

- Specifies willingness to receive typed items

receive window fpage	000s

  - ⋆ s-bit indicates willingness to receive string items
  - ⋆ fpage defines receive window for mappings
    - ➜ limits region where mappings can be established

Buffer Register BR_0

- Specifies willingness to receive typed items

receive window fpage	000s

  - ⋆ s-bit indicates willingness to receive string items
  - ⋆ fpage defines receive window for mappings
    - ➜ limits region where mappings can be established

- Eg. page fault at address 0x2002
  - ⋆ kernel sends

Fault IP					MR_2
0x2002					MR_1
-2	0rwx	0 (4)	0	2	MR_0

... and sets up receive as

0	0x3f	0 (4)	BR_0

Buffer Register BR_0

- Specifies willingness to receive typed items

receive window fpage	000s

  - ⋆ s-bit indicates willingness to receive string items
  - ⋆ fpage defines receive window for mappings
    - ➜ limits region where mappings can be established

- Eg. page fault at address 0x2002
  - ⋆ kernel sends

Fault IP					MR_2
0x2002					MR_1
-2	0rwx	0 $_{(4)}$	0	2	MR_0

... and sets up receive as

0	0x3f	0 $_{(4)}$	BR_0

- Works same way for explicit mapping receive

MAPITEM/GRANTITEM

- Specifies an fpage to be mapped or granted

send fpage		$0rwx$
send base/1024	0 $_{(6)}$	$10gC$

➜ rxw: permissions
➜ g: 1: grant, 0: map
➜ C: continuation bit, 1: more items follow

MapItem/GrantItem

- Specifies an fpage to be mapped or granted

send fpage		$0rwx$
send base/1024	0 $_{(6)}$	$10gC$

  - ➜ rxw: permissions
  - ➜ g: 1: grant, 0: map
  - ➜ C: continuation bit, 1: more items follow

- E.g., pager handles write page fault at 0x2002
  - ⋆ to map 4kB page at 0xc0000, pager sends:

0x300		12	6	MR_2
0x80		0 $_{(6)}$	$10gC$	MR_1
0	0	0	2	MR_0

  - ⋆ send base (use fault address) determines where to map the page
  - ⋆ kernel expects message with a single MapItem or GrantItem only

Mapping Rules

- Receive window size = fpage size: mapping fully defined
- Otherwise, send base is used to disambiguate:

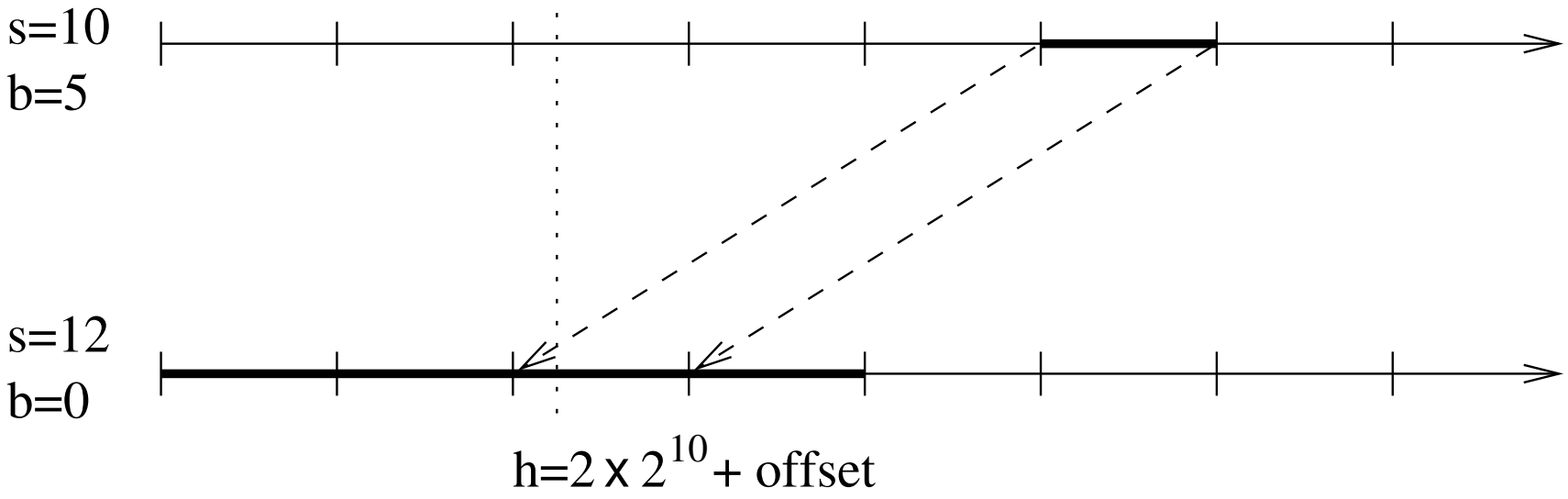
Mapping Rules

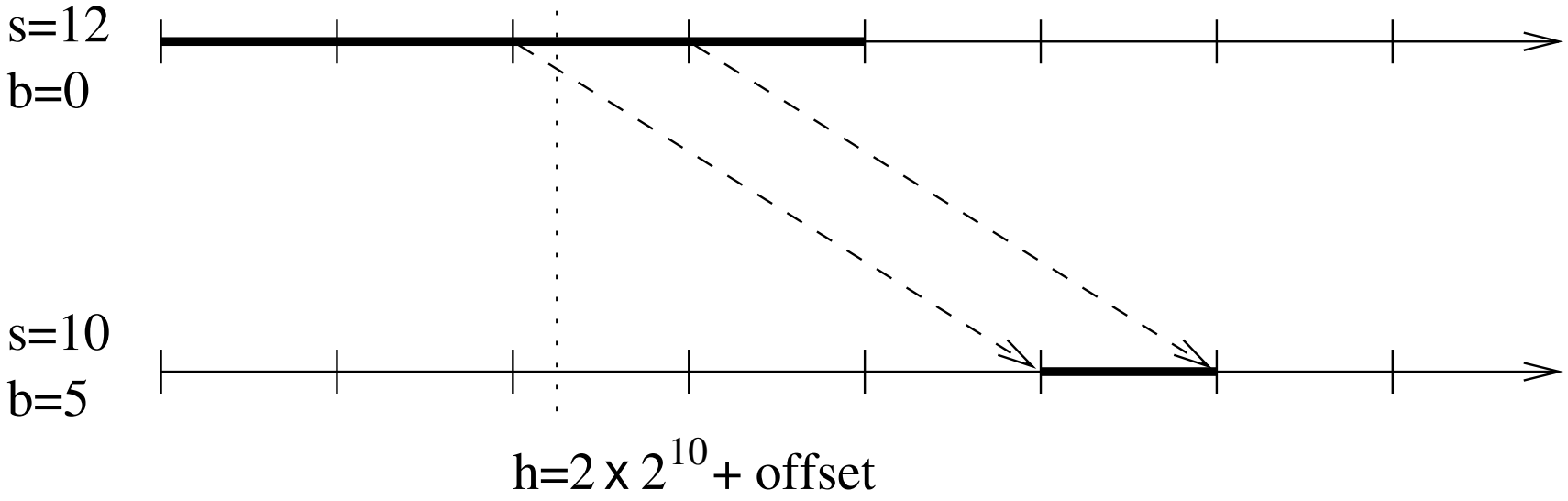
- Receive window size = fpage size: mapping fully defined
- Otherwise, send base is used to disambiguate:
  - ⋆ Send fpage is taken modulo receive fpage size.
    - ➜ Uniquely determines a page in the receiver's address space which will receive a mapping.
  - ⋆ Send fpage is also taken modulo send fpage size.
    - ➜ Uniquely determines a page in the send fpage which will be mapped.

Mapping Rules

- Receive window size = fpage size: mapping fully defined
- Otherwise, send base is used to disambiguate:
  - ⋆ Send fpage is taken modulo receive fpage size.
    - ➜ Uniquely determines a page in the receiver's address space which will receive a mapping.
  - ⋆ Send fpage is also taken modulo send fpage size.
    - ➜ Uniquely determines a page in the send fpage which will be mapped.
- In other words, the smaller fpage is mapped to/from the larger one so that:
  - ➜ it is aligned according to its size, and
  - ➜ it contains the send base

FPAGE MAPPING EXAMPLES





UNMAP()

- Revokes mappings
- Very similar to mapping IPC:
  - ➜ fpages in MRs specify region to be unmapped

UNMAP()

- Revokes mappings
- Very similar to mapping IPC:
  - ➜ fpages in MRs specify region to be unmapped
- Also *partial unmap*
  - ➜ remove some access rights from pages
  - ➜ e.g., RW ➜R

UNMAP( ) ARGUMENTS

- Control:

$0_{(25)}$	f	$k_{(6)}$

  - ⋆ f:
    - ➜ =0: unmap from all spaces which directly or indirectly received the page from the caller address space
    - ➜ =1: also *flush* from caller's address space

UNMAP( ) ARGUMENTS

- Control:

0 $_{(25)}$	f	k $_{(6)}$

  - ⋆ f:
    - ➜ =0: unmap from all spaces which directly or indirectly received the page from the caller address space
    - ➜ =1: also *flush* from caller's address space
  - ⋆ k: MR_0–MR_k contain $k+1$ fpages to unmap

- Fpages:

fpage	0rwx	MR_x

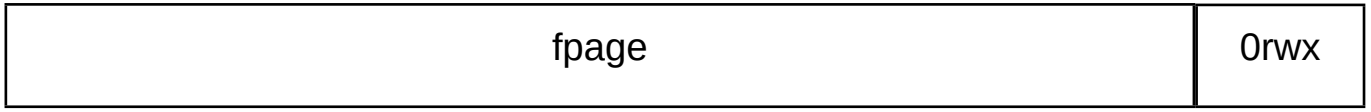
  - ⋆ rwx: rights to invoke, if 0x7 page is completely unmapped

UNMAP() ARGUMENTS

- Control:

  - ⋆ f:
    - ➜ =0: unmap from all spaces which directly or indirectly received the page from the caller address space
    - ➜ =1: also *flush* from caller's address space
  - ⋆ k: MR_0–MR_k contain $k+1$ fpages to unmap

- Fpages:



MR_x

  - ⋆ rwx: rights to invoke, if 0x7 page is completely unmapped

- On return, the kernel updates the rwx bits in the MRs
  - ➜ to reflect hardware-maintained reference, dirty, executed bits
  - ➜ only on architectures with hardware-maintained bits (ix86)

System Calls

- KernelInterface
- ThreadControl
- ExchangeRegisters
- IPC
- ThreadSwitch
- Schedule
- SystemClock
- Unmap
- ➜ SpaceControl
- ProcessorControl
- MemoryControl

SpaceControl()

- Controls layout of new address spaces
  - ➜ KIP location
  - ➜ UTCB area location

SpaceControl()

- Controls layout of new address spaces
  - ➜ KIP location
  - ➜ UTCB area location
- 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:** unimplemented in released version (no restrictions possible)

System Calls

- KernelInterface
- ThreadControl
- ExchangeRegisters
- IPC
- ThreadSwitch
- Schedule
- SystemClock
- Unmap
- SpaceControl
- ➜ ProcessorControl
- MemoryControl

ProcessorControl()

- Sets processor core voltage and frequency (where supported)
- not in R4700

System Calls

- KernelInterface
- ThreadControl
- ExchangeRegisters
- IPC
- ThreadSwitch
- Schedule
- SystemClock
- Unmap
- SpaceControl
- ProcessorControl
- ➜ MemoryControl

MEMORYCONTROL()

- Sets cache attributes of pages
- ***FIXME!***

System Calls

- KernelInterface
- ThreadControl
- ExchangeRegisters
- IPC
- ThreadSwitch
- Schedule
- SystemClock
- Unmap
- SpaceControl
- ProcessorControl
- MemoryControl

That's it!

L4 Protocols

- Page fault
  - ➜ already covered
- Thread start
  - ➜ already covered
- Interrupt
  - ➜ already covered
- ➜ Preemption
- Exception
- Sigma0

PREEMPTION PROTOCOL

- On preemption, the kernel sends a message on behalf of the preempted thread

- Format:

clock high					MR_2
clock low					MR_1
-3	0	0	0	2	MR_0

- Gives the time at which the preemption occurred

L4 Protocols

- Page fault
- Thread start
- Interrupt
- Preemption
- ➜ Exception
- Sigma0

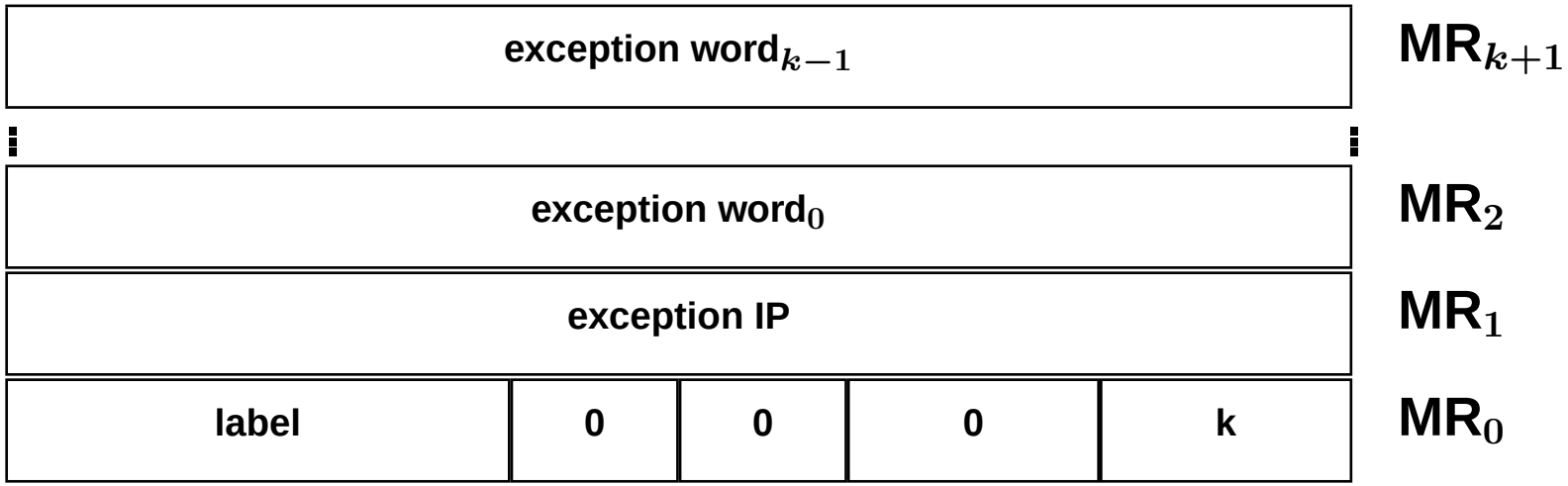
Exception Protocol

- Exceptions also converted into IPC to *exception handler*

Exception Protocol

- Exceptions also converted into IPC to *exception handler*
  - ➜ e.g., arithmetic exceptions, illegal instructions

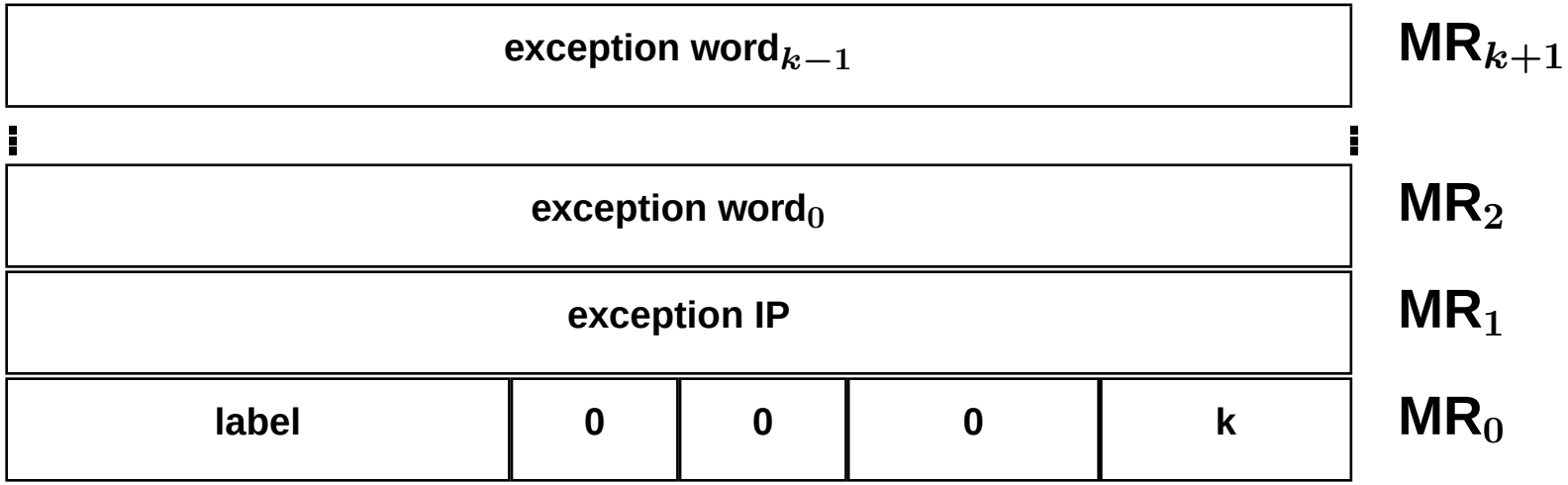
- Exception IPC
  - ⋆ kernel sends (partial) thread state



  - ⋆ label:
    - ➜ -4: standard exceptions, architecture independent
    - ➜ -5: architecture-specific exception

EXCEPTION PROTOCOL

- Exceptions also converted into IPC to *exception handler*
  - ➜ e.g., arithmetic exceptions, illegal instructions
- Exception IPC
  - ⋆ kernel sends (partial) thread state



  - ⋆ 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`**

  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

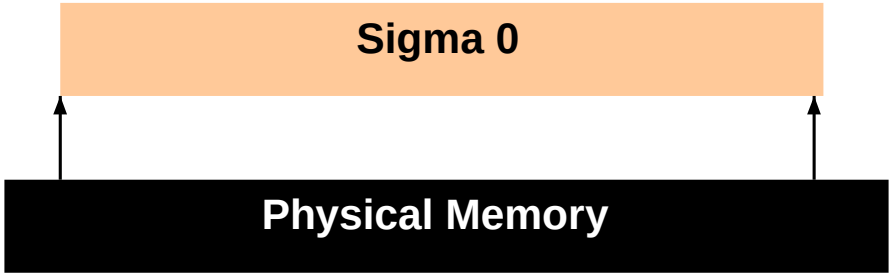
- Page fault
- Thread start
- Interrupt
- Preemption
- Exception
- ➜ Sigma0

SIGMA0

- Initial address space, magically created at boot time
  - ⋆ has a mapping of each free physical frame
    - ➜ everything not reserved for kernel use
    - ➜ identical mapping
  - ⋆ maps pages to other tasks on request
    - ➜ each page is only mapped once
    - ➜ first come, first served
    - ➜ sets send base = page base (identical mapping)
    - ➜ receiver can force arbitrary location via mapping window

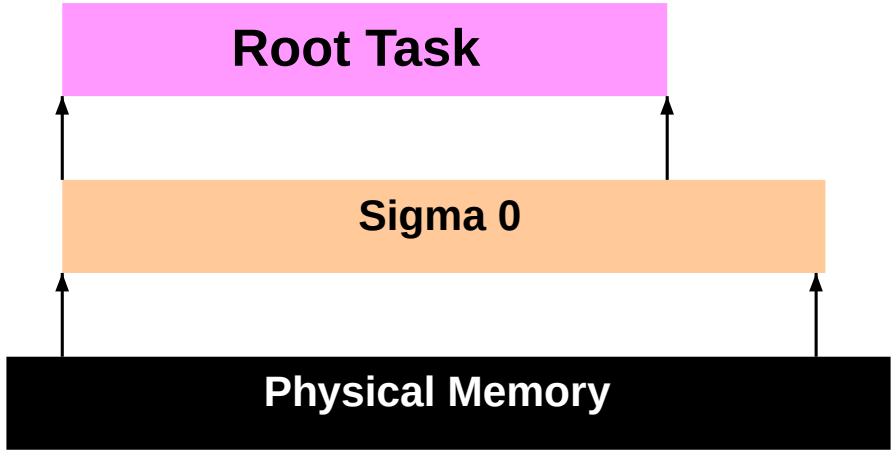
SIGMA0

- Initial address space, magically created at boot time
  - ⋆ has a mapping of each free physical frame
    - ➜ everything not reserved for kernel use
    - ➜ identical mapping
  - ⋆ maps pages to other tasks on request
    - ➜ each page is only mapped once
    - ➜ first come, first served
    - ➜ sets send base = page base (identical mapping)
    - ➜ receiver can force arbitrary location via mapping window



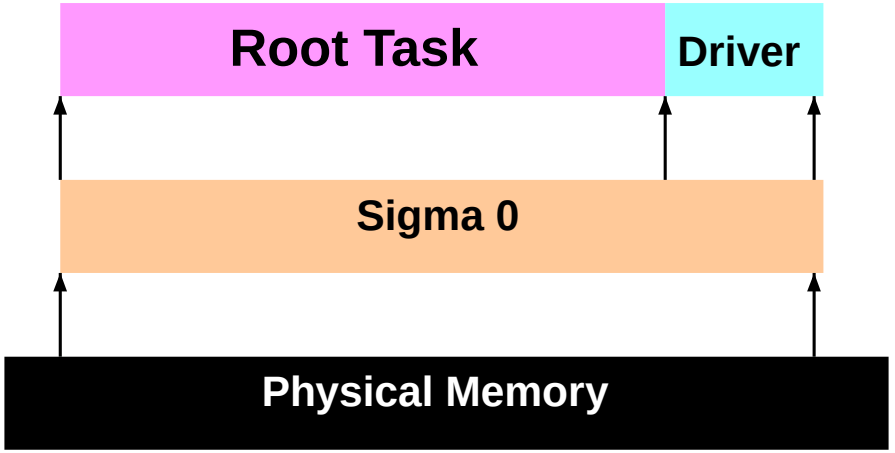
SIGMA0

- Initial address space, magically created at boot time
  - ⋆ has a mapping of each free physical frame
    - ➜ everything not reserved for kernel use
    - ➜ identical mapping
  - ⋆ maps pages to other tasks on request
    - ➜ each page is only mapped once
    - ➜ first come, first served
    - ➜ sets send base = page base (identical mapping)
    - ➜ receiver can force arbitrary location via mapping window



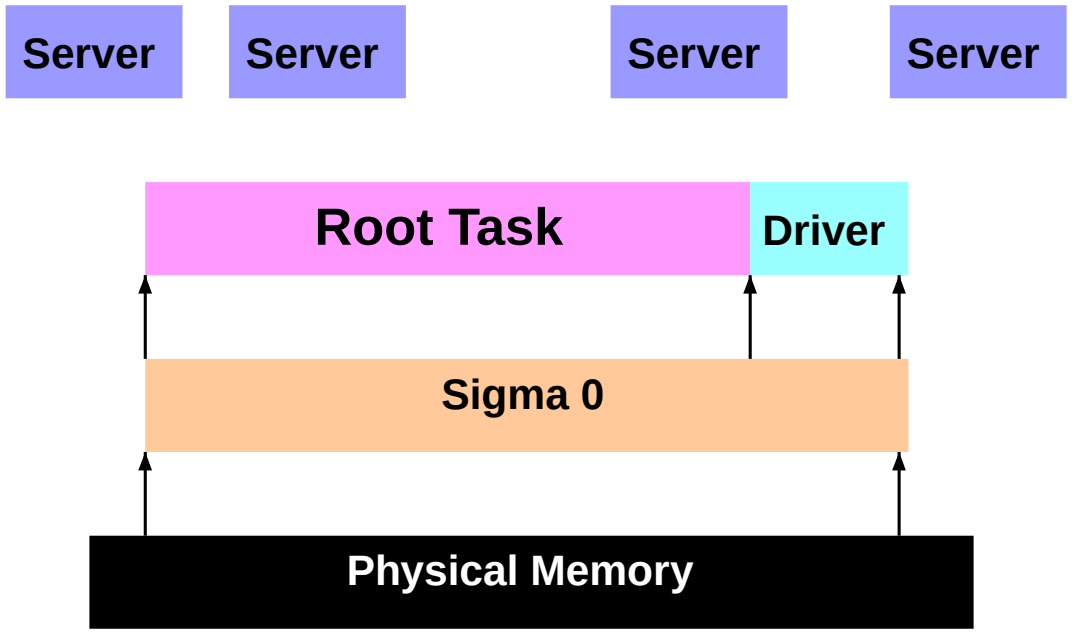
SIGMA0

- Initial address space, magically created at boot time
  - ⋆ has a mapping of each free physical frame
    - ➜ everything not reserved for kernel use
    - ➜ identical mapping
  - ⋆ maps pages to other tasks on request
    - ➜ each page is only mapped once
    - ➜ first come, first served
    - ➜ sets send base = page base (identical mapping)
    - ➜ receiver can force arbitrary location via mapping window



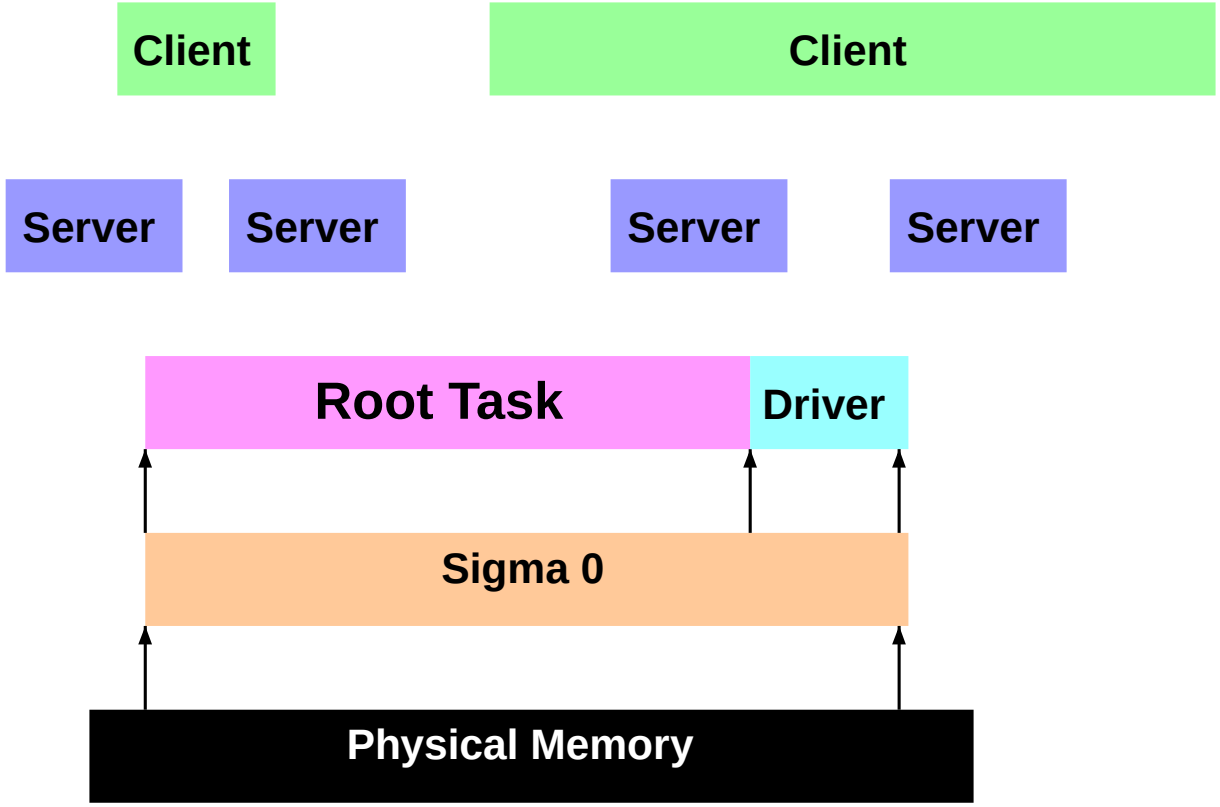
SIGMA0

- Initial address space, magically created at boot time
  - ⋆ has a mapping of each free physical frame
    - ➜ everything not reserved for kernel use
    - ➜ identical mapping
  - ⋆ maps pages to other tasks on request
    - ➜ each page is only mapped once
    - ➜ first come, first served
    - ➜ sets send base = page base (identical mapping)
    - ➜ receiver can force arbitrary location via mapping window



SIGMA0

- Initial address space, magically created at boot time
  - ⋆ has a mapping of each free physical frame
    - ➜ everything not reserved for kernel use
    - ➜ identical mapping
  - ⋆ maps pages to other tasks on request
    - ➜ each page is only mapped once
    - ➜ first come, first served
    - ➜ sets send base = page base (identical mapping)
    - ➜ receiver can force arbitrary location via mapping window



SIGMA0 PROTOCOL

- Memory request to σ_0:

requested memory attributes					MR_2
requested fpage				0rwx	MR_1
-6	0	0	0	k	MR_0

- σ_0 response:
  - ⋆ single fpage mapping (if page available)
  - ⋆ null fpage (if page has already been mapped to someone)

L4 Protocols

- Page fault
- Thread start
- Interrupt
- Preemption
- Exception
- Sigma0