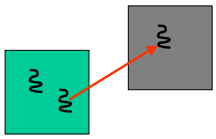
L4 Programming Introduction

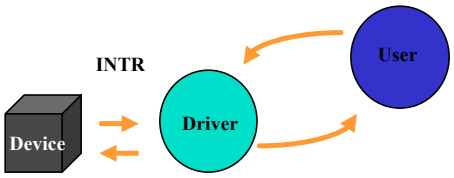
Fundamental Concepts

- Address Spaces
  - Unit of protection, resource management
- Threads
  - Execution abstraction and provide unique identifiers



- Communication: IPC
  - Synchronous
  - Identification: uids
- AS construction: mapping
  - Via IPC
  - Flexpages
    - Architecture independent page abstraction

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**

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 version and configuration of the kernel and the machine it's running on
  - Examples: Page sizes supported, API version, physical memory layout.

`KernelInterface`

- System call provided to locate the kernel information page
- In 'C' api

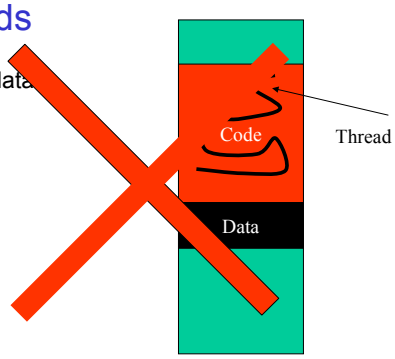
```
void * L4_KernelInterface (L4_Word_t *ApiVersion,
                           L4_Word_t *ApiFlags,
                           L4_Word_t *KernelId)
```

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
- Three basic types
  - 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

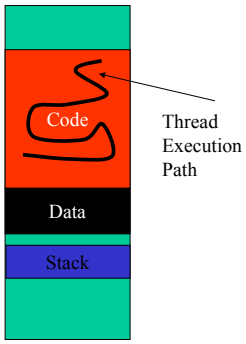
Threads

- Code, data



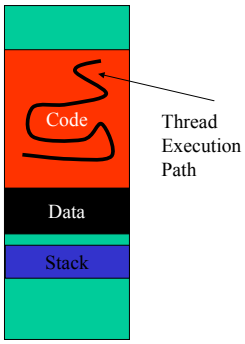
Traditional Thread

- 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 TCRs, MRs, BRs
- A priority and a timeslice
- A unique thread identifier
- An associated address space
- L4 provides a fixed number of threads in the entire system
  - Root task responsible for creating/deleting threads and assigning them to address spaces.
  - System, User and “Hardware” threads

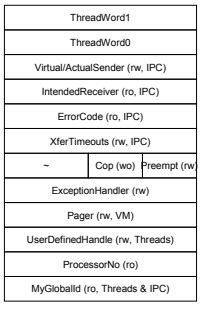


Thread Control Blocks (TCBs)

- State of a thread is stored in it’s thread control block
- Some state can only be modified via a controlled interface (system calls) (e.g. address space associated with the thread)
- Some state can be freely visible and modifiable by user-level applications without compromising the system
- Why not put this information in a user-level TCB (UTCB) for efficiency of access

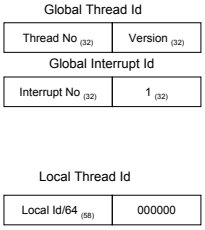
Thread Control Registers

- Stored in UTCB
- 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



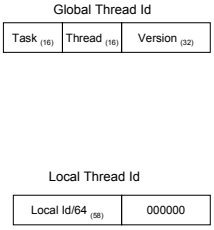
Thread Identifiers

- Global Identifiers
  - Identify a thread uniquely within the system
- 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.
- Can translate one to another



Thread Identifiers

- Global Identifiers
  - Thread number and version number assigned by root task accord to whatever policy you like
  - Example:
    - Version numbers are unique to give unique Ids
    - Threads number are grouped into tasks to allow upper bits to be task numbers
- Local Identifiers
  - Assigned by the system



ThreadControl

- Used to create, destroy, and modify threads
- Determines:
  - The global thread identifier associated with the thread
  - The address space the thread is associated with
  - The thread permitted to control scheduling parameters of the new thread
  - The pager
  - Location of the UTCB within the address spaces allotted UTCB area (See SpaceControl later)
- Threads can be created *active* or *inactive*.
  - *Inactive* is used to create and manipulate a new address space, or allocate a new thread to an existing address space.

ThreadControl

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


Steps in Creating a New "Task"

- Task = Address Space + Thread
- A "task" has
  - Thread
    - Identifier, IP, SP, pager, scheduler, utcb location
  - Address space
    - UTCB area, kernel info page area, redirector
  - Code, data, and stack mapped to address space

Steps in Creating a New "Task"

1. Create an inactive thread in a new address space
   - L4_ThreadControl (task, /* new tid */
     task, /* new space identifier */
     me, /* scheduler of new thread */
     L4_nilthread, /* pager = nil, inactive,
     (void *) -1); /* NOP Utcb location */

Steps in Creating a New "Task"

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

```
L4_SpaceControl (task,
        0, /* control (ignored on mips) */
        kip_area,
        utcb_area,
        L4_anythread, /* redirector */
        &control);
```


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.

This is a little cumbersome!

- We provide the following support function in the sample project code.
  - Read it and understand what it does!!!!

```
l4e_task_new(L4_ThreadId_t task,
    L4_ThreadId_t pager,
    void *entrypoint, void *stack)
```

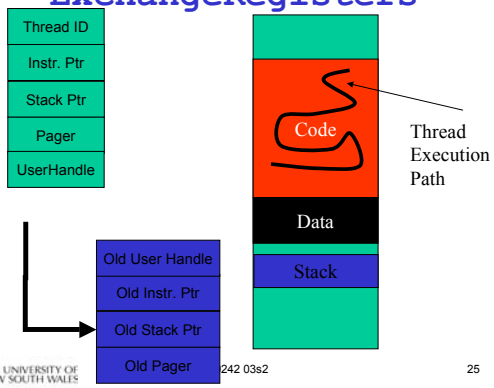

Adding extra inactive threads to a task

- Use ThreadControl to assign new inactive threads to an existing address space.
  - L4_ThreadControl (newtid, /* new thread id */
    ExistingId, /* address space identifier */
    me, /* scheduler of new thread */
    L4_nilthread, /* pager = nil, inactive,
    (void *) -1); /* NOP Utcb location */
- Note: Can also add active threads

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.

ExchangeRegisters

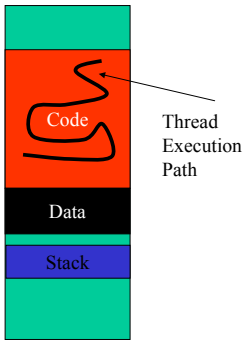


ExchangeRegisters

```
L4_ThreadId_t L4_ExchangeRegisters (L4_ThreadId_t dest,
                                    L4_Word_t control,
                                    L4_Word_t sp,
                                    L4_Word_t ip,
                        /* ignore */ L4_Word_t flags,
                                    L4_Word_t UserDefHandle,
                                    L4_ThreadId_t pager,
                                    L4_Word_t *old_control,
                                    L4_Word_t *old_sp,
                                    L4_Word_t *old_ip,
                        /* ignore */ L4_Word_t *old_flags,
                                    L4_Word_t *old_UserDefHandle,
                           L4_ThreadId_t *old_pager)
```

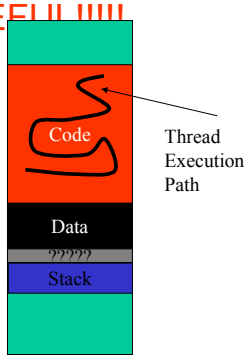

Threads

- 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 allocation, deallocation
  - Entry point



Be CAREFUL!!!!

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



Communication

Ignoring Address Spaces

IPC Overview

- Single system call that implements several variants of synchronous unbuffered IPC
  - Arguments determine IPC system call behaviour
  - Operations are
    - **`Send()`** send a message to a specified thread
    - **`Receive()`** "closed" receive from a specific sender (might be an interrupt)
    - **`Wait()`** "open" receive from any sender (incl. interrupt).
    - **`Call()`** send and wait for reply (usual 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

Thread No (32)	Version (32)
Interrupt No (32)	1 (32)
$0_{(64)}$	
$-1_{(64)}$	

- Local Identifiers
  - Special Ids
    - Any local thread

Local Id/64 (58)	000000
-1 (58)	000000

In <l4/types.h>

```
typedef union {
    L4_Word_t           raw;
    struct {
        L4_BITFIELD2( L4_Word_t,
                version : __14,
                thread_no : __18);
    } X;
} L4_GthreadId_t;

typedef union {
    L4_Word_t           raw;
    struct {
        L4_BITFIELD2(L4_Word_t,
                __zeros : 6,
                local_id : 26 __PLUS32);
    } X;
} L4_LthreadId_t;

typedef union {
    L4_Word_t           raw;
    L4_GthreadId_t      global;
    L4_LthreadId_t      local;
} L4_ThreadId_t;
```


In <l4/types.h>

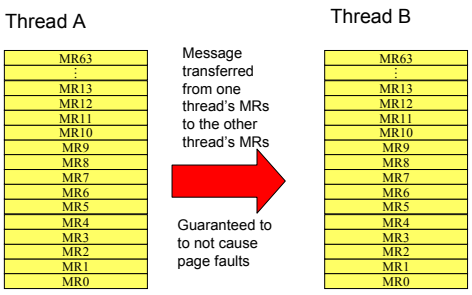
```
#define L4_nilthread            ((L4_ThreadId_t) { raw : 0UL})
#define L4_anythread            ((L4_ThreadId_t) { raw : ~0UL})
#define L4_anylocalthread       ((L4_ThreadId_t) { local : { X : {L4_SHUFFLE2(0, ~0UL)}}})
```


IPC Registers

- Message Registers
  - 64 “registers”
  - Form a message
  - Used to transfer typed items and untyped words
    - Typed items
      - MapItem
      - GrantItem
      - StringItem
- Buffer Registers
  - 34 “registers”
  - Specify where
    - MapItems and GrantItems are received
    - StringItems are received

  if any are permitted to be in the message

Message Register Only IPC



Overview of IPC operations

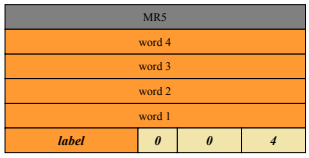
- L4_Ipc system call performs all IPC operations (both sending and receiving)
- Helper functions for frequent operations (see <l4/ipc.h>)
  - L4_Send
    - Send a message to a thread (blocking)
  - L4_Receive
    - Receive a message from a specified thread
  - L4_Wait
    - Receive a message from any 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

- Message content specified by MR_0
  - *u* the number of untyped words
  - *t* the number of words holding typed items
  - *label* free for the sender to use as part of the message (usually a "label" or "tag")
  - *flags* specifies option for the IPC operation
    - Not used for project (set = 0)
    - Specifies propagation

Example: Sending 4 untyped words

- Only 5 MRs tranferred
  - Note: On MIPS64, 9 MRs are transferred in CPU registers
    - Fast
    - The rest (if used) are copied from and to memory



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 MR_0

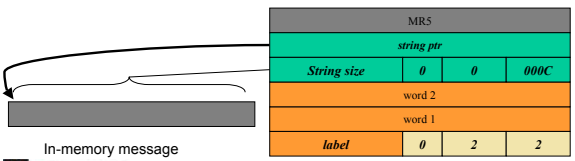
- MsgTag [MR_0]
  - *u* untyped words received (*u* = 0, send only IPC)
  - *t* typed words sent/received (*t* = 0, send only IPC)
  - Flags EXrp
    - E: error occurred (send or receive), see ErrorCode TCR for details
    - X: received cross processor IPC (ignore)
    - r: received redirected IPC (ignore)
    - p: received propagated IPC (ignore)

The StringItem Type

- Example sends a single simple string + two untyped words
  - More complex variations are possible (see the reference manual)
- Used to send a message in place
  - Avoid marshalling costs

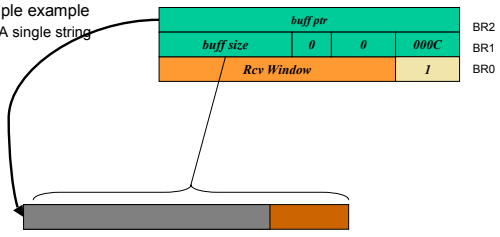
Note: The typed items always follow the untyped words

C: specifies if typed items follow



Receiving Strings

- Buffer Registers used to specify area and size of memory region to receive strings
- Simple example
  - A single string



Note!!!!

- Currently, StringItems are not supported on the MIPS-64 version of L4Ka::Pistachio
- We will discuss alternatives later
  - Example: Break long messages into many short messages

IPC Timeouts

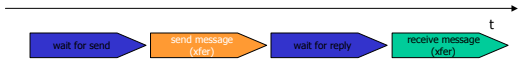
- Used to control the duration of IPC
- Two *timeout* types
  - Rcv/Snd Timeouts
    - Used to control how long the IPC syscall will block prior to
      - The send phase beginning (*SndTimeout*)
      - The receive phase beginning (*RcvTimeout*)
  - XferTimeouts (Snd/Rcv)
    - Used to limit how long the IPC transfer takes
      - Only used for StringItems (I.e. you can ignore them on MIPS-64)
      - Needed to limit time waiting for sender/receiver pagefaults on memory.
        - » More later in the course

Timeouts

- Timeouts
  - snd timeout, rcv timeout, xfer timeout

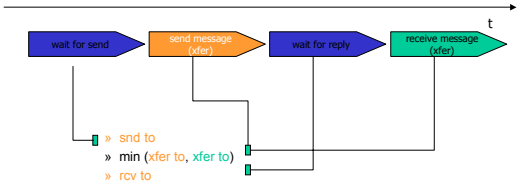
Timeouts

- Timeouts
  - snd timeout, rcv timeout, xfer timeout



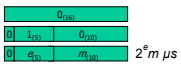
Timeouts

- Timeouts
  - snd timeout, rcv timeout, xfer timeout



Timeouts

- Timeouts
  - snd timeout, rcv timeout, xfer timeout
    - relative timeout values
      - 0
      - infinite
      - 1us … 610 h (log)



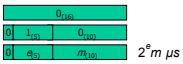
Timeout Value Range

e	m = 1	m = 1023
0	1.00E-06	1.02E-03
1	2.00E-06	2.05E-03
2	4.00E-06	4.09E-03
3	8.00E-06	8.18E-03
4	1.60E-05	1.64E-02
5	3.20E-05	3.27E-02
6	6.40E-05	6.55E-02
7	1.28E-04	1.31E-01
8	2.56E-04	2.62E-01
9	5.12E-04	5.24E-01
10	1.02E-03	1.05E+00
11	2.05E-03	2.10E+00
12	4.10E-03	4.19E+00
13	8.19E-03	8.38E+00
14	1.64E-02	1.68E+01
15	3.28E-02	3.35E+01

e	m = 1	m = 1023
16	6.55E-02	6.70E+01
17	1.31E-01	1.34E+02
18	2.62E-01	2.68E+02
19	5.24E-01	5.36E+02
20	1.05E+00	1.07E+03
21	2.10E+00	2.15E+03
22	4.19E+00	4.29E+03
23	8.39E+00	8.58E+03
24	1.68E+01	1.72E+04
25	3.36E+01	3.43E+04
26	6.71E+01	6.87E+04
27	1.34E+02	1.37E+05
28	2.68E+02	2.75E+05
29	5.37E+02	5.49E+05
30	1.07E+03	1.10E+06
31	2.15E+03	2.20E+06

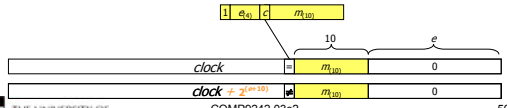
Timeouts

- Timeouts
  - snd timeout, rcv timeout, xfer timeout
    - relative timeout values
      - 0
      - infinite
      - 1us … 610 h (log)



$2^e m\ \mu s$

- absolute timeout values

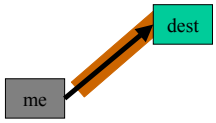


IPC

- to / from
- FromSpecifier
- Timeouts
- MR_0

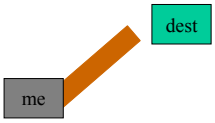
IPC

- Send
  - dest to / from
  - nilthread FromSpecifier
  - Timeouts
  - MR_0



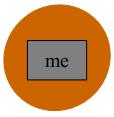
IPC

- Receive
  from *dest*
  - nilthread to / from
  - dest FromSpecifier
  - Timeouts
  - MR_0



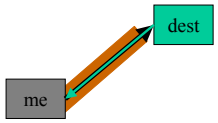
IPC

- Wait
  Receive from *anyone*
  - nilthread to / from
  - anythread FromSpecifier
  - Timeouts
  - MR_0



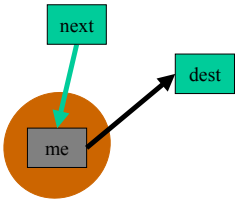
IPC

- Call
  - dest to / from
  - dest FromSpecifier
  - Timeouts
  - MR_0



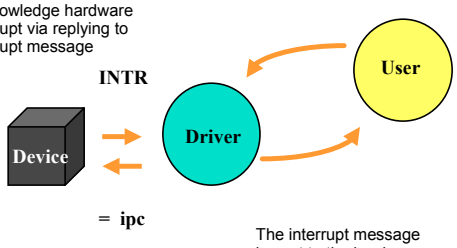
IPC

- ReplyWait
  - dest to / from
  - anythread FromSpecifier
  - Timeouts
  - MR_0



Interrupts

- Interrupts: messages from "hardware" threads
- Acknowledge hardware interrupt via replying to interrupt message



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

Interrupt Associated

- Association is done via the priviledged 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

Sample Code

```
int
register_cpu_interrupt_handler(host_handle_t host, int irq, void *fn, void *data)
{
    L4_ThreadId_t tid;
    int res;

    tid.global.X.thread_no = irq;
    tid.global.X.version = 1;

    res = L4_ThreadControl(tid, tid, L4_nilthread, L4_Pager(), (void*) -1);
                                                        ^--- The tid we want
                                                             to associate the irq
                                                             with (this is a hack
                                                             right now)
    if (res != 1) {
        l4e_printf("BADNESS ON THREAD CONTROL\n");
    }

    irq_handlers[irq].function = fn;
    irq_handlers[irq].data = data;


    return 1;
}
```


Microkernel System Calls

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

ThreadSwitch

- Yields the current thread's remaining timeslice, or donate remaining timeslice to another thread
  - You should not do this, but…
    - Can use to reduce impact of busy-wait
    - Ensure progress of resource (lock) holder
- `L4_ThreadSwitch (thread);`
  - Thread = nilthread: Yield the processor
  - Thread = threadID: Donate timeslice
  - See <l4/schedule.h> for derived functions

Schedule

- L4 implements a mostly multi-level round robin scheduler
- **`Schedule`** is used:
  - to change scheduling parameters of threads (which you should not need to do).
    - Timeslice
    - Priority
    - Total quantum (don't use, set to infinity)
  - for controlling preemption parameters: not implemented
  - set the processor the thread should run on: not needed, you have only one ☺

Schedule

- Only a thread's scheduler can invoke the schedule call
- The scheduler is set using thread control
  - Typically the root task will remain the scheduler
- Syscall:

```
L4_Schedule (L4_ThreadId_t dest,
                    L4_Word_t TimeControl,
                    L4_Word_t ProcessorControl,
                    L4_Word_t prio,
                    L4_Word_t PreemptionControl,
                    L4_Word_t * old_TimeControl)
```

- Derived functions in <l4/schedule.h>

SystemClock

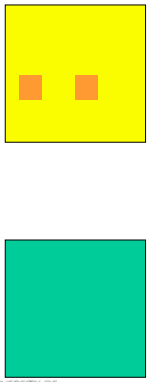
- Returns the current system clock
  - 64-bit number that counts µ-seconds
- Usually not a real system call

Microkernel System Calls

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

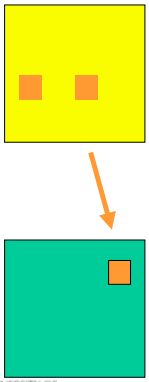
Address Spaces

Address Spaces



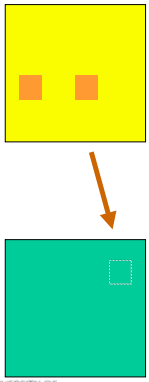
Address Spaces

- map



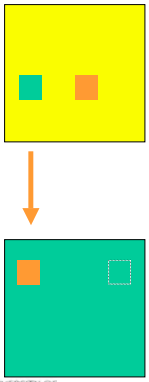
Address Spaces

- map
- unmap

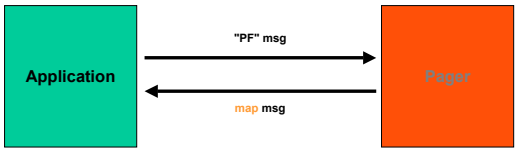


Address Spaces

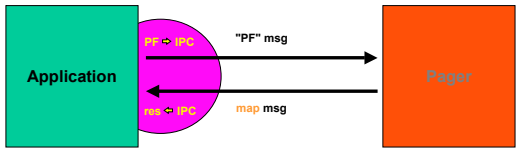
- map
- unmap
- grant



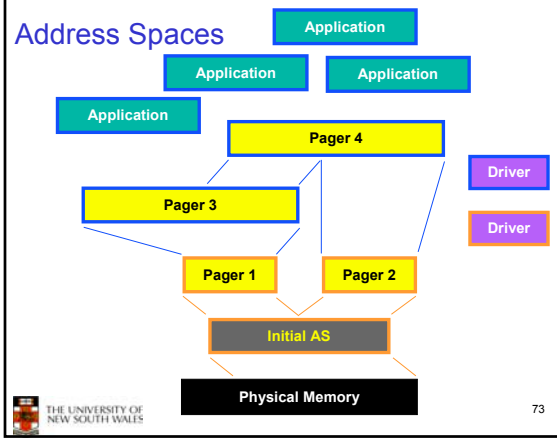
Page Fault Handling



Page Fault Handling

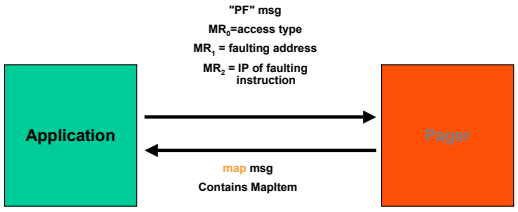


Address Spaces

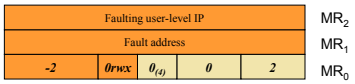


Address Space Management

Page Fault Protocol



PageFault Message



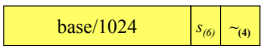
- Note: Applications can synthesis page fault messages
  - Not a problem as the application could do it anyway by directly accessing the memory it wishes to cause a fault on

Mapping Questions

- How is the mapping to be sent specified?
- How is the mapping to be received specified?
- How do they combine to produce the end result?
- What is the end result?

Fpage Data Type

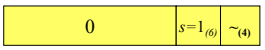
- Fpage
  - fpage size = 2^s



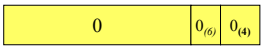
  - Specifies a region of memory that is
    - A power of 2 in size
    - Aligned to its size
  - Note: Smallest supported size is architecture specific
    - MIPS-64 supports 4K (s = 12)

Fpage Data Type

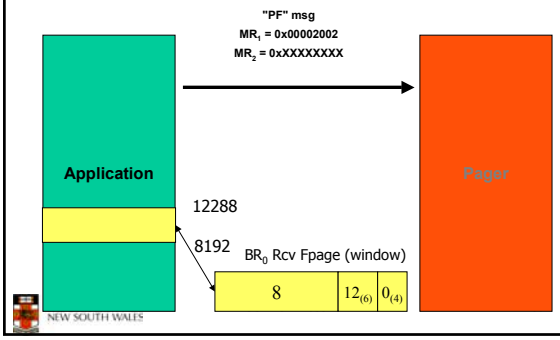
- Complete Address Space



- Nilpage

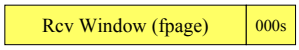


Receiving a mapping

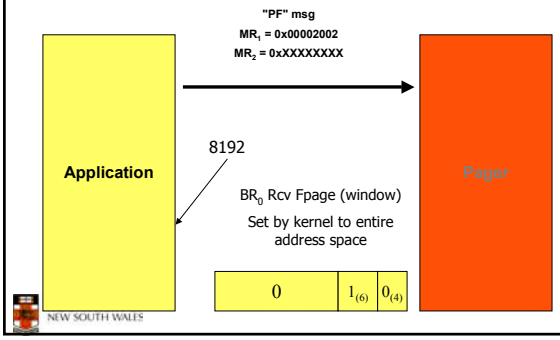


Buffer Register 0

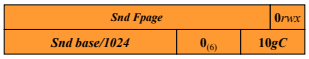
- Specifies
  - Willingness and locations to receive StringItems
    - s = 1
  - The receive window for mappings
    - Which location in the address space mappings are allowed



Normal Page Fault

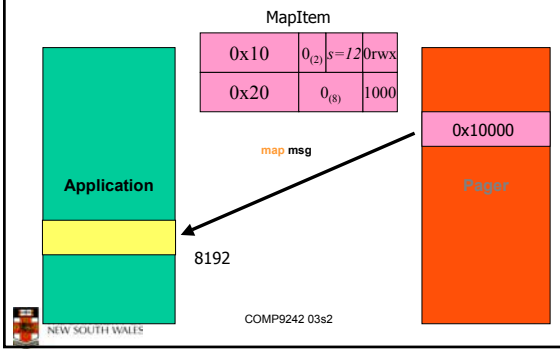


MapItem/GrantItem Data Type

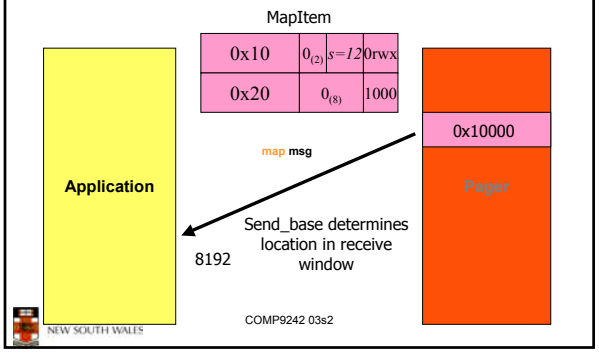


- Permissions
  - R: read
  - w: write (1) mapping
  - X: execute
  - Note: Not all architectures support all combinations
    - » MIPS-64: rx, rwx are supported by hardware
- g: mapping (0) or granting (1)

Send a mapping



Receive window > mapping size



Mappings and Window Sizes

- Se reference manual for precise definition of what happens for mismatched mappings and window sizes
- Advice:
  - Simply use 4K pages for all mappings

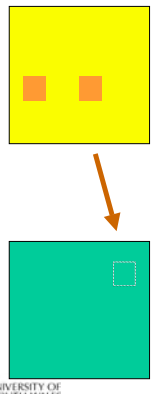
A Map Message

- For page faults, the kernel expects the following map message response
  - No untyped words
  - 1 MapItem

Snd Fpage			0_{rwx}
Snd base/1024		$0_{(6)}$	*10gC*
label	*0*	*2*	*0*

Explicit Mapping Receive

- BR_0 determines whether a receive/wait IPC can include a mapping
  - Set it prior to invoking IPC receive/wait



`Unmap`

- Used to revoke a mappings established in other address spaces that are derived from mappings in the current address space
- Unmap is also used to revoke access rights to existing mappings
  - Example: RW -> RO
- The mapping to revoke are specified by fpages in MRs

`Unmap` Arguments

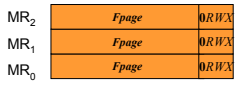
- Control
  - *f*: specifies whether fpage is *flushed* from the current address space in addition to revoking derived mapping
  - *k*: specifies the highest number MR that contain an Fpage to unmap
- Fpages
  - Fpages specify the regions in the local address space
  - *rwx:* the access rights to revoke

$0_{(57)}$	$f_{(1)}$	$k_{(6)}$

MR_2	*Fpage*	0_{rwx}
MR_1	*Fpage*	0_{rwx}
MR_0	*Fpage*	0_{rwx}

Unmap Results

- RWX
  - Reference (r), Dirty (w), and Executed (x) bits
    - Reset as a result of the unmap operation
    - Bit returned set if corresponding access has occurred on any derived mapping
- Note: Should not need to use
  - Behaviour is not heavily tested



SpaceControl

- Used to control the layout of newly created address spaces
  - Specifically
    - Location of Kernel Info Page
    - Location of UTCB region
- Redirector
  - All IPC from threads within the address space is redirected to a controlling thread
    - Used to enforce security policy
- Note: Should not need to change what is already done in the example code

Microkernel System Calls

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

ProcessorControl

- Privileged system call
- Sets processor frequency, voltage and other processor specific stuff.
- You can safely ignore it

MemoryControl

- Privileged system call
- Used to set cache architecture attributes on pages in memory
  - Machine specific
  - See Appendix E for details

Microkernel System Calls

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

That's all folks

Protocols

- Page Fault
- Thread Start
- Interrupt
- Preemption
  - Not supported, used to control preemption
- Exception
- Sigma0

Already covered

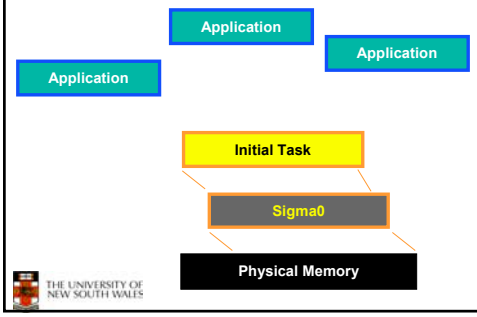
Exception Protocol

- Exception include: Divide by Zero, etc.
- Exceptions are indicated via *Exception IPC* to the thread's exception handler thread
  - The IPC contains
    - IP of where to resume the thread after handling the exception
    - Exception type
    - Other machine specific stuff
  - The exception handler can respond with an IPC specifying a new IP and other state to recover from the exception
- You should not need to do anything other than kill the task that caused the exception

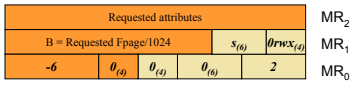
Sigma 0

- Contains all physical memory in the machine
  - Except that reserved for kernel use
  - Mapped idempotently
    - One-to-one
- Sigma0 distributes physical memory to start-up tasks at boot time
  - It maps each page once (and only once)
- Initial tasks request memory via an IPC protocol that allows mappings to be received
- Sigma0 responds (if possible) with an idempotent mapping giving access to a frame of physical memory

Sigma0



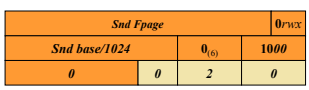
Sigma0 Request Message



- Requested attributes
  - Architecture specific
    - Use default = 0
- Requested Fpage
  - B != -1
    - Request a specific region of physical memory

Sigma0 Response

- If a successful request, Sigma0 responds with an idempotent mapping giving access to the physical memory request



Sigma0 Response

- If a unsuccessful request, Sigma0 responds with the following