

L4 PROGRAMMING

COMP9242
2006/S2 Week 2

RECAP: L4 ABSTRACTIONS AND MECHANISMS

Three basic abstractions:

- Address spaces
- Threads
- Time (second-class abstraction)

Two basic mechanisms:

- Inter-process communication (IPC)
- Mapping

L4 API:

- 10 system calls (N2, other APIs have slightly different numbers)
- 6–8 kernel-defined protocols

L4 SYSTEM CALLS

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

KERNEL INTERFACE PAGE (KIP)

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

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

SYSTEM CALLS

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

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 privileged *root task*

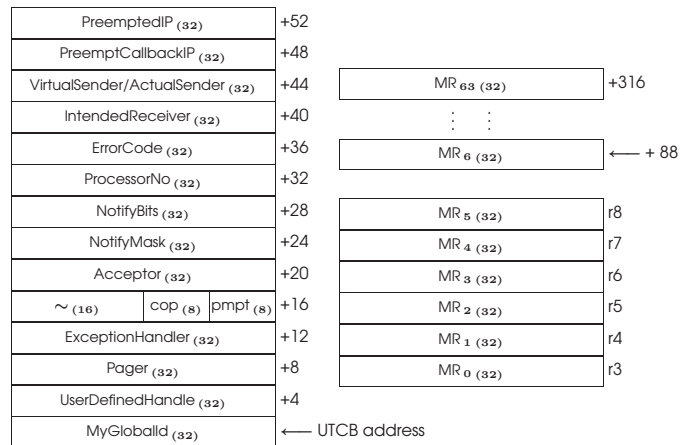
VIRTUAL REGISTERS

- 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

THREAD CONTROL BLOCK (TCB)

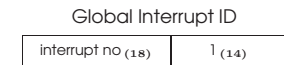
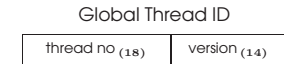
- Contains thread state
 - kernel-controlled state, must only be modified by syscalls
 - kept in kernel TCB (KTCB)
 - state that can be exposed to user w/o compromising security
 - kept in *user-level TCB* (UTCB)
 - includes virtual registers (as far as not bound to real registers)
 - *must only be modified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

USER-LEVEL TCB (ARM)



THREAD IDENTIFIERS

- Global thread IDs
 - uniquely identify a thread system-wide
 - defined by root task at thread creation
 - ... according to some policy
 - Note: version_[5..0] ≠ 0
- Note: V4 local thread IDs removed



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 handler ("pager")
 - location of thread's UTCB within the UTCB area of the thread's address space
 - ARM: UTCB address defined by kernel, not ThreadControl()

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)
```

→ ARM: utcb must be zero!

TASK

- L4 does not define a concept of a "task"
- We use it informally meaning:
 - 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

CREATING A TASK

1. Create inactive thread in a new address space (AS)
 - 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);     /* leave alone ;-) */
```

3. Define UTCB address of new thread

```
utcb_base = l4_nilpage;  
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 MR₁, RM₂
 - 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);
```

- 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
- 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
 - MapControl
 - SpaceControl
 - ProcessorControl
 - CacheControl

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()

CPSR bits affected by `flags` (ARM):

Bit	Name	Effect
31	N	negative
30	Z	zero
29	C	carry
28	V	overflow

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
 - MapControl
 - SpaceControl
 - ProcessorControl
 - CacheControl

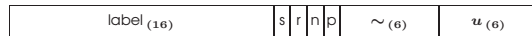
IPC OVERVIEW

- 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
 - *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 TAG MR₀



- 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
 - r**: blocking send
 - if unset, fail immediately if receiver not waiting
 - label**: user-defined (e.g., opcode)

EXAMPLE: SENDING 4 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);
```

Note: *u*, *s*, *r* set implicitly by L4_MsgAppendWord and convenience function

Delivers MR₀, ..., MR₄ to thread tid

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

IPC RESULT MR₀



- Returns to receiver details of message
 - u**: number of untyped 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

IPC: OBSOLETE FEATURES

- String items in message
 - used to send *out-of-line* data
 - arbitrarily sized and aligned buffers
 - non-essential feature that should not be in the kernel
- Map/grant items in message
 - used to send page mappings through IPC
 - replaced by MapControl() syscall
- Timeouts on IPC
 - limit blocking time
 - practically not very useful
 - replaced by send/receive block bits (*s*, *r* respectively)

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

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

SYSTEM CALLS

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

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
- 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

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()` 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

- 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)

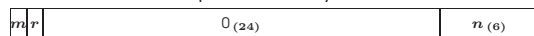
MAPCONTROL()

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

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

dest: denominates target address space

control: determines operation of syscall



r: read operation — returns (pre-syscall) mapping info
→ eg reference bits where hardware-maintained (x86)

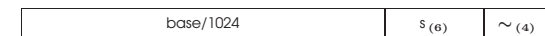
m: modify operation — changes mappings

n: number of *map items* used to describe mappings
→ map items are contained in message registers MR₀...MR_{2n-1}

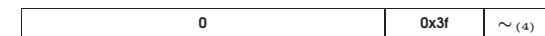
SPECIFYING MAPPINGS: FPGES

- 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

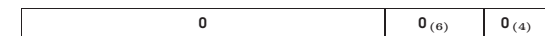
– fpage of size 2^s is specified as



– special fpages:



full AS



nil page

– On ARM, $s \geq 12$

MAP ITEM

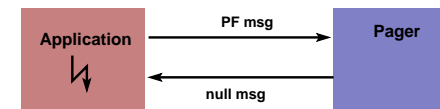
- Specifies a mapping to be created in destination AS

<i>fpage</i> (2a)	<i>Orwx</i>
<i>phys adr/1024</i> (2e)	<i>attr</i> (6)

- *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 (eg cached/uncached)
- *Orwx*: 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:
 - app triggers page fault
 - kernel exception handler generates IPC from faulter to pager
 - pager establishes mapping
 - calls `MapControl()` (if privileged) otherwise asks root task to do it
 - pager replies to page-fault IPC
 - kernel intercepts message, discards
 - kernel restarts faulting thread



PAGE FAULT MESSAGE

- Format of kernel-generated page fault message

Fault IP					MR ₂
Fault address					MR ₁
-2	Orwx	0 (4)	0 (6)	2	MR ₀

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

Fault IP					MR ₂
0x2002					MR ₁
-2	Orwx	0 (4)	0	2	MR ₀

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

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 faulter 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)
- 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

- ✓ KernellInterface
- ✓ 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

- ✓ KernellInterface
- ✓ ThreadControl
- ✓ ExchangeRegisters
- ✓ IPC
- ✓ ThreadSwitch
- ✓ Schedule
- ✓ MapControl
- ✓ SpaceControl
- ✓ ProcessorControl
- ✓ CacheControl

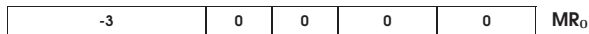
That's it!

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 prio
- 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*:



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

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