Sel4报告

sel4组成部分

- CSpace
- 进程间通信（ipc）
- 虚拟内存（vspace）
- 中断异常
- 进程控制块

CSpace

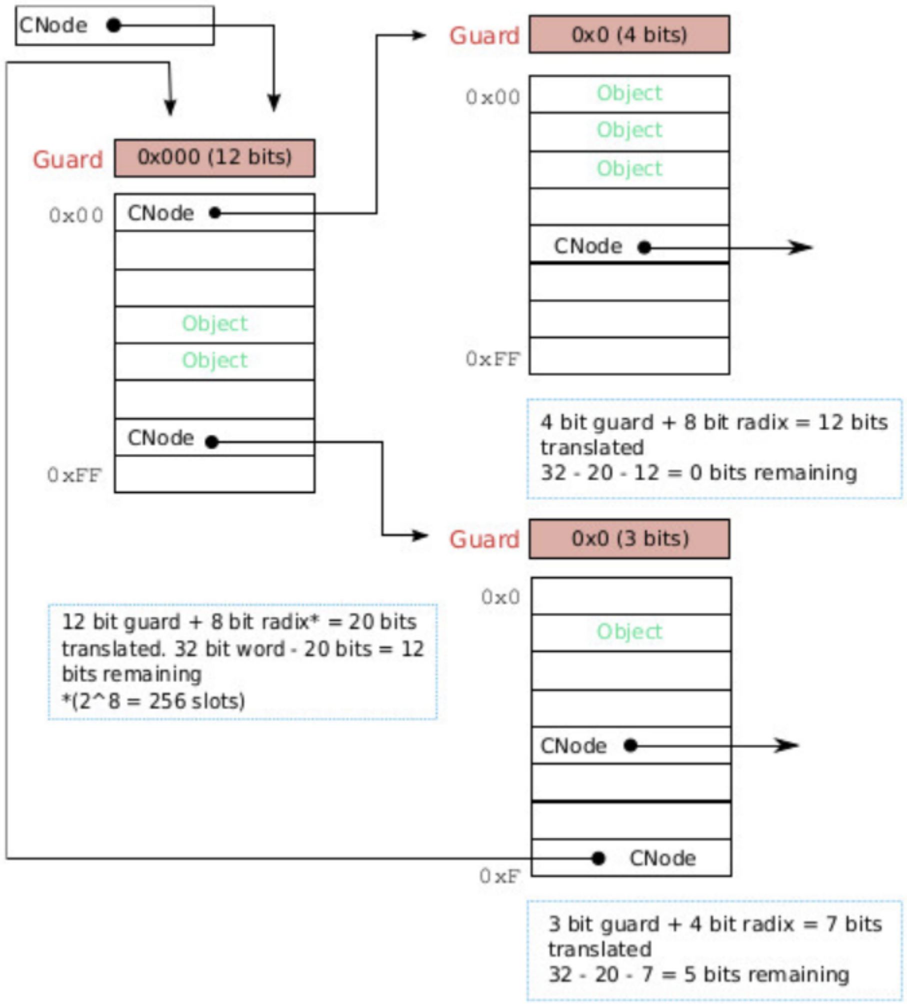
存放能力的空间，内核对象 内存空间均由能力管理

```
#sel4中所有能力均由两个字节组成，存储了具体能力的特定信息。
#sel4将所有的内存全部用untyped_cap管理
block untyped_cap {
    field capFreeIndex 39 //
    padding 18
    field capIsDevice 1
    field capBlockSize 6 //内存大小 2^capBlockSize + capFreeIndex
    field capType 5
    padding 20
    field_high capPtr 39 //指向内存起始地址的指针
},
#sel4具体的内核对象，用于进程间同步通信的端点能力
block endpoint_cap(capEPBadge, capCanGrantReply, capCanGrant, capCanSend,
                   capCanReceive, capEPPtr, capType) {
    field capEPBadge 64
    field capType 5
    field capCanGrantReply 1
    field capCanGrant 1
    field capCanReceive 1
    field capCanSend 1
    padding 16
    field_high capEPPtr 39 //指向内核对象 端点 的指针
}
```

Cspace由cte以链表的形式构成

```
struct cte {
    cap_t cap; //用于存放具体的能力
    mdb_node_t cteMDBNode;//构成一个双向的链表，将同一进程的cte连接起来
};

block mdb_node {
    padding 25
    field_high mdbNext 37
    field mdbRevocable 1
```

```
    field mdbFirstBadged 1
    field mdbPrev 64
}


exception_t invokeCNodeRevoke(cte_t *destSlot);
exception_t invokeCNodeDelete(cte_t *destSlot);
exception_t invokeCNodeInsert(cap_t cap, cte_t *srcSlot, cte_t *destSlot);
exception_t invokeCNodeMove(cap_t cap, cte_t *srcSlot, cte_t *destSlot);
exception_t invokeCNodeRotate(cap_t cap1, cap_t cap2, cte_t *slot1,
                              cte_t *slot2, cte_t *slot3);


//srcSlot<=>next
//srcSlot<=>destSlot<=>next
void cteInsert(cap_t newCap, cte_t *srcSlot, cte_t *destSlot)
{
    mdb_node_t srcMDB, newMDB;
    cap_t srcCap;
    bool_t newCapIsRevocable;

    srcMDB = srcSlot->cteMDBNode;
    srcCap = srcSlot->cap;

    newCapIsRevocable = isCapRevocable(newCap, srcCap);

    //srcSlot<-destSlot->nextSlot
    newMDB = mdb_node_set_mdbPrev(srcMDB, ((word_t)(srcSlot)));
    newMDB = mdb_node_set_mdbRevocable(newMDB, newCapIsRevocable);
    newMDB = mdb_node_set_mdbFirstBadged(newMDB, newCapIsRevocable);
                                                                    ;

    destSlot->cap = newCap;
    destSlot->cteMDBNode = newMDB;
    //srcSlot<=>destSlot->nextSlot
    mdb_node_ptr_set_mdbNext(&srcSlot->cteMDBNode, ((word_t)(destSlot)));
    if (mdb_node_get_mdbNext(newMDB)) {
        srcSlot<=>destSlot<=>next
        mdb_node_ptr_set_mdbPrev(
            &((cte_t *)(mdb_node_get_mdbNext(newMDB)))->cteMDBNode,
            ((word_t)(destSlot)));
    }
}
```




```
//cnode查询
block cnode_cap(capCNodeRadix, capCNodeGuardSize, capCNodeGuard,
                capCNodePtr, capType) {
    field capCNodeGuard 64
    field capType 5
    field capCNodeGuardSize 6
    field capCNodeRadix 6
    padding 9
    field_high capCNodePtr 38
}
resolveAddressBits_ret_t resolveAddressBits(cap_t nodeCap, cptr_t capptr, word_t
n_bits)
{
    ...
    while (1) {
            radixBits = cap_cnode_cap_get_capCNodeRadix(nodeCap);//槽号位数
            guardBits = cap_cnode_cap_get_capCNodeGuardSize(nodeCap);//保护位位数
            levelBits = radixBits + guardBits;//当前cnode占用的位数

            capGuard = cap_cnode_cap_get_capCNodeGuard(nodeCap);//当前cnode的保护位

            /* The MASK(wordRadix) here is to avoid the case where
             * n_bits = wordBits (=2^wordRadix) and guardBits = 0, as it
violates
```

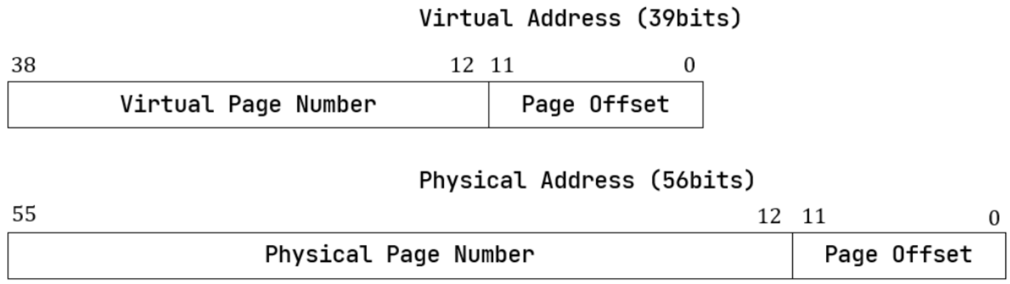
```
             * the C spec to shift right by more than wordBits-1.
             */
            guard = (capptr >> ((n_bits - guardBits) & MASK(wordRadix))) &
MASK(guardBits);//读取要查询的cap的保护位
            if (unlikely(guardBits > n_bits || guard != capGuard)) {//如果两个保护
位不相等就报错
                current_lookup_fault =
                    lookup_fault_guard_mismatch_new(capGuard, n_bits,
guardBits);
                ret.status = EXCEPTION_LOOKUP_FAULT;
                return ret;
            }

            if (unlikely(levelBits > n_bits)) {//如果当前cnode占用位数已超过要查询
slot的要求位数则报错
                current_lookup_fault =
                    lookup_fault_depth_mismatch_new(levelBits, n_bits);
                ret.status = EXCEPTION_LOOKUP_FAULT;
                return ret;
            }

            offset = (capptr >> (n_bits - levelBits)) & MASK(radixBits);
            slot = CTE_PTR(cap_cnode_cap_get_capCNodePtr(nodeCap)) + offset;

            if (likely(n_bits == levelBits)) {//如果当前cnode占用位数刚好与slot要求查
询的位数想等，则匹配
                ret.status = EXCEPTION_NONE;
                ret.slot = slot;
                ret.bitsRemaining = 0;
                return ret;
            }
            //否则，就是cnode占用位数小于slot查询位数的情况，这说明，当前找到的slot是一个
cnode，仍需向下一层查询

            n_bits -= levelBits;
            nodeCap = slot->cap;

            if (unlikely(cap_get_capType(nodeCap) != cap_cnode_cap)) {
                ret.status = EXCEPTION_NONE;
                ret.slot = slot;
                ret.bitsRemaining = n_bits;
                return ret;
            }
        }
}



lookupSlot_raw_ret_t lookupSlot(tcb_t *thread, cptr_t capptr)
{
    cap_t threadRoot;
    resolveAddressBits_ret_t res_ret;
    lookupSlot_raw_ret_t ret;
```

```
    threadRoot = (((cte_t *)((word_t)(thread)&~((1ul << (10)) - 1ul)))+
(tcbCTable))->cap;
    res_ret = resolveAddressBits(threadRoot, capptr, (1ul << (6)));

    ret.status = res_ret.status;
    ret.slot = res_ret.slot;
    return ret;
}
```

虚拟地址（VSpace)

sel4的页表与其他操作系统的页表并无太大差别，都采用了sv39的页表设计

地址格式与组成




```
 *                   +-----------------------------+ 2^64
 *                   |        Kernel Devices       |
 *                -> +-------------------KDEV_BASE-+ 2^64 - 1GiB
 *                |  |         Kernel ELF          |
 *            ----|  +-------------KERNEL_ELF_BASE-+ --+ 2^64 - 2GiB +
(KERNEL_ELF_PADDR_BASE % 1GiB)
 *            |   |  |                             |
 *            |   -> +-----------------------------+ --+ 2^64 - 2GiB =
(KERNEL_ELF_BASE % 1GiB)
 * Shared 1GiB|      |                             |   |
 * table entry|      |           PSpace            |   |
 *            |      |  (direct kernel mappings)   |   +----+
 *            ------>|                             |   |    |
 *                   |                             |   |    |
 *                   +-------------------PPTR_BASE-+ --+ 2^64 - 2^b
 *                   |                             |        |         +---------
----------------+
 *                   |                             |        |         |
                |
 *                   |                             |        |         |
                |
 *                   |          Invalid            |        |         |
                |
 *                   |                             |        |         |
  not           |
 *                   |                             |        |         |
kernel          |
 *                   |                             |        |         |
addressable       |
```

```
 *                   +--------------------USER_TOP-+  2^c   |         |
                |
 *                   |                             |        |         |
                |
 *                   |                             |        |         |
                |
 *                   |                             |        |      +- ----------
----------------+  PADDR_TOP =
 *                   |                             |        |      |  |
                |    PPTR_TOP - PPTR_BASE
 *                   |                             |        |      |  |
                |
 *                   |                             |        |      |  |
                |
 *                   |            User             |        |      |  |
                |
 *                   |                             |        |      |  |
                |
 *                   |                             |        +------+  +---------
----------------+  KDEV_BASE - KERNEL_ELF_BASE + PADDR_LOAD
 *                   |                             |     kernel    |  |
 Kernel ELF        |
 *                   |                             |   addressable |  +---------
----------------+  KERNEL_ELF_PADDR_BASE
 *                   |                             |               |  |
                |
 *                   |                             |               |  |
                |
 *                   +-----------------------------+  0            +- +---------
----------------+  0 PADDR_BASE
 *
 *                      virtual address space                          physical
address space
 *
```

```
block frame_cap \ page_table_cap {
    field capPTMappedASID 16
    field_high capPTBasePtr 39//页表的基地址
    padding 9
    field capType 5
    padding 19
    field capPTIsMapped 1 //是否被映射
    field_high capPTMappedAddress 39//自身映射的虚拟地址
}

BOOT_CODE void map_it_frame_cap(cap_t vspace_cap, cap_t frame_cap)
{
    pte_t *lvl1pt   = PTE_PTR(pptr_of_cap(vspace_cap));
    pte_t *frame_pptr   = PTE_PTR(pptr_of_cap(frame_cap));
    vptr_t frame_vptr = cap_frame_cap_get_capFMappedAddress(frame_cap);
```

```
    /* We deal with a frame as 4KiB */
    lookupPTSlot_ret_t lu_ret = lookupPTSlot(lvl1pt, frame_vptr);
    assert(lu_ret.ptBitsLeft == seL4_PageBits);

    pte_t *targetSlot = lu_ret.ptSlot;

    *targetSlot = pte_new(
                      (pptr_to_paddr(frame_pptr) >> seL4_PageBits),
                      0, /* sw */
                      1, /* dirty (leaf) */
                      1, /* accessed (leaf) */
                      0,  /* global */
                      1,  /* user (leaf) */
                      1,  /* execute */
                      1,  /* write */
                      1,  /* read */
                      1   /* valid */
                  );
    sfence();
}


lookupPTSlot_ret_t lookupPTSlot(pte_t *lvl1pt, vptr_t vptr)
{
    lookupPTSlot_ret_t ret;

    word_t level = CONFIG_PT_LEVELS - 1;
    pte_t *pt = lvl1pt; //当前页表指向1级页表的基址

    ret.ptBitsLeft = PT_INDEX_BITS * level + seL4_PageBits;
    ret.ptSlot = pt + ((vptr >> ret.ptBitsLeft) & MASK(PT_INDEX_BITS));//获得1级页表中对应的槽

    while (isPTEPageTable(ret.ptSlot) && likely(0 < level)) {
        level--;
        ret.ptBitsLeft -= PT_INDEX_BITS;
        pt = getPPtrFromHWPTE(ret.ptSlot);//获得当前对应槽的ppn并将其转换为虚拟地址（下一级页表的基地址）
        ret.ptSlot = pt + ((vptr >> ret.ptBitsLeft) & MASK(PT_INDEX_BITS));//获得对应的槽
    }

    return ret;
}

BOOT_CODE cap_t create_it_address_space(cap_t root_cnode_cap, v_region_t
it_v_reg)
{
    ...
    lvl1pt_cap =
        cap_page_table_cap_new(
            IT_ASID,               /* capPTMappedASID    */
            (word_t) rootserver.vspace,  /* capPTBasePtr       */
            1,                     /* capPTIsMapped      */
            (word_t) rootserver.vspace   /* capPTMappedAddress */
```

```
        );

    /* create all n level PT caps necessary to cover userland image in 4KiB pages
*/
    for (int i = 0; i < CONFIG_PT_LEVELS - 1; i++) {

        for (pt_vptr = ROUND_DOWN(it_v_reg.start, RISCV_GET_LVL_PGSIZE_BITS(i));
             pt_vptr < it_v_reg.end;
             pt_vptr += RISCV_GET_LVL_PGSIZE(i)) {
            if (!provide_cap(root_cnode_cap,
                             create_it_pt_cap(lvl1pt_cap, it_alloc_paging(),
pt_vptr, IT_ASID))
               ) {
                return cap_null_cap_new();
            }
        }

    }
    ...
    return lvl1pt_cap;
}
```

IPC

主要通过端点来进行ipc的通信

```
block endpoint {
    field epQueue_head 64
    padding 25
    field_high epQueue_tail 37
    field state 2
}

//tcb结构
struct tcb {
    ...
    word_t tcbIPCBuffer;
    struct tcb *tcbEPNext;
    struct tcb *tcbEPPrev;
    ...
};


block endpoint_cap(capEPBadge, capCanGrantReply, capCanGrant, capCanSend,
                   capCanReceive, capEPPtr, capType) {
    field capEPBadge 64
    field capType 5
    field capCanGrantReply 1
    field capCanGrant 1
    field capCanReceive 1
    field capCanSend 1
    padding 16
    field_high capEPPtr 39 //指向内核对象 端点 的指针
}
```

```
block endpoint_cap(capEPBadge, capCanGrantReply, capCanGrant, capCanSend,
                   capCanReceive, capEPPtr, capType) {
    field capEPBadge 64
    field capType 5
    field capCanGrantReply 1
    field capCanGrant 1
    field capCanReceive 1
    field capCanSend 1
    padding 16
    field_high capEPPtr 39 //指向内核对象 端点 的指针
}

//涉及的函数为sendIPC\receiveIPC ， 均为一个状态机。
void sendIPC(bool_t blocking, bool_t do_call, word_t badge,
             bool_t canGrant, bool_t canGrantReply, tcb_t *thread, endpoint_t
*epptr)

{
    switch (endpoint_ptr_get_state(epptr)) {
    case EPState_Idle:
    case EPState_Send:
        if (blocking) {
            tcb_queue_t queue;

            scheduleTCB(thread);//因为当前线程阻塞在发送端，所以要进行schedule

            queue = ep_ptr_get_queue(epptr);//endpoint对象内部包含一个队列，用于存储当
前等待发送的线程
            queue = tcbEPAppend(thread, queue);
            endpoint_ptr_set_state(epptr, EPState_Send);
            ep_ptr_set_queue(epptr, queue);
        }
        break;

    case EPState_Recv: {
        tcb_queue_t queue;
        tcb_t *dest;

        queue = ep_ptr_get_queue(epptr);
        dest = queue.head;

        queue = tcbEPDequeue(dest, queue);
        ep_ptr_set_queue(epptr, queue);

        if (!queue.head) {
            endpoint_ptr_set_state(epptr, EPState_Idle);
        }

        /* Do the transfer */
        doIPCTransfer(thread, epptr, badge, canGrant, dest);
        bool_t replyCanGrant = thread_state_ptr_get_blockingIPCCanGrant(&dest-
>tcbState);;
```

```c
        setThreadState(dest, ThreadState_Running);
        possibleSwitchTo(dest);

        break;
    }
    }
}


void receiveIPC(tcb_t *thread, cap_t cap, bool_t isBlocking)

{
    endpoint_t *epptr;
    epptr = ((endpoint_t *)(cap_endpoint_cap_get_capEPPtr(cap)));
    switch (endpoint_ptr_get_state(epptr)) {
        case EPState_Idle:
        case EPState_Recv: {
            tcb_queue_t queue;

            if (isBlocking) {
                scheduleTCB(thread);

                /* Place calling thread in endpoint queue */
                queue = ep_ptr_get_queue(epptr);
                queue = tcbEPAppend(thread, queue);
                endpoint_ptr_set_state(epptr, EPState_Recv);
                ep_ptr_set_queue(epptr, queue);
            } else {
                doNBRecvFailedTransfer(thread);
            }
            break;
        }

        case EPState_Send: {
            tcb_queue_t queue;
            tcb_t *sender;
            word_t badge;
            bool_t canGrant;
            bool_t canGrantReply;
            bool_t do_call;

            queue = ep_ptr_get_queue(epptr);
            sender = queue.head;

            queue = tcbEPDequeue(sender, queue);
            ep_ptr_set_queue(epptr, queue);

            if (!queue.head) {
                endpoint_ptr_set_state(epptr, EPState_Idle);
            }

            /* Get sender IPC details */
            badge = thread_state_ptr_get_blockingIPCBadge(&sender->tcbState);
            canGrant =
                thread_state_ptr_get_blockingIPCCanGrant(&sender->tcbState);
```

```
            canGrantReply =
                thread_state_ptr_get_blockingIPCCanGrantReply(&sender->tcbState);

            /* Do the transfer */
            doIPCTransfer(sender, epptr, badge,
                          canGrant, thread);

            setThreadState(sender, ThreadState_Running);
            possibleSwitchTo(sender);

            break;
        }
    }
}


void doNormalTransfer(tcb_t *sender, word_t *sendBuffer, endpoint_t *endpoint,
                      word_t badge, bool_t canGrant, tcb_t *receiver,
                      word_t *receiveBuffer)
{
    word_t msgTransferred;
    seL4_MessageInfo_t tag;
    exception_t status;

    tag = messageInfoFromWord(getRegister(sender, msgInfoRegister));

    if (canGrant) { //如果endpoint设置为传递能力，则需将sender的sendbuffer中的能力加入receiver的CSpace
        status = lookupExtraCaps(sender, sendBuffer, tag);
        if (__builtin_expect(!!(status != EXCEPTION_NONE), 0)) {
            current_extra_caps.excaprefs[0] = ((void *)0);
        }
    } else {
        current_extra_caps.excaprefs[0] = ((void *)0);
    }

    msgTransferred = copyMRs(sender, sendBuffer, receiver, receiveBuffer,
                             seL4_MessageInfo_get_length(tag));

    tag = transferCaps(tag, endpoint, receiver, receiveBuffer);

    tag = seL4_MessageInfo_set_length(tag, msgTransferred);
    setRegister(receiver, msgInfoRegister, wordFromMessageInfo(tag));
    setRegister(receiver, badgeRegister, badge);
}
```

Thread

```
//tcb结构
struct tcb {
    /* arch specific tcb state (including context)*/
    arch_tcb_t tcbArch; //保存了全部的寄存器，还多保存了SCAUSE,SSTATUS,SEPC,NextIP（下一次跳转地址），
```

```
    /* Thread state, 3 words */
    thread_state_t tcbState;

    /* Current fault, 2 words */
    seL4_Fault_t tcbFault;//记录当前的错误

    /*  maximum controlled priority, 1 byte (padded to 1 word) */
    prio_t tcbMCP;

    /* Priority, 1 byte (padded to 1 word) */
    prio_t tcbPriority;

    /* Timeslice remaining, 1 word */
    word_t tcbTimeSlice;

    /* Capability pointer to thread fault handler, 1 word */
    cptr_t tcbFaultHandler;

    /* userland virtual address of thread IPC buffer, 1 word */
    word_t tcbIPCBuffer;

    /* Previous and next pointers for scheduler queues , 2 words */
    struct tcb *tcbSchedNext;
    struct tcb *tcbSchedPrev;

    /* Preivous and next pointers for endpoint and notification queues, 2 words
*/
    struct tcb *tcbEPNext;
    struct tcb *tcbEPPrev;

};
```

管理不同优先级的tcb的数据结构为位图和队列构成的结构，不同的优先级放入位图的不同位置的队列中。

```
tcb_t* ksSchedulerAction;
tcb_t* SchedulerAction_ResumeCurrentThread=0;
tcb_t* SchedulerAction_ChooseNewThread=1;
void schedule(void)
{
    if (NODE_STATE(ksSchedulerAction) != SchedulerAction_ResumeCurrentThread)
{//如果scheduleraction要求重启当前线程，那就重启当前线程，调度直接完成。
        bool_t was_runnable;
        if (isSchedulable(NODE_STATE(ksCurThread))) {//如果当前线程仍然可执行，那就把
它加入到ready队列头部
            was_runnable = true;
            SCHED_ENQUEUE_CURRENT_TCB;
        } else {
            was_runnable = false;
        }
```

```
        if (NODE_STATE(ksSchedulerAction) == SchedulerAction_ChooseNewThread)
{//如果scheduler要求必须执行新线程，就选择新线程执行。选择的新进程是最高优先级队列中的第一个进程。
            scheduleChooseNewThread();
        } else {
            tcb_t *candidate = NODE_STATE(ksSchedulerAction);//否则，此时
scheduleraction代表的为竞争线程
            assert(isSchedulable(candidate));
            bool_t fastfail =
                NODE_STATE(ksCurThread) == NODE_STATE(ksIdleThread)
                || (candidate->tcbPriority < NODE_STATE(ksCurThread)-
>tcbPriority);
            if (fastfail &&
                !isHighestPrio(ksCurDomain, candidate->tcbPriority)) {//（竞争线程优先级小于当前线程||当前线程优先级与初始线程相同）&&竞争线程并非最高优先级。
                SCHED_ENQUEUE(candidate);
                /* we can't, need to reschedule */
                NODE_STATE(ksSchedulerAction) =
SchedulerAction_ChooseNewThread;//选择新线程而并非竞争线程执行。
                scheduleChooseNewThread();
            } else if (was_runnable && candidate->tcbPriority ==
NODE_STATE(ksCurThread)->tcbPriority) {
                /* We append the candidate at the end of the scheduling queue,
that way the
                 * current thread, that was enqueued at the start of the
scheduling queue
                 * will get picked during chooseNewThread */
                //如果两者优先级相同，仍选择当前线程执行。
                SCHED_APPEND(candidate);
                NODE_STATE(ksSchedulerAction) = SchedulerAction_ChooseNewThread;
                scheduleChooseNewThread();
            } else {//选择新线程执行
                assert(candidate != NODE_STATE(ksCurThread));
                switchToThread(candidate);
            }
        }
    }
    NODE_STATE(ksSchedulerAction) = SchedulerAction_ResumeCurrentThread;
}



void chooseThread(void)
{
    word_t prio;
    word_t dom;
    tcb_t *thread;

    if (numDomains > 1) {
        dom = ksCurDomain;
    } else {
        dom = 0;
    }

    if (likely(NODE_STATE(ksReadyQueuesL1Bitmap[dom]))) {
```

```
        prio = getHighestPrio(dom);
        thread = NODE_STATE(ksReadyQueues)[ready_queues_index(dom, prio)].head;
        switchToThread(thread);
    } else {
        switchToIdleThread();
    }
}
```

Sel4 中断处理

Sel4的中断处理并无特色，遵循riscv 基本理念

```
/*traps.S:*/
.global trap_entry
.extern c_handle_syscall
.extern c_handle_fastpath_reply_recv
.extern c_handle_fastpath_call
.extern c_handle_interrupt
.extern c_handle_exception

/*中断异常处理的入口，stvec设置的中断处理地址*/
trap_entry:
csrrw sp, sscratch, sp ;内核栈与用户栈的切换
csrrw t0, sscratch, t0
STORE ra, (0*REGBYTES)(t0)
STORE sp, (1*REGBYTES)(t0) ;在用户占保存返回地址与内核栈的指针
... ;此处省略保存通用寄存器的过程，所有通用寄存器均被保存


  csrr x1, sstatus
  STORE x1, (32*REGBYTES)(t0)

  csrr s0, scause
  STORE s0, (31*REGBYTES)(t0) ;读取scause sstatus寄存器

  la sp, (kernel_stack_alloc + BIT(CONFIG_KERNEL_STACK_BITS)) :加载内核栈地址

  csrr x1,  sepc
  STORE   x1, (33*REGBYTES)(t0);保存sepc

  ...;接下来根据s0的值判断是否为中断异常，如果是，就调入对应的c函数去处理
```

中断处理部分比较简单，其中`getActiveIrq`函数会获得当前被触发的中断，然后进入handleInterrupt函数具体处理，在此仅展示部分代码：

```
void handleInterrupt(irq_t irq)
{
    switch (intStateIRQTable[(irq)]) {
            ...
        case IRQTimer:
```

```
        timerTick();
        resetTimer();

        break;
    }
    ackInterrupt(irq);
}
```

当完成中断处理后，会进入用户态的函数restore_user_context，这一部分用c的内联汇编编写,基本内容与`trap.S`中内容类似，不做展示。

这一部分的内容的大致流程与中断类似，具体处理函数为c_handler_exception

```
void VISIBLE NORETURN c_handle_exception(void)
{
    NODE_LOCK_SYS;

    c_entry_hook();

    word_t scause = read_scause();
    switch (scause) {
    case RISCVInstructionAccessFault:
    case RISCVLoadAccessFault:
    case RISCVStoreAccessFault:
    case RISCVLoadPageFault:
    case RISCVStorePageFault:
    case RISCVInstructionPageFault:
        handleVMFaultEvent(scause);
        break;
    default:
        handleUserLevelFault(scause, 0);
        break;
    }

    restore_user_context();
    UNREACHABLE();
}
```