Pop quiz 7

CS5250 Advanced Operating Systems

Pop Quiz 7

(Due: 10 Mar 2022, 11.59pm)

Name: ______________Ding Fan____________________

Student Number: ______A0248373X_______________

Please do a code walkthrough of the Linux 5.16.1 kernel and explain how, starting with `context_switch()` in **kernel/sched/core.c:4921**, how context switching is achieved. In particular, staying on the 64 bit x86 architecture, trace the control flow and:

(1)Identify the macros and procedures encountered, and try to explain what they do;

some data structures definitions used in `context_switch()`:

macros involved in `context_switch()`:

procedures in `context_switch()`:

control flow of `context_switch`:

(2)Identify the stacks involved and where the stack switches occur;

(3)At key points where the control flow changes, what is on the top of the current stack? ("Key" is up to you to define but it should be used to clearly explain the flow that you have identified.)

(4)Show how control will return back to the current task being switched out eventually.

Also, answer the following questions:

Why are RBP, RBX, R12-R15 pushed and then popped in **__switch_to_asm** (found in **arch/x86/entry/entry_64.S:225**)?

What is the effect of the do-while loop in the **switch_to** macro?

Submit your answer in a PDF into the corresponding Luminus submission folder.

Please do a code walkthrough of the Linux 5.16.1 kernel and explain how, starting with `context_switch()` in kernel/sched/core.c:4921, how context switching is achieved. In particular, staying on the 64 bit x86 architecture, trace the control flow and:

/kernel/sched/core.c

```
/*
 * context_switch - switch to the new MM and the new thread's register state.
 */
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
         struct task_struct *next, struct rq_flags *rf)
{
  prepare_task_switch(rq, prev, next);

  /*
   * For paravirt, this is coupled with an exit in switch_to to
   * combine the page table reload and the switch backend into
   * one hypercall.
   */
  arch_start_context_switch(prev);

  /*
   * kernel -> kernel   lazy + transfer active
   *   user -> kernel   lazy + mmgrab() active
   *
   * kernel ->   user   switch + mmdrop() active
   *   user ->   user   switch
   */
  if (!next->mm) {                                // to kernel
    enter_lazy_tlb(prev->active_mm, next);

    next->active_mm = prev->active_mm;
    if (prev->mm)                           // from user
      mmgrab(prev->active_mm);
    else
      prev->active_mm = NULL;
  } else {                                        // to user
    membarrier_switch_mm(rq, prev->active_mm, next->mm);
    /*
     * sys_membarrier() requires an smp_mb() between setting
```

```
	 * rq->curr / membarrier_switch_mm() and returning to userspace.
	 *
	 * The below provides this either through switch_mm(), or in
	 * case 'prev->active_mm == next->mm' through
	 * finish_task_switch()'s mmdrop().
	 */
	switch_mm_irqs_off(prev->active_mm, next->mm, next);

	if (!prev->mm) {                        // from kernel
	  /* will mmdrop() in finish_task_switch(). */
	  rq->prev_mm = prev->active_mm;
	  prev->active_mm = NULL;
	}
  }

  rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);

  prepare_lock_switch(rq, next, rf);

  /* Here we just switch the register state and the stack. */
  switch_to(prev, next, prev);
  barrier();

  return finish_task_switch(prev);
}
```

(1)Identify the macros and procedures encountered, and try to explain what they do;

some data structures definitions used in `context_switch()`:

1. **`rq` is the main per-CPU run queue data structure**

   defined in /kernel/sched/sched.h

2. **`task_struct` it is traditional called PCB(process control blocked), which hold all the information about a process/task**

   defined in /include/linux/sched.h

3. **`rq_flags` it is the flag of the per-CPU run queue `rq`**

   defined in /kernel/sched/sched.h

macros involved in `context_switch()`:

1. the following is the macro definition of the `switch_to` function (64 bit x86)

   /arch/x86/include/asm/switch_to.h

```
#define switch_to(prev, next, last)                \
do {                                               \
  ((last) = __switch_to_asm((prev), (next)));      \
} while (0)
```

- `switch_to` is used to switch the CPU state during the `context_switch()`(switch old task to new task)
- it include saving the old task CPU states(register, stack); restoring new task CPU state(register, stack)

2. the following is the macro definition of the `arch_start_context_switch` function (64 bit x86)

   [/include/linux/pgtable.h]()

```
/*
 * A facility to provide batching of the reload of page tables and
 * other process state with the actual context switch code for
 * paravirtualized guests.  By convention, only one of the batched
 * update (lazy) modes (CPU, MMU) should be active at any given time,
 * entry should never be nested, and entry and exits should always be
 * paired.  This is for sanity of maintaining and reasoning about the
 * kernel code.  In this case, the exit (end of the context switch) is
 * in architecture-specific code, and so doesn't need a generic
 * definition.
 */
#ifndef __HAVE_ARCH_START_CONTEXT_SWITCH
#define arch_start_context_switch(prev) do {} while (0)
#endif
```

- here is `do {} while (0)`, because the exit (end of the context switch) is in architecture-specific code, and so doesn't need a generic definition.
- This is for sanity of maintaining and reasoning about the kernel code.

3. the following is the macro definition of the `barrier` function

   [/include/linux/compiler.h]()

```
/* Optimization barrier */
#ifndef barrier
/* The "volatile" is due to gcc bugs */
# define barrier() __asm__ __volatile__("": : :"memory")
#endif
```

procedures in `context_switch()`:

1. `prepare_task_switch(rq, prev, next);`

   - what do?

     - *prepare to switch tasks*
     - *is called with the rq lock held and interrupts off. It must be paired with a subsequent `finish_task_switch` after the context switch.*
     - *`prepare_task_switch` sets up locking and calls architecture specific hooks.*

   - definition:/kernel/sched/core.c

```
/**
 * prepare_task_switch - prepare to switch tasks
 * @rq: the runqueue preparing to switch
 * @prev: the current task that is being switched out
 * @next: the task we are going to switch to.
 *
 * This is called with the rq lock held and interrupts off. It must
 * be paired with a subsequent finish_task_switch after the context
 * switch.
 *
 * prepare_task_switch sets up locking and calls architecture specific
 * hooks.
 */
static inline void
prepare_task_switch(struct rq *rq, struct task_struct *prev,
        struct task_struct *next)
{
  kcov_prepare_switch(prev);
  sched_info_switch(rq, prev, next);
  perf_event_task_sched_out(prev, next);
  rseq_preempt(prev);
  fire_sched_out_preempt_notifiers(prev, next);
  kmap_local_sched_out();
  prepare_task(next);
  prepare_arch_switch(next);
}
```

2. `arch_start_context_switch(prev)`

   - what do?

     - *This is for sanity of maintaining and reasoning about the kernel code.*
     - *because the exit (end of the context switch) is in architecture-specific code, and so doesn't need a generic definition.*
     - *telling the specific architecture that the `prev` task is starting to switch*

- definitions: /include/linux/pgtable.h

```
/*
 * A facility to provide batching of the reload of page tables and
 * other process state with the actual context switch code for
 * paravirtualized guests.  By convention, only one of the batched
 * update (lazy) modes (CPU, MMU) should be active at any given time,
 * entry should never be nested, and entry and exits should always be
 * paired.  This is for sanity of maintaining and reasoning about the
 * kernel code.  In this case, the exit (end of the context switch) is
 * in architecture-specific code, and so doesn't need a generic
 * definition.
 */
#ifndef __HAVE_ARCH_START_CONTEXT_SWITCH
#define arch_start_context_switch(prev) do {} while (0)
#endif
```

  - here is `do {} while (0)`, because the exit (end of the context switch) is in architecture-specific code, and so doesn't need a generic definition.
  - This is for sanity of maintaining and reasoning about the kernel code.

3. `enter_lazy_tlb(prev->active_mm, next)`

- what do?
  - *it is only called when the CPU will switch to a new kernel task (or other context) without an mm*
  - *it notify the underlying architecture that there is no need to switch the user virtual address space. it is used to accelerate context switching.*
  - *this technology is called lazy TLB. it aims to minimize TLB flushes*
- definition: /arch/x86/mm/tlb.c

```
/*
 * Please ignore the name of this function.  It should be called
 * switch_to_kernel_thread().
 *
 * enter_lazy_tlb() is a hint from the scheduler that we are entering a
 * kernel thread or other context without an mm.  Acceptable implementations
 * include doing nothing whatsoever, switching to init_mm, or various clever
 * lazy tricks to try to minimize TLB flushes.
 *
 * The scheduler reserves the right to call enter_lazy_tlb() several times
 * in a row.  It will notify us that we're going back to a real mm by
 * calling switch_mm_irqs_off().
 */
void enter_lazy_tlb(struct mm_struct *mm, struct task_struct *tsk)
{
```

```
  if (this_cpu_read(cpu_tlbstate.loaded_mm) == &init_mm)
    return;

  this_cpu_write(cpu_tlbstate_shared.is_lazy, true);
}
```

4. `mmgrab(prev->active_mm)`

   - what do?
     - *Make sure that @mm(which is the active_mm of the previous task) will not get freed even after the owning task exits*
     - *ensure if needed, the next task can still use that @mm*
   - definition:/include/linux/sched/mm.h

```
/**
 * mmgrab() - Pin a &struct mm_struct.
 * @mm: The &struct mm_struct to pin.
 *
 * Make sure that @mm will not get freed even after the owning task
 * exits. This doesn't guarantee that the associated address space
 * will still exist later on and mmget_not_zero() has to be used before
 * accessing it.
 *
 * This is a preferred way to pin @mm for a longer/unbounded amount
 * of time.
 *
 * Use mmdrop() to release the reference acquired by mmgrab().
 *
 * See also <Documentation/vm/active_mm.rst> for an in-depth explanation
 * of &mm_struct.mm_count vs &mm_struct.mm_users.
 */
static inline void mmgrab(struct mm_struct *mm)
{
  atomic_inc(&mm->mm_count);
}
```

5. `membarrier_switch_mm(rq, prev->active_mm, next->mm)`

   - what do?
     - *The scheduler provides memory barriers required by membarrier between:*
       - prior user-space memory accesses and store to `rq->membarrier_state`,
       - store to `rq->membarrier_state` and following user-space memory accesses.
   - definition:/kernel/sched/sched.h

```
#ifdef CONFIG_MEMBARRIER
/*
 * The scheduler provides memory barriers required by membarrier between:
 * - prior user-space memory accesses and store to rq->membarrier_state,
 * - store to rq->membarrier_state and following user-space memory accesses.
 * In the same way it provides those guarantees around store to rq->curr.
 */
static inline void membarrier_switch_mm(struct rq *rq,
          struct mm_struct *prev_mm,
          struct mm_struct *next_mm)
{
  int membarrier_state;

  if (prev_mm == next_mm)
    return;

  membarrier_state = atomic_read(&next_mm->membarrier_state);
  if (READ_ONCE(rq->membarrier_state) == membarrier_state)
    return;

  WRITE_ONCE(rq->membarrier_state, membarrier_state);
}
#else
static inline void membarrier_switch_mm(struct rq *rq,
          struct mm_struct *prev_mm,
          struct mm_struct *next_mm)
{
}
#endif
```

6. `switch_mm_irqs_off(prev->active_mm, next->mm, next)`

- what do?
    - *Switch the process memory address space. For each process, there is a process memory address space, which is a virtual memory address space isolated by processes. Therefore, switching is also required here, including page tables*
- definition:/arch/x86/mm/tlb.c

```
void switch_mm_irqs_off(struct mm_struct *prev, struct mm_struct *next,
      struct task_struct *tsk)
{
  struct mm_struct *real_prev = this_cpu_read(cpu_tlbstate.loaded_mm);
  u16 prev_asid = this_cpu_read(cpu_tlbstate.loaded_mm_asid);
  bool was_lazy = this_cpu_read(cpu_tlbstate_shared.is_lazy);
  unsigned cpu = smp_processor_id();
  u64 next_tlb_gen;
  bool need_flush;
  u16 new_asid;
```

```
  /*
   * NB: The scheduler will call us with prev == next when switching
   * from lazy TLB mode to normal mode if active_mm isn't changing.
   * When this happens, we don't assume that CR3 (and hence
   * cpu_tlbstate.loaded_mm) matches next.
   *
   * NB: leave_mm() calls us with prev == NULL and tsk == NULL.
   */

  /* We don't want flush_tlb_func() to run concurrently with us. */
  if (IS_ENABLED(CONFIG_PROVE_LOCKING))
    WARN_ON_ONCE(!irqs_disabled());

  /*
   * Verify that CR3 is what we think it is.  This will catch
   * hypothetical buggy code that directly switches to swapper_pg_dir
   * without going through leave_mm() / switch_mm_irqs_off() or that
   * does something like write_cr3(read_cr3_pa()).
   *
   * Only do this check if CONFIG_DEBUG_VM=y because __read_cr3()
   * isn't free.
   */
```

7. `prepare_lock_switch(rq, next, rf)`

- what do?
    - *Since the runqueue lock will be released by the next task (which is an invalid locking op but in the case of the scheduler it's an obvious special-case),*
    - *so we do an early lockdep release here:*
- definition:[/kernel](#)/[sched](#)/[core.c](#)

```
static inline void
prepare_lock_switch(struct rq *rq, struct task_struct *next, struct rq_flags
 *rf)
{
  /*
   * Since the runqueue lock will be released by the next
   * task (which is an invalid locking op but in the case
   * of the scheduler it's an obvious special-case), so we
   * do an early lockdep release here:
   */
  rq_unpin_lock(rq, rf);
  spin_release(&__rq_lockp(rq)->dep_map, _THIS_IP_);
#ifdef CONFIG_DEBUG_SPINLOCK
  /* this is a valid case when another task releases the spinlock */
  rq_lockp(rq)->owner = next;
#endif
}
```

8. `switch_to(prev, next, prev)`

- what do?
    - *switch to the new register state and the stack.*
    - `switch_to` is used to switch the CPU state during the `context_switch()` (switch old task to new task)
    - it includes saving the old task CPU states(register, stack); restoring new task CPU state(register, stack)
- definition:/arch/x86/include/asm/switch_to.h

  the following is the macro definition of the `switch_to` function (64 bit x86)

```
#define switch_to(prev, next, last)              \
do {                                             \
  ((last) = __switch_to_asm((prev), (next)));    \
} while (0)
```

9. `barrier()`

- what do?
    - *In order that the execution order of the instructions after the program is compiled will not change due to the optimization of the compiler, the kernel provides a barrier to ensure the execution order of the program*
    - *during `context_switch` it is used to ensure the execution order that `switch_to` execute before `finish_task_switch`*
- definition:

10. `finish_task_switch(prev)`

- What do?
    - *clean up after a task-switch*
    - *we may have delayed dropping an mm in context_switch(). If so, we finish that here outside of the runqueue lock.*
- definition:/kernel/sched/core.c

```
/**
 * finish_task_switch - clean up after a task-switch
 * @prev: the thread we just switched away from.
 *
```

```
 * finish_task_switch must be called after the context switch, paired
 * with a prepare_task_switch call before the context switch.
 * finish_task_switch will reconcile locking set up by prepare_task_switch,
 * and do any other architecture-specific cleanup actions.
 *
 * Note that we may have delayed dropping an mm in context_switch(). If
 * so, we finish that here outside of the runqueue lock. (Doing it
 * with the lock held can cause deadlocks; see schedule() for
 * details.)
 *
 * The context switch have flipped the stack from under us and restored the
 * local variables which were saved when this task called schedule() in the
 * past. prev == current is still correct but we need to recalculate this_rq
 * because prev may have moved to another CPU.
 */
static struct rq *finish_task_switch(struct task_struct *prev)
	__releases(rq->lock)
{
	struct rq *rq = this_rq();
	struct mm_struct *mm = rq->prev_mm;
	long prev_state;

	/*
	 * The previous task will have left us with a preempt_count of 2
	 * because it left us after:
	 *
	 *  schedule()
	 *    preempt_disable();			// 1
	 *    __schedule()
	 *      raw_spin_lock_irq(&rq->lock)	// 2
	 *
	 * Also, see FORK_PREEMPT_COUNT.
	 */
	if (WARN_ONCE(preempt_count() != 2*PREEMPT_DISABLE_OFFSET,
		      "corrupted preempt_count: %s/%d/0x%x\n",
		      current->comm, current->pid, preempt_count()))
		preempt_count_set(FORK_PREEMPT_COUNT);

	rq->prev_mm = NULL;

	/*
	 * A task struct has one reference for the use as "current".
	 * If a task dies, then it sets TASK_DEAD in tsk->state and calls
	 * schedule one last time. The schedule call will never return, and
	 * the scheduled task must drop that reference.
	 *
	 * We must observe prev->state before clearing prev->on_cpu (in
	 * finish_task), otherwise a concurrent wakeup can get prev
	 * running on another CPU and we could rave with its RUNNING -> DEAD
	 * transition, resulting in a double drop.
	 */
	prev_state = READ_ONCE(prev->__state);
	vtime_task_switch(prev);
	perf_event_task_sched_in(prev, current);
	finish_task(prev);
	tick_nohz_task_switch();
	finish_lock_switch(rq);
	finish_arch_post_lock_switch();
```

```
  kcov_finish_switch(current);
  /*
   * kmap_local_sched_out() is invoked with rq::lock held and
   * interrupts disabled. There is no requirement for that, but the
   * sched out code does not have an interrupt enabled section.
   * Restoring the maps on sched in does not require interrupts being
   * disabled either.
   */
  kmap_local_sched_in();

  fire_sched_in_preempt_notifiers(current);
  /*
   * When switching through a kernel thread, the loop in
   * membarrier_{private,global}_expedited() may have observed that
   * kernel thread and not issued an IPI. It is therefore possible to
   * schedule between user->kernel->user threads without passing though
   * switch_mm(). Membarrier requires a barrier after storing to
   * rq->curr, before returning to userspace, so provide them here:
   *
   * - a full memory barrier for {PRIVATE,GLOBAL}_EXPEDITED, implicitly
   *   provided by mmdrop(),
   * - a sync_core for SYNC_CORE.
   */
  if (mm) {
    membarrier_mm_sync_core_before_usermode(mm);
    mmdrop_sched(mm);
  }
  if (unlikely(prev_state == TASK_DEAD)) {
    if (prev->sched_class->task_dead)
      prev->sched_class->task_dead(prev);

    /* Task is done with its stack. */
    put_task_stack(prev);

    put_task_struct_rcu_user(prev);
  }

  return rq;
}
```

control flow of `context_switch`:

1. Using the `prepare_task_switch` function to prepare for the task switch; *sets up locking and calls architecture specific hooks.*
2. Using `arch_start_context_switch` function tell the architecture that previous task is going to be switched.
3. four scenarios:
    a. if `kernel -> kernel`

```
enter_lazy_tlb(prev->active_mm, next);
next->active_mm = prev->active_mm;
prev->active_mm = NULL;
```

- `enter_lazy_tlb` function *notify the underlying architecture that there is no need to switch the user virtual address space. it is used to accelerate context switching. this technology is called lazy TLB. it aims to minimize TLB flushes*
- Set the `next -> active_mm` equal to `prev->active_mm` ,The kernel is pointing to the user state address space
- set the `prev->active_mm` equal to `null`

b. if *`user -> kernel`*

```
enter_lazy_tlb(prev->active_mm, next);
next->active_mm = prev->active_mm;
mmgrab(prev->active_mm);
```

- `enter_lazy_tlb` function *notify the underlying architecture that there is no need to switch the user virtual address space. it is used to accelerate context switching. this technology is called lazy TLB. it aims to minimize TLB flushes*
- Set the `next -> active_mm` equal to `prev->active_mm` ,The kernel is pointing to the user state address space
- `mmgrab` function make sure `prev->active_mm` will not get freed even after the owning task exits

c. if *`kernel ->   user`*

```
membarrier_switch_mm(rq, prev->active_mm, next->mm);
switch_mm_irqs_off(prev->active_mm, next->mm, next);
rq->prev_mm = prev->active_mm;
prev->active_mm = NULL;
```

- using `membarrier_switch_mm` function set the memory barriers required by membarrier between:
  - prior user-space memory accesses and store to `rq->membarrier_state`,

  - store to `rq->membarrier_state` and following user-space memory accesses.

  - using `switch_mm_irqs_off` function Switch the process memory address space.
  - set the `pre_mm` of the cpu runqueue `rq` to the value `prev->active_mm`
  - set the `prev->active_mm` equal to `null`

d. if *`user ->    user`*

```
membarrier_switch_mm(rq, prev->active_mm, next->mm);
switch_mm_irqs_off(prev->active_mm, next->mm, next);
```

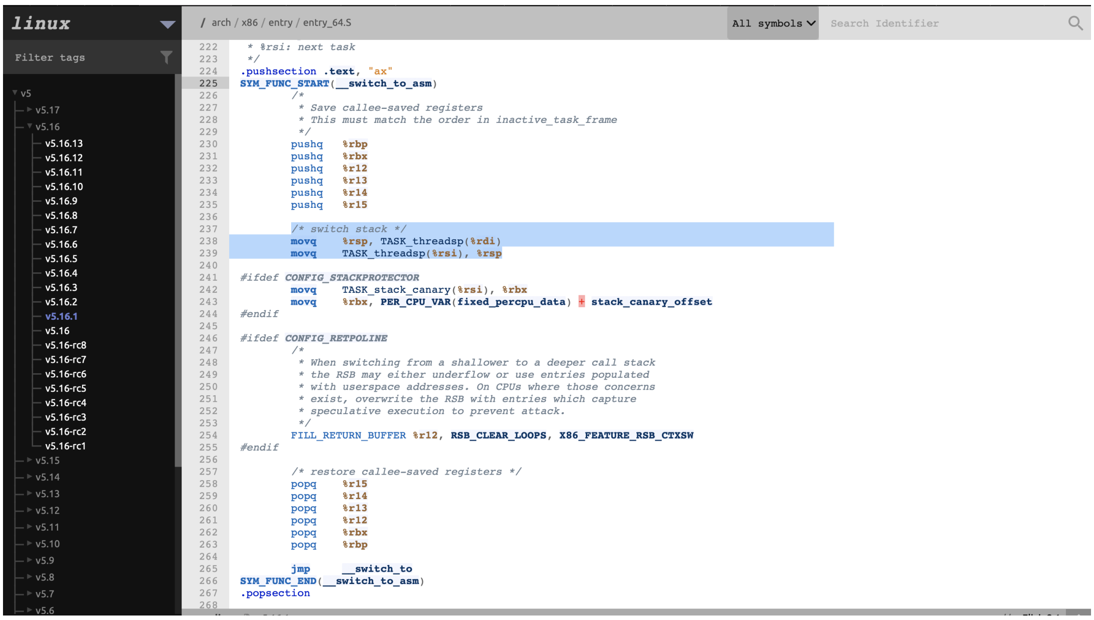
  - using `membarrier_switch_mm` function set the memory barriers required by membarrier between:
    - prior user-space memory accesses and store to `rq->membarrier_state`,
    - store to `rq->membarrier_state` and following user-space memory accesses.
  - using `switch_mm_irqs_off` function Switch the process memory address space.

4. using `prepare_lock_switch(rq, next, rf)`
   - do an early lockdep release
   - Since the runqueue lock will be released by the next task (which is an invalid locking op but in the case of the scheduler it's an obvious special-case)
5. using `switch_to` function
   - `switch_to` is used to switch the CPU state during the `context_switch()`(switch old task to new task)
   - it include saving the old task CPU states(register, stack); restoring new task CPU state(register, stack)
6. using `barrier` function to ensure the execution order that `switch_to` execute before `finish_task_switch`
7. using `finish_task_switch` function
   - clean up after a task-switch

- we may have delayed dropping an mm in context_switch(). If so, we finish that here outside of the runqueue lock.

(2)Identify the stacks involved and where the stack switches occur;

- stacks involved：
  - the stack of the `pre task`
  - the stack of the `next task`
  - the stack type of the `pre task` and `next task`, can be `user model stack` or `kernel mode stack`
- where the stack switches occur？
  - /arch/x86/entry/entry_64.S: 238,239



(3)At key points where the control flow changes, what is on the top of the current stack? (“Key” is up to you to define but it should be used to clearly explain the flow that you have identified.)

suppose the CPU currently run task `A` and will switch to task `B`

1. before the `switch_to` process switches from pre task `A` to next task `B`

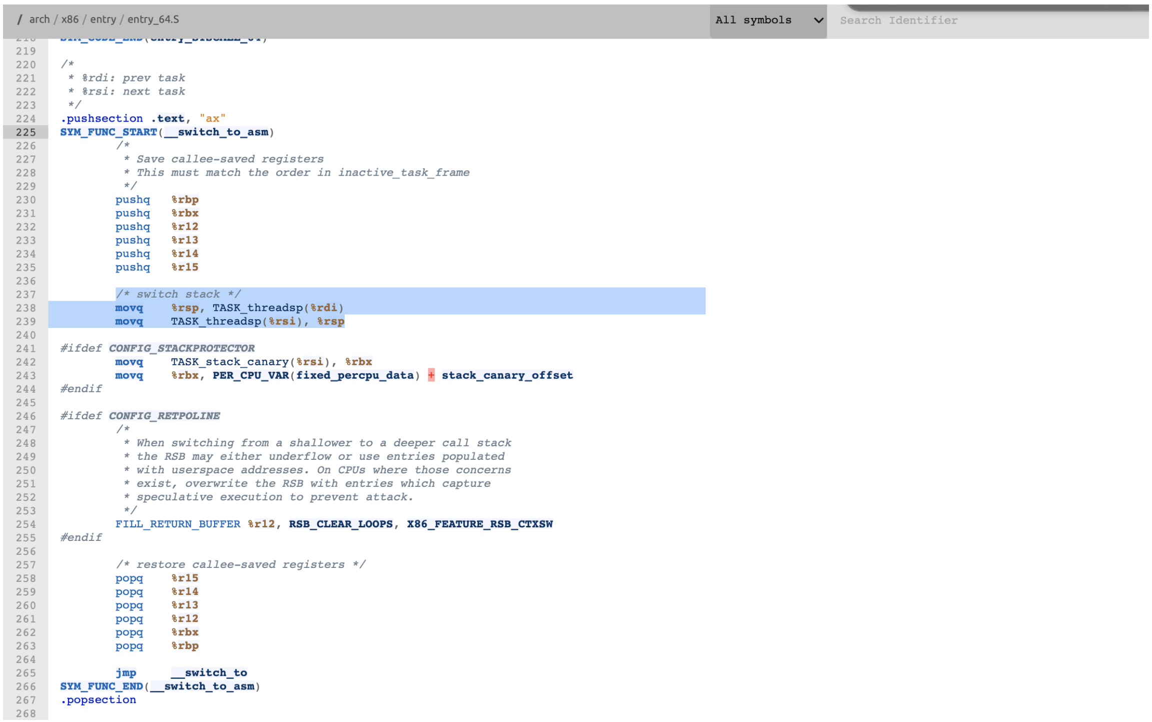
The current stack is the stack of the prev task `A`, and the top of the stack stores the value of the next instruction of task `A`

2. when the code have executed the following highlight part

```
.pushsection .text, "ax"
SYM_FUNC_START(__switch_to_asm)
	/*
	 * Save callee-saved registers
	 * This must match the order in inactive_task_frame
	 */
	pushq	%rbp
	pushq	%rbx
	pushq	%r12
	pushq	%r13
	pushq	%r14
	pushq	%r15

	/* switch stack */
	movq	%rsp, TASK_threadsp(%rdi)
	movq	TASK_threadsp(%rsi), %rsp

#ifdef CONFIG_STACKPROTECTOR
	movq	TASK_stack_canary(%rsi), %rbx
	movq	%rbx, PER_CPU_VAR(fixed_percpu_data) + stack_canary_offset
#endif

#ifdef CONFIG_RETPOLINE
	/*
	 * When switching from a shallower to a deeper call stack
	 * the RSB may either underflow or use entries populated
	 * with userspace addresses. On CPUs where those concerns
	 * exist, overwrite the RSB with entries which capture
	 * speculative execution to prevent attack.
	 */
	FILL_RETURN_BUFFER %r12, RSB_CLEAR_LOOPS, X86_FEATURE_RSB_CTXSW
#endif

	/* restore callee-saved registers */
	popq	%r15
	popq	%r14
	popq	%r13
	popq	%r12
	popq	%rbx
	popq	%rbp

	jmp	__switch_to
SYM_FUNC_END(__switch_to_asm)
.popsection
```

- The current stack is the stack of the prev task `A`, but the top of the stack stores the value of `RBP, RBX, R12-R15` of task `A`

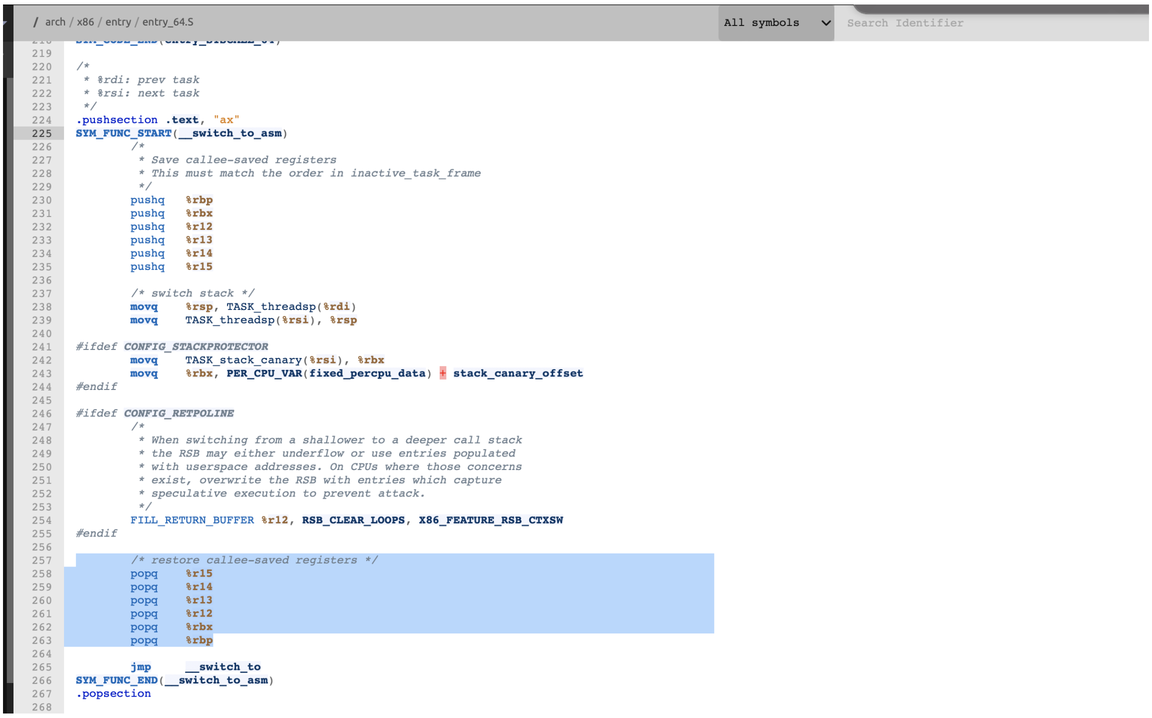
3. when the code have executed the following highlight part




```
/*
 * %rdi: prev task
 * %rsi: next task
 */
.pushsection .text, "ax"
SYM_FUNC_START(__switch_to_asm)
	/*
	 * Save callee-saved registers
	 * This must match the order in inactive_task_frame
	 */
	pushq	%rbp
	pushq	%rbx
	pushq	%r12
	pushq	%r13
	pushq	%r14
	pushq	%r15

	/* switch stack */
	movq	%rsp, TASK_threadsp(%rdi)
	movq	TASK_threadsp(%rsi), %rsp

#ifdef CONFIG_STACKPROTECTOR
	movq	TASK_stack_canary(%rsi), %rbx
	movq	%rbx, PER_CPU_VAR(fixed_percpu_data) + stack_canary_offset
#endif

#ifdef CONFIG_RETPOLINE
	/*
	 * When switching from a shallower to a deeper call stack
	 * the RSB may either underflow or use entries populated
	 * with userspace addresses. On CPUs where those concerns
	 * exist, overwrite the RSB with entries which capture
	 * speculative execution to prevent attack.
	 */
	FILL_RETURN_BUFFER %r12, RSB_CLEAR_LOOPS, X86_FEATURE_RSB_CTXSW
#endif

	/* restore callee-saved registers */
	popq	%r15
	popq	%r14
	popq	%r13
	popq	%r12
	popq	%rbx
	popq	%rbp

	jmp	__switch_to
SYM_FUNC_END(__switch_to_asm)
.popsection
```

- The current stack is the stack of the next task `B`, but the top of the stack stores the value of `RBP, RBX, R12-R15` of task `B`

4. when the code have executed the following highlight part




```
/*
 * %rdi: prev task
 * %rsi: next task
 */
.pushsection .text, "ax"
SYM_FUNC_START(__switch_to_asm)
	/*
	 * Save callee-saved registers
	 * This must match the order in inactive_task_frame
	 */
	pushq	%rbp
	pushq	%rbx
	pushq	%r12
	pushq	%r13
	pushq	%r14
	pushq	%r15

	/* switch stack */
	movq	%rsp, TASK_threadsp(%rdi)
	movq	TASK_threadsp(%rsi), %rsp

#ifdef CONFIG_STACKPROTECTOR
	movq	TASK_stack_canary(%rsi), %rbx
	movq	%rbx, PER_CPU_VAR(fixed_percpu_data) + stack_canary_offset
#endif

#ifdef CONFIG_RETPOLINE
	/*
	 * When switching from a shallower to a deeper call stack
	 * the RSB may either underflow or use entries populated
	 * with userspace addresses. On CPUs where those concerns
	 * exist, overwrite the RSB with entries which capture
	 * speculative execution to prevent attack.
	 */
	FILL_RETURN_BUFFER %r12, RSB_CLEAR_LOOPS, X86_FEATURE_RSB_CTXSW
#endif

	/* restore callee-saved registers */
	popq	%r15
	popq	%r14
	popq	%r13
	popq	%r12
	popq	%rbx
	popq	%rbp

	jmp	__switch_to
SYM_FUNC_END(__switch_to_asm)
.popsection
```

the current stack is the task `B`, and the top of the stack stores the address of the pointer to the next instruction of task `B`.

(4)Show how control will return back to the current task being switched out eventually.

1. If currently the cpu is executing task `A` and want to switch to task `B`, we need to schedule() with `prev` pointing to `A` and next pointing to `B`. After `context_switch` complete, task `A` will be suspended from the CPU and the CPU will execute task `B`.
2. after some time, the CPU is currently execute task `C`
3. If we want to return back to task `A`, we need to suspend task `C` and switch to task `A`. At this time, we need to point prev to `C` and next to `A`. After `context_switch` complete, task `C` will be suspended and the CPU will restore the state of task `A` and execute task `A`.

Also, answer the following questions:

Why are RBP, RBX, R12-R15 pushed and then popped in __switch_to_asm (found in arch/x86/entry/entry_64.S:225)?

- We need to push the `RBP, RBX, R12-R15,` because we need to save the current task registers. Therefore if we want to switch to this task in the future, we can restore these registers to restore the original state of the CPU.

```
.pushsection .text, "ax"
SYM_FUNC_START(__switch_to_asm)
	/*
	 * Save callee-saved registers
	 * This must match the order in inactive_task_frame
	 */
	pushq	%rbp
	pushq	%rbx
	pushq	%r12
	pushq	%r13
	pushq	%r14
	pushq	%r15
```

- We need to pop `RBP, RBX, R12-R15`

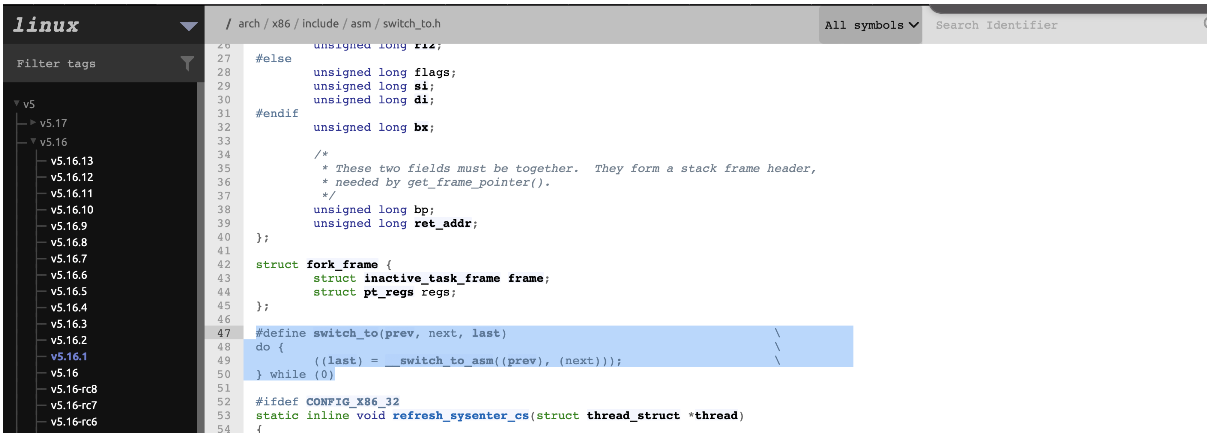
```
	/* restore callee-saved registers */
	popq	%r15
	popq	%r14
	popq	%r13
	popq	%r12
	popq	%rbx
	popq	%rbp
```

- because here, the code has completed the stack switch

```
/* switch stack */
movq    %rsp, TASK_threadsp(%rdi)
movq    TASK_threadsp(%rsi), %rsp
```

- `%rsp` has been pointed to the stack of `next task`
- we need to pop `RBP, RBX, R12-R15`, to restore the register state of the `next task`

What is the effect of the do-while loop in the switch_to macro?



- Using the `do{...} while(0)` macro definition ensures that `(( last ) = __switch_to_asm (( prev ), (next)));` will always be called and run the way we expect. it will not be affected by braces, semicolons, etc.
- In the compilation process, `(( last ) = __switch_to_asm (( prev ), (next)));` is wrapped into an independent syntax unit, so that it will not be confused with the context

Submit your answer in a PDF into the corresponding Luminus submission folder.