Troca de Contexto Entre Processos

Grupo M:
Anthony Bernardo Kamers
Gabriel da Silva Cardoso
Julio Gonçalves Ramos

Grupo I:
Marcelo José Dias
Nelson Luiz Joppi Filho
Igor May Wensing

Fundamentos

O que é o Contexto?

- Estado computacional de um processo ou thread.
- Pode incluir: registradores de uso geral, contador de programa (PC), ponteiro para a pilha (SP), etc.
- A possibilidade de alternar entre processos ou threads dá a impressão de execução simultânea, mesmo em CPU single core.
- O contexto deve ser carregado e restaurado de acordo para garantir uma execução coerente.

Diferenças entre contexto de Thread e Processo

Processo

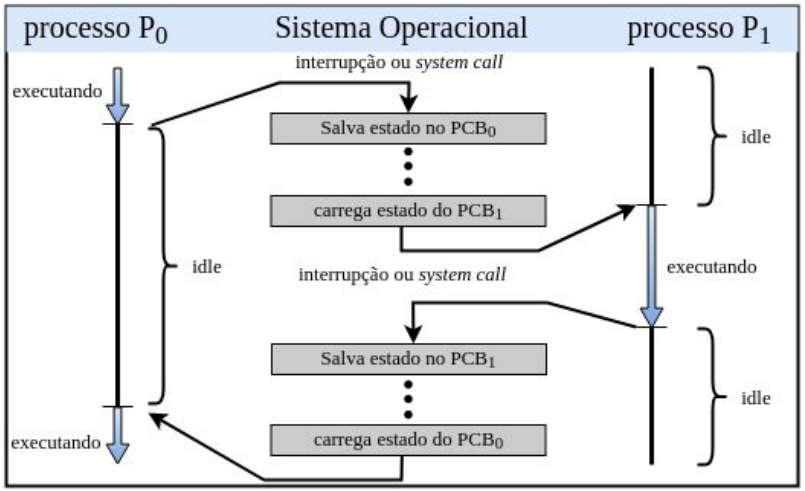
- Espaço de endereçamento é trocado.
- TLB e caches são limpos (“flushed”).
- Maior custo computacional.

Thread

- Espaço de endereçamento é mantido.
- TLB e caches não sofrem alteração.
- Menor custo computacional.

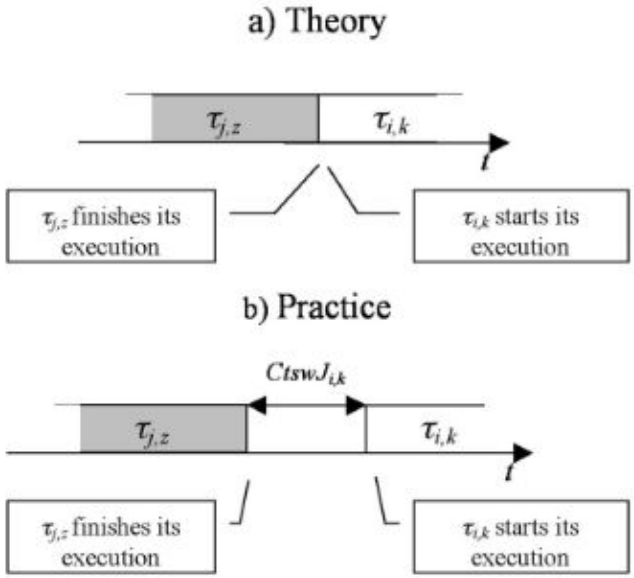
Process Context Switch

- O contexto deve ser salvo em memória para que não seja perdido quando o conteúdo dos registradores seja alterado.
- As principais causas de um context switch são: escalonamento de threads/processos, interrupções, mode change, etc.
- O contexto é salvo em uma estrutura de dados adequada:
  - Contexto da thread: Salvo nas informações do processo.
  - Contexto do processo: Salvo em uma PCB.



Custos de context switching

- A troca não é imediata e todo tempo gasto salvando e restaurando contextos é tempo perdido de processamento, logo é overhead.
- Os custos diretos incluem os tempos de acesso à memória para salvar e carregar os contextos.
  - Esses tempos são proporcionais ao tamanho do contexto.
- Existem também custos indiretos, como as perdas causadas pela limpeza da cache do processo atual.
- Além disso, também deve ocorrer “flushing” do pipeline para evitar que dados inválidos sejam processados.

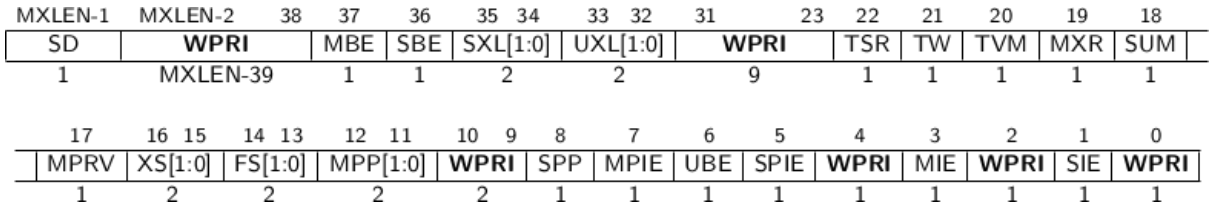



RISC-V

Registradores importantes

Machine Status (mstatus): O registrador mstatus é um CSR importante para a troca de contexto. Alguns de seus campos são:

- xIE: Ativa interrupções no modo de privilégio x.
- xPP: Guarda modo do sistema antes de entrar no modo x.
- xPIE: Guarda valor anterior do registrador xIE.
- MXR/SUM: Modifica o nível de privilégio em que loads acessam a memória virtual.
- xBE: Determina endianness dos acessos à memória em modo x (não afeta instruction fetches, que são sempre Little Endian).



Registradores importantes

Supervisor Address Translation and Protection (SATP): Registrador que controla a tradução e proteção de endereços em modo supervisor (S-Mode). Ele possui os campos:

- MODE: Determina quantos bits serão utilizados para o endereçamento.
- Address Space ID: Identifica o espaço de endereçamento atual.
- PPN: Guarda o physical page number (endereço supervisor da página dividido por 4KiB) atual.

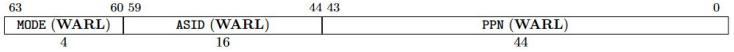


Figure 4.12: RV64 Supervisor address translation and protection register `satp`, for MODE values Sv39 and Sv48.

RV32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing (see Section **??**).

RV64		
Value	Name	Description
0	Bare	No translation or protection.
1–7	—	*Reserved for standard use*
8	Sv39	Page-based 39-bit virtual addressing (see Section **??**).
9	Sv48	Page-based 48-bit virtual addressing (see Section **??**).
10	*Sv57*	*Reserved for page-based 57-bit virtual addressing.*
11	*Sv64*	*Reserved for page-based 64-bit virtual addressing.*
12–13	—	*Reserved for standard use*
14–15	—	*Designated for custom use*

Encoding of satp MODE field.

SATP e Espaço de Endereçamento

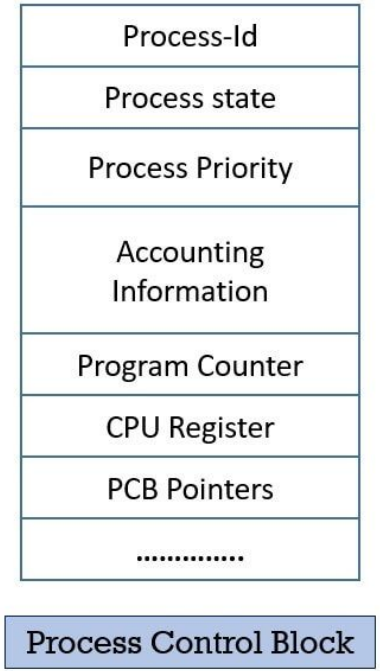
- Durante uma mudança de contexto, o sistema operacional deve gerenciar os espaços de endereçamento e mapeamentos de memória para cada processo.
- O sistema operacional **salva o valor atual do registrador SATP** para o processo que está sendo interrompido. Em seguida, o sistema operacional **carrega o registrador SATP** com o valor salvo correspondente ao processo que está sendo retomado.
- O campo ASID no registro SATP é essencial para o gerenciamento eficiente do Translation Lookaside Buffer (TLB) durante mudanças de contexto.
    - Como cada processo possui seu próprio espaço de endereço e mapeamentos de endereço virtual para físico, o TLB deve ser atualizado para refletir os mapeamentos corretos para o novo processo.
    - No entanto, em vez de limpar todo o TLB, o ASID permite que o processador mantenha entradas separadas do TLB para diferentes processos, reduzindo o overhead das mudanças de contexto.

Process Context Switch

Process Control Block

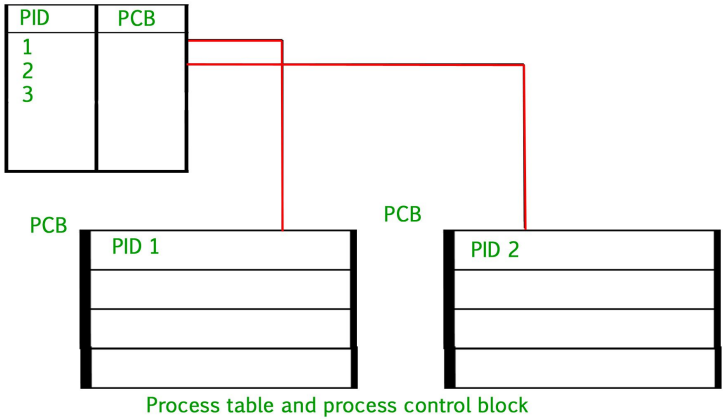
- O Process Control Block (PCB) contém informações importantes sobre cada processo. Isso pode incluir: PID, estado, PC, registradores, espaço de endereçamento, etc.

- Quando a troca de contexto está relacionada à troca de dois processos diferentes, os PCB são usados para guardar e restaurar o estado de cada processo.



Process Table

- A Process Table é uma estrutura de dados que contém informações sobre todos os processos.
- Ela possui um ponteiro para o PCB de cada processo, sendo utilizada na hora da troca de contexto para determinar onde salvar e carregar o contexto de cada processo.



Process table and process control block

Page Tables e Translation Lookaside Buffer (TLB)

- Cada processo pode possuir uma ou mais tabelas de páginas.
- Registrador guarda ponteiro para tabela única ou base do endereço onde as múltiplas tabelas se encontram.
- Esse registrador é salvo e carregado na hora da troca de contexto.

- A TLB faz parte da MMU e pode ser entendida como um cache para a MMU, traduzindo endereços de memória virtuais para físicos. ➔
- Deve ser limpa após uma troca de contexto, já que contém traduções de endereços em cache inválidas.

Exemplo de Troca de Contexto

E quão complicado é chegar à esse ponto

Infraestrutura necessária

Requisitos:

- Mode change
- MMU
- Paginação (SATP)
- Stacks separadas
- Tratamento de interrupções
  - Interrupção de tempo (para escalonador)
- A própria troca de contexto dos processos
  - Papel da stack do kernel como auxiliar da troca

Modos de privilégio — RISC-V

O RISC-V possui três modos de privilégio principais:

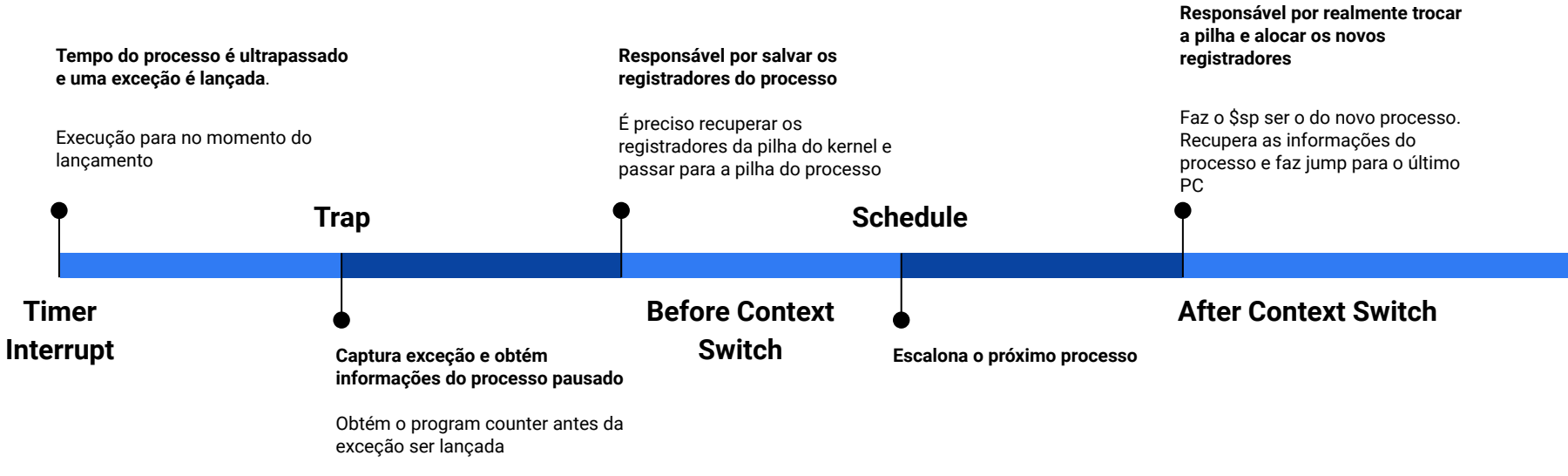
1. Machine (M) Mode: Bootloader, Firmware.
2. Supervisor (S) Mode: Kernel (módulos e drivers de dispositivo).
3. User (U) Mode: Código do usuário.

Cada um desses modos está relacionado a um conjunto de registradores de controle e estado (CSR) que guardam informações importantes sobre o funcionamento do programa.

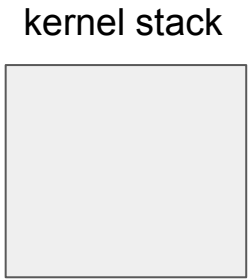
Escalonador

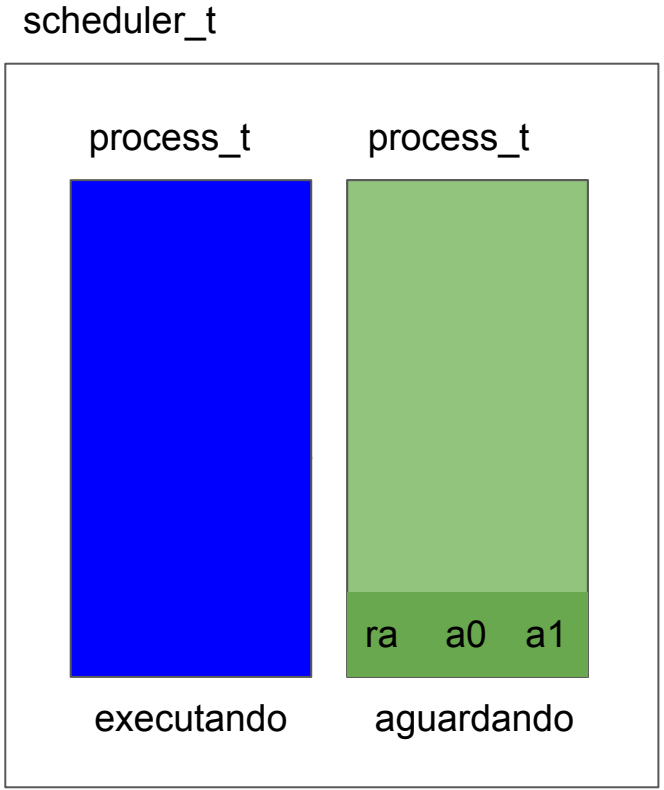
- Chamado quando interrupção de tempo é alcançada
  - Precisamos identificar qual trap foi chamada (usando o registrador mcause)
- Então começa o funcionamento do escalonador (e por conseguinte a troca de contexto dos processos)

Passo-a-passo de uma Troca de Processo



mie	Habilita exceções de timer
mtimecmp	Diz o tempo de execução até disparar exceção
trap	Aguarda exceção para tratar





mie

Habilita exceções de timer

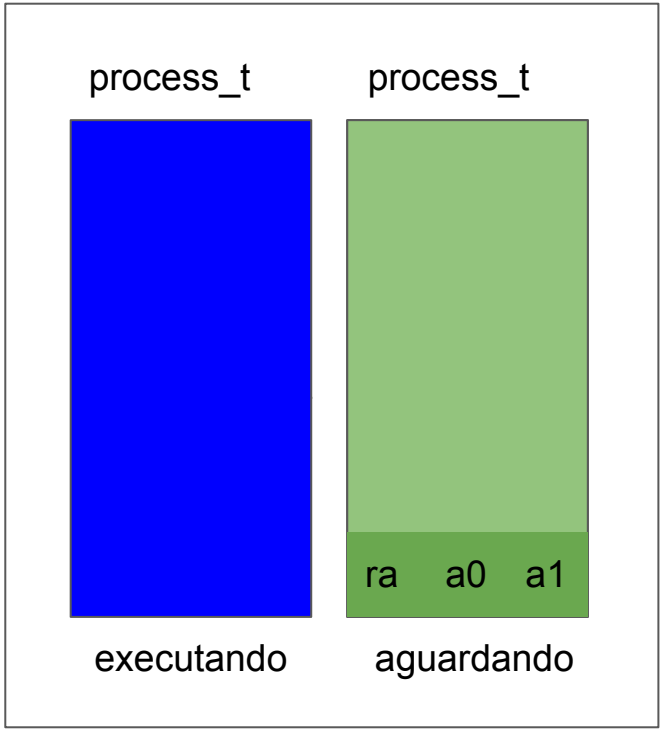
mtimecmp

Diz o tempo de execução até disparar exceção

trap

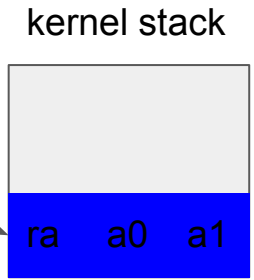
Detecta exceção. Inicia-se a troca de contexto

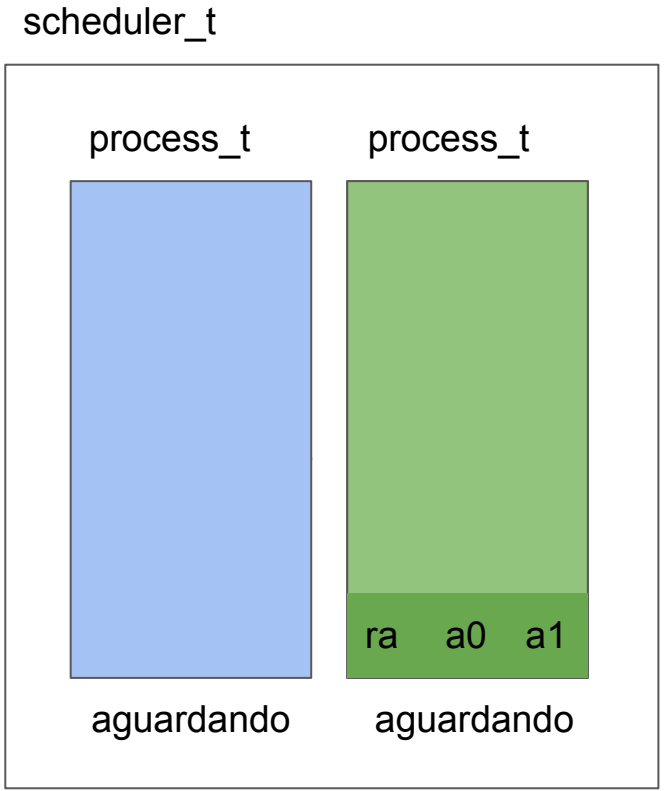
kernel stack



Salva-se os registradores na kernel_stack (para manter entre as trocas de funções).
Salva também o ra (return address, que será o PC no próximo escalonamento)

ABI Name	Description
zero	Hard-wired zero
ra	Return address
sp	Stack pointer
gp	Global pointer
tp	Thread pointer
t0	Temporary/alternate link register
t1–2	Temporaries
s0/fp	Saved register/frame pointer
s1	Saved register
a0–1	Function arguments/return values
a2–7	Function arguments
s2–11	Saved registers
t3–6	Temporaries

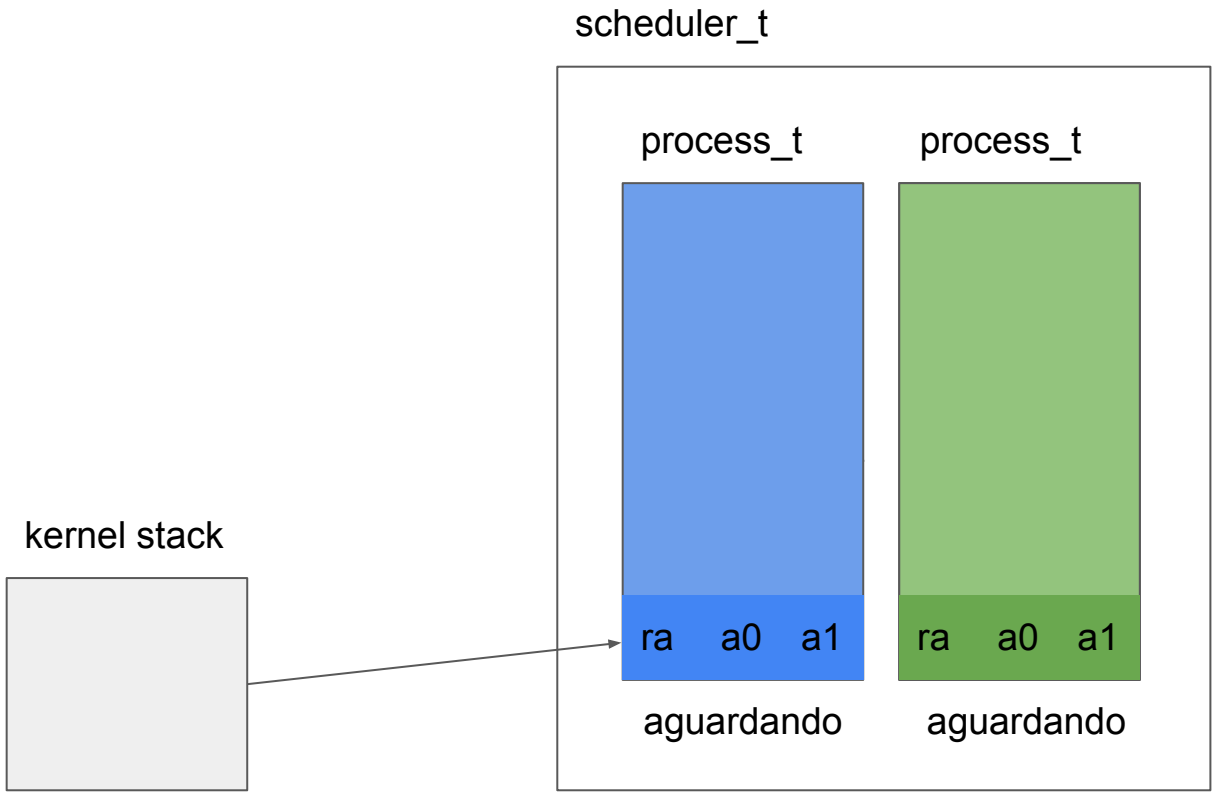




Chama-se o before_context_switch, responsável por transferir da kernel_stack para a stack do processo atual e é escalonado um novo

```
.global before_context_switch
before_context_switch:
    # kernel stack
    ld      s0, 0(sp)   # load ra (PC on the
    ld      s1, 8(sp)   # load satp
    ld      a0, 16(sp)   # load a0
    ld      a1, 24(sp)   # load a1
    ld      a2, 32(sp)   # load a2
    ld      a3, 40(sp)   # load a3
    ld      a4, 48(sp)   # load a4
    ld      a5, 56(sp)   # load a5
    ld      a6, 64(sp)   # load a6
    ld      a7, 72(sp)   # load a7
    ld      t0, 80(sp)   # load t0
    ld      t1, 88(sp)   # load t1
    ld      t2, 96(sp)   # load t2
    ld      t3, 104(sp)   # load t3
    ld      t4, 112(sp)   # load t4
    ld      t5, 120(sp)   # load t5
    ld      t6, 128(sp)   # load t6
    addi    sp, sp, 136

    # user stack (push info into that)
    jal     switch_user_stack
    addi    sp, sp, -136
    sd      s0, 0(sp)   # store PC
    sd      s1, 8(sp)   # store satp
```

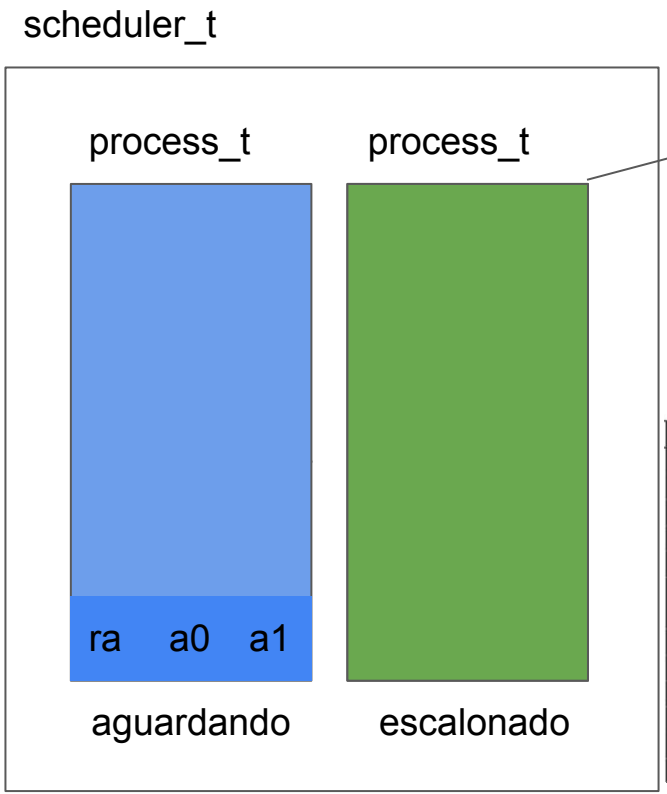


É escalonado o próximo processo. Por fim, restaura-se o contexto e o SATP e é feito um jump para o antigo program counter

```
# a0 = old context
# a1 = new context
.global after_context_switch
after_context_switch:
    # store sp into PCB of old process
    sw      sp, 0(a0)

    # load sp from PCB of new process
    mv      sp, a1

    # load PC and other registers
    ld      s0, 0(sp)  # load pc
    ld      s1, 8(sp)  # load satp that was
    ld      a0, 16(sp)   # load a0
    ld      a1, 24(sp)   # load a1
    ld      a2, 32(sp)   # load a2
    ld      a3, 40(sp)   # load a3
    ld      a4, 48(sp)   # load a4
    ld      a5, 56(sp)   # load a5
    ld      a6, 64(sp)   # load a6
    ld      a7, 72(sp)   # load a7
    ld      t0, 80(sp)   # load t0
    ld      t1, 88(sp)   # load t1
    ld      t2, 96(sp)   # load t2
    ld      t3, 104(sp)   # load t3
    ld      t4, 112(sp)   # load t4
```



ABI Name	Description
zero	Hard-wired zero
ra	Return address
sp	Stack pointer
gp	Global pointer
tp	Thread pointer
t0	Temporary/alternate link register
t1–2	Temporaries
s0/fp	Saved register/frame pointer
s1	Saved register
a0–1	Function arguments/return values
a2–7	Function arguments
s2–11	Saved registers
t3–6	Temporaries

Detalhes de implementação

- Feito exclusivamente em machine mode
- Não funciona para versões mais novas do QEMU
- Baseado em um u-kernel ARM, adaptado para RISCV

Código realizado

Estruturas utilizadas

```c
// PCB
typedef struct stack {
    volatile uint64_t * stack;
    uint64_t stack_base[MAX_STACK];
} process_t;

// Scheduler
typedef struct {
    unsigned int length;
    volatile unsigned int current_id;
    process_t process[MAX_PROCESSES];
} scheduler_t;

scheduler_t scheduler;
process_t kernel_stack;
```

Main criando os processos

```
extern "C" int main() {
    uint64_t satp0 = make_process_pagetable();
    create_process(&process1_entry, satp0);

    uint64_t satp1 = make_process_pagetable();
    create_process(&process2_entry, satp1);

    while (TRUE) {
        print("0000000000\n");
        halt();
    }
    return 0;
}
```

```
void create_process(void (*process_entry)(void), uint64_t satp) {
    unsigned int id = scheduler.length;
    scheduler.process[id].stack = scheduler.process[id].stack_base
     + (MAX_STACK - 1);

    // add first information into the stack
    asm_create_process(scheduler.process[id].stack, (uint64_t)
     process_entry, satp);

    // increase scheduler process length
    scheduler.length++;
}
```

Main criando os processos

```
# a0 = stack
# a1 = process_entry
# a2 = satp
.global asm_create_process
asm_create_process:
    addi    a0, a0, -136
    sd      a1, 0(a0)       # process entry (PC)
    sd      a2, 8(a0)       # satp
    sd      zero, 16(a0)     # a0
    sd      zero, 24(a0)    # a1
    sd      zero, 32(a0)    # a2
    sd      zero, 40(a0)    # a3
    sd      zero, 48(a0)    # a4
    sd      zero, 56(a0)    # a5
    sd      zero, 64(a0)    # a6
    sd      zero, 72(a0)    # a7
    sd      zero, 80(a0)    # t0
    sd      zero, 88(a0)    # t1
    sd      zero, 96(a0)    # t2
    sd      zero, 104(a0)    # t3
    sd      zero, 112(a0)    # t4
    sd      zero, 120(a0)    # t5
    sd      zero, 128(a0)    # t6
    ret
```

```
void process1_entry(void) {
    const char * message = "process1\n";
    while (TRUE) {
        print( s: message);
        halt();
    }
}

void process2_entry(void) {
    const char * message = "process2\n";
    while (TRUE) {
        print( s: message);
        halt();
    }
}
```

Funcionamento do escalonador

```
extern "C" void schedule() {
    int current_id = scheduler.current_id;
    int next_id = current_id + 1;

    // round-robin: circular queue
    if (next_id >= scheduler.length) next_id = 0;

    // current sp (old process)
    volatile uint64_t * sp = get_sp() + 4;
    scheduler.process[current_id].stack = sp;

    // do context switch
    scheduler.current_id = next_id;

    after_context_switch(
            scheduler.process[current_id].stack,
            scheduler.process[next_id].stack);
}
```

Processos sendo escalonados

```
kinit
0000000000
Timer count: 1
process1
Timer count: 2
process2
Timer count: 3
```