循序渐进，学习开发一个 RISC-V 上的操作系统

第 9 章 上下文切换和协作式多任务

汪辰

2021/4

本章内容

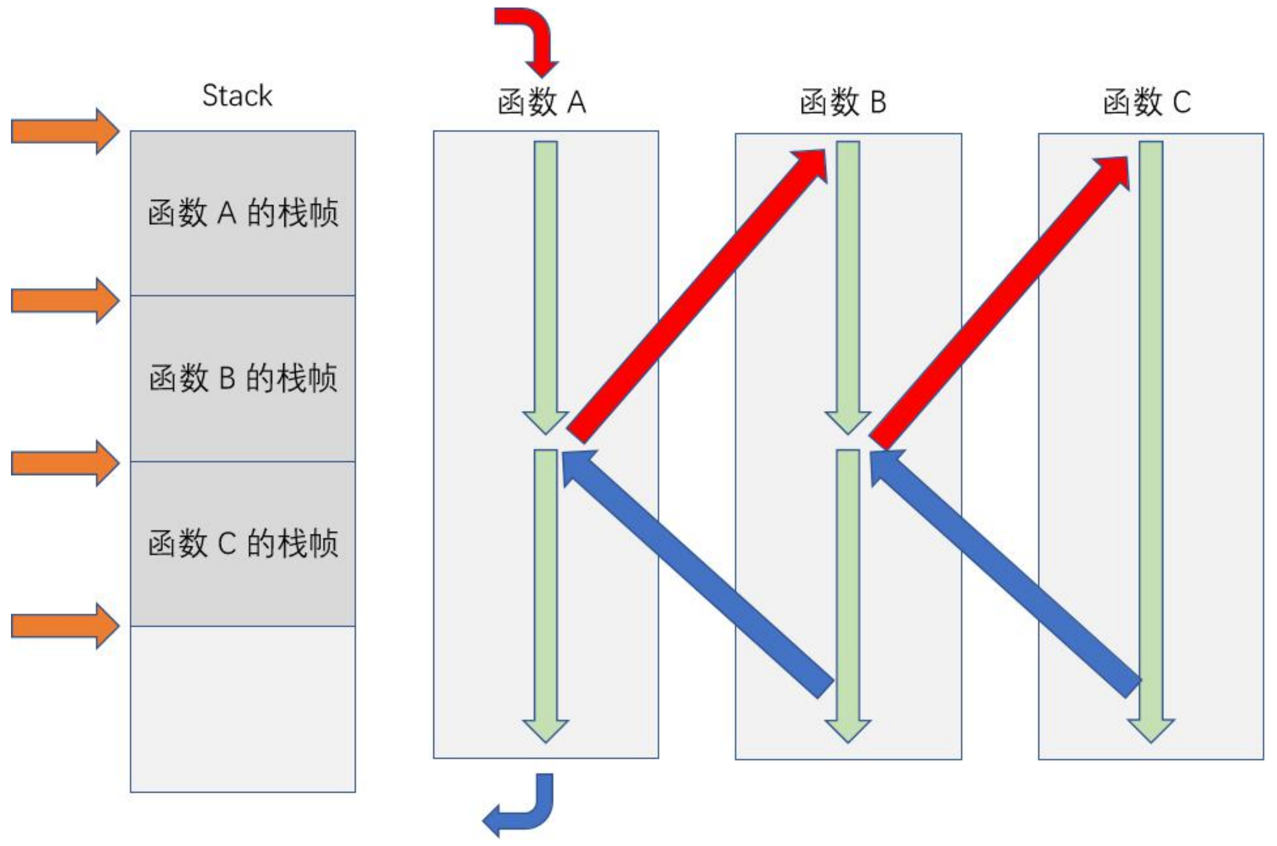
- 多任务与上下文
- 协作式多任务的设计与实现

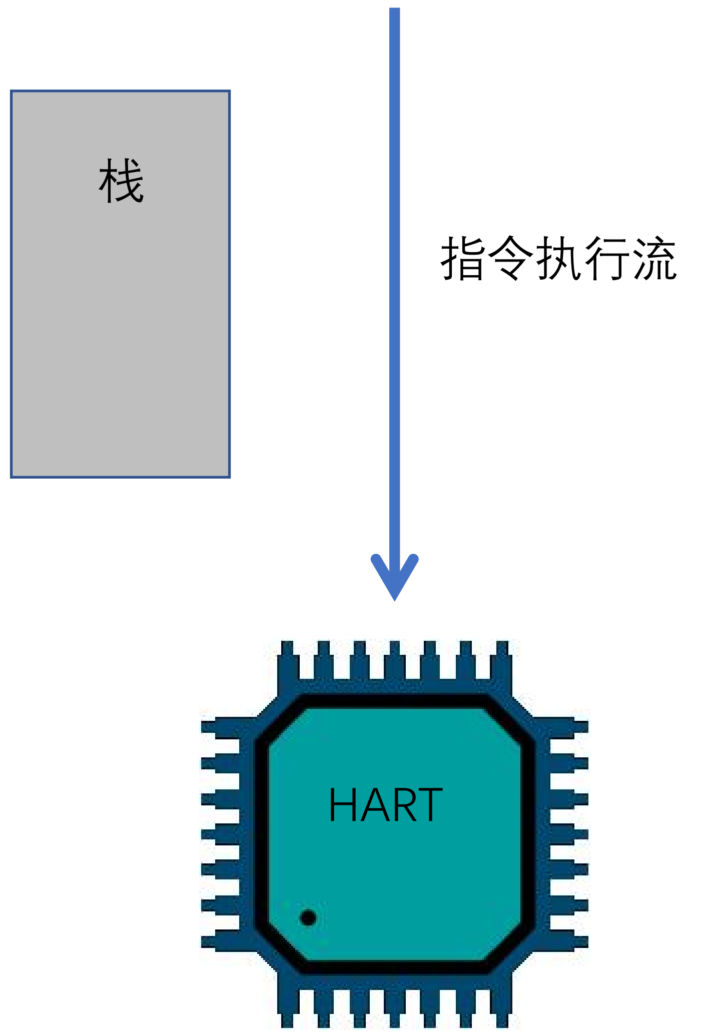
本章内容

- **多任务与上下文**
  - 任务的概念
  - 多任务的概念
  - 任务上下文的概念

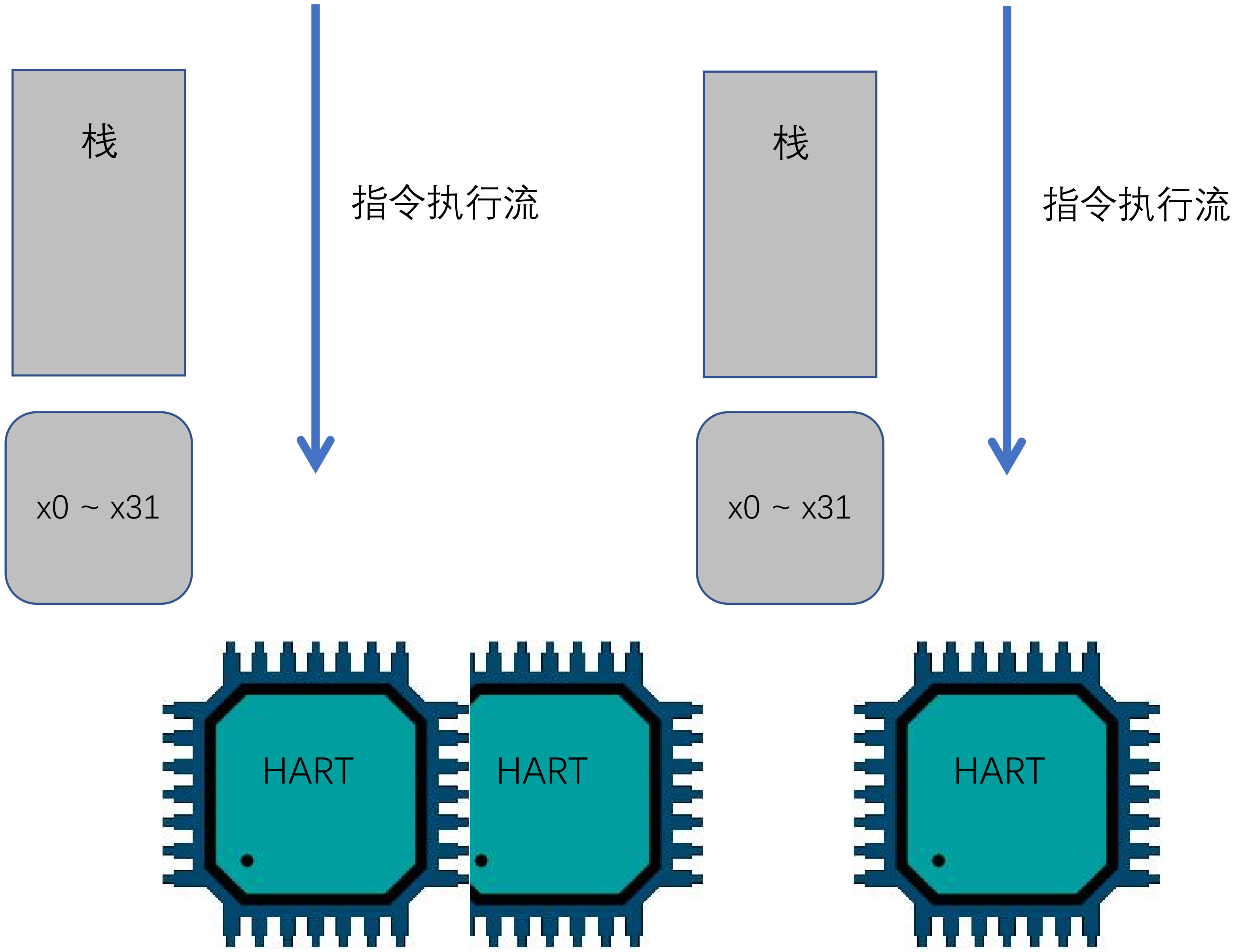
- **协作式多任务的设计与实现**

任务（task）

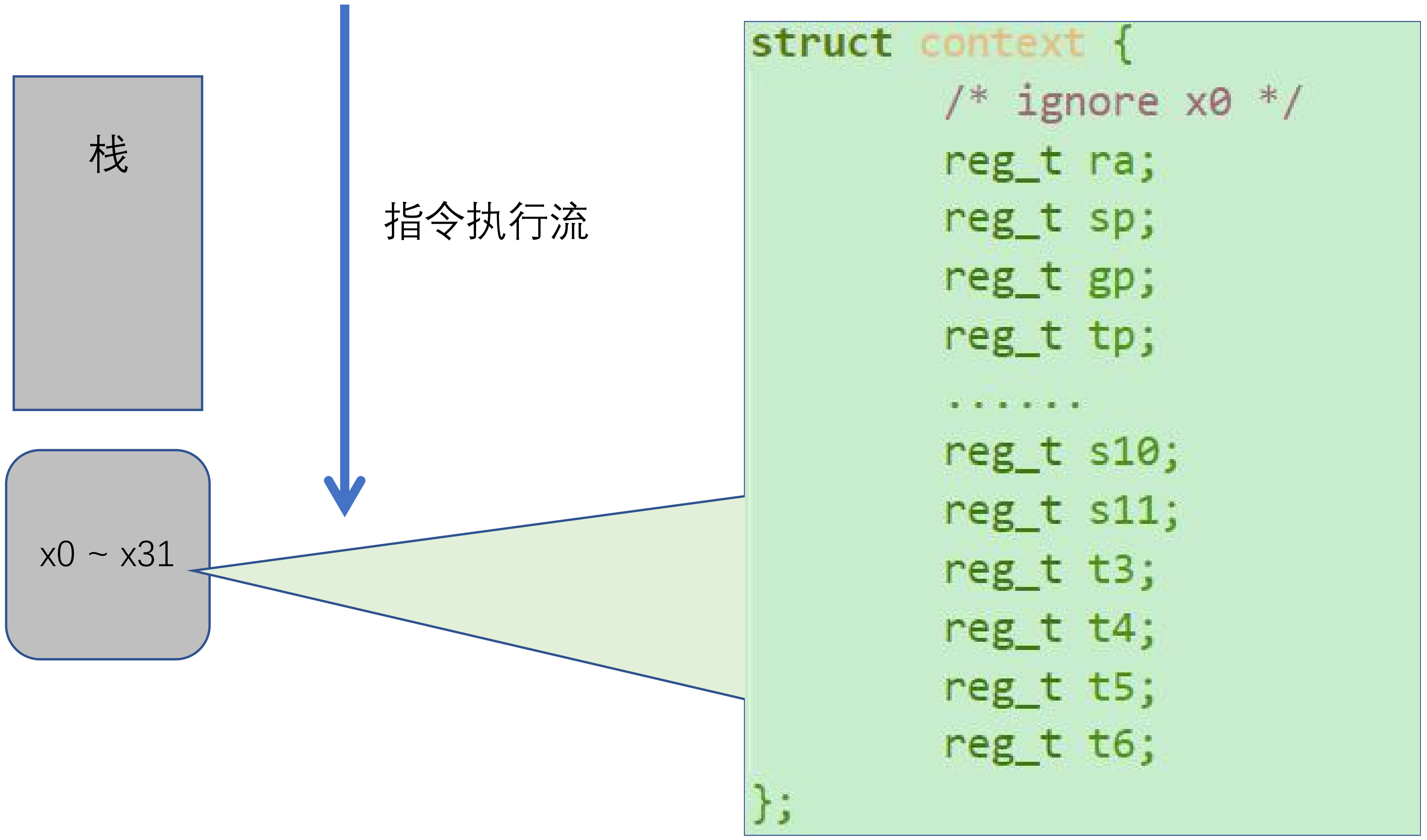




多任务 （Multitask）



任务上下文（Context）



```
struct context {
        /* ignore x0 */
        reg_t ra;
        reg_t sp;
        reg_t gp;
        reg_t tp;
        .......
        reg_t s10;
        reg_t s11;
        reg_t t3;
        reg_t t4;
        reg_t t5;
        reg_t t6;
};
```

本章内容

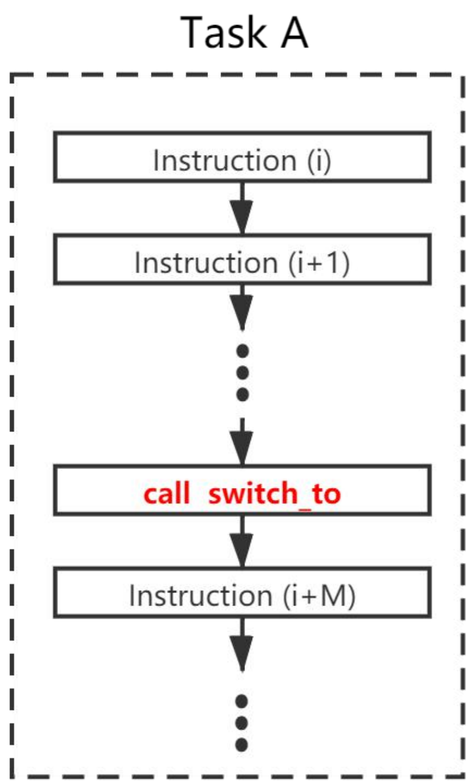
- 多任务与上下文
- **协作式多任务的设计与实现**
  - 协作式多任务和抢占式多任务
  - 协作式多任务的设计思路
  - 协作式多任务的关键实现

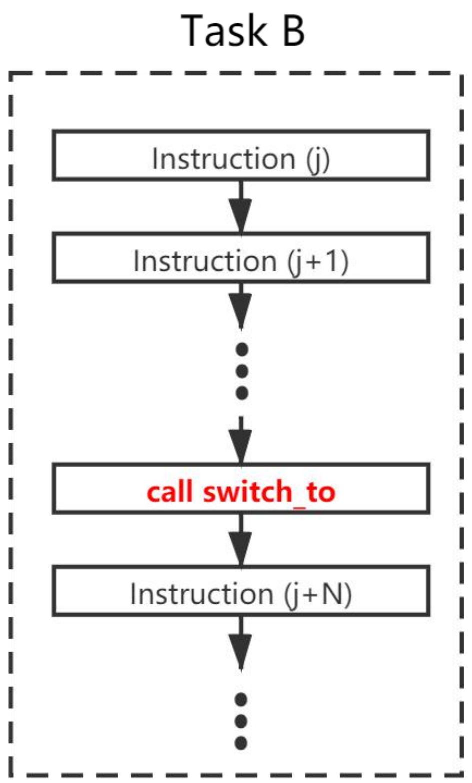
多任务系统的分类

- **协作式多任务 (Cooperative Multitasking)**：协作式环境下，下一个进程被调度的前提是当前进程主动放弃时间片。

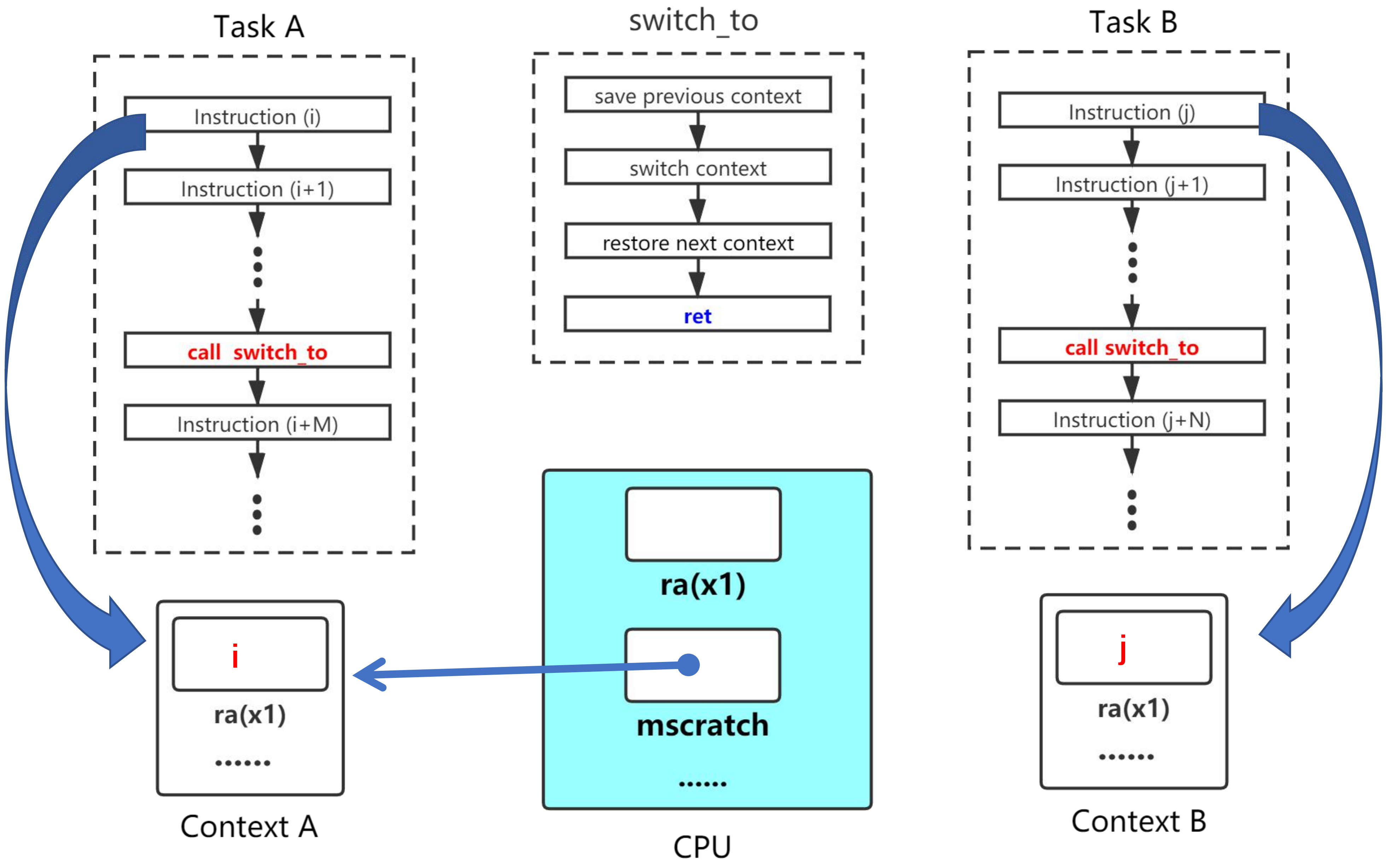
- **抢占式多任务 (Preemptive Multitasking)**：抢占式环境下，操作系统完全决定进程调度方案，操作系统可以剥夺耗时长的进程的时间片，提供给其它进程。

协作式多任务

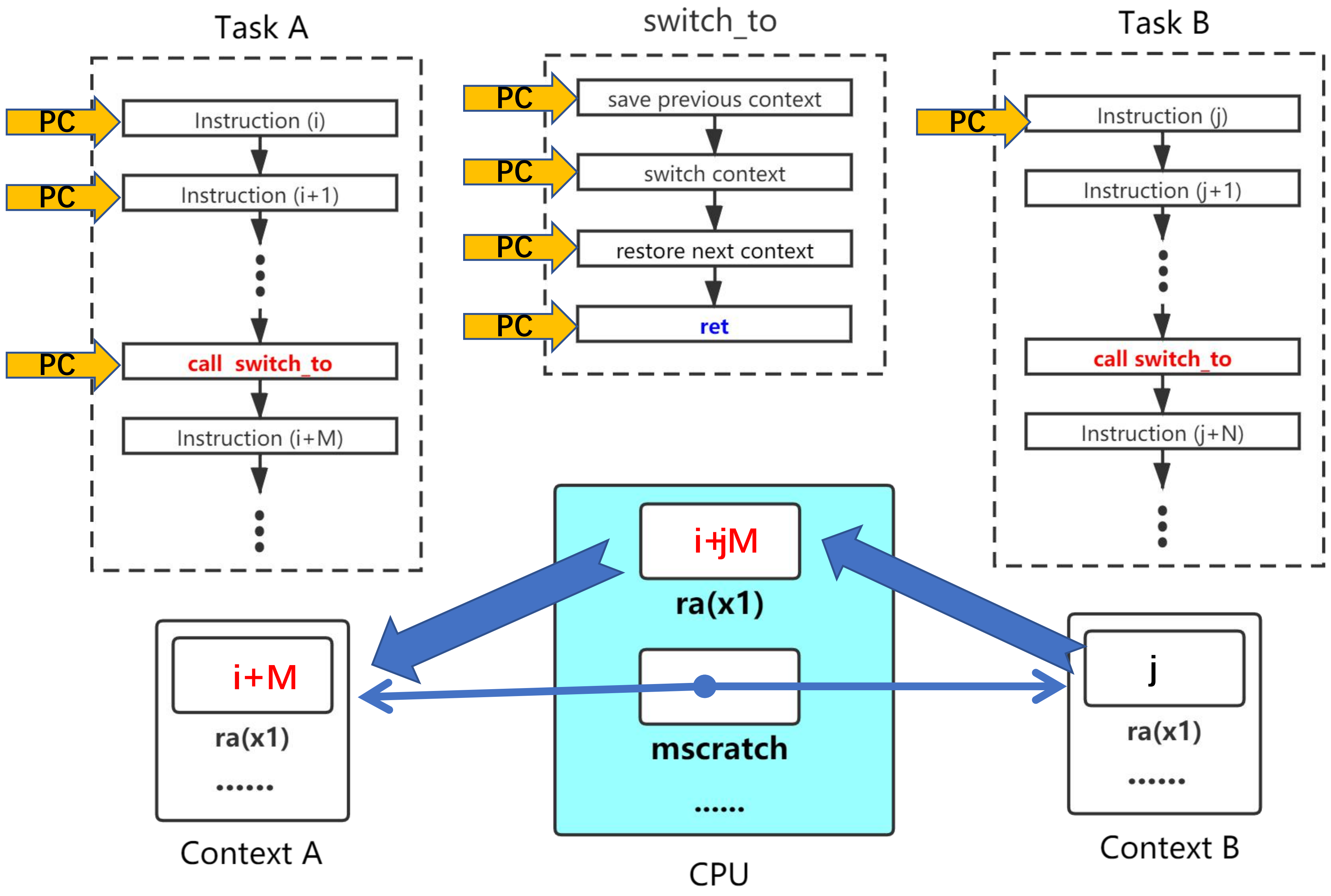

Task A
Instruction (i)
Instruction (i+1)
call switch_to
Instruction (i+M)


Task B
Instruction (j)
Instruction (j+1)
call switch_to
Instruction (j+N)

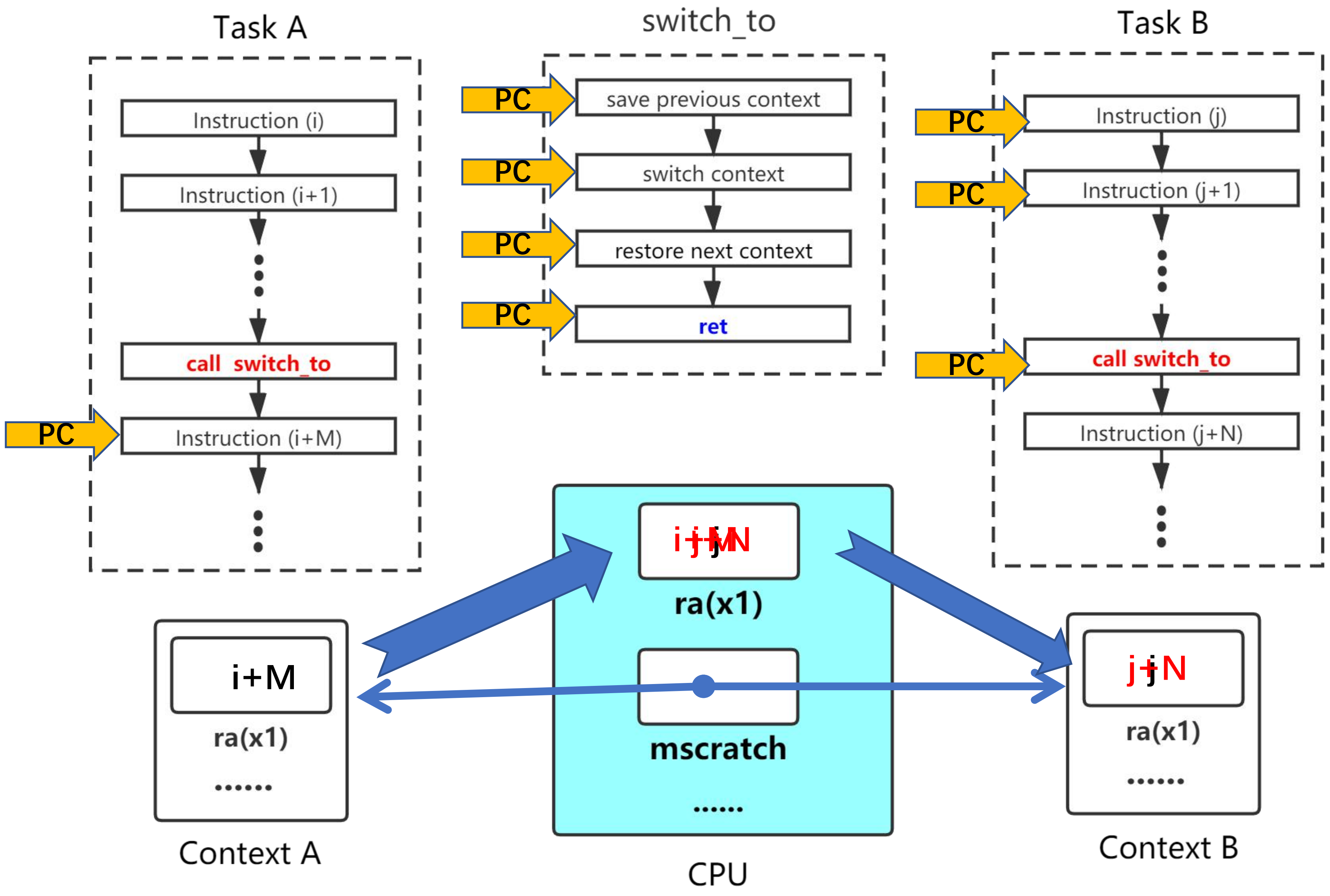
协作式多任务


Task A
Instruction (i)
Instruction (i+1)
call switch_to
Instruction (i+M)
switch_to
save previous context
switch context
restore next context
ret
Task B
Instruction (j)
Instruction (j+1)
call switch_to
Instruction (j+N)
i
ra(x1)
......
Context A
ra(x1)
mscratch
......
CPU
j
ra(x1)
......
Context B

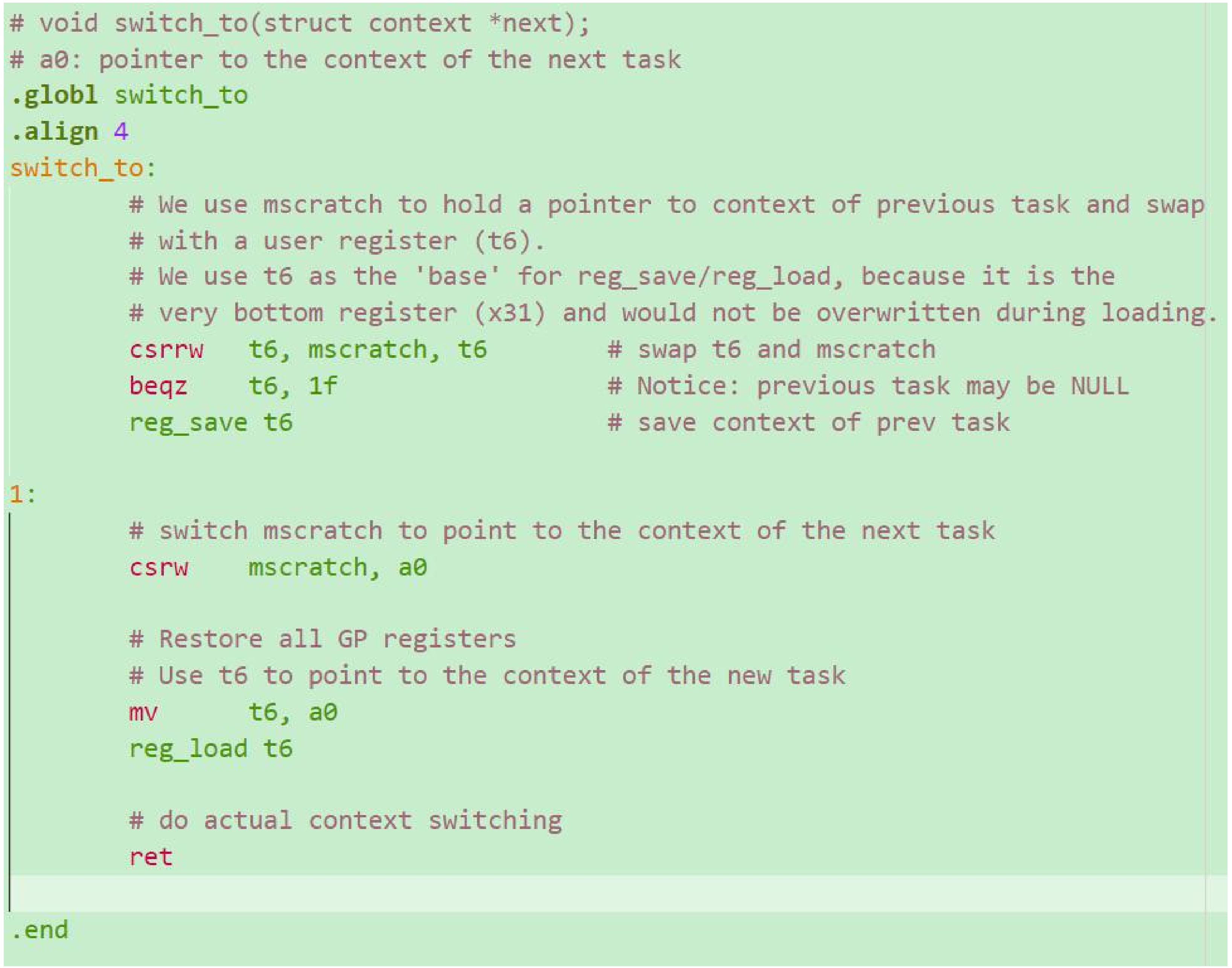
协作式多任务


Task A
PC
Instruction (i)
PC
Instruction (i+1)
PC
call switch_to
Instruction (i+M)
switch_to
PC
save previous context
PC
switch context
PC
restore next context
PC
ret
Task B
PC
Instruction (j)
Instruction (j+1)
call switch_to
Instruction (j+N)
i+M
ra(x1)
......
Context A
i+jM
ra(x1)
mscratch
......
CPU
j
ra(x1)
......
Context B

协作式多任务


Task A
Instruction (i)
Instruction (i+1)
call switch_to
Instruction (i+M)
PC
switch_to
save previous context
switch context
restore next context
ret
PC
Task B
Instruction (j)
Instruction (j+1)
call switch_to
Instruction (j+N)
PC
i+M
ra(x1)
......
Context A
i+j+M+N
ra(x1)
mscratch
......
CPU
j+j+N
ra(x1)
......
Context B

关键函数（switch_to）



```
# void switch_to(struct context *next);
# a0: pointer to the context of the next task
.globl switch_to
.align 4
switch_to:
        # We use mscratch to hold a pointer to context of previous task and swap
        # with a user register (t6).
        # We use t6 as the 'base' for reg_save/reg_load, because it is the
        # very bottom register (x31) and would not be overwritten during loading.
        csrrw   t6, mscratch, t6        # swap t6 and mscratch
        beqz    t6, 1f                  # Notice: previous task may be NULL
        reg_save t6                     # save context of prev task

1:
        # switch mscratch to point to the context of the next task
        csrw    mscratch, a0

        # Restore all GP registers
        # Use t6 to point to the context of the new task
        mv      t6, a0
        reg_load t6

        # do actual context switching
        ret

.end
```

context 数据结构设计

```
#define STACK_SIZE 1024
uint8_t task_stack[STACK_SIZE];
struct context ctx_task;
```

```
w_mscratch(0);

ctx_task.sp = (reg_t) &task_stack[STACK_SIZE - 1];
ctx_task.ra = (reg_t) user_task0;
```

```
struct context *next = &ctx_task;
switch_to(next);
```

```
struct context {
        /* ignore x0 */
        reg_t ra;
        reg_t sp;
        reg_t gp;
        reg_t tp;
        ......
        reg_t s10;
        reg_t s11;
        reg_t t3;
        reg_t t4;
        reg_t t5;
        reg_t t6;
};
```

谢 谢

欢迎交流合作