Operating Systems

Stack and Context Switch

Giorgio Grisetti
grisetti@diag.uniroma1.it

Department of Computer Control and Management Engineering
Sapienza University of Rome

Context of a Process

Process: running program

Context:

- CPU registers
- Process Memory
    - Stack (.stack)
    - Program code (.text), typically read only during execution
    - Initialized Variables (.data)
    - Global Uninitialized Variables (.bss)

If each process uses only its own memory, the execution can be stopped and later recovered by saving/restoring the CPU registers

Coroutines

Coroutine: piece of program that can be "jumped in" and "out"

- In assembly "jmp".
- In C: need to preserve consistency of the stack.

- Ucontext: portable C library for user level control of contexts

Ucontext: concepts

`struct ucontext_t`; datatype to store a context

- **`ucontext_t *uc_link`**: pointer to the context that will be resumed when this context returns
- **`sigset_t uc_sigmask`**: the set of signals that are blocked when this context is active
- **`stack_t uc_stack`**: the stack used by this context
- **`mcontext_t uc_mcontext`**: a machine-specific representation of the saved context

getContext

`int getcontext(ucontext_t *ucp);`

- Saves the current context in ucp.
- A subsequent call to setcontext(ucp) will result in the flow of the program continuing from the instruction following setcontext(ucp);

```
struct ucontext_t ctx;      ← Here we store our jump point

int f2(){
  setContext(&ctx);
}

int f1(){
  ….
  getContext(&ctx);         ← set ctx to jump to the next instruction after
  …                           getcontext
  f2();
}
```

setContext

`int setcontext(const ucontext_t *ucp)`

- Sets the current context to ucp, a context that was previously saved.
- The flow will continue from the instruction following the
- getcontext(ucp) call issued when SAVING the context

```
struct ucontext_t ctx;

int f2(){
  setContext(&ctx);      ← We jump to a saved context [1]
}

int f1(){                 [1]
  ….
  getContext(&ctx);

  f2();
}
```

makeContext

`void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);`

- creates a trampoline context for function func.
- the context is initialized so that when jumping to it it will start executing the function func
- ucp should have the stack and the signal mask already set before calling makecontext

```
struct ucontext_t ctx;

int f2(){
    …
}

int main(){
    ….
    makeContext(&ctx, f2,...);          ← ctx initialized so that when called starts f2
    …
    setContext(&ctx);                   ← flow continues from f2
}
```

swapContext

```
int swapcontext(ucontext_t *oucp,
const ucontext_t *ucp);
```

- saves the current context in oucp, and jumps to ucp

Full example

```
ucontext_t main_context, f1_context, f2_context;

void f1(){
  printf("f1 started\n");
  for (int i=0; i<num_iterations; i++) {
    printf("f1: %d\n", i);
    swapcontext(&f1_context, &f2_context);
  }
  setcontext(&main_context);
}

void f2(){
  printf("f2 started\n");
  for (int i=0; i<num_iterations; i++) {
    printf("f2: %d\n", i);
    swapcontext(&f2_context, &f1_context);
  }
  setcontext(&main_context);
}

char f1_stack[STACK_SIZE];
char f2_stack[STACK_SIZE];
```

```
int main(){
  //get a context from main
  getcontext(&f1_context);

  // set the stack of f1 to the right place
  f1_context.uc_stack.ss_sp=f1_stack;
  f1_context.uc_stack.ss_size = STACK_SIZE;
  f1_context.uc_stack.ss_flags = 0;
  f1_context.uc_link=&main_context;

  // create a trampoline for the first function
  makecontext(&f1_context, f1, 0, 0);

  // always remember to initialize
  // a new context from something known
  f2_context=f1_context;

  f2_context.uc_stack.ss_sp=f2_stack;
  f2_context.uc_stack.ss_size = STACK_SIZE;
  f2_context.uc_stack.ss_flags = 0;
  f2_context.uc_link=&main_context;

  // create a trampoline for the second function
  makecontext(&f2_context, f2, 0, 0);

  // this passes control to f2.
  // and saves the current context in main_context

  swapcontext(&main_context, &f1_context);
  // we will jump back here

  printf("exiting\n");
}
```

Exercise

- Modify the program above to spin on 10 different contexts instead of two

Preemptive multitasking on AVR

We want to implement an timer controlled preemptive task switcher on our arduino.

- **Task Control Blocks**: stored in double linked list
- Always at least one process in running

Initialization

- Fill in TCB data
- Prepare all stack frames so that the Program Counter stored on the stack points to a launcher for the thread function, and all registers clean

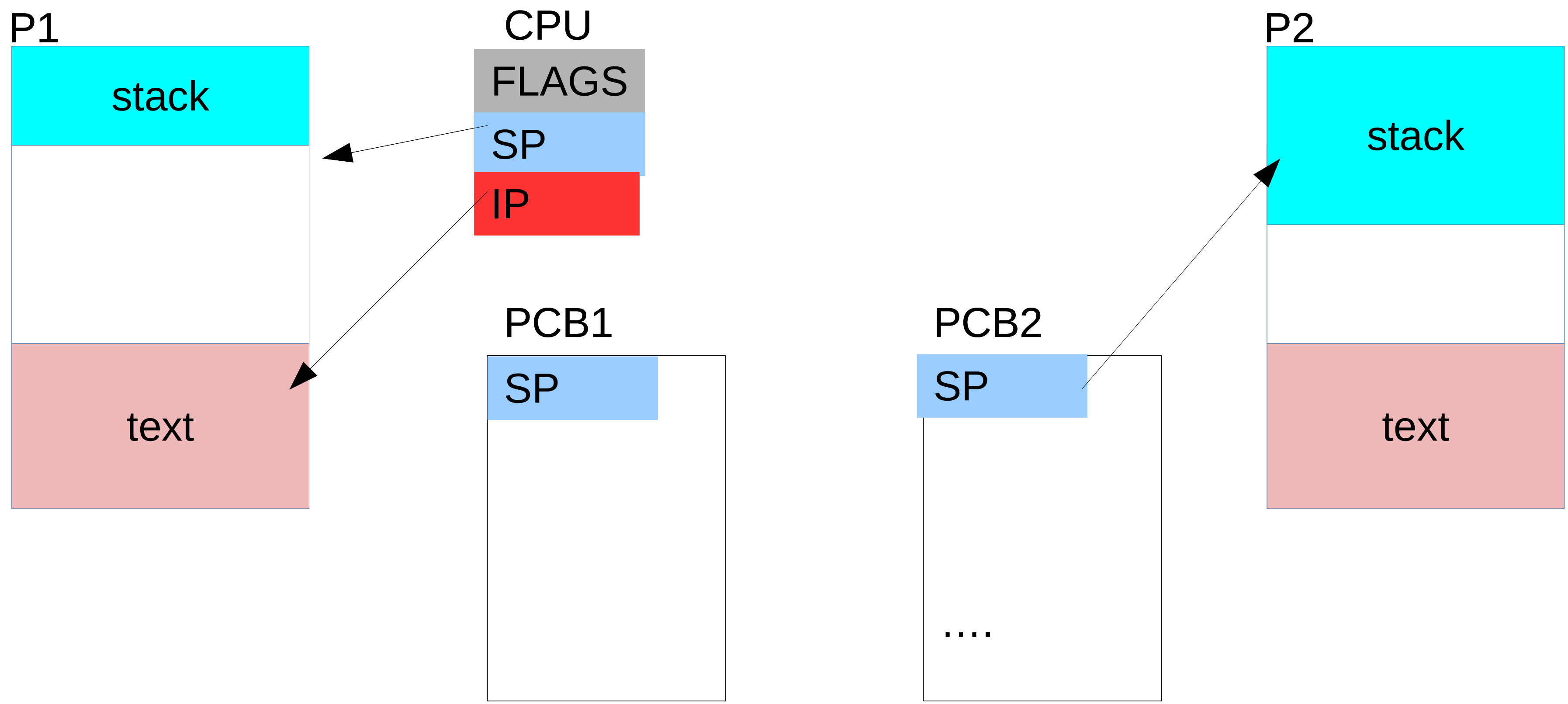
Start

- Change stack pointer to first tcb
- Pull all registers
- Return from function

Context switch (once all is set), on interrupt:

- Save all registers on stack
- Change stack pointer
- Pull all registers from stack
- Return from interrupt

Context Switch On AVR

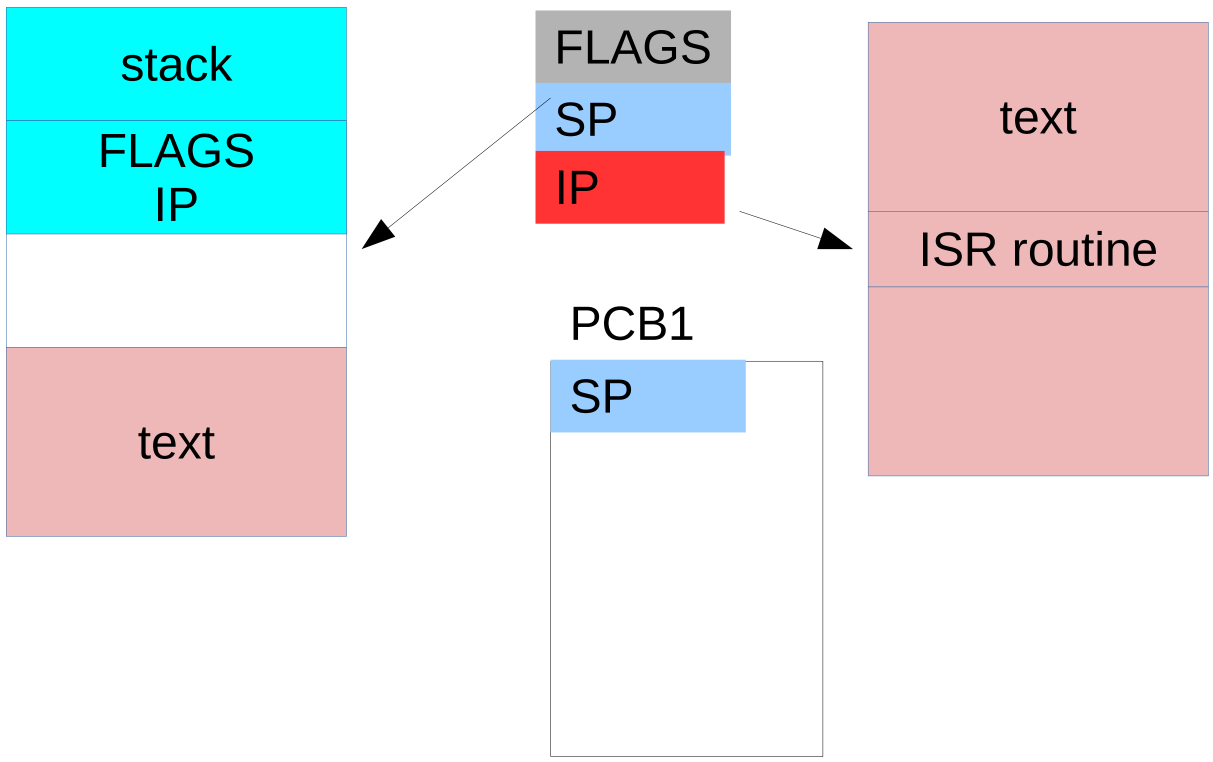


Scenario

- we have two threads: P1 in running and P2 in ready
- P2 was previously running but it has been preempted before. His status is in PCB2
- P1 is running

What should happen such that after an interrupt the CPU continues executing P2?

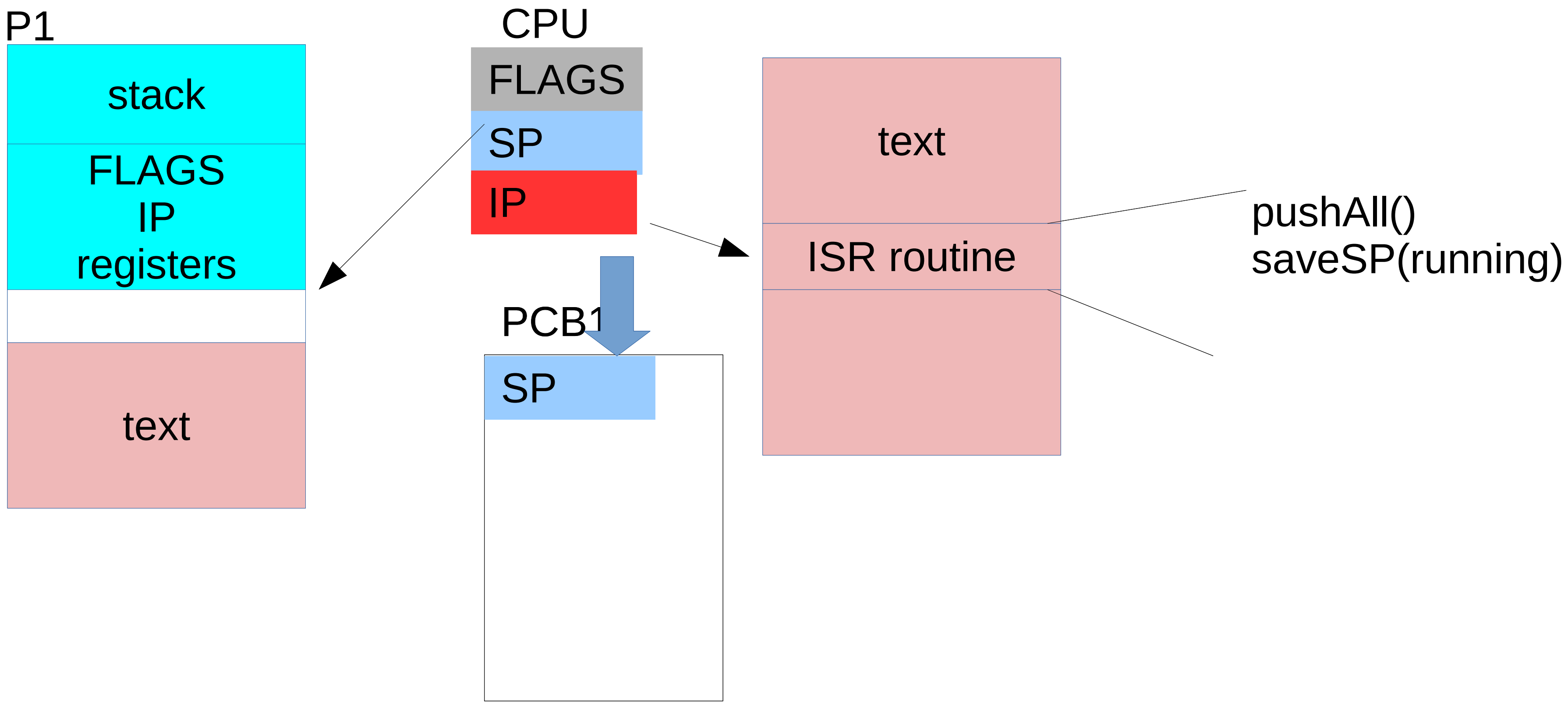
Context Switch on AVR



The interrupt comes, thus the CPU

- saves flags and instruction counter on the stack
- calls the appropriate ISR (that should terminate with IRET)

Context Switch on AVR

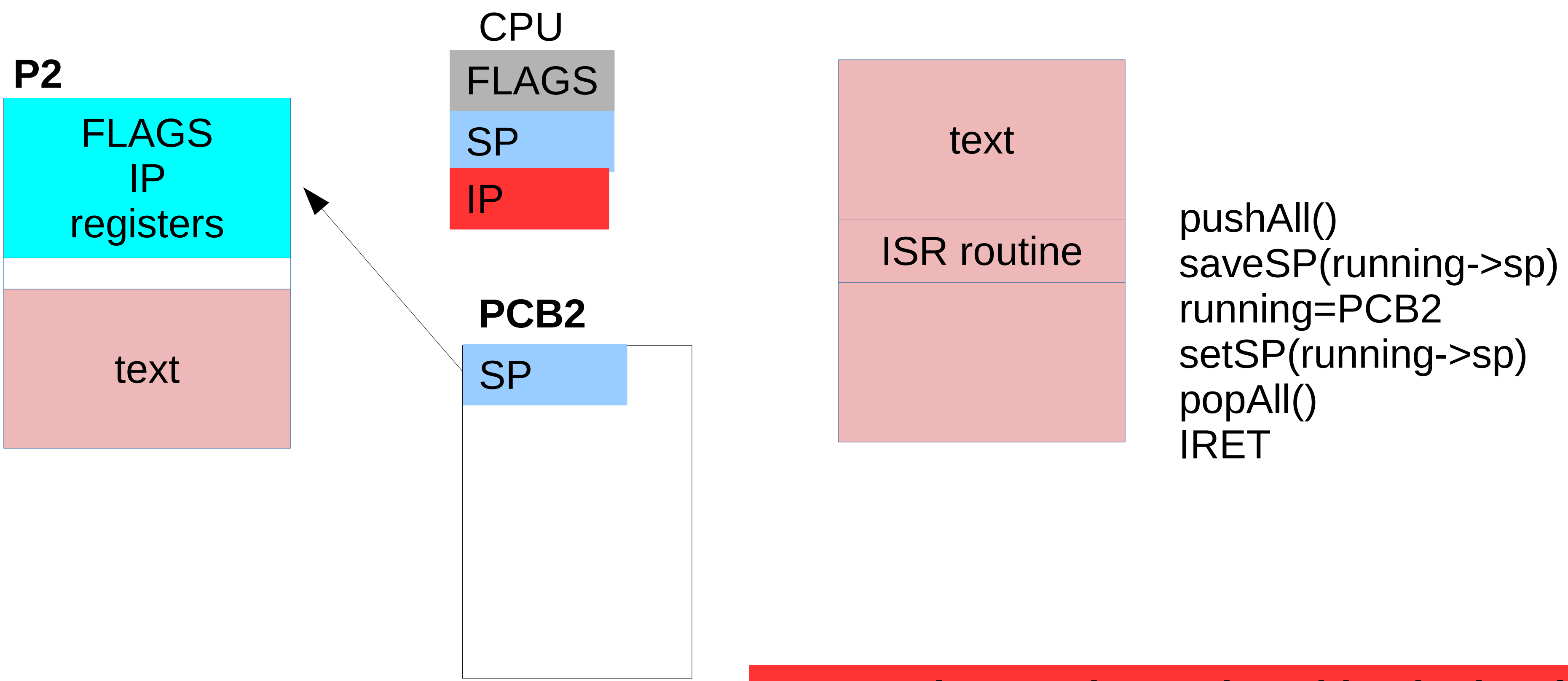


It is responsibility of the ISR to save the clobbered registers and be able to resume the state of the interrupted process

To recover P1 in the future, we need to save its CPU state in the PCB.

The state is on the stack, so in this example we save in the PCB just the stack pointer.

Context Switch on AVR



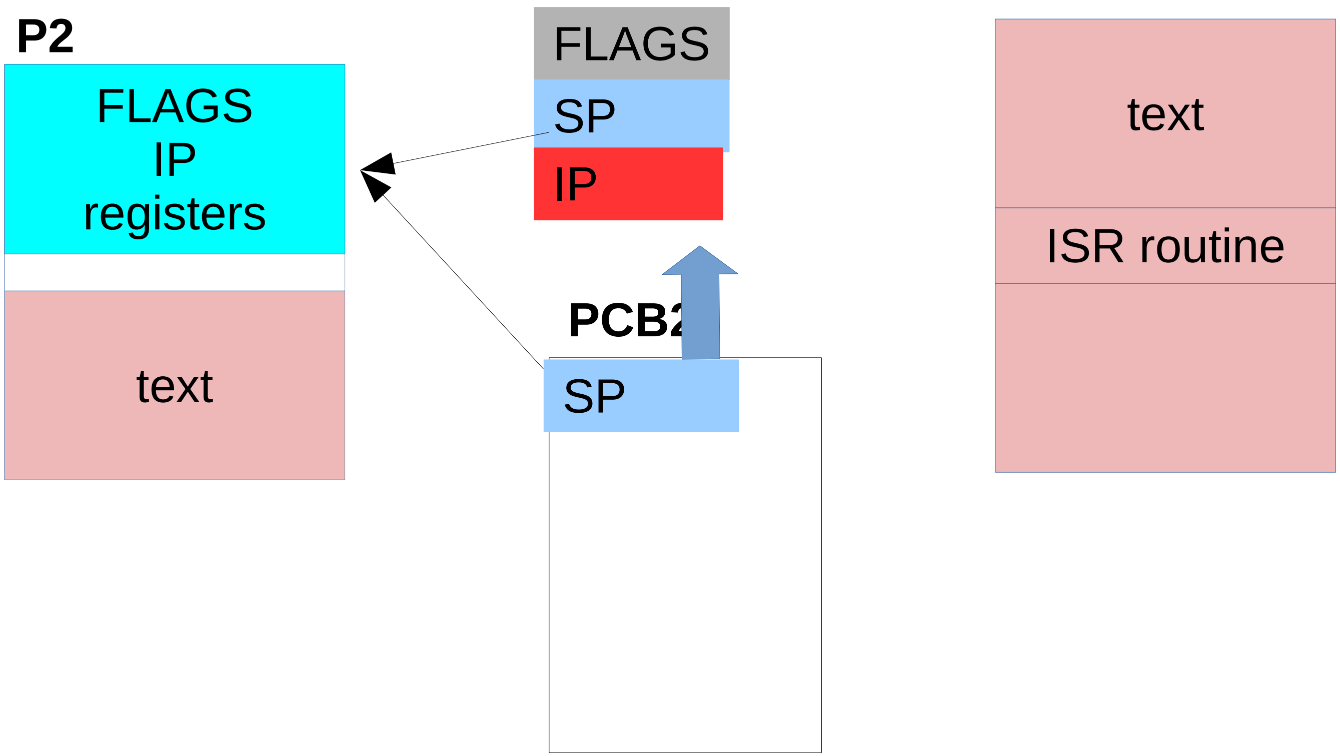
Beware that P2 changed position in the slide

Let us assume P2 is our next running, we need to start it again.

Since P2 was preempted, we know its structures are consistent

We know that the last instruction being executed by the ISR will be a return from interrupt (IRET), that recovers the flags.

Context Switch on AVR

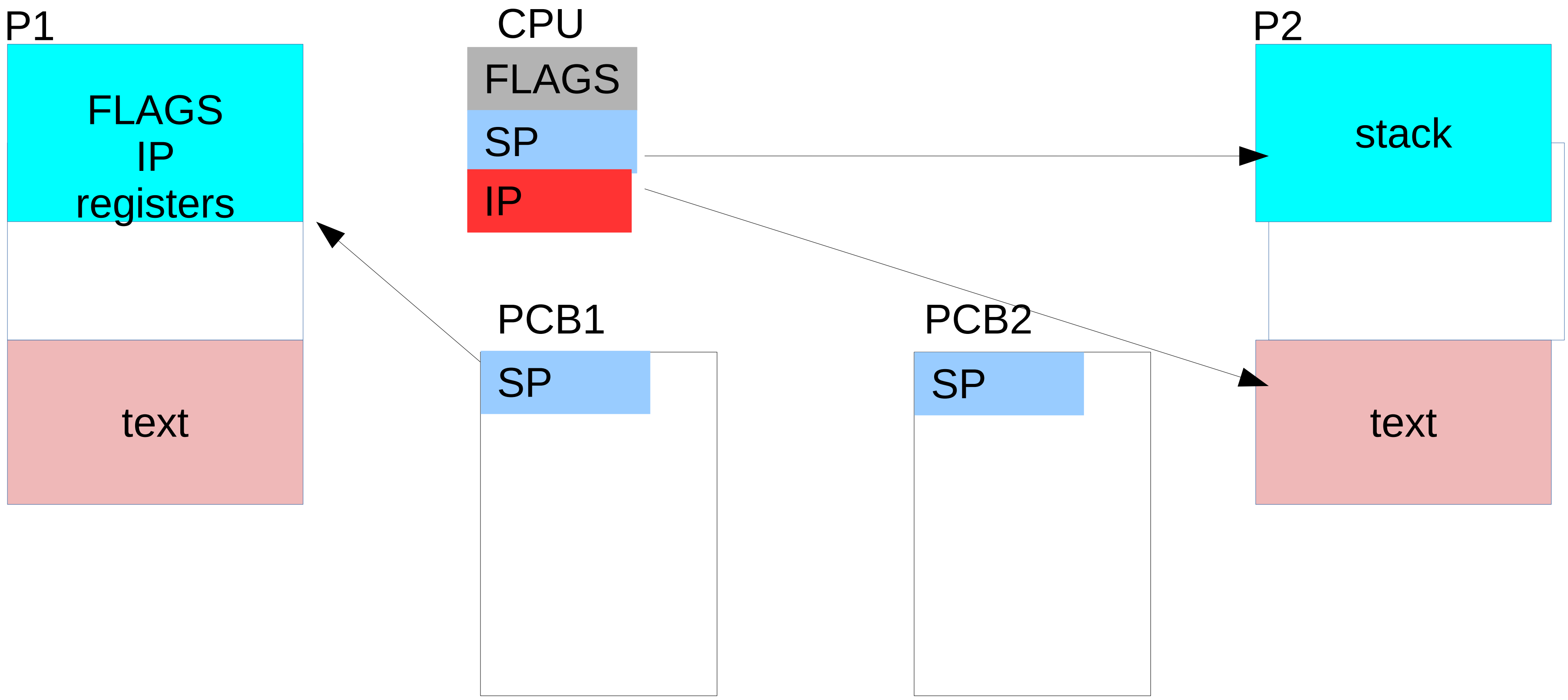



```
pushAll()
saveSP(running->sp)
running=PCB2
setSP(running->sp)
popAll()
IRET
```

To continue the execution, we

- change the stack back the one stored in the running pcb
- restore the state in the CPU
- return from interrupt

Context Switch on AVR



Et voila' P2 is running again as if nothing has happened

Task Control Block

```
#pragma once
#include <stdint.h>
#include <stddef.h>

#define OK     0
#define ERROR -1

typedef uint8_t* Pointer;
typedef void (* ThreadFn)(uint32_t thread_args);

typedef enum {Running=0x0, Terminated=0x1, Ready=0x2} ThreadStatus;

// thread control block
typedef struct TCB {
  Pointer sp_save_ptr;

  ThreadFn thread_fn;
  uint32_t thread_arg;

  struct TCB* next;
  struct TCB* prev;

  Pointer stack_bottom;         /* Pointer to bottom of stack allocation */
  uint32_t stack_size;          /* Size of stack allocation in bytes */
  ThreadStatus status;
} TCB;


void TCB_create(TCB* tcb, Pointer stack_top, ThreadFn thread_fn, uint32_t thread_arg);
```

TCB Create

```
void TCB_create(TCB* tcb, Pointer stack_top, ThreadFn thread_fn, uint32_t thread_arg){
  //initialize variables
  tcb->thread_fn=thread_fn;
  tcb->thread_arg=thread_arg;
  tcb->prev=NULL;
  tcb->next=NULL;
  tcb->status=Ready;

  /** prepare stack for process **/

  uint8_t *stack_ptr = (uint8_t *)stack_top;
  //write the return address of the function being called (the trampoline)
  *stack_ptr-- = (uint8_t)((uint16_t)_trampoline & 0xFF);
  *stack_ptr-- = (uint8_t)(((uint16_t)_trampoline >> 8) & 0xFF);
  *stack_ptr-- = 0; // store an additional segment register (atMega2560)


  /**
   * Store starting register values for R2-R17, R28-R29
   */
  *stack_ptr-- = 0x00;    /* R2 */
  *stack_ptr-- = 0x00;    /* R3 */
  ……………………… // here we save all other registers………
  *stack_ptr-- = 0x00;    /* R28 */
  *stack_ptr-- = 0x00;    /* R29 */
  *stack_ptr-- = 0x00;    /* RAMPZ */
  *stack_ptr-- = 0x00;    /* EIND */

  // store stack pointer
  tcb->sp_save_ptr = stack_ptr;

}
```

TCB Create, trampoline

- The trampoline is a convenient function without parameters that calls the function whose pointer is stored in the current_tcb global variable
- Not to mess up with calling conventions ;-)

```
static void _trampoline(void){
  sei();
  /* Call the thread entry point */
  if (current_tcb && current_tcb->thread_fn) {
    (*current_tcb->thread_fn)(current_tcb->thread_arg);
  }

  // set the thread to terminated, when the above function finishes
  current_tcb->status=Terminated;
}
```

TCB Queue

The TCBs are stored in a double linked list

No memory allocation

- Two actions:
  - Take out the element at the beginning of the list
  - Put an element out of the list at its tail

```
// simple double linked list of TCBs
typedef struct {
  struct TCB* first;
  struct TCB* last;
  uint8_t size;
} TCBList;

// global list of tcbs containing the running processes
extern TCBList tcb_queue;

// removes (if any) first tcb from the list
TCB* TCBList_dequeue(TCBList* list);

// adds new detached tcb to the list
uint8_t TCBList_enqueue(TCBList* list, TCB* tcb);
```

Context Switch

```
//void archContextSwitch (ATOM_TCB *old_tcb_ptr, ATOM_TCB *new_tcb_ptr)

.global archContextSwitch
archContextSwitch:

    /**
     * Parameter locations:
     *  old_tcb_ptr = R25-R24
     *  new_tcb_ptr = R23-R22
     */


    /**
     * Save registers R2-R17, R28-R29.
     */
    push r2
    ………………………
    push r29
     // save RAMPZ and EIND
    in r0,_SFR_IO_ADDR(RAMPZ)
    push r0
    in r0,_SFR_IO_ADDR(EIND)
    push r0

    // Save the final stack pointer to the TCB.
    in  r16,_SFR_IO_ADDR(SPL)
    in  r17,_SFR_IO_ADDR(SPH)
    mov r28,r24
    mov r29,r25
    st  Y,r16
    std Y+1,r17
```

```
//get SP from  new TCB
mov r28,r22
mov r29,r23
ld r16,Y
ldd r17,Y+1

 // switch stack
out _SFR_IO_ADDR(SPL),r16
out _SFR_IO_ADDR(SPH),r17

// restore status
pop r0
in r0,_SFR_IO_ADDR(EIND)
pop r0
in r0,_SFR_IO_ADDR(RAMPZ)
pop r29
…
pop r2
ret
```

First Thread Restore

Is just the bottom part of the context switch

```
 void archFirstThreadRestore (ATOM_TCB *new_tcb_ptr)
 */
.global archFirstThreadRestore

archFirstThreadRestore:

    /**
     * Parameter locations:
     *  new_tcb_ptr = R25-R24
     */

    //get SP from  new TCB
    mov r28,r24
    mov r29,r25
    ld r16,Y
    ldd r17,Y+1

     // switch stack
    out _SFR_IO_ADDR(SPL),r16
    out _SFR_IO_ADDR(SPH),r17

    // restore status
    pop r0
    in r0,_SFR_IO_ADDR(EIND)
    pop r0
    in r0,_SFR_IO_ADDR(RAMPZ)
    pop r29
    …
    pop r2
    ret
```

Schedule

The final schduler consists of:

- The current process, and the head of a list of thread control blocks
- Two functions:
  - startSchedule (initializes timers, and gives control to first thread)
  - schedule (called in the timer interrupt), that switches context

```
TCB* current_tcb=NULL;

// the running queue
TCBList running_queue={
  .first=NULL,
  .last=NULL,
  .size=0
};


void startSchedule(void){
  cli();
  current_tcb=TCBList_dequeue(&running_queue);
  assert(current_tcb);
  timerStart();
  archFirstThreadRestore(current_tcb);
}

void schedule(void) {
  TCB* old_tcb=current_tcb;
  // we put back the current thread in the queue
  TCBList_enqueue(&running_queue, current_tcb);

  // we fetch the next;
  current_tcb=TCBList_dequeue(&running_queue);
  // we jump to it
  //(useless if it is the only process)
  if (old_tcb!=current_tcb)
    archContextSwitch(old_tcb, current_tcb);
}
```

Run, baby run

```
TCB idle_tcb;
uint8_t idle_stack[IDLE_STACK_SIZE];
void idle_fn(uint32_t thread_arg){
  while(1) {
    cli();
    printf("i\n");
    sei();
    _delay_ms(10);
  }
}

TCB p1_tcb;
uint8_t p1_stack[THREAD_STACK_SIZE];
void p1_fn(uint32_t arg ){
  while(1){
    cli();
    printf("p1\n");
    sei();
    _delay_ms(10);
  }
}

TCB p2_tcb;
uint8_t p2_stack[THREAD_STACK_SIZE];
void p2_fn(uint32_t arg ){
  while(1){
    cli();
    printf("p2\n");
    sei();
    _delay_ms(10);
  }
}
```

```
int main(void){
  // we need printf for debugging
  printf_init();

  TCB_create(&idle_tcb,
             idle_stack+IDLE_STACK_SIZE-1,
             idle_fn,
             0);

  TCB_create(&p1_tcb,
             p1_stack+THREAD_STACK_SIZE-1,
             p1_fn,
             0);

  TCB_create(&p2_tcb,
             p2_stack+THREAD_STACK_SIZE-1,
             p2_fn,
             0);


  TCBList_enqueue(&running_queue, &p1_tcb);
  TCBList_enqueue(&running_queue, &p2_tcb);
  TCBList_enqueue(&running_queue, &idle_tcb);

  printf("starting\n");
  startSchedule();
}
```