SUPSI

Lab: The Bthread library

Operating Systems

Amos Brocco, Lecturer & Researcher

14 settembre 2018

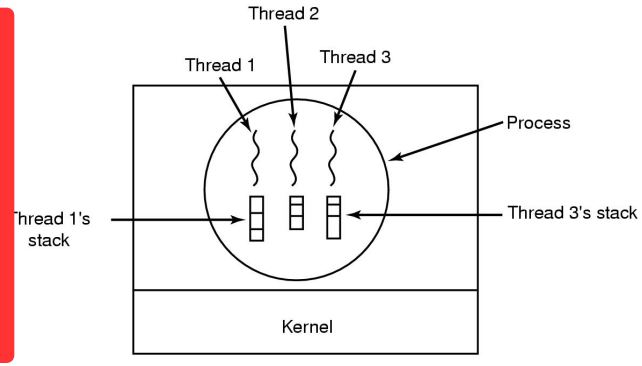
Objectives

- Understand the goal of the Bthread library
- Discover how to implement user-level threads

The active part of a process: threads

- A process can have one or more **threads** (or **paths**) **of execution** *
  - Threads in a process share some resources (→ concurrency problems)

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

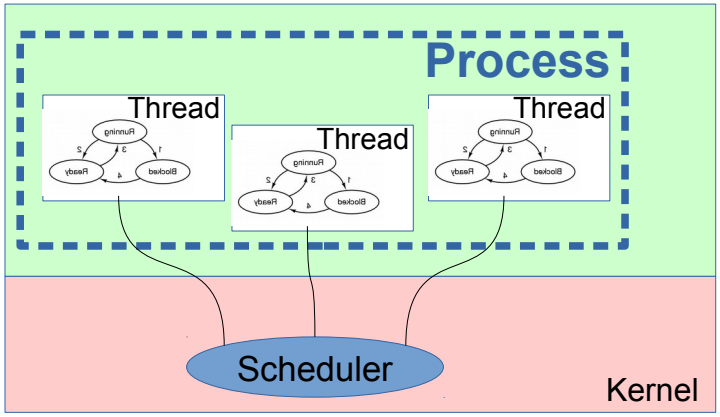


A. Tanenbaum, Modern Operating Systems, 2nd ed

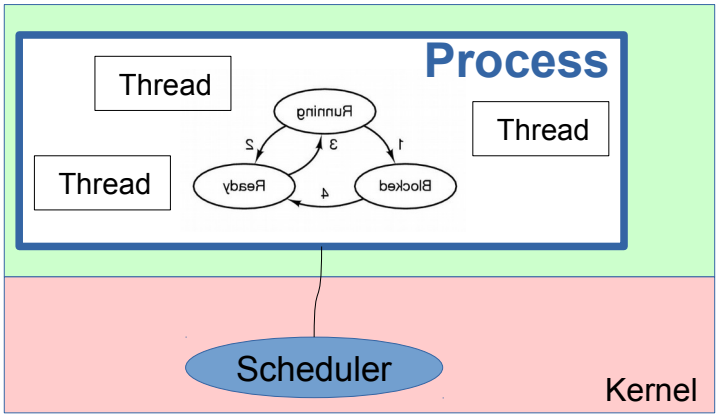
  - When a process has multiple threads of execution we call it a **multi-threaded process**, otherwise it is called a **single-threaded process**

* typically simply referred to as **threads**

Threads implementation



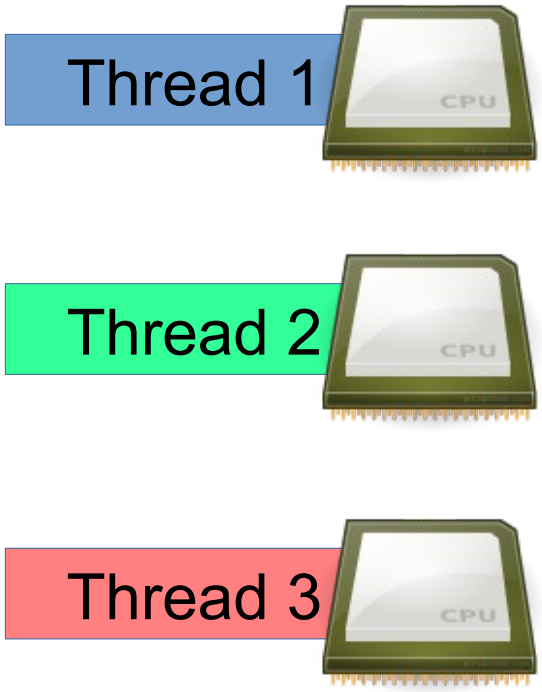
- **Kernel level threads**
  - Thread scheduling is done by the kernel
  - If a thread blocks, other threads within the same process can continue executing



- **User level threads**
  - Thread scheduling is done by the process
  - If a thread blocks, the whole process (including other user threads) is blocked

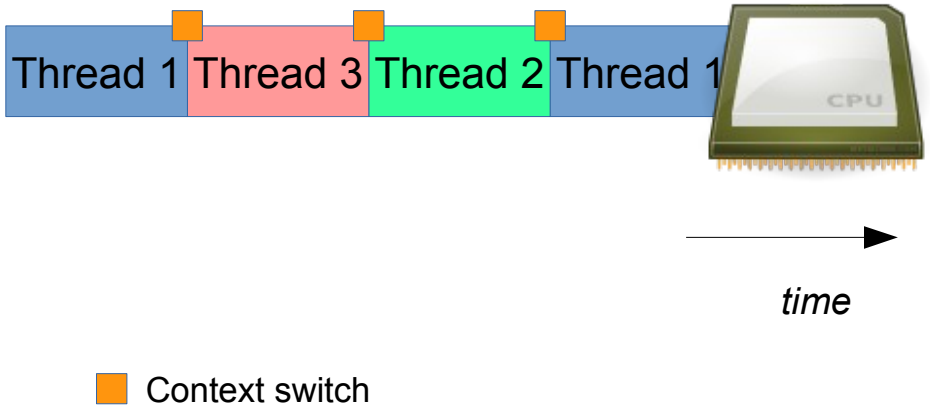
Execution

- **# threads <= # cpu cores**



Parallel execution

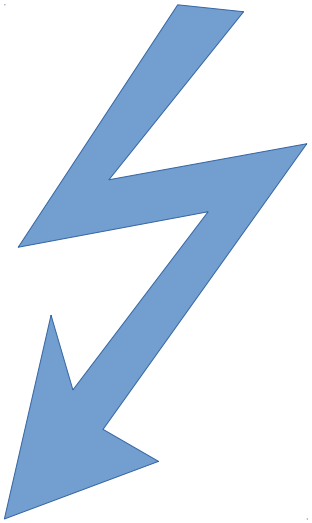
- **# threads > # cpu cores**



Pseudo-parallel execution

Implementation

```
void thread1() {
        // do something
}

void thread2() {
        // do something
}

void main() {
        while(1) {
                thread1();
                thread2();
        }
}
```



Implementation

```
void thread1() {
        for(int i=0; i<10000; i++) {
                // Do some computation
                yield();
        }
}

void thread2() {
        for(int i=0; i<10000; i++) {
                // Do some computation
                yield();
        }
}

void main() {
        while(1) {
                thread1();
                thread2();
        }
}
```

Yield

```
void thread1() {
        for(int i=0; i<10000; i++) {
                // Do some computation
                yield();
        }
}

void thread2() {
        for(int i=0; i<10000; i++) {
                // Do some computation
                yield();
        }
}

void main() {
        while(1) {
                thread1();
                thread2();
        }
}
```

Save context

Restore context

Yield

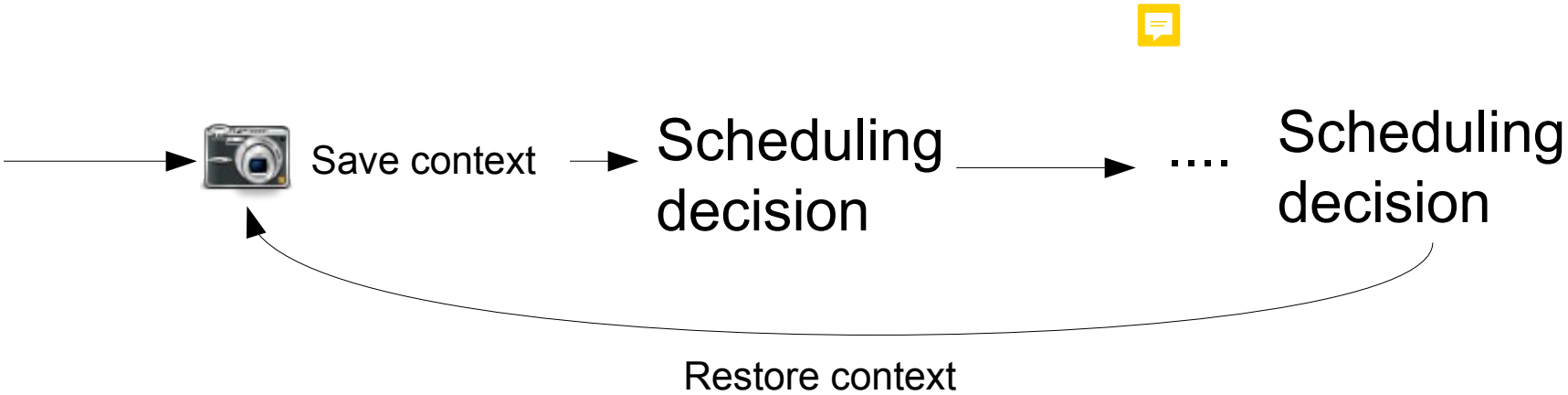
```
void thread1() {
        for(int i=0; i<10000; i++) {
                // Do some computation
                yield();
        }
}

void thread2() {
        for(int i=0; i<10000; i++) {
                // Do some computation
                yield();
        }
}

void main() {
        while(1) {
                thread1();
                thread2();
        }
}
```

Restore context

Save context

Scheduling threads

- To move to another process/thread we must first **save the execution context** of the current process/thread, in order to be able to **restore it later**
  - It's like taking a snapshot of the execution context



What's the context?

- Register values
  - General registers
  - Base pointer (**“Where are local variables?”**)
  - Stack pointer (**“Where is the top of the stack?”**)
  - Program counter (**“Where are we in the code?”**)

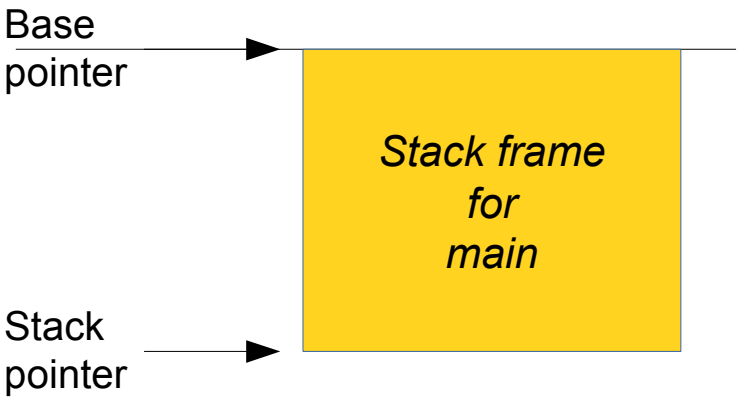
Longjmp, setjmp

```
#include <setjmp.h>

// Save context
int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);

// Restore context
void longjmp(jmp_buf env, int val);
void siglongjmp(sigjmp_buf env, int val);
```

Example 3.2.1

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void dothings() {
   printf("Now I'm here\n");
   sleep(3);
   longjmp(buf, 42);
   printf("This is never printed\n");
}

int main() {
    if (!setjmp(buf)) { // the first time returns 0
        dothings();
    } else {
        printf("Now I'm there\n");
    }
    return 0;
}
```

Example 3.2.1

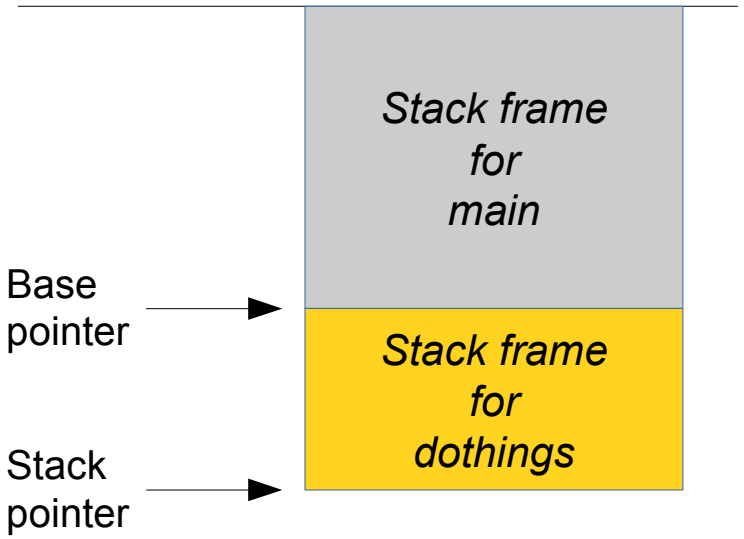



```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void dothings() {
   printf("Now I'm here\n");
   sleep(3);
   longjmp(buf, 42);
   printf("This is never printed\n");
}

int main() {
    if (!setjmp(buf)) {
        dothings();
    } else {
        printf("Now I'm there\n");
    }
    return 0;
}
```

Saves the context into 'buf', returns 0

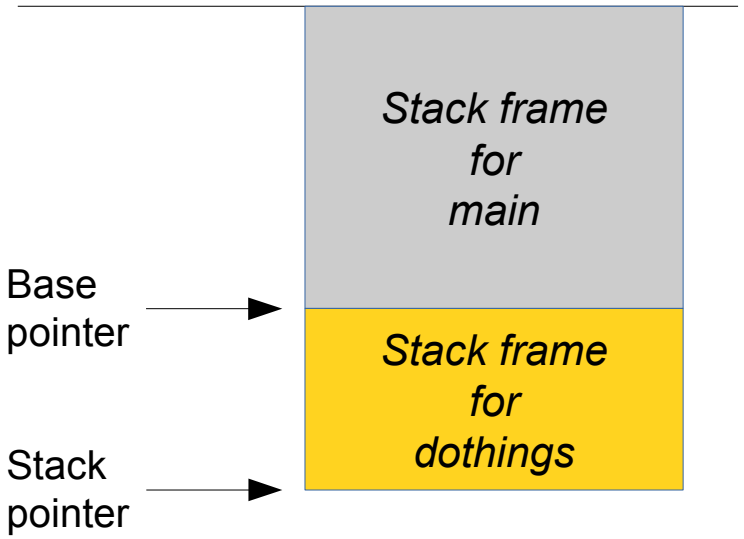
Example 3.2.1

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void dothings() {
    printf("Now I'm here\n");
    sleep(3);
    longjmp(buf, 42);
    printf("This is never printed\n");
}

int main() {
    if (!setjmp(buf)) { // the first time returns 0
        dothings();
    } else {
        printf("Now I'm there\n");
    }
    return 0;
}
```



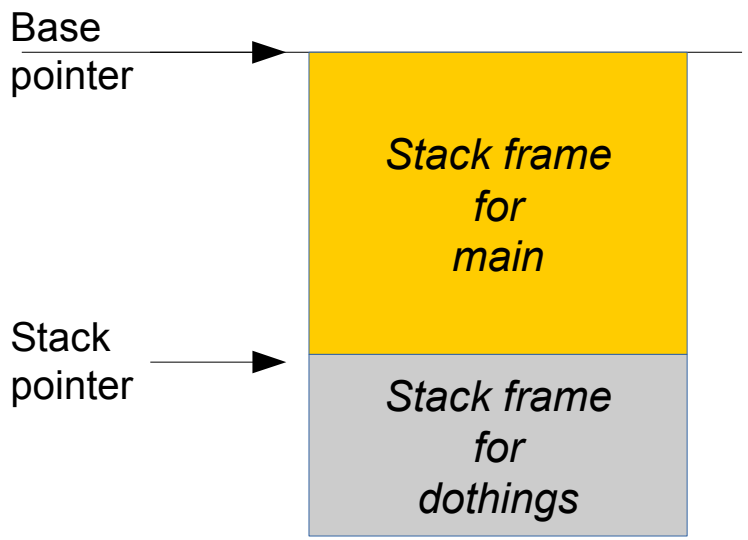
Example 3.2.1

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void dothings() {
    printf("Now I'm here\n");
    sleep(3);
    longjmp(buf, 42);
    printf("This is never printed\n");
}

int main() {
    if (!setjmp(buf)) { // the first time returns 0
        dothings();
    } else {
        printf("Now I'm there\n");
    }
    return 0;
}
```



Example 3.2.1

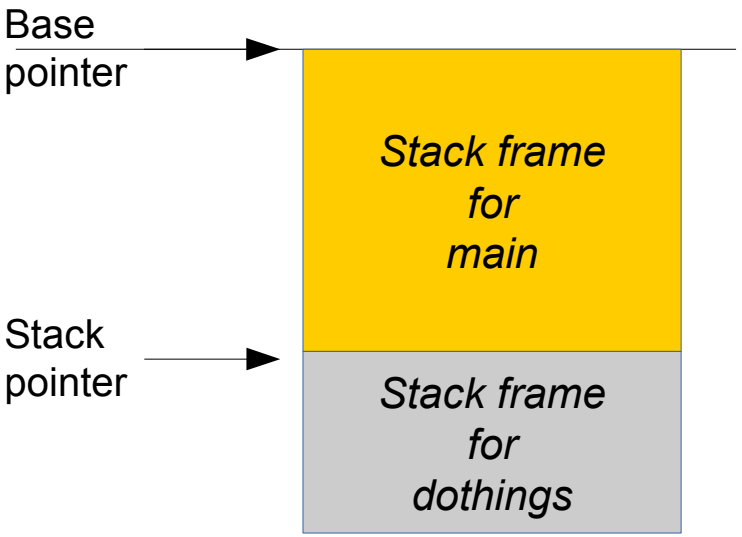
```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void dothings() {
   printf("Now I'm here\n");
   sleep(3);
   longjmp(buf, 42);
   printf("This is never p
}

int main() {
    if (!setjmp(buf)) { // the first time returns 0
        dothings();
    } else {
        printf("Now I'm there\n");
    }
    return 0;
}
```



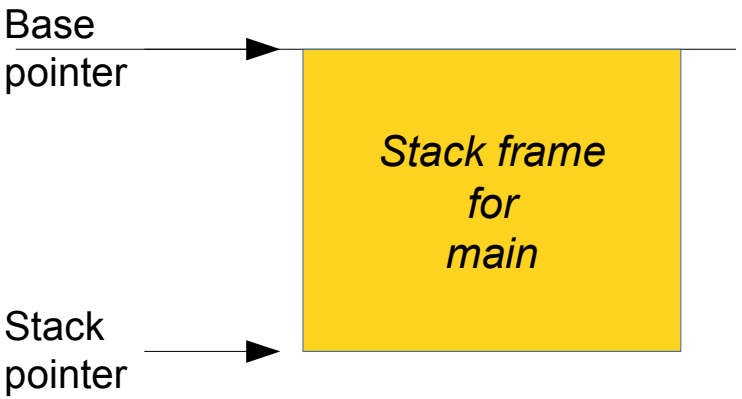
Restores the context from 'buf', tells 'setjmp' to return 42

Example 3.2.1

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void dothings() {
   printf("Now I'm here\n");
   sleep(3);
   longjmp(buf, 42);
   printf("This is never printed\n");
}

int main() {
    if (!setjmp(buf)) { // the first time returns 0
        dothings();
    } else {
        printf("Now I'm there\n");
    }
    return 0;
}
```



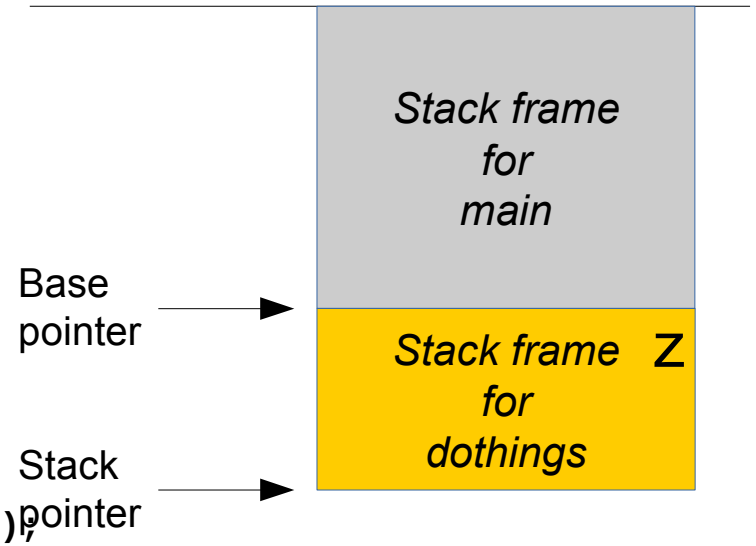
Example 3.2.2




```c
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

static jmp_buf main_buf, dothings_buf;

void dothings() {
        int z = 1313;
        if(!setjmp(dothings_buf)) {
                printf("Now I'm here, z=%d\n", z);
                longjmp(main_buf, 42);
        } else {
                printf("Now I'm back here, z=%d\n", z);
                exit(0);
        }
}

int main() {
    if (!setjmp(main_buf)) {
        dothings();
    } else {
        longjmp(dothings_buf, 17);
    }
    return 0;
}
```

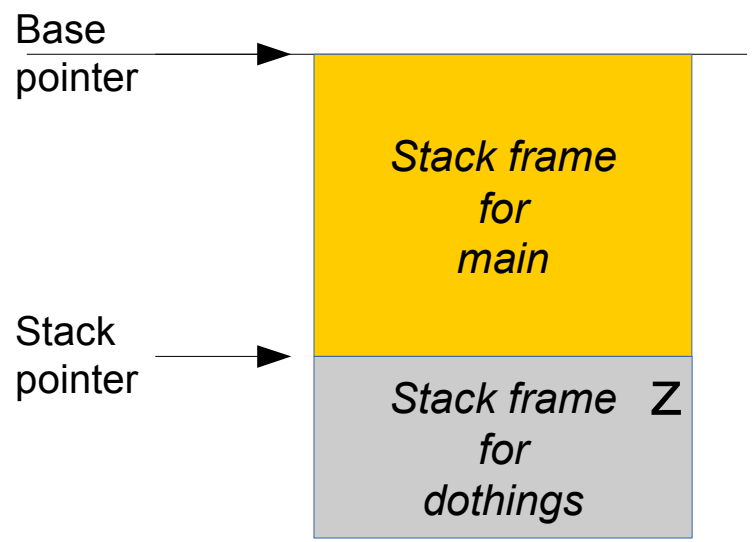
Example 3.2.2

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

static jmp_buf main_buf, dothings_buf;

void dothings() {
        int z = 1313;
        if(!setjmp(dothings_buf)) {
                printf("Now I'm here, z=%d\n", z);
                longjmp(main_buf, 42);
        } else {
                printf("Now I'm back here, z=%d\n", z);
                exit(0);
        }
}

int main() {
    if (!setjmp(main_buf)) {
        dothings();
    } else {
        longjmp(dothings_buf, 17);
    }
    return 0;
}
```



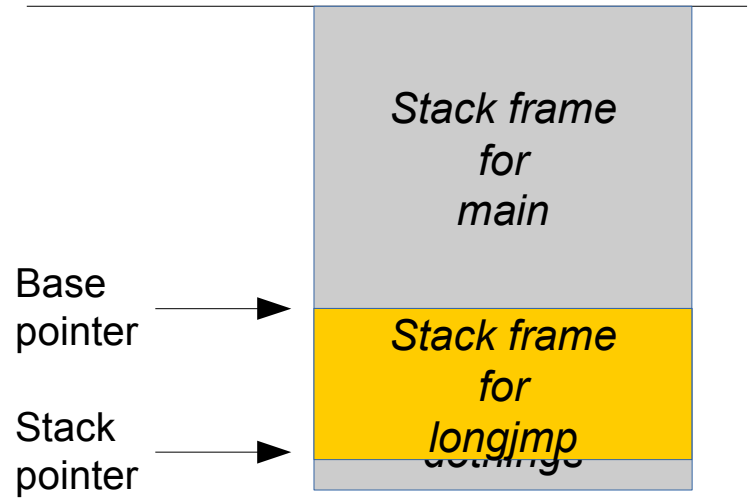
Example 3.2.2

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

static jmp_buf main_buf, dothings_buf;

void dothings() {
        int z = 1313;
        if(!setjmp(dothings_buf)) {
                printf("Now I'm here, z=%d\n", z);
                longjmp(main_buf, 42);
        } else {
                printf("Now I'm back here, z=%d\n", z);
                exit(0);
        }
}

int main() {
    if (!setjmp(main_buf)) {
        dothings();
    } else {
        longjmp(dothings_buf, 17);
    }
    return 0;
}
```



Example 3.2.2 (stack smashing)

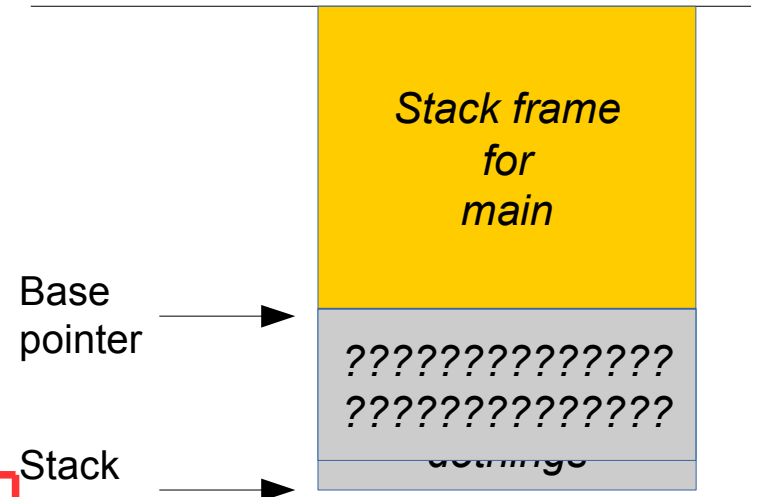
```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

static jmp_buf main_buf, dothings_buf;

void dothings() {
        int z = 1313;
        if(!setjmp(dothings_buf)) {
                printf("Now I'm here, z=%d\n", z);
                longjmp(main_buf, 42);
        } else {
                printf("Now I'm back here, z=%d\n", z);
                exit(0);
        }
}

int main() {
    if (!setjmp(main_buf)) {
        dothings();
    } else {
        longjmp(dothings_buf, 17);
    }
    return 0;
}
```



longjmp is also a function!

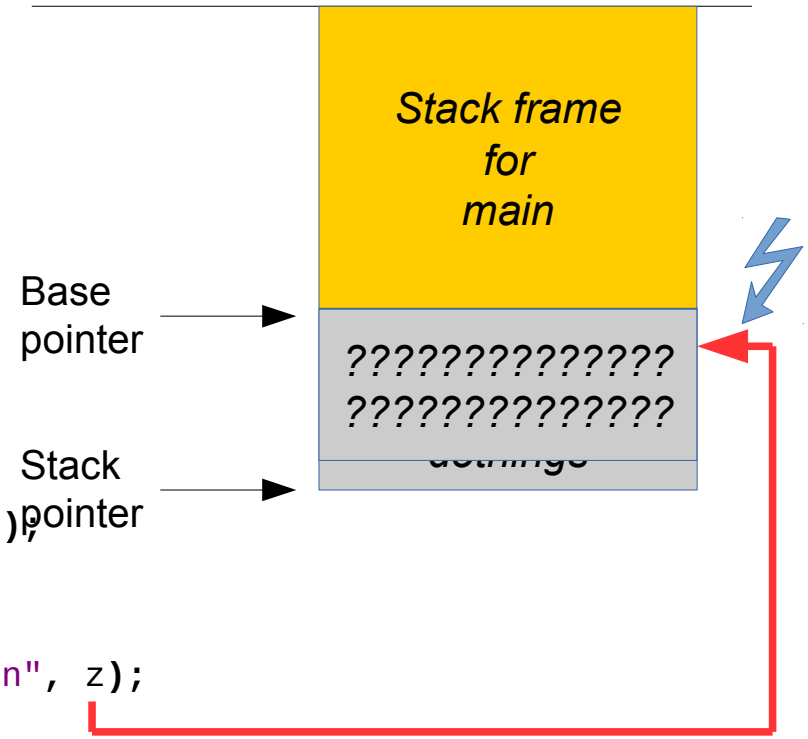
Example 3.2.2

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

static jmp_buf main_buf, dothings_buf;

void dothings() {
        int z = 1313;
        if(!setjmp(dothings_buf)) {
                printf("Now I'm here, z=%d\n", z);
                longjmp(main_buf, 42);
        } else {
                printf("Now I'm back here, z=%d\n", z);
                exit(0);
        }
}

int main() {
    if (!setjmp(main_buf)) {
        dothings();
    } else {
        longjmp(dothings_buf, 17);
    }
    return 0;
}
```



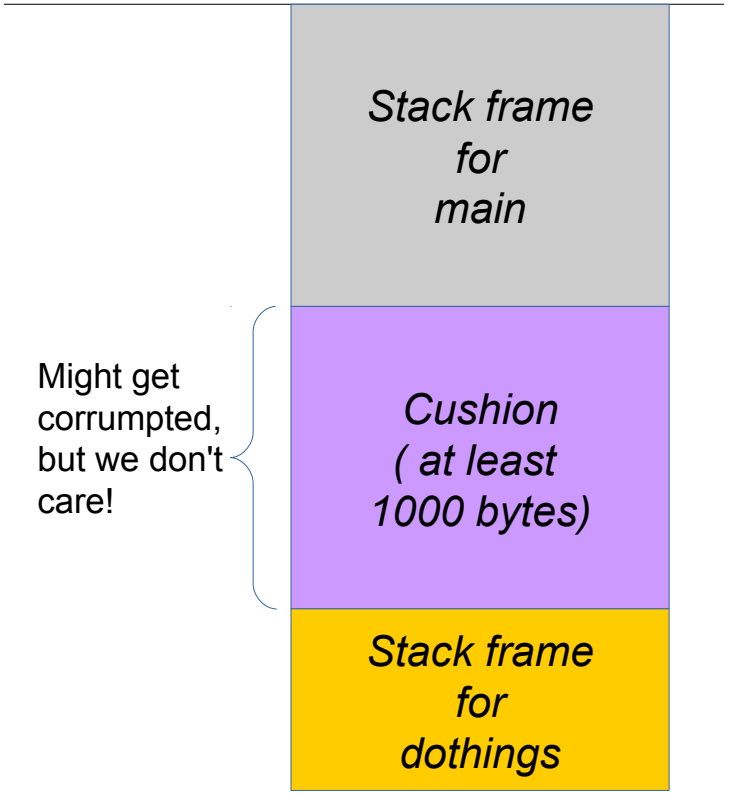
Example 3.2.2

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

static jmp_buf main_buf, dothings_buf;

void dothings() {
        int z = 1313;
        if(!setjmp(dothings_buf)) {
                printf("Now I'm here, z=%d\n", z);
                longjmp(main_buf, 42);
        } else {
                printf("Now I'm back here, z=%d\n", z);
                exit(0);
        }
}

int main() {
    if (!setjmp(main_buf)) {
        dothings();
    } else {
        longjmp(dothings_buf, 17);
    }
    return 0;
}
```



Cushion

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>
static jmp_buf main_buf, dothings_buf;
void dothings() {
        int z = 1313;
        if(!setjmp(dothings_buf)) {
                printf("Now I'm here, z=%d\n", z);
                longjmp(main_buf, 42);
        } else {
            printf("Now I'm back here, z=%d\n", z);
            exit(0);
        }
}
void cushion() {
        char data_cushion[1000];
        data_cushion[999] = 1;
        dothings();
}
int main() {
    if (!setjmp(main_buf)) {
        cushion();
    } else {
        longjmp(dothings_buf, 17);
    }
    return 0;
}
```



Cooperative threads example (4)

```
#include <setjmp.h>
#include <stdio.h>

#define CUSHION_SIZE 10000
#define save_context(CONTEXT) setjmp(CONTEXT)
#define restore_context(CONTEXT) longjmp(CONTEXT, 1)

typedef enum { __BTHREAD_UNINITIALIZED, __BTHREAD_READY } bthread_state;
typedef void *(*bthread_routine) (void *);

void create_cushion_and_call(bthread_routine fn, bthread_state* state);
void* bthread1(void* arg);
void* bthread2(void* arg);
void* bthread3(void* arg);

jmp_buf bthread1_buf, bthread2_buf, bthread3_buf;
bthread_state bthread1_state = __BTHREAD_UNINITIALIZED;
bthread_state bthread2_state = __BTHREAD_UNINITIALIZED;
bthread_state bthread3_state = __BTHREAD_UNINITIALIZED;

void create_cushion_and_call(bthread_routine fn, bthread_state* state)
{
    char cushion[CUSHION_SIZE];
    cushion[CUSHION_SIZE-1] = cushion[0];
    *state = __BTHREAD_READY;
    fn(NULL);
}
```

Cooperative threads example (4)

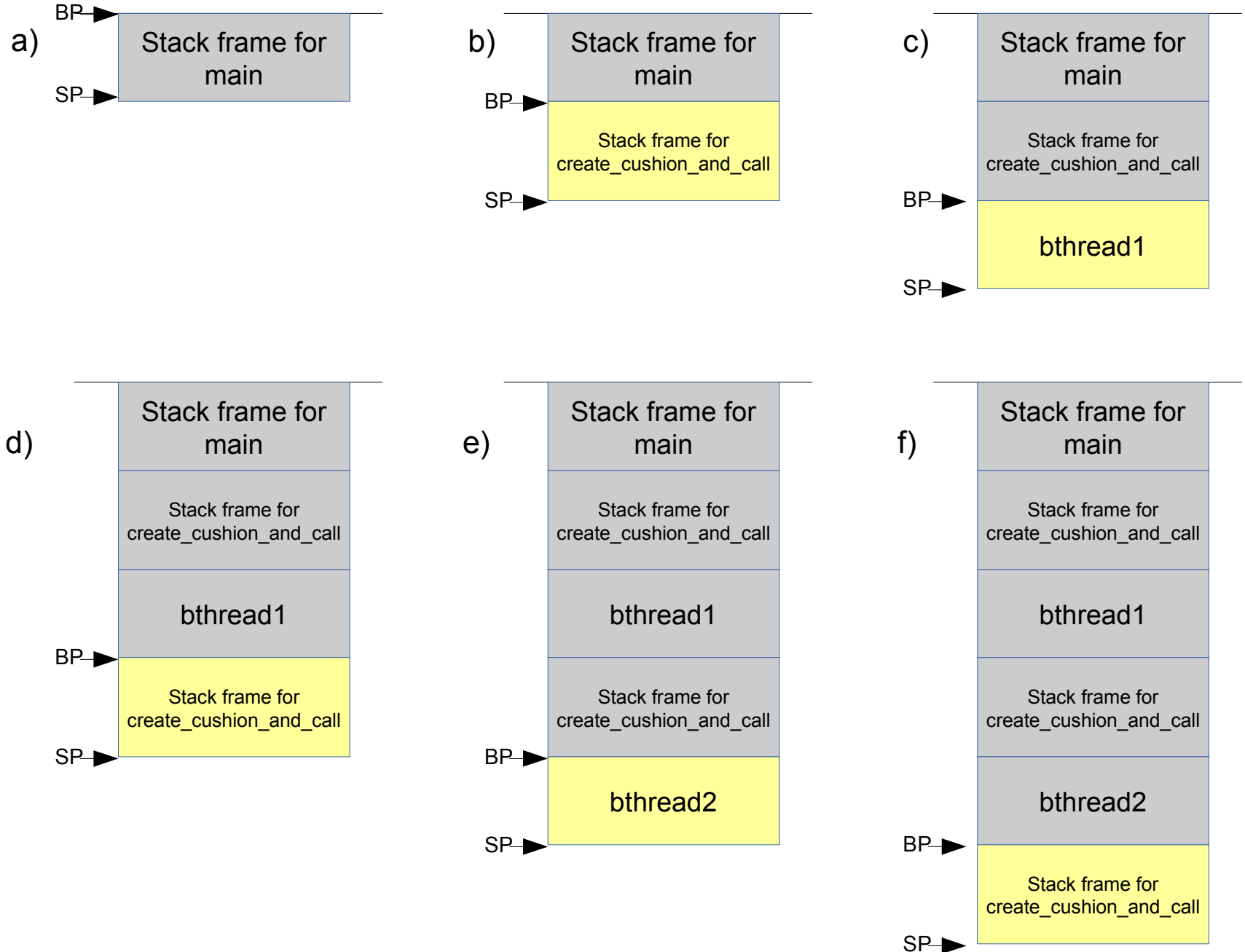
```
void* bthread1(void* arg)
{
    int i;
    for(i=0;i<10000;i++) {
        printf("BThread1, i=%d\n", i);
        /* Yield to next bthread */
        if (!save_context(bthread1_buf)) {
            if (bthread2_state == __BTHREAD_UNINITIALIZED) {
                create_cushion_and_call(bthread2, &bthread2_state);
            } else {
                restore_context(bthread2_buf);
            }
        }
    }
}

void* bthread2(void* arg)
{
    int i;
    for(i=0;i<10000;i++) {
        printf("BThread2, i=%d\n", i);
        /* Yield to next bthread */
        if (!save_context(bthread2_buf)) {
            if (bthread3_state == __BTHREAD_UNINITIALIZED) {
                create_cushion_and_call(bthread3, &bthread3_state);
            } else {
                restore_context(bthread3_buf);
            }
        }
    }
}
```

Cooperative threads example (4)

```
void* bthread3(void* arg)
{
    int i;
    for(i=0;i<10000;i++) {
        printf("BThread3, i=%d\n", i);
        /* Yield to next bthread */
        if (!save_context(bthread3_buf)) {
            // We assume that bthread1 is already
initialized
            restore_context(bthread1_buf, 1);
        }
    }
}

void main()
{
    create_cushion_and_call(bthread1, &bthread1_state);
}
```

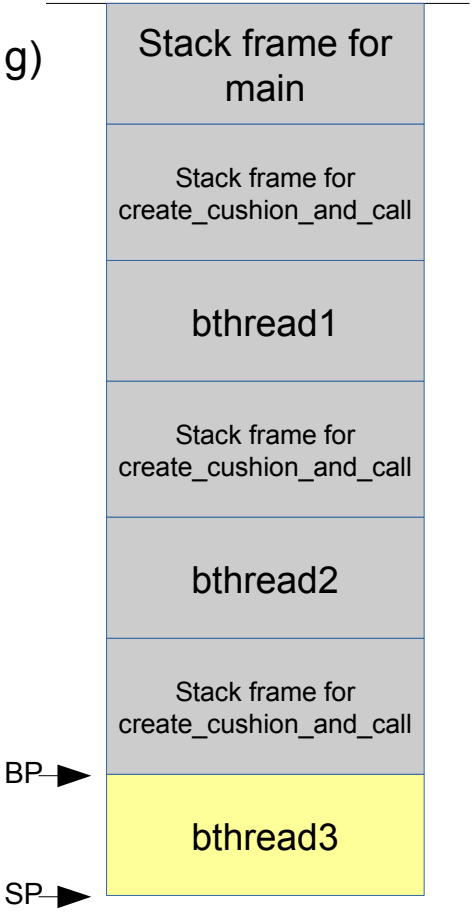
Stack evolution

BP = Base Pointer, SP = Stack Pointer



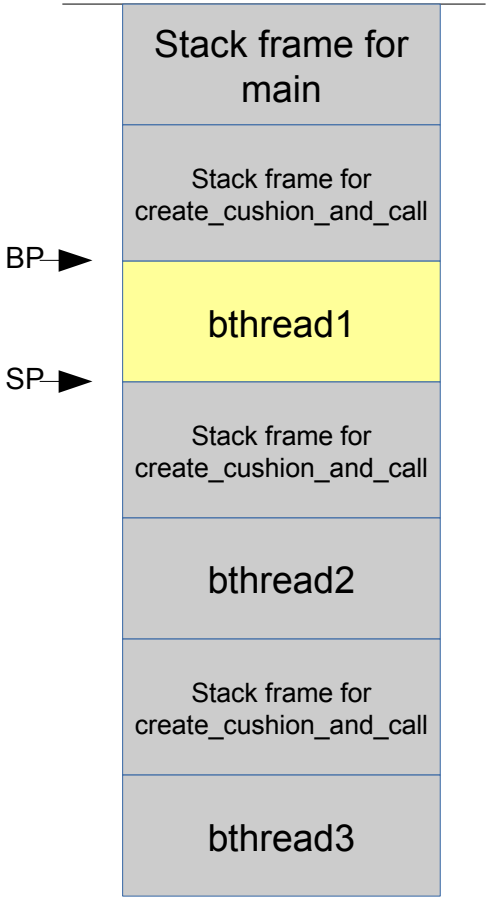
Stack evolution

g)



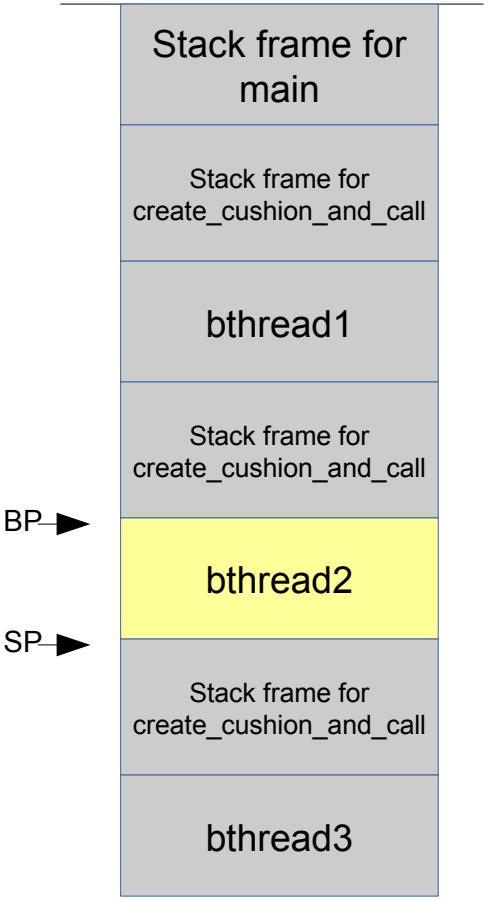
Context change

Switching (yield) to bthread1



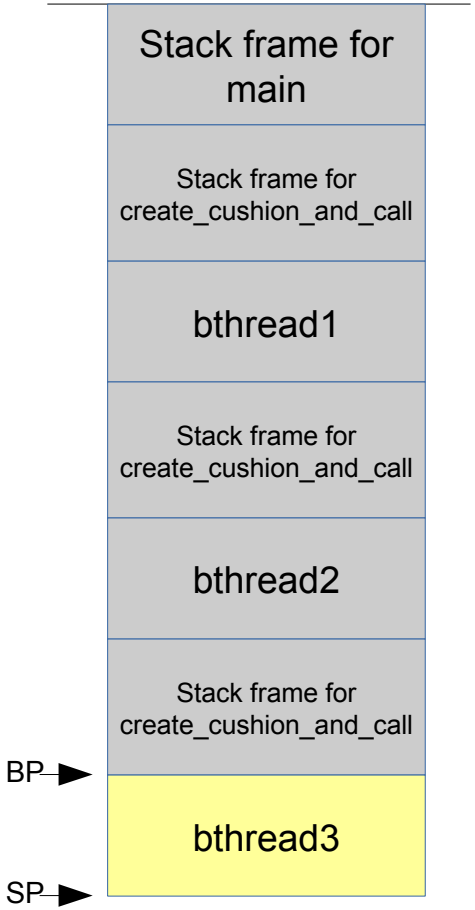
Context change

Switching (yield) to bthread2



Context change

Switching (yield) to bthread3



Exercise (2)

Mandatory

- **Study, compile and test the proposed example code;**
- **Try to add another thread and verify that your code executes as expected.**

Optional (but recommended):

- What happens if a thread exits the for loop earlier than the others?
- How could you solve this problem?
- How could you implement a simple priority mechanism where one receives more CPU time than the other threads?

… continue reading the tutorial from Section 4.1 !