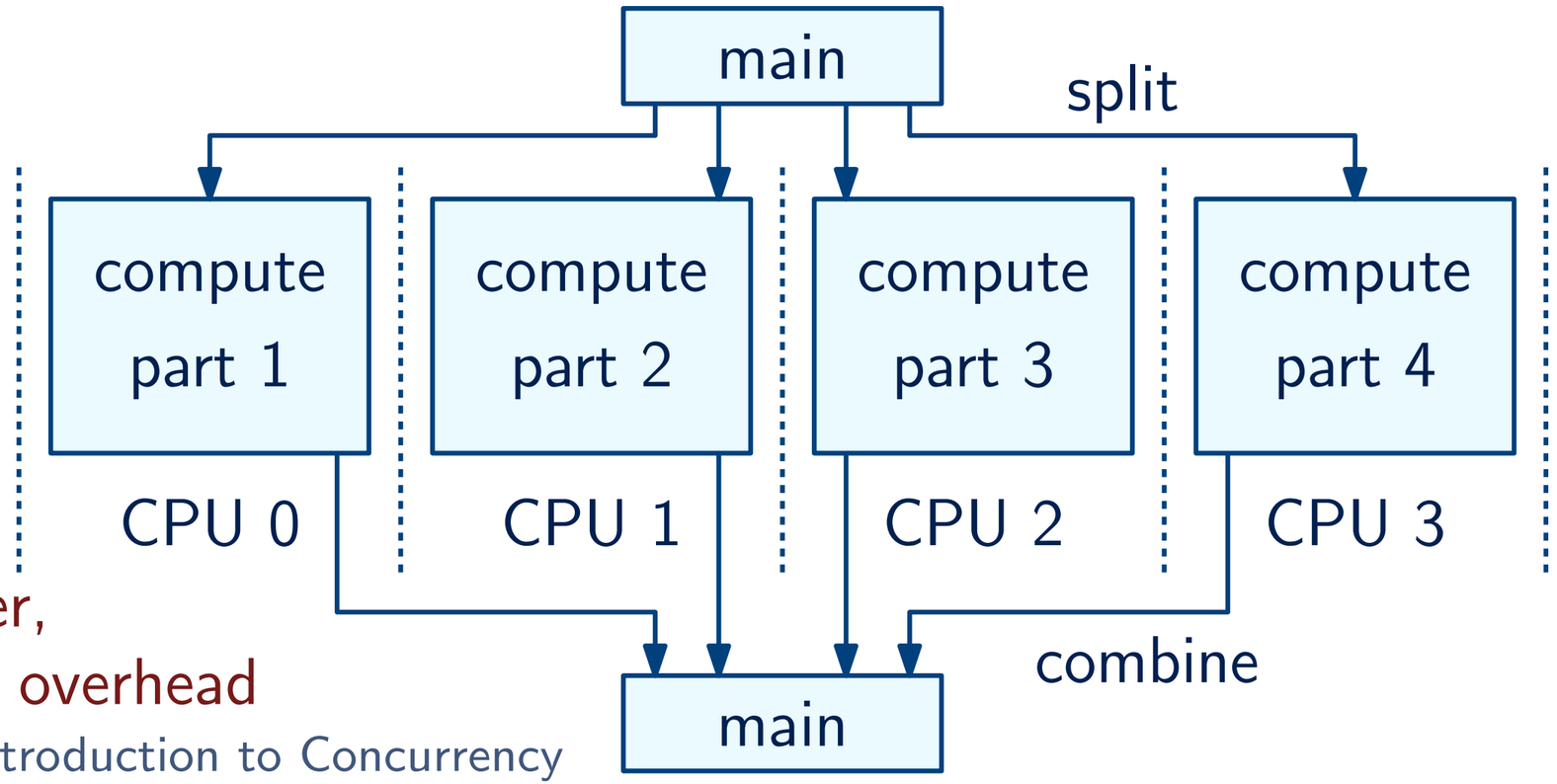
INF113: Introduction to Concurrency

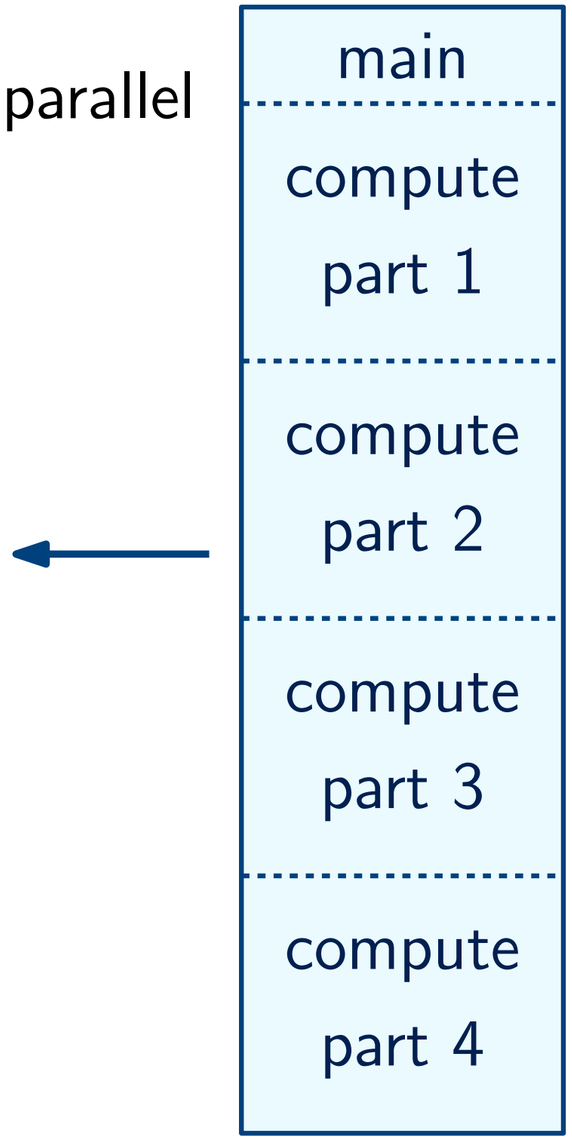
Kirill Simonov

15.10.2025

Parallel computation

- A program may split computation into smaller pieces to run in parallel
- Each "subprocess" gets a piece
- Examples: encoding a video, reverting a hash, solving TSP...

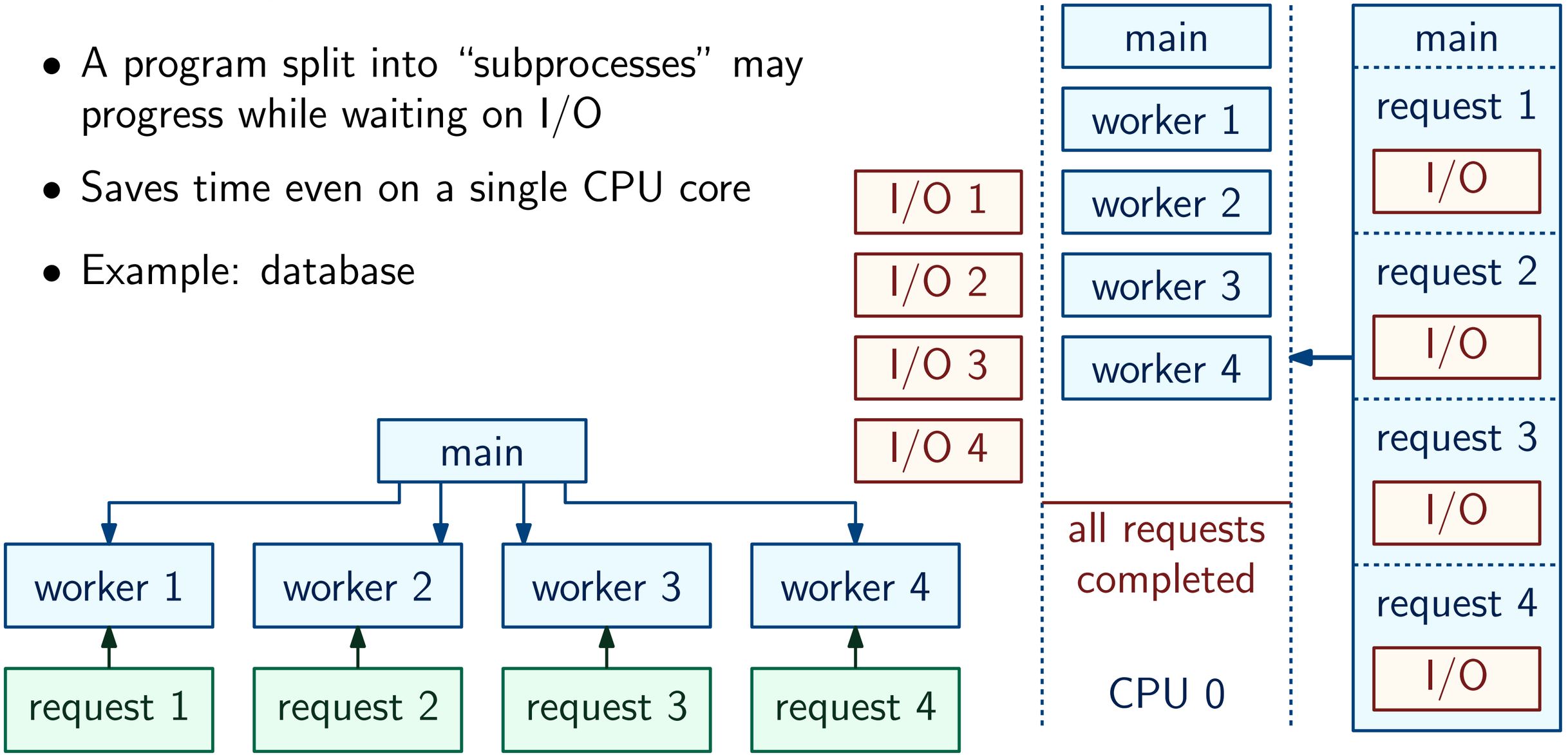




4x faster,
modulo overhead

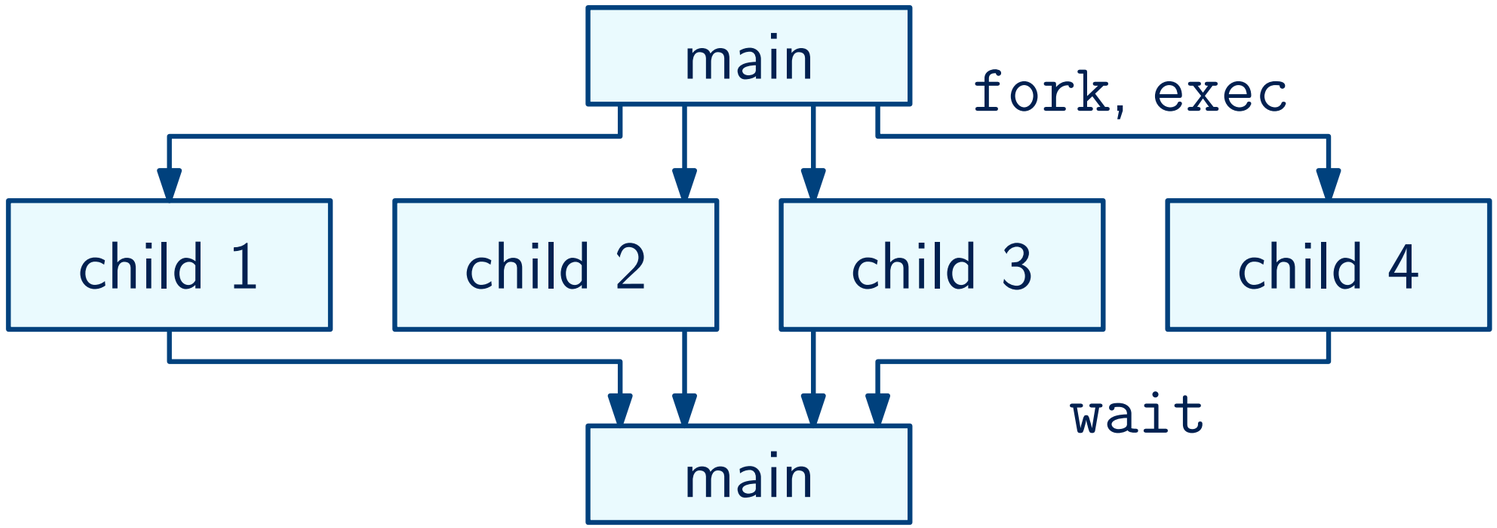
Servicing requests

- A program split into "subprocesses" may progress while waiting on I/O
- Saves time even on a single CPU core
- Example: database



Child processes

- We could implement parallelization by spawning child processes
  - "Prime" each child with any data
  - Run independently, also in parallel
  - Blocked independently, e.g., with I/O
  - Wait for completion if needed
- Issue: almost no interaction between the processes
  - Possible via files, but very slow

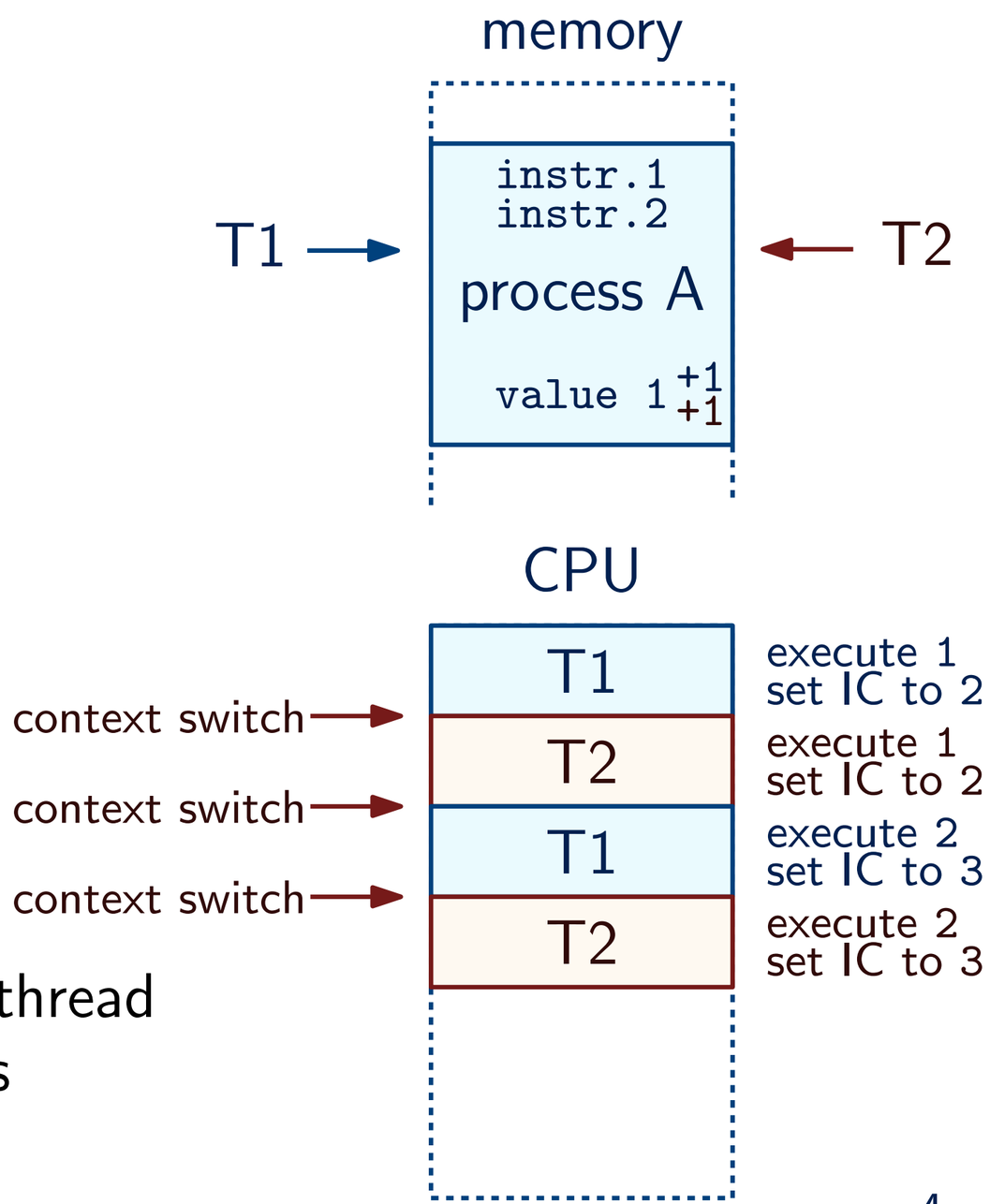



```
for (int i = 0; i < 4; ++i) {
  int rc = fork();
  if (rc == 0) {
    //preparation
    //code
    exec();
  }
}

while (wait() > 0) {};
//parent
//continues
```

Threads

- "Subprocesses" need to have shared memory
- **Thread** is a point of execution within a process
  - Its own value of Instruction Counter
  - And other registers
  - But no own memory
- Each process may have multiple threads
- OS schedules threads independently
- On a context switch, OS saves the state of the thread
  - No need to flush TLB if still the same process



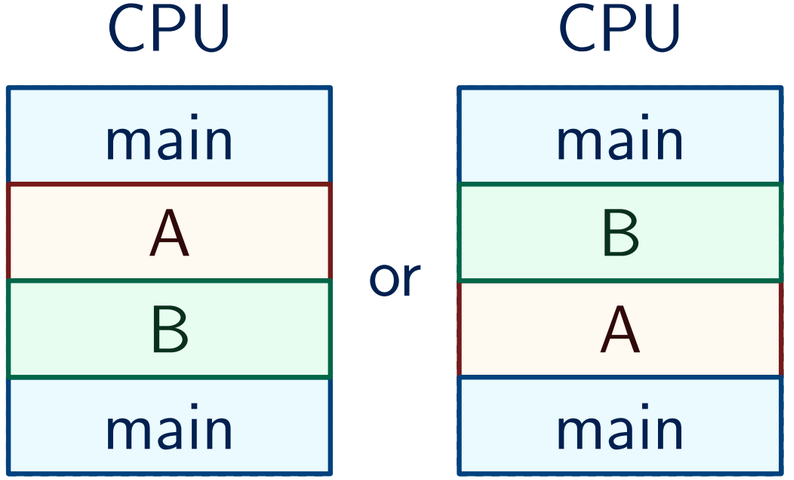
Example in C

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

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main() {
    pthread_t p1, p2;
    printf("main: begin\n");
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

- `pthread_create` starts a new thread with a given entry point
- `pthread_join` waits for a given thread
- `void*` allows to pass/return any data
- order is not fixed:



C tricks: Wrapping arguments

```
...
typedef struct {
    int arg1, arg2;
} args_t;

void *mythread(void *arg) {
    args_t *a = (args_t *) arg;
    printf("%d %d\n", a->arg1, a->arg2);
    return NULL;
}

int main() {
...
    args_t a = {3, 2};
    pthread_create(&p, NULL, mythread, &a);
...
}
```

- Any collection of arguments can be arranged in a `struct` and passed as a `void*` pointer

C tricks: Wrapping arguments

```
void *mythread(void *arg) {
    args_t *a = (args_t *) arg;
    args_t *ret = malloc(sizeof(args_t));
    ret->arg1 = a->arg1 + a->arg2;
    ret->arg2 = a->arg1 - a->arg2;
    return ret;
}

int main() {
...
    void *ret;
    pthread_join(p, &ret);
    args_t *aret = ret;
    printf("main: %d %d\n",
         aret->arg1, aret->arg2);
...
}
```

- Any collection of arguments can be arranged in a `struct` and passed as a `void*` pointer
- Same for return values

C tricks: Wrapping arguments

```
...
void *mythread(void *arg) {
    int a = *(int *)arg;
    int b = a + 1;
    return &b;
}

int main() {
    pthread_t p;
    printf("main: begin\n");
    int a = 3;
    pthread_create(&p, NULL, mythread, &a);
    void *ret;
    pthread_join(p, &ret);
    printf("main: %d\n", *(int *)ret);
    printf("main: end\n");
    return 0;
}
```

pointer to b will not be valid!

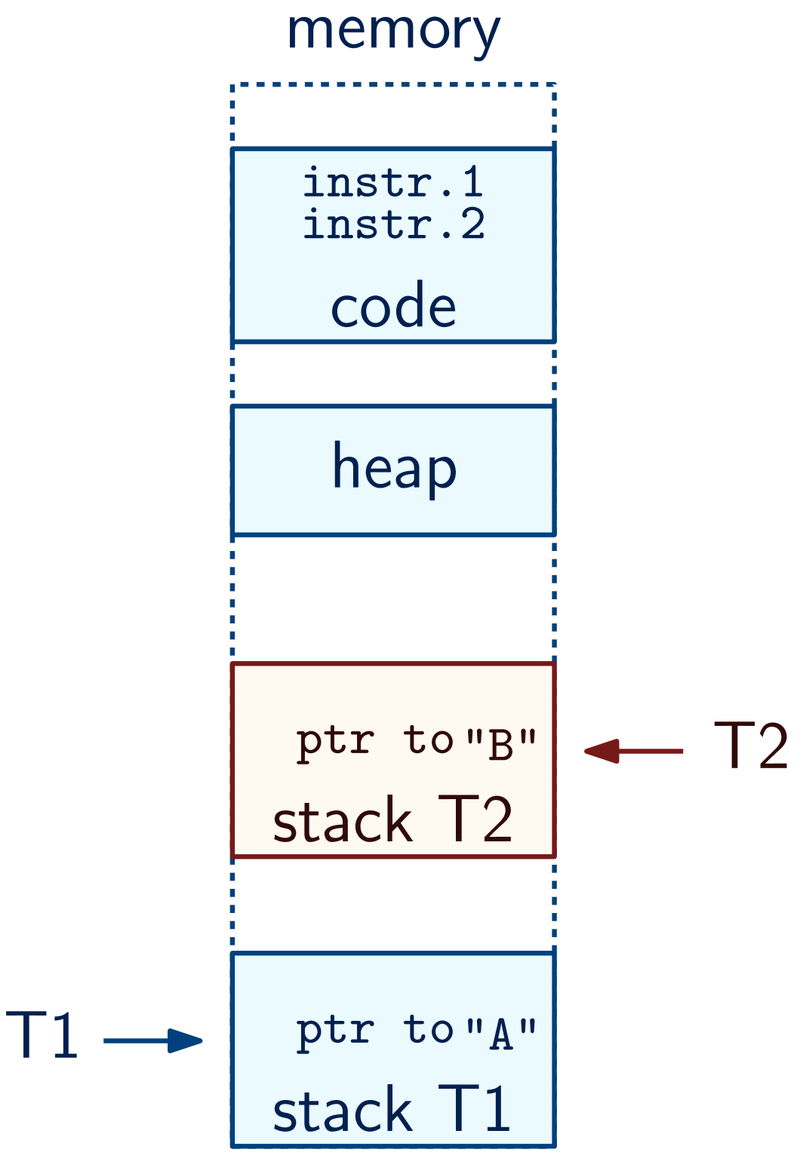
- Any collection of arguments can be arranged in a `struct` and passed as a `void*` pointer
- Same for return values
- **Beware:** Variables allocated on stack get invalidated!

Threads and stack

- If threads possess no memory, how the args are different?

```
...
void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}
...
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
...
```

- Actually, each thread gets its own place for stack
  - Hence, local variables are “private” for the thread
  - But not protected, since threads share the same address space

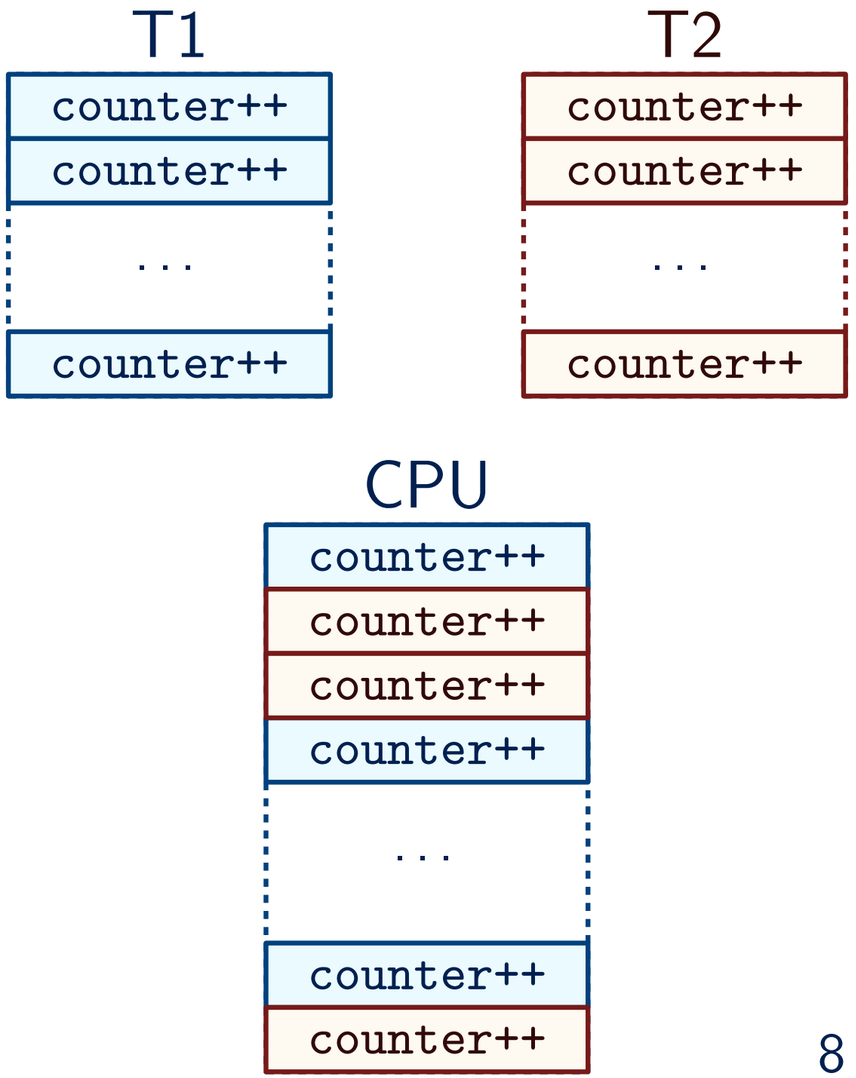


Example 2

- If two threads modify the same counter, the result is very unpredictable—why?

```
...
volatile int counter = 0;

void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    for (int i = 0; i < 1e7; i++) {
      counter = counter + 1;
    }
    printf("%s: end\n", (char *) arg);
    return NULL;
}
...
```



- Seemingly, the order in which the threads increment the counter should not matter

Example 2: Explanation

- `counter = counter + 1` is not **atomic**!
- Context switch may happen **within** the operation

```
counter = counter + 1;
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

T1 → `mov 0x8049a1c, %eax`

T2 ← `mov %eax, 0x8049a1c`

CPU	counter	%eax in T1	%eax in T2
T1: mov 0x8049a1c, %eax	0	0	
T2: mov 0x8049a1c, %eax	0	0	0
T2: add $0x1, %eax	0	0	1
T2: mov %eax, 0x8049a1c	1	0	1
T2: mov 0x8049a1c, %eax	1	0	1
T2: add $0x1, %eax	1	0	2
T2: mov %eax, 0x8049a1c	2	0	2
T1: add $0x1, %eax	2	1	2
T1: mov %eax, 0x8049a1c	1	1	2
T2: mov 0x8049a1c, %eax	1	1	1
T2: add $0x1, %eax	1	1	2
T2: mov %eax, 0x8049a1c	2	1	2

after 1 iteration of T1 and 3 iterations of T2, counter is at 2!

Some terminology

- **Critical section** is a part of code accessing a **shared resource**
- **Race condition** occurs when multiple threads enter a critical section simultaneously
- A program that admits race conditions is **indeterminate**
- To avoid this, there must be **mutual exclusion**: if one thread enters the critical section, the others cannot

```
...
volatile int counter = 0;
     shared resource
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    for (int i = 0; i < 1e7; i++) {
      counter = counter + 1;
    }          critical section
    printf("%s: end\n", (char *) arg);
    return NULL;
}
...
```

Atomic instructions

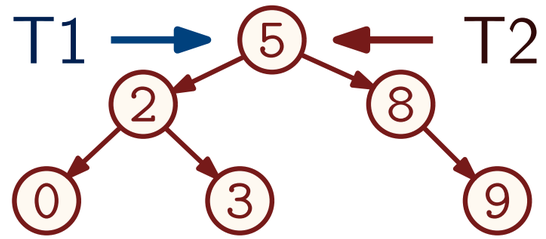
- We could hope that hardware implements common modifications **atomically**
- I.e., there is a single instruction instead of several
- While some exist, it is not sufficient in general
- Example: Concurrently modify a BST
  - Hardware should support atomics on our specific node structure?

```
counter = counter + 1;
```

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```



```
mem-add 0x8049a1c, $0x1
```



T1

```
int search(node *v, int key) {
  if (v->key < key) {
    search(v->left, key);
  }
  ...
```

T2

```
int remove(node *v, int key) {
  if (v->key == key) {
    tree* sub = merge(v->left, v->right);
    free(v);
    return sub;
  }
  ...
```

Summary

- We aim to design a set of general **synchronization primitives** based on limited hardware instructions
- Allow to safely "block" parts of code when in use by a thread
- Allow to "wake" threads when the wait condition is fulfilled

```
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    for (int i = 0; i < 1e7; i++) {
      counter = counter + 1;
    }
    printf("%s: end\n", (char *) arg);
    return NULL;
}
```

Summary

- We aim to design a set of general **synchronization primitives** based on limited hardware instructions
- Allow to safely "block" parts of code when in use by a thread
- Allow to "wake" threads when the wait condition is fulfilled
- OS viewpoint: what support from hardware and the OS is needed to make concurrency in programs easy?
- OS viewpoint: OS is the original concurrent process
  – Write to disk, start/finish process, . . . , all in presense of interrupts