Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T16 Concurrency and Threads

Referência principal
Ch.26 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 17 de setembro de 2018

Arthur João Catto, PhD

2º semestre de 2018

Introduction to Concurrency

- We use the word concurrency to refer to multiple activities that can happen at the same time.
- Correctly managing concurrency is a key challenge for operating system developers and, more recently, has also become a concern for many application developers.
- From the programmer's perspective, it is much easier to think sequentially than to keep track of many simultaneous activities.

How can you write a correct program with dozens of events happening at once?

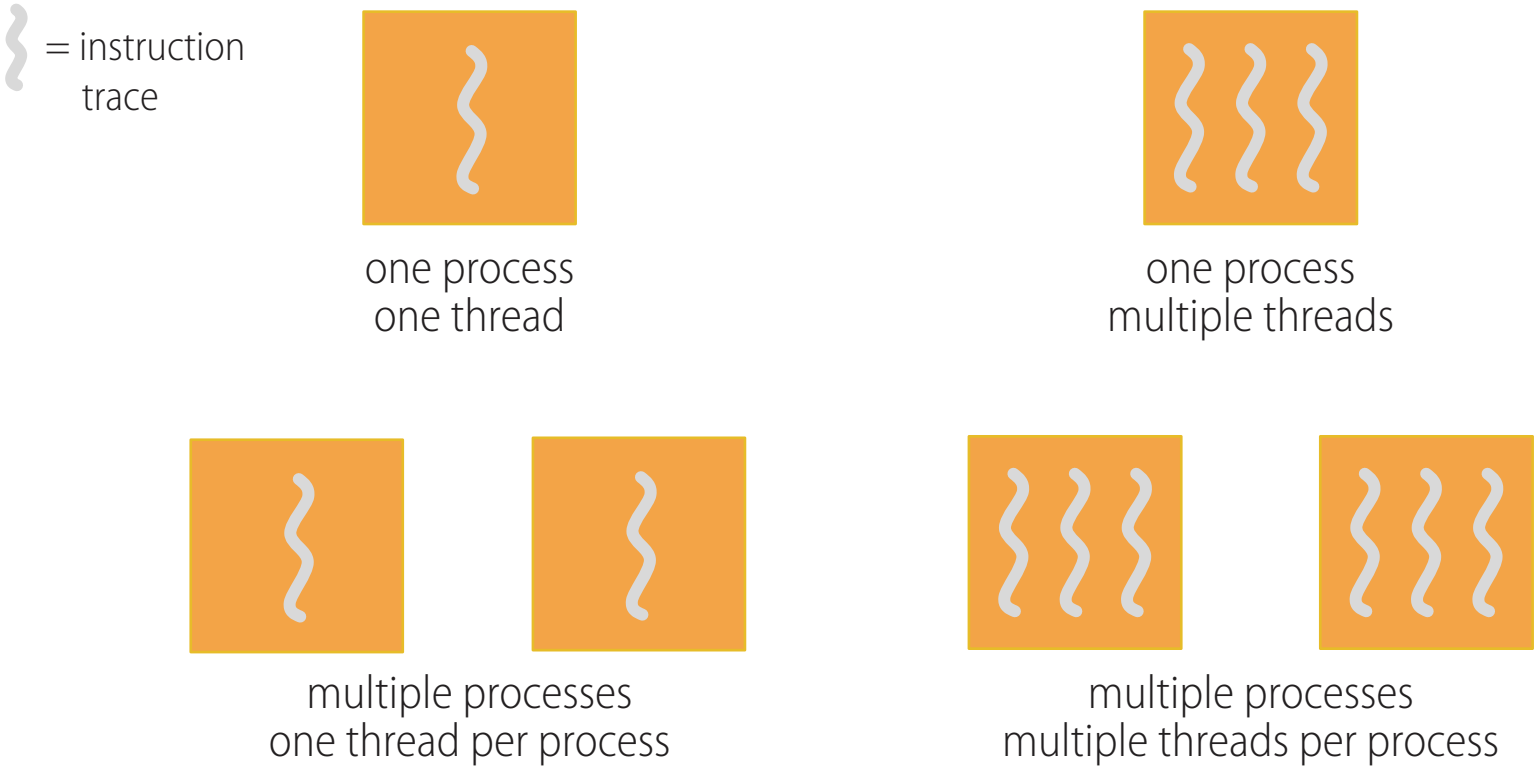
How do you debug such program if each execution is unique?

Concurrency within Applications

- We have already studied two abstractions
  - **Virtual CPUs** – enabling concurrency between multiple running programs (i.e. processes).
  - **Virtual Memory** – defining **address spaces** which enable each program to feel like having its own private memory.
- We will now introduce a new abstraction: **threads**
  - Threads behave nearly as separate processes, except for **one big difference**:
    - All the threads of a program share the same address space.
  - Threads enable concurrency within a single process.
  - A **multi-threaded program** will have more than one point of execution but only a single address space.

Processes and threads

- A traditional (heavyweight) process has a single thread of control
- A multi-threaded process can have more than one active task at a time



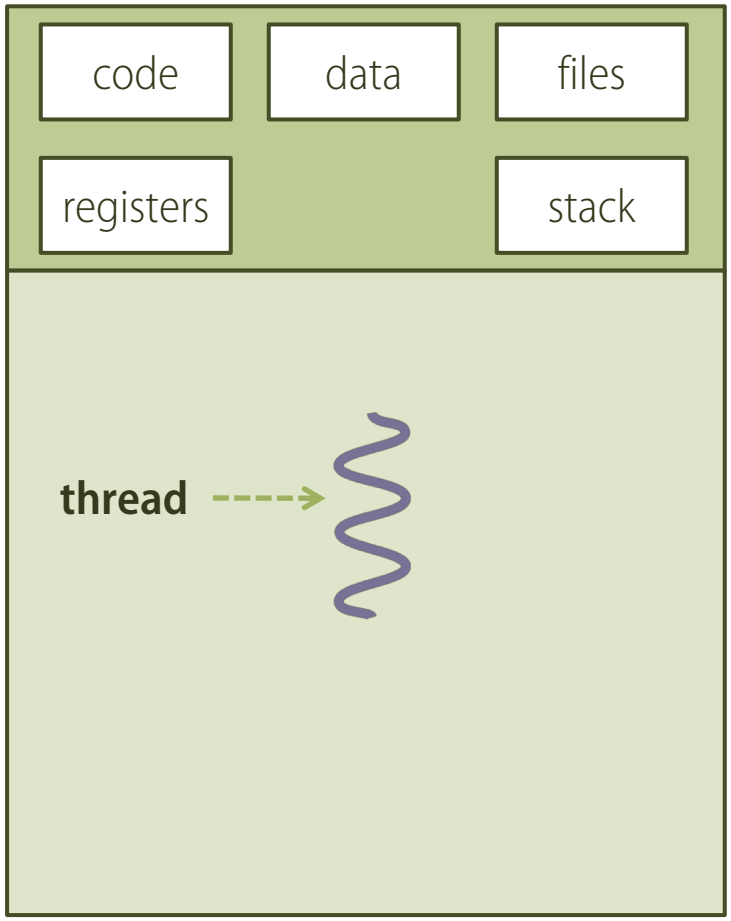
What's our goal?

- The key idea is to write a concurrent program as a set of sequential streams of execution, or **threads**, that interact and share results in a very precise way.
- **Threads** let us define a set of **sequential tasks that run concurrently**.
- We will now
  - define the thread abstraction
  - show how you can use it
  - explain how the OS can implement threads on top of a limited number of processors.

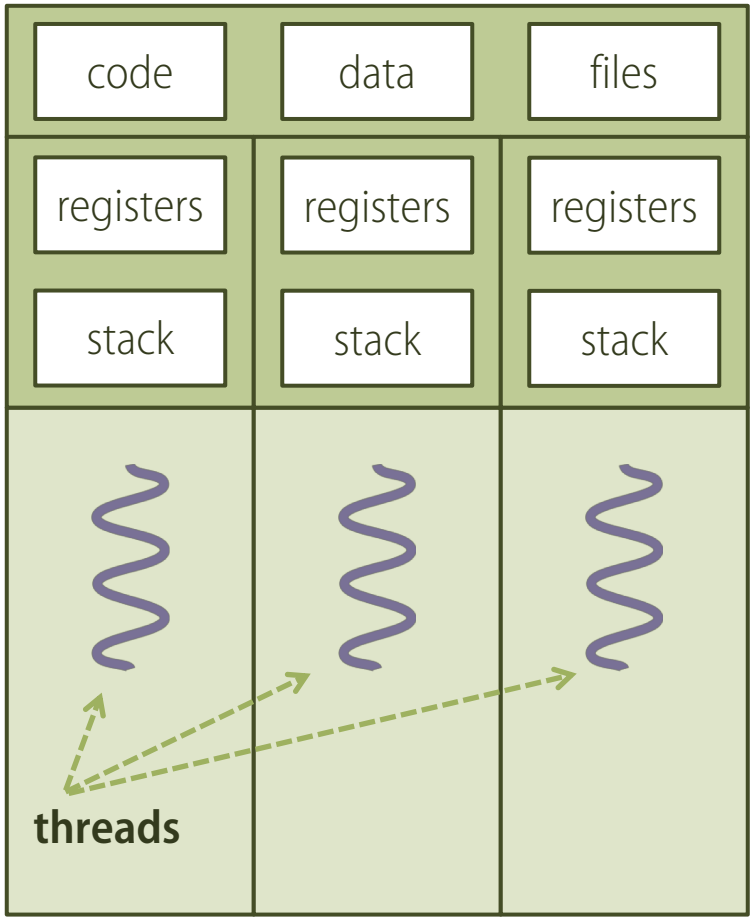
Thread Control Blocks (TCBs)

- Each thread has its own private set of registers it uses for computation
- If there are two threads that are running on a single processor, switching from running one to running the other, requires a context switch.
- The context switch between threads is quite similar to the context switch between processes, as the register state of the first must be saved and the register state of the second restored before running it.
- For processes, we saved state to a Process Control Block (PCB).
  For threads, we'll need one or more Thread Control Blocks (TCBs).

A process may have many threads

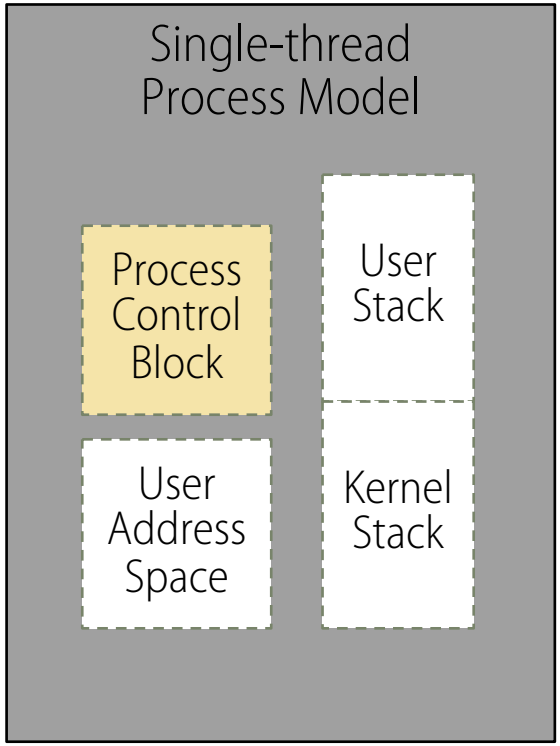


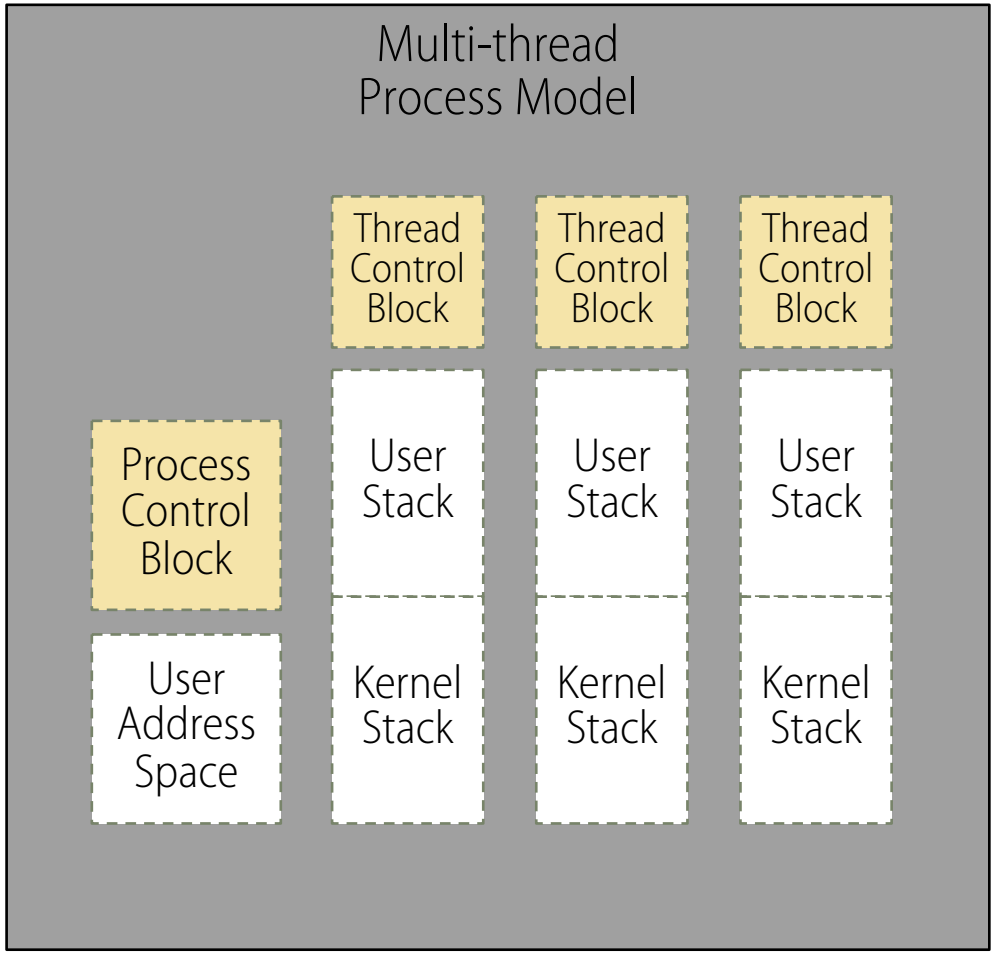
Single-threaded process



Multi-threaded process

Single-thread and multi-thread process models





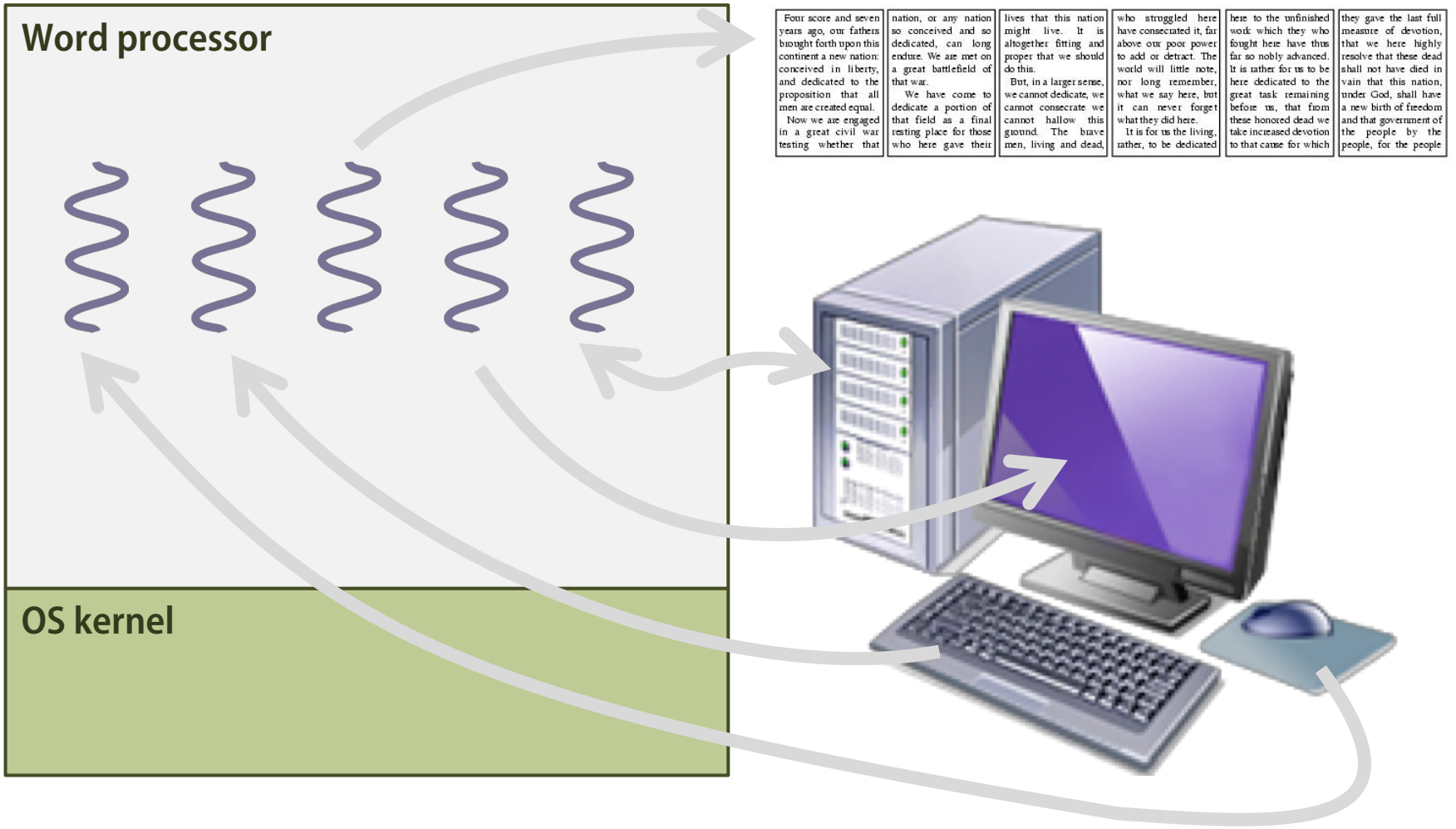
Examples of process and thread level items

Per process items (shared by all threads in a process)	Per thread items (private to each thread)
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Processes and threads

- An OS can be seen as a manager of
  - resources
  - time
- Early OSs treated both problems at process level.
- Modern OSs use a finer grain for time management:
  - The unit of **dispatching** is now a thread or lightweight process.
  - The unit of **resource** ownership still is a process or task.

A multithreaded word processor



A thread is a single execution sequence that represents a separately executable task.

Single execution sequence

- Each thread executes a sequence of instructions just as in the familiar sequential programming model.

Separately executable task

- The operating system can run, suspend or resume a thread at any time.

In summary, a thread . . .

- Has an execution state.
  - Running, ready, etc.
- Has an execution stack.
- Has some static storage for local variables.
- Has its context saved when not running.
- Has access to the memory and resources of its process.
  - These are shared by all threads of the process
- Is blocked when its process is blocked.
- Is terminated when its process terminates.

Why should you use threads at all?

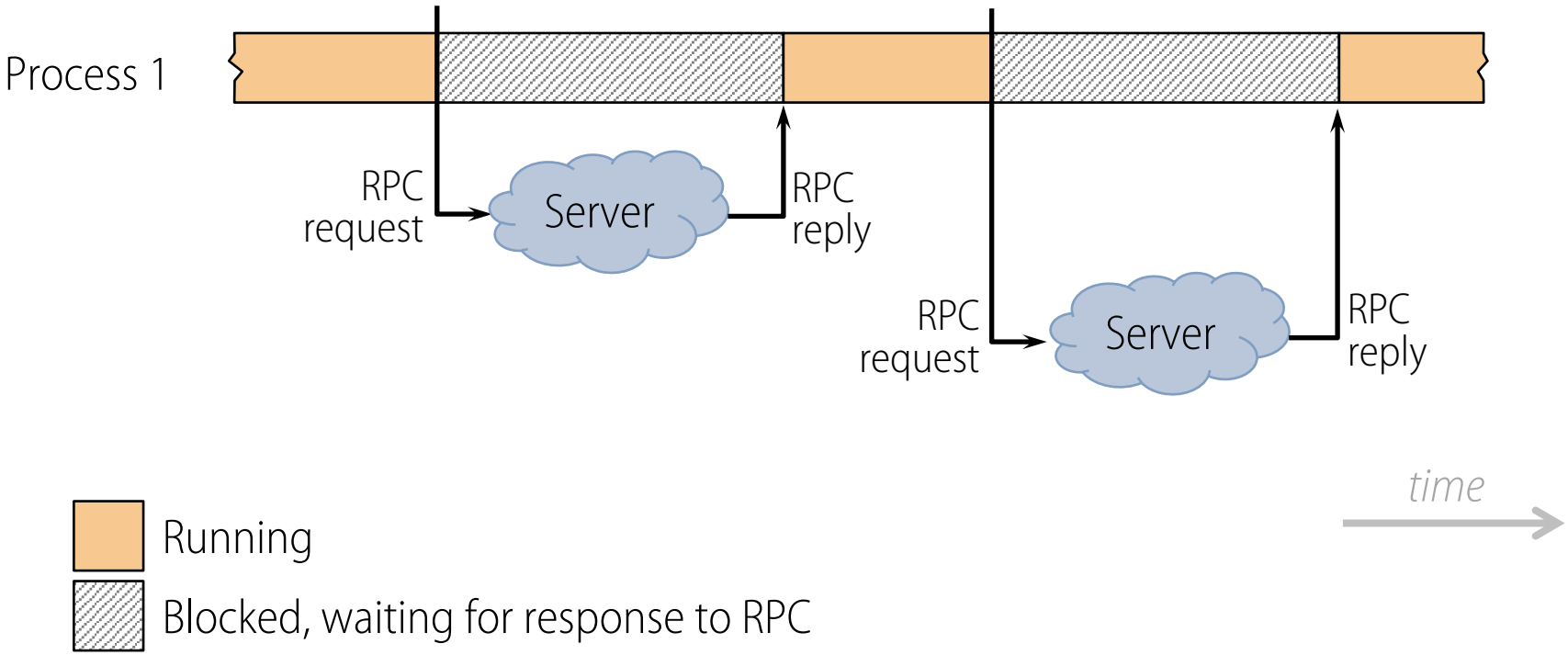
Exploiting parallelism	• E.g. by doing foreground and background work at the same time.
Asynchronous processing	• E.g. by implementing asynchronous elements of a program (power failure protection, backup, spell check, etc.) as independent threads.
Speeding execution	• E.g. by overlapping record reading and processing.
Modular program structure	• Programs involving different activities or data sources and destinations may be easier to design and implement using separate threads.

Thread benefits

Responsiveness	• A multithreaded process may run even if part of it is blocked or very busy.
Resource sharing	• The threads of a process share memory and files, so intercommunication may not require kernel involvement.
Performance	• It is faster to create a new thread than a process, to terminate a thread than a process and to switch between threads than between processes.
Scalability	• A multi-threaded process can benefit of a multiprocessor or multicore architecture.

Example

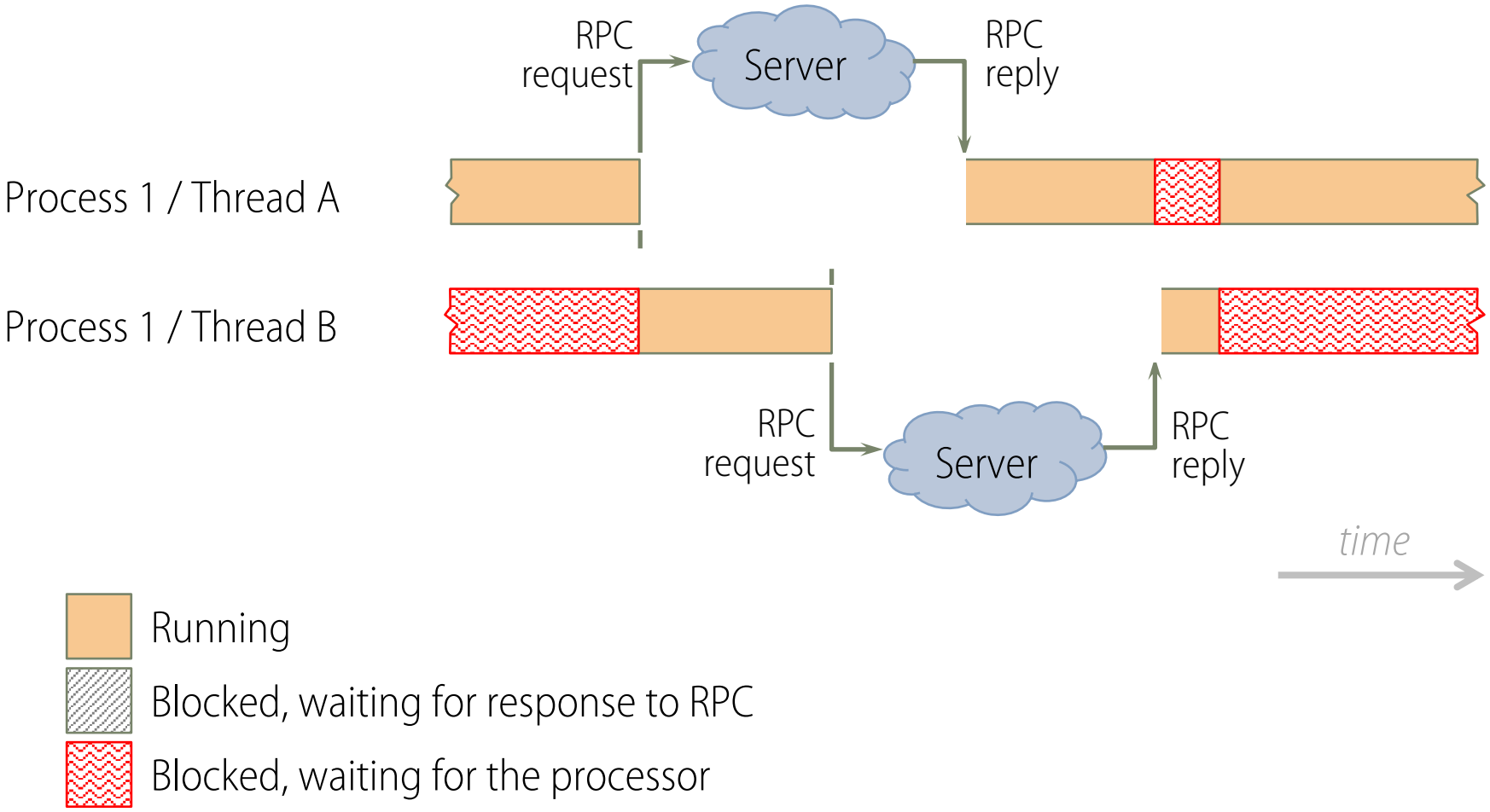
Synchronous RPC in a single thread uniprocessor



RPC ≡ Remote Procedure Call

Example

Synchronous RPC in a multi thread uniprocessor



Multicore programming challenges

Dividing activities	• Finding program blocks that can be run in parallel.
Achieving balance	• Adjusting block run duration and value.
Reducing data dependency	• Minimizing data dependencies between blocks. • Assuring proper synchronization of the unavoidable dependencies.
Testing and debugging	• Process traces are of much lower value than in single-core or single-processor environments due to potential nondeterminism

Example: Thread creation

- Let's design a 2-thread application so that one of the threads will print "A" while the other thread will print "B".
- We will use two functions from the **pthread** API of POSIX
  - Create a new thread

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *func, void *arg);
```

  - Wait for the termination of another thread

```
int pthread_join(pthread_t thread, void **thread_return);
```

- The POSIX API provides a way to invoke an asynchronous procedure call.
  - An asynchronous procedure call separates the call from the return.
    - The caller starts the function and continues executing concurrently with the called function.
    - Later, the caller can wait for function completion.

Before we go . . . let's wrap the pthread functions

- If you examine pthread's specification, you'll see that some of its functions return a success indicator.
- In order to avoid processing the success indicator at each point of call, we'll define wrapper functions, which will just call their similarly-named counterparts and make sure that they don't return any funny results.
- The "`mythreads.h`" header file will contain our wrapper functions, which differ from the original ones by having their names starting with a capital "P".

Our wrapper functions

```
1   #ifndef __MYTHREADS_h__
2   #define __MYTHREADS_h__
3
4   #include <pthread.h>
5   #include <assert.h>
6   #include <sched.h>
7
8   void Pthread_create(pthread_t *thread, const pthread_attr_t *attr,
9                       void *(*start_routine)(void*), void *arg) {
10      int rc = pthread_create(thread, attr, start_routine, arg);
11      assert(rc == 0);
12  }
13
14  void Pthread_join(pthread_t thread, void **value_ptr) {
15      int rc = pthread_join(thread, value_ptr);
16      assert(rc == 0);
17  }
18
19  #endif // __MYTHREADS_h__
```

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "mythreads.h"

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

```

```
10
11  int main(int argc, char *argv[]) {
12
13      pthread_t p1, p2;
14      printf("main : begin\n");
15      Pthread_create(&p1, NULL, mythread, "A");
16      Pthread_create(&p2, NULL, mythread, "B");
17
18      // join waits for the threads to finish
19      Pthread_join(p1, NULL);
20      Pthread_join(p2, NULL);
21
22      printf("main : end\n");
23      return 0;
24  }
```

```
SUP080:code arthur.catto$ gcc -o p0 p0.c -Wall -pthread
SUP080:code arthur.catto$ ./p0
main : begin
A
B
main : end
SUP080:code arthur.catto$
```

Is this the only possible output we could get?

Example(?)

Making things worse: sharing data

Design a 2-thread application so that

- Each thread performs the summation of the sequence of non-negative integers up to a given limit.
- The main function prints the accumulated result of the threads' calculation.

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

int max;
volatile int counter = 0; // shared global variable

void *mythread(void *arg) {
    char *letter = arg;
    int i;              // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: main-first <loopcount>\n");
        exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: begin [counter = %d] [%x]\n", counter,
           (unsigned int) &counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [counter: %8d]\n [should:  %8d]\n",
           counter, max*2);
    return 0;
}
```

```
SUP080:nb180917 arthur.catto$ gcc -o t1 t1.c -Wall -pthread
SUP080:nb180917 arthur.catto$ ./t1 10000000
main: begin [counter = 0] [dc2f060]
A: begin [addr of i: 0x700010038efc]
B: begin [addr of i: 0x7000100bbefc]
B: done
A: done
main: done
 [counter: 10455431]
 [should:  20000000]
SUP080:nb180917 arthur.catto$ ./t1 10000000
main: begin [counter = 0] [5c5060]
A: begin [addr of i: 0x700000293efc]
B: begin [addr of i: 0x700000316efc]
B: done
A: done
main: done
 [counter: 10338269]
 [should:  20000000]
SUP080:nb180917 arthur.catto$
```

Where is the bug?

Example: Summing a range of integers

Design a proper multithreaded application that performs the summation of the sequence of non-negative integers up to a given limit.

- Main idea
  - Split the sequence into nt ranges.
  - Create nt threads and let each of them sum one range.
  - Sum the results of all threads.

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

#define N 100
#define NUM_THREADS 6

void *adder(void *param); /* the thread */

int main(void) {
    pthread_t tid[NUM_THREADS];                    /* thread ids */
    int thread_arg[NUM_THREADS][4];
    int sum = 0;
    int range = N / NUM_THREADS;

    /* create the threads */
```

...

```c

    /* wait for the threads to exit and calculate total */
```

...

```c

    printf("Main again... sum = %d.\n", sum);
}

```

```
16
17      /* create the threads */
18      for (int i = 0; i < NUM_THREADS; ++i) {
19          thread_arg[i][0] = i;
20          thread_arg[i][1] = i * range + 1;
21          thread_arg[i][2] = (i == NUM_THREADS-1) ? N : (i+1) * range;
22          printf("Main creating thread %d to sum %3d to %3d.\n",  i,
 •          thread_arg[i][1], thread_arg[i][2]);
23          Pthread_create(&tid[i], NULL, adder,  (void *) &thread_arg[i][0]);
24      }
25
```

```
25
26      /* wait for the threads to exit and calculate total */
27      for (int i = 0; i < NUM_THREADS; ++i) {
28          Pthread_join(tid[i], NULL);
29          printf("Thread %d returned %d.\n", i, thread_arg[i][3]);
30          sum += thread_arg[i][3];
31      }
32
```

```
35
36  void *adder(void *param) {
37      int i;
38      int ix    = *((int *)param);
39      int lower = *((int *)param + 1);
40      int upper = *((int *)param + 2);
41
42      printf("Thread %d summing %3d to %3d...\n",  ix, lower, upper);
43
44      int sum = 0;
45
46      for (i = lower; i <= upper; i++)
47          sum += i;
48
49      *((int *)param + 3) = sum;
50      printf("Thread %d summed %3d to %3d... Result %d.\n",  ix, lower, upper,
 •      sum);
51
52      pthread_exit(NULL);
53  }
```

Sample result output

```
SUP080:nb180917 arthur.catto$ gcc -o t2 t2.c -Wall -pthread
SUP080:nb180917 arthur.catto$ ./t2
Main creating thread 0 to sum   1 to  20.
Main creating thread 1 to sum  21 to  40.
Main creating thread 2 to sum  41 to  60.
Thread 0 summing   1 to  20...
Thread 1 summing  21 to  40...
Main creating thread 3 to sum  61 to  80.
Thread 2 summing  41 to  60...
Thread 3 summing  61 to  80...
Thread 2 summed  41 to  60... Result 1010.
Thread 0 summed   1 to  20... Result 210.
Main creating thread 4 to sum  81 to 100.
Thread 3 summed  61 to  80... Result 1410.
Thread 1 summed  21 to  40... Result 610.
Thread 4 summing  81 to 100...
Thread 4 summed  81 to 100... Result 1810.
Thread 0 returned 210.
Thread 1 returned 610.
Thread 2 returned 1010.
Thread 3 returned 1410.
Thread 4 returned 1810.
Main again... sum = 5050.
SUP080:nb180917 arthur.catto$
```

What do the messages tell you about program execution?