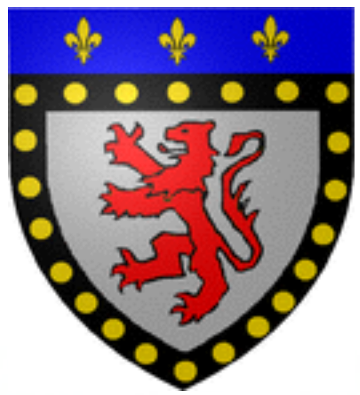
# Asynchronous and Parallel Programming

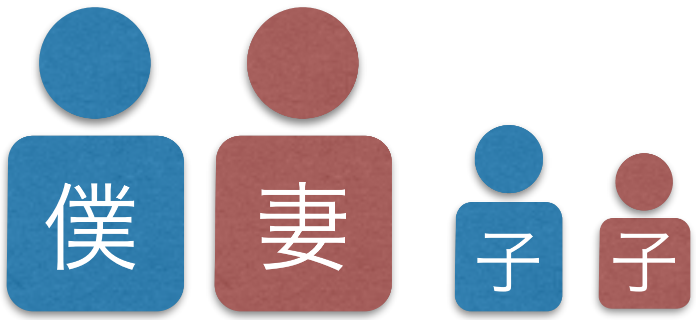
Antoine Trouvé
2015/05/25

# Self Introduction

- Family name: **Trouvé (トルヴェ)**
- Given name: **Antoine (アントワン)**
- Origin: **Poitiers, France (ポワチエ)**
  - http://ja.wikipedia.org/wiki/ポワチエ



- Study
  - **Master:** Bordeaux Institute of Technology
  - **PhD:** Kyushu University
- Now:
  - Assistant professor at Kyushu University
  - Family



# About this Lecture

- **Two sessions**
  - 2015/5/25 (today)
  - 2015/6/1 (next Monday)
- **Content**
  - 13:00 ~ 14:30: Lecture
  - 14:50 ~ 16:20: Exercise

Slides in
English

話は
日本語

# What you will Learn

C Programming

Operating System

Debug with printf

Use Linux

Virtual Machine

Remote coding

Connect via SSH

**Parallel Programming**

Launch a simple web server

Computer Architecture

Image processing

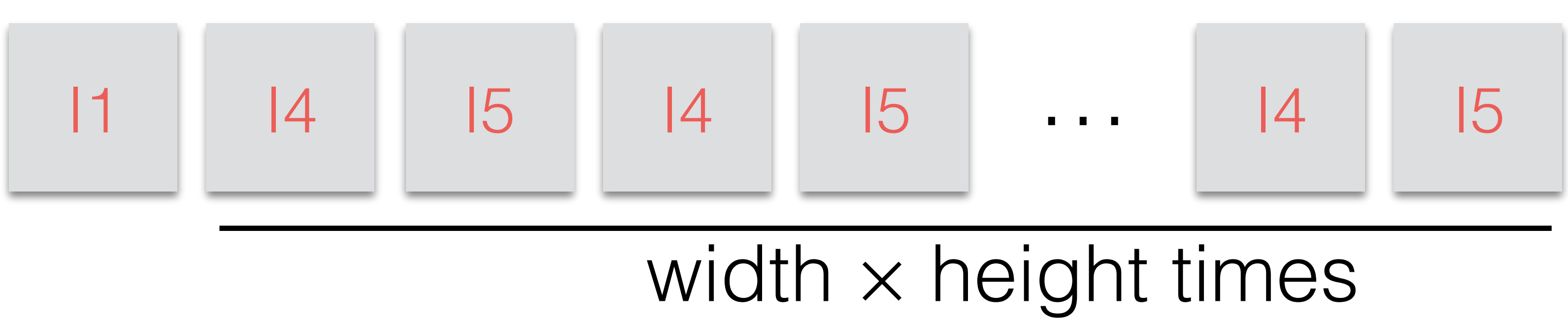
# Why Parallel Programming ?

# How Traditional Program are Executed

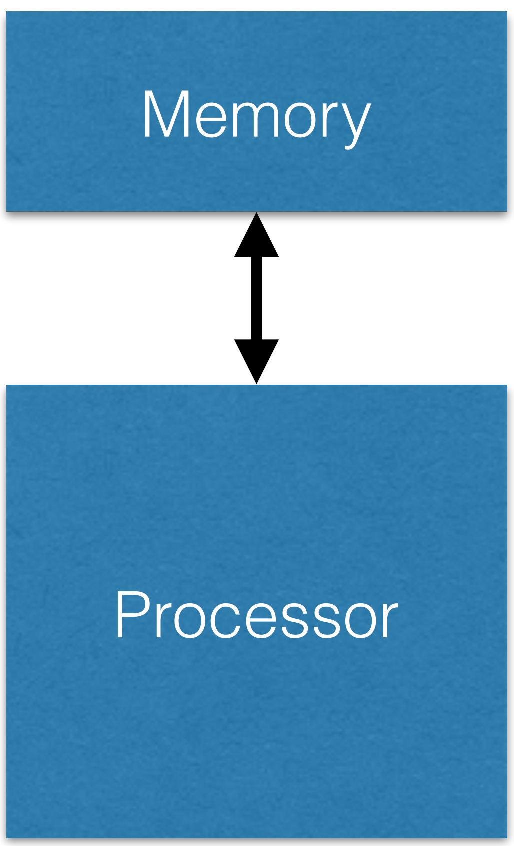
- Let us consider this program (pseudo code):

```
I1 image = read image file
I2 for(x=1; x<width-1; x++)
I3   for(y=1; y<height-1; y++)
I4     pixel = image[x][y]
I5     pixel *= -1
```

- I is executed as follows (if we ignore I3 and I4)

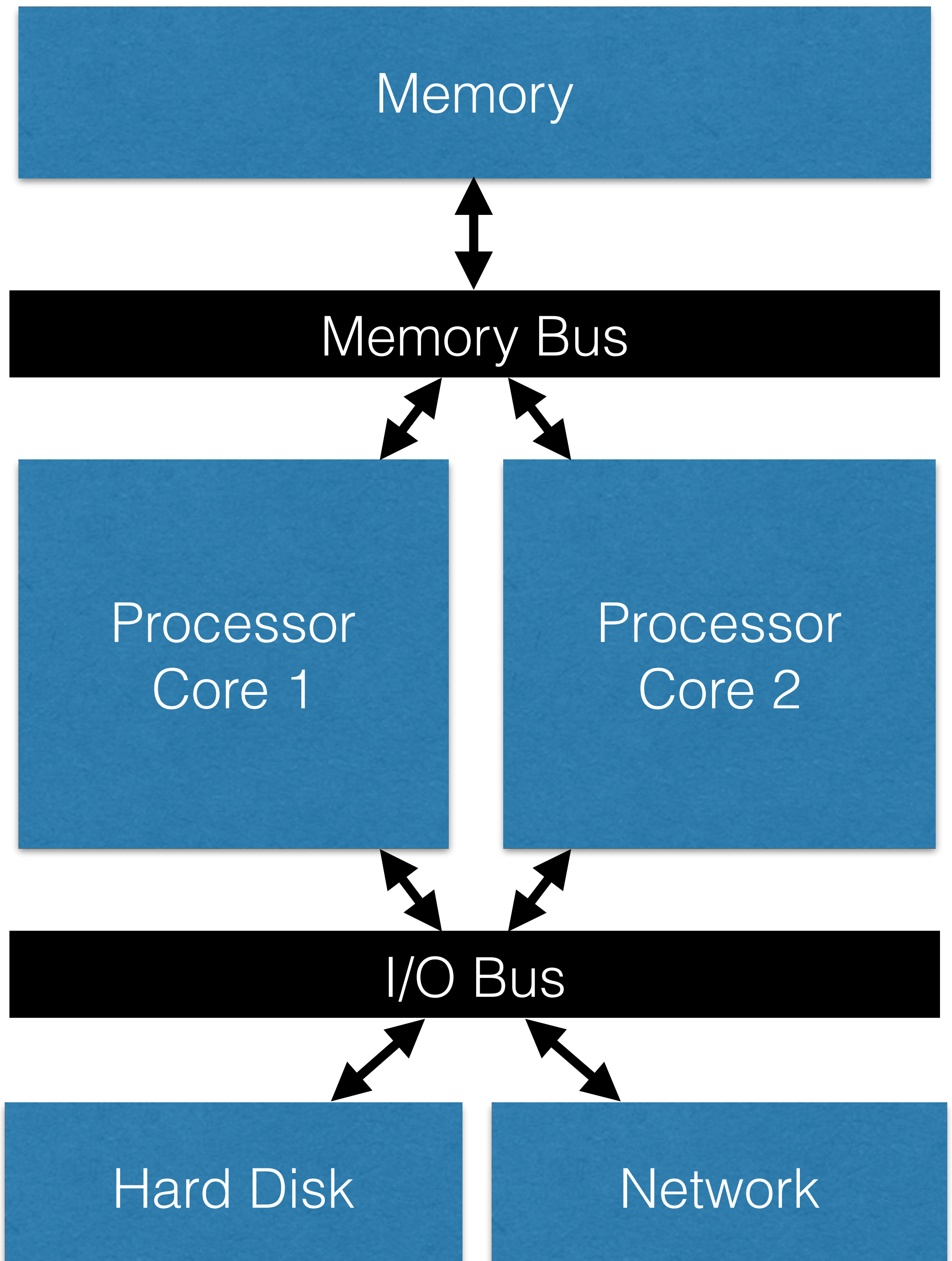


# Hw Architecture: What this program supposes




- The memory stores all the data
- The processor executes the instructions
- **But …**

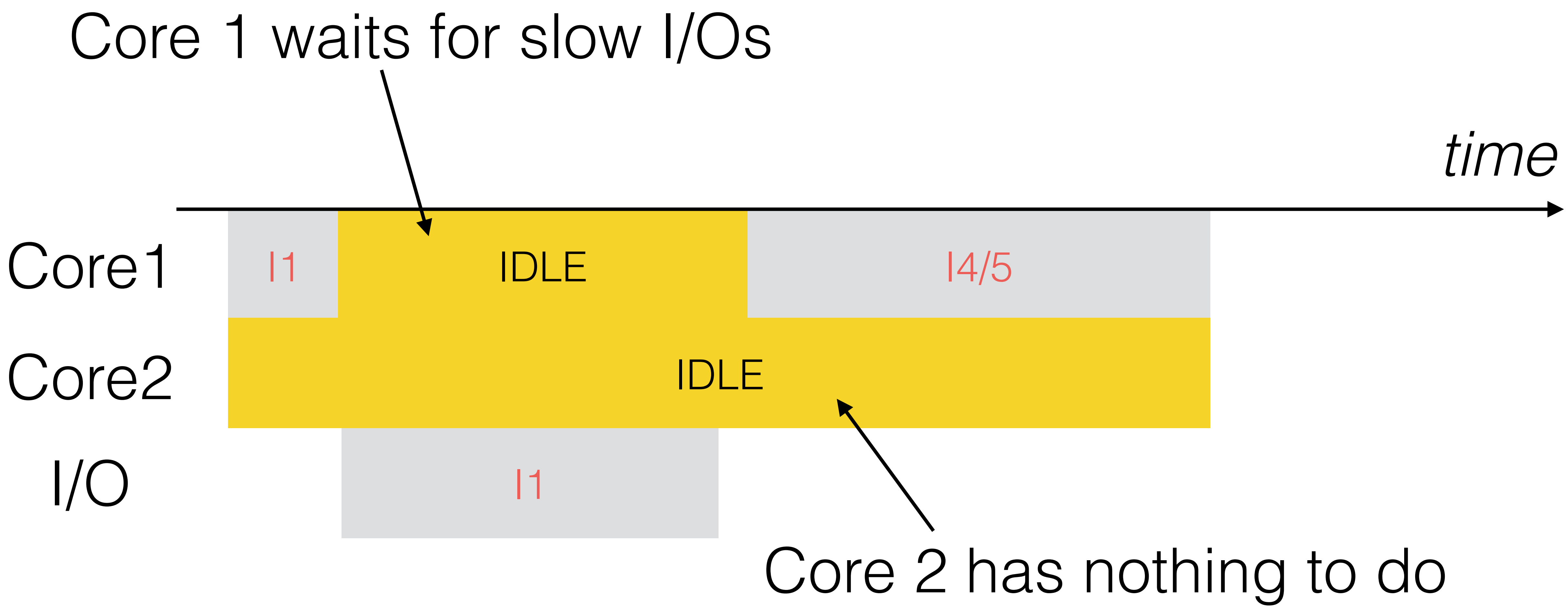
# Hw Architecture
# What **Really** Exists



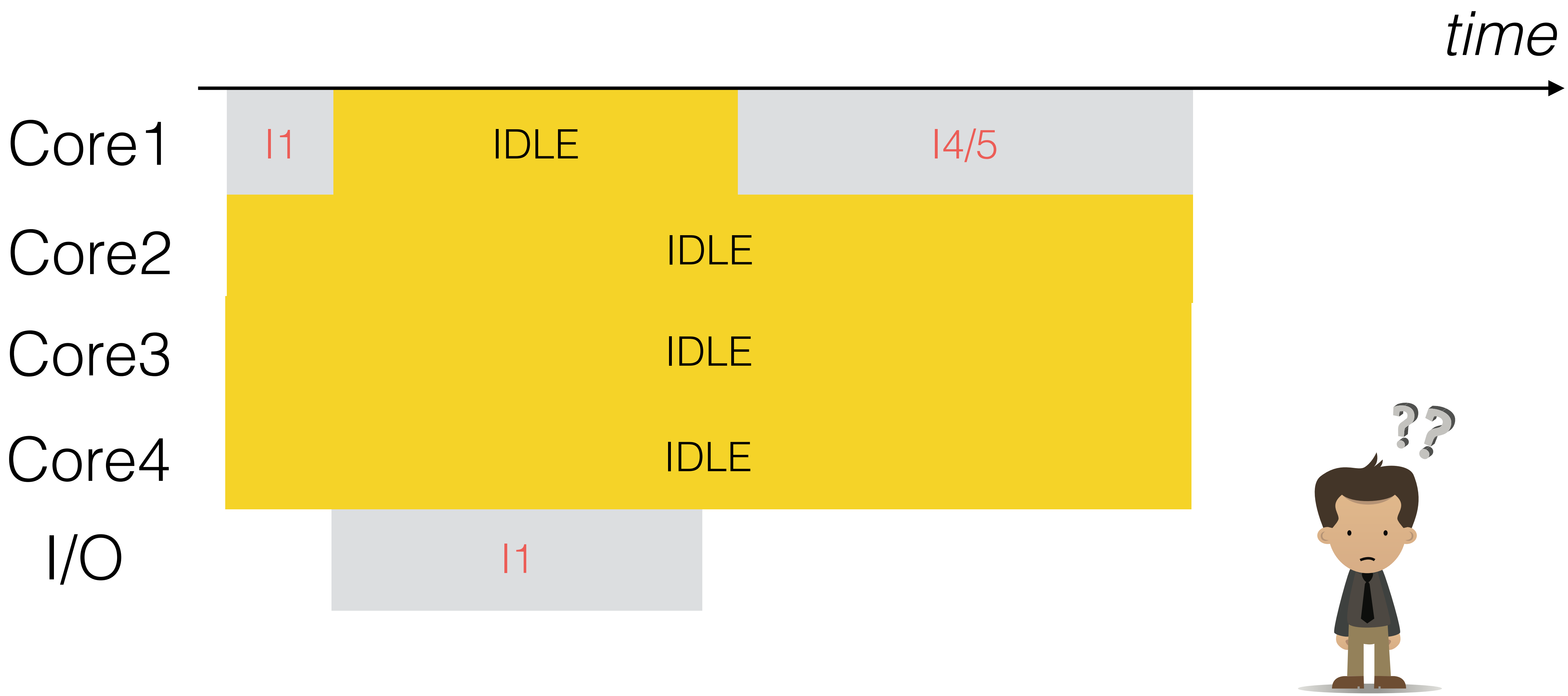

- Multi core processor
  - 2 on this figure
  - Can be 4, 6, 8 … more !
- Files are stored in slow I/Os
  - Hard drive / SSD access: 1 ~ 10ms
  - Network access: 100ms

That is 100 000 000 cycles on a 1GHz processor !

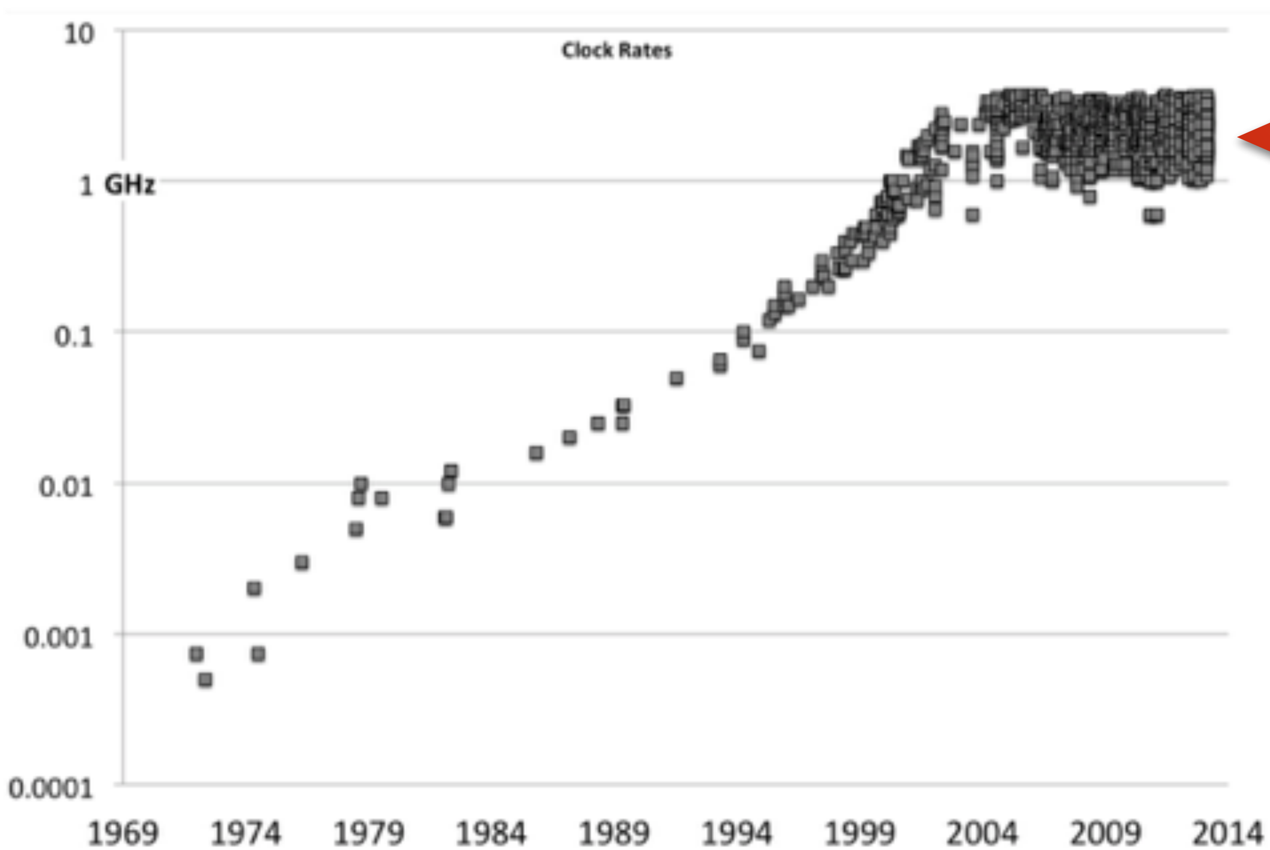
# Traditional Program on Modern Hw Architecture



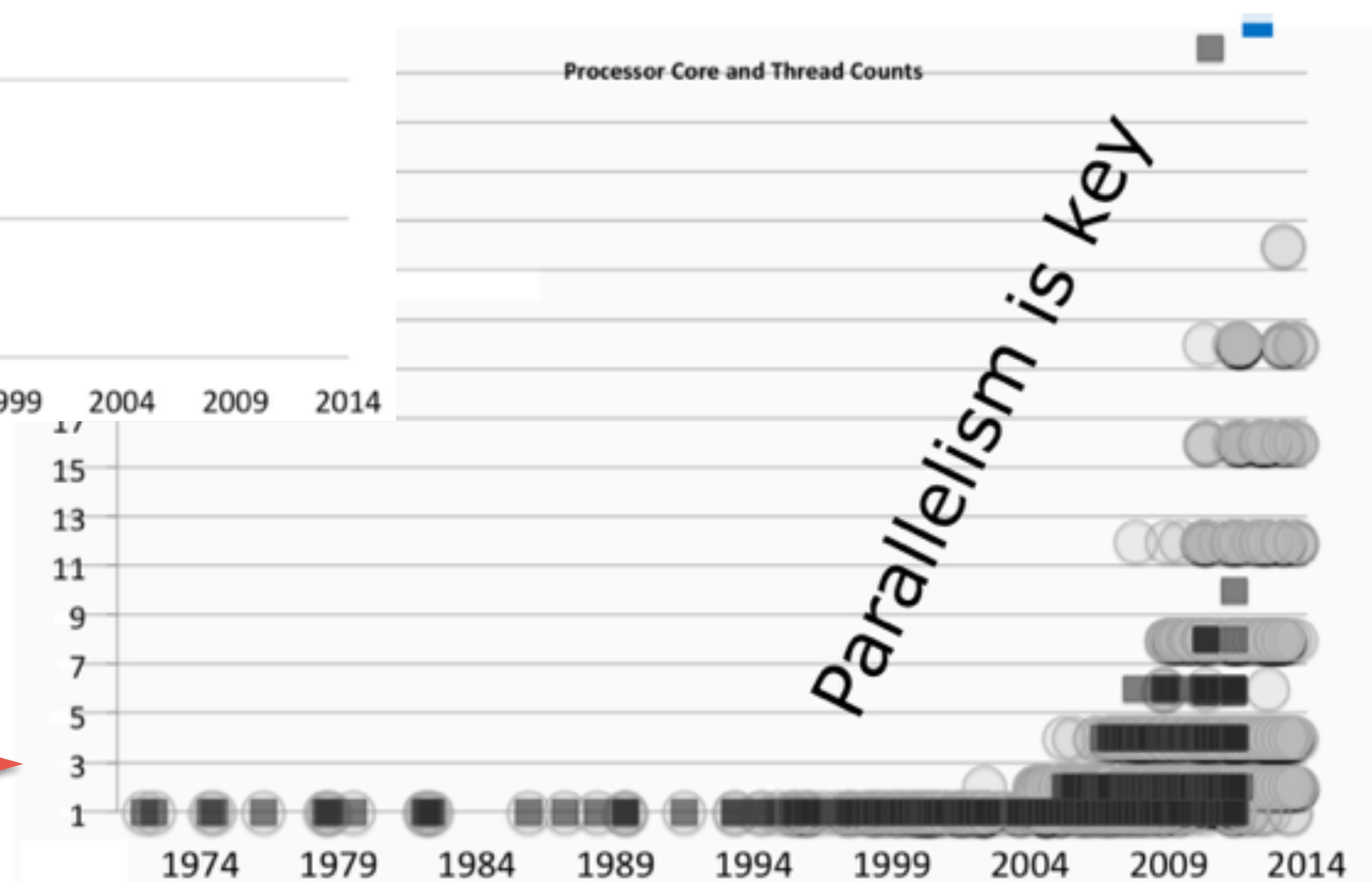
# Traditional Program on Modern Hw Architecture (4 core)



# Current Processor Trends




Frequency is stalling / diminishing




# core is raising

Source: http://ipcc.cs.uoregon.edu/lectures/lecture-16-spp.pdf

# Mini-Test

- I have the following hw architecture
  - 2 processor cores at 2.6GHz (average IPC=1.5)
  - Average HDD access time: 2ms + 1Gb/s
  - Average RAM access time: 100ns
  - Average cache access time: 5ns
  - Cache line size: 128 bits
- **Question:** Calculate the execution time of the following program (only consider I1, I4 and I5)

```
I1 image = read image file
I2 for(x=1; x<width-1; x++)
I3   for(y=1; y<height-1; y++)
I4     pixel = image[x][y]
I5     pixel *= -1
```

(the image is 20 MB)

# Mini-Test

- I got rich, so I bought a new processor with 8 cores at 1.6GHz and an IPC per core of 1.6

- **Question:** Will the program run faster ?

# Conclusion

We need to better use our computing resources !

Asynchronous
Parallel
Distributed
Concurrent

# Asynchronous and Parallel Programming

非同期

- **Asynchronous** = Not Synchronous
  - We don’t execute tasks in sequential orders
  - Tasks are started before the others end
  - This is useful to
    - Hide the time spent in I/Os
    - Give the impression of simultaneity on single core
- **Parallel** 並列
  - When asynchronous tasks actually run simultaneously we use the term parallel programming
  - This is only possible if you have multiple processor cores

# Use case of Asynchronous Programming (1)

**Make video games both fluid and interactive**

```
/* We want 60 frames per second */
#define FRAMERATE 60

/* Defines some functions and structure for my game */
#include "MyGame.h"

/* GameState is a structure defined in MyGame.h */
GameState *game_state;

int main() {
  /* Some variables to store the time elapsed between two frames */
  clock_t last_frame = clock();
  clock_t now;
  /* The number of clocks between frames */
  clock_t delta = CLOCKS_PER_SEC / FRAMERATE;
  /* Stores the key pressed by the user */
  char c;

  /* init_game_state is a function defined in MyGame.h */
  game_state = init_game_state();

  while(true) {
    /* Updates the display if enough clocks are elapsed */
    now = clock();
    if(now-last_frame > delta) {
      /* render_frame is a function defined in MyGame.h */
      /* It updates the display */
      render_frame(game_state);
      last_frame = now;
    }
    /* Captures user input */
    c = getch();
    if(c!=ERR) {
      /* update_game_state is a function defined in MyGame.h */
      /* It updates the state of the game depending on user input */
      update_game_state(game_state);
    }
  }
}
```

This program is a “game loop”, the base of almost any game

# Use case of Asynchronous Programming (1)

**Make video games both fluid and interactive**

```c
/* We want 60 frames per second */
#define FRAMERATE 60

/* Defines some functions and structure for my game */
#include "MyGame.h"

/* GameState is a structure defined in MyGame.h */
GameState *game_state;

int main() {
  /* Some variables to store the time elapsed between two frames */
  clock_t last_frame = clock();
  clock_t now;
  /* The number of clocks between frames */
  clock_t delta = CLOCKS_PER_SEC / FRAMERATE;
  /* Stores the key pressed by the user */
  char c;

  /* init_game_state is a function defined in MyGame.h */
  game_state = init_game_state();

  while(true) {
    /* Updates the display if enough clocks are elapsed */
    now = clock();
    if(now-last_frame > delta) {
      /* render_frame is a function defined in MyGame.h */
      /* It updates the display */
      render_frame(game_state);
      last_frame = now;
    }
    /* Captures user input */
    c = getch();
    if(c!=ERR) {
      /* update_game_state is a function defined in MyGame.h */
      /* It updates the state of the game depending on user input */
      update_game_state(game_state);
    }
  }
}
```

Initializes the game

Updates the display (draws the screen)

Processes user input (keyboard hit)

# Use case of Asynchronous Programming (1)

**Make video games both fluid and interactive**

- The functions `render_frame`, `getc` and `update_game_state` should be executed asynchronously
- **Question:** what happens otherwise ?

```
/* We want 60 frames per second */
#define FRAMERATE 60

/* Defines some functions and structure for my game */
#include "MyGame.h"

/* GameState is a structure defined in MyGame.h */
GameState *game_state;

int main() {
  /* Some variables to store the time elapsed between two frames */
  clock_t last_frame = clock();
  clock_t now;
  /* The number of clocks between frames */
  clock_t delta = CLOCKS_PER_SEC / FRAMERATE;
  /* Stores the key pressed by the user */
  char c;

  /* init_game_state is a function defined in MyGame.h */
  game_state = init_game_state();

  while(true) {
    /* Updates the display if enough clocks are elapsed */
    now = clock();
    if(now-last_frame > delta) {
      /* render_frame is a function defined in MyGame.h */
      /* It updat
      render_frame(game_state);
      last_frame = now;
    }
    /* Captures user input */
    c = getch();
    if(c!=ERR) {
      /* update_game_state is a function defined in MyGame.h */
      /* It updates the state of the game depending on user input */
      update_game_state(game_state);
    }
  }
}
```

Updates the display (draws the screen)

Captures user input (keyboard hit)

# Use case of Asynchronous Programming (2)

**Execute programs simultaneously on a single core**

- Most modern operating systems are **multitasked**
  - They run multiple programs (or tasks) at the same time
  - This works even on a single core !
- **Question:** how is that possible ?

Mini-test

# A first Parallel Program

# Our First Parallel Program

## Example of our Program with 2 Processing Cores

```
I1 image = read image file
I2 for(x=1; x<width-1; x++)
I3   for(y=1; y<height-1; y++)
I4     pixel = image[x][y]
I5     pixel *= -1
```

Let us to divide calculations between two processor cores

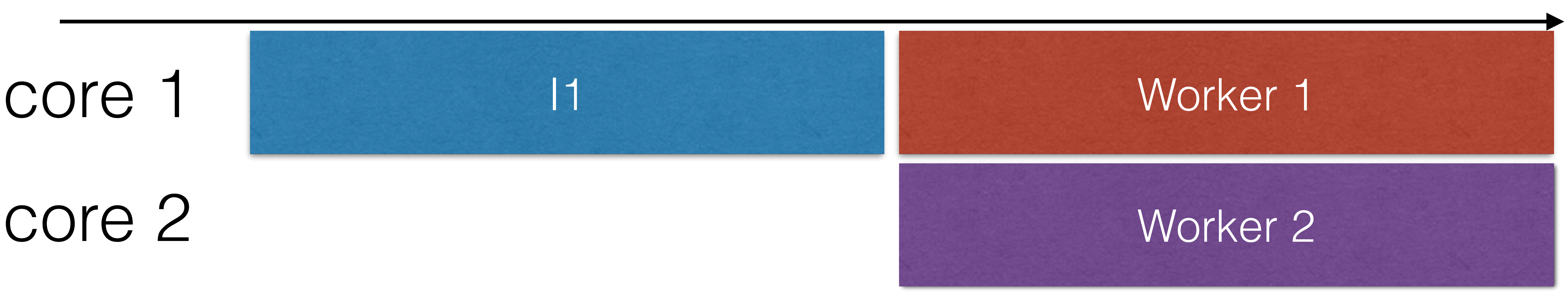
# Our First Parallel Program

## Divide the image among Worker

```
Initialization
I1  image = read image file
```

```
Worker 1
I12 for(x=1; x<width-1; x++)
I13   for(y=1; y<height/2-1; y++)
I14     pixel = image[x][y]
I15     pixel *= -1
```

```
Worker 2
I12 for(x=1; x<width-1; x++)
I13   for(y=height/2; y<height-1; y++)
I14     pixel = image[x][y]
I15     pixel *= -1
```

# Our First Parallel Program

## Divide tasks among Workers

```
Worker 1
I1 image = read image file
```

```
Worker 2
I2 for(x=1; x<width-1; x++)
I3    for(y=1; y<height-1; y++)
I4       pixel = image[x][y]
I5       pixel *= -1
```

We read the data while processing it.
Warning:
- it requires worker 2 to wait for worker 1 to read the data: this is synchronization
- we will study that next week

# Two Approaches to Parallelize Programs

- Data-parallelism
    - All workers are doing the same job, with different data
- Task-parallelism
    - All workers are doing a different task, sub-part of the algorithm
    - Often looks like pipelined processing

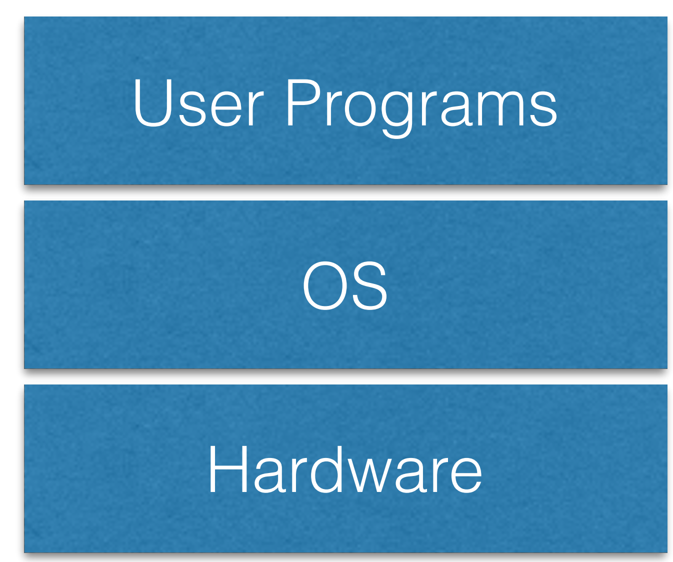
# Mini Test

- I have the following hw architecture
  - 2 processor cores at 2.6GHz (average IPC=1.5)
  - Average memory access time:10 ns
  - Average HDD access time: 2ms + 1Gb/s
- The image is 20MB
- We ignore
  - The cache
  - Instructions I2 and I3
- **Question:** Calculate the execution time of the programs of slide 31, 32, 33. Which one is the fastest ?

# How Modern OS Support Parallelism

# Why are we Talking about the OS ?

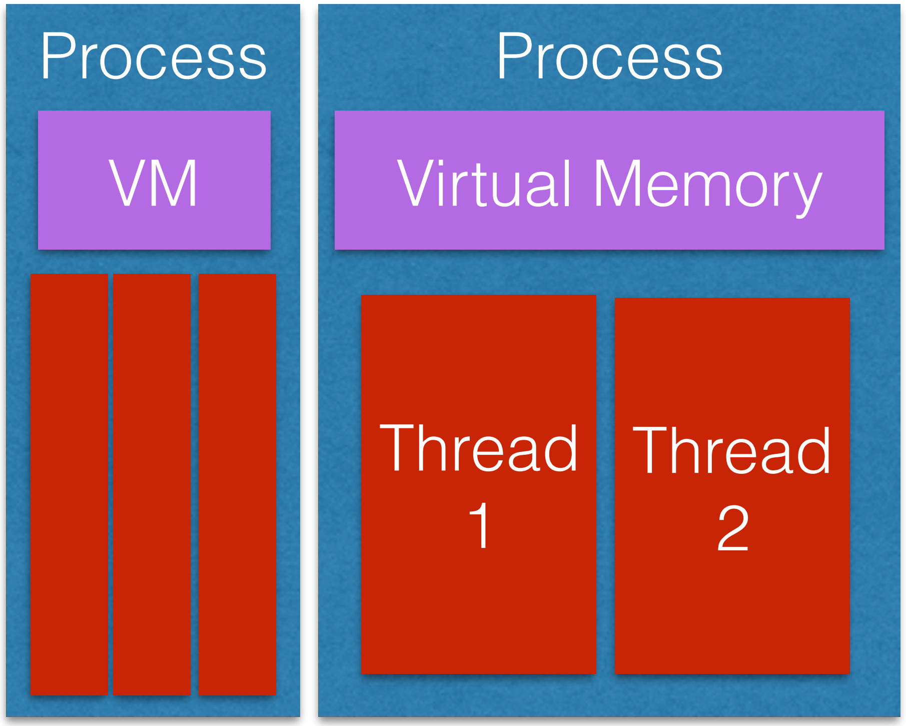
- Programs that we execute are user programs
- They run above the OS, that is, they cannot access the hw directly
- Therefore, the OS needs to support parallelism for user programs to benefit from it




The hw/sw stack

# Threads and Processes

- Most modern OS (Linux, Windows, MacOSX, BSD) support two kinds of parallel facilities
- Facility 1: **Process**
  - Have their own virtual memory
- Facility 2: **Threads**
  - Have their own stack and processor state
  - Threads are affected to processes
  - One process owns at least one thread
  - Threads of a same thread share the same virtual memory



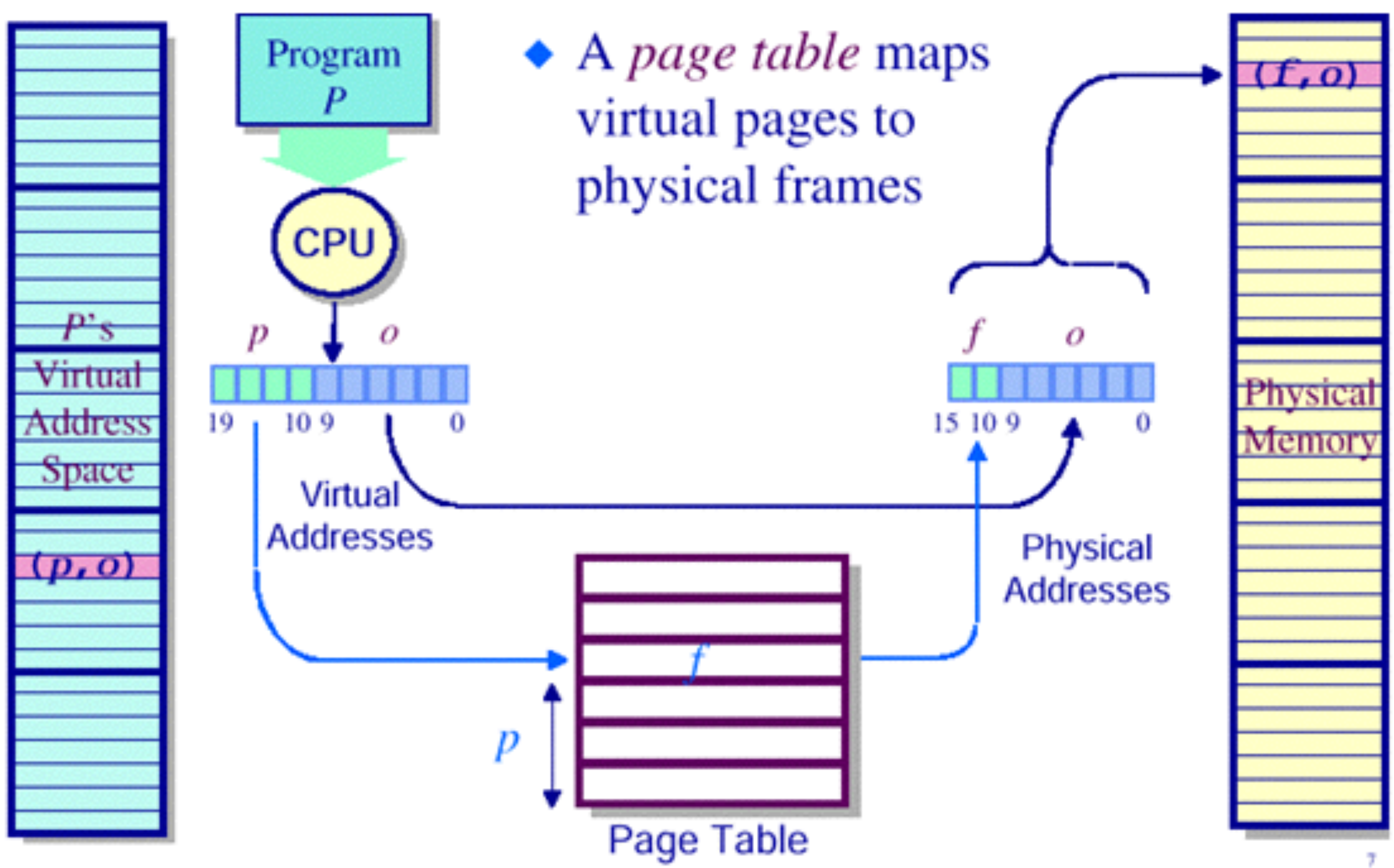
# Reminder: Virtual Memory

- Programs store their data in
  - The processor's registers - a few KB
  - The memory ("the RAM") - several GB
- Data in the memory are designated by addresses, stored in pointers
- In old OS, programs used to manipulate address directly to the real memory, however
  - This made impossible for programs to manipulate data larger than the size of the memory
  - This made possible for programs to modify the data of other programs
- Therefore, modern OS hide real addresses to programs, and give them virtual addresses
- The memory addressed by virtual addresses is the virtual memory

# Reminder: Virtual Address Translation

- Data in the virtual memory may be physically stored in
  - he memory
  - the hard drive
- The OS translates virtual addresses to “real addresses”: this is called address translation
- Address translation is executed at each memory accesses
- In order to speedup address translation, modern processors feature a hardware called the TLB (translation lookup buffer)
- The TLB stores the correspondence between virtual and real addresses

# Reminder: Virtual Address Translation

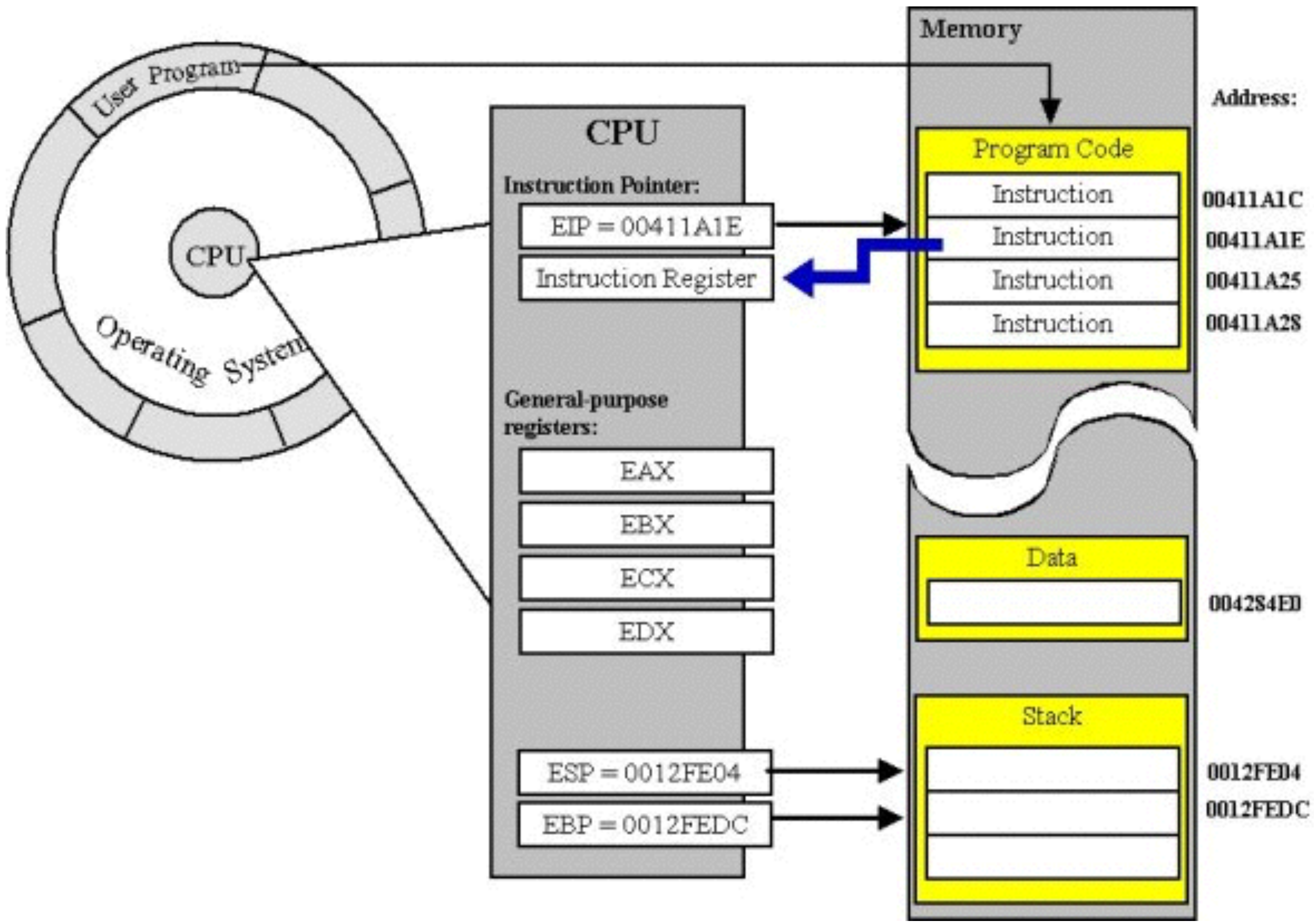



Source: http://bug7015.tistory.com/category/study/Computer%20Architecture

# Reminder: The state of a program

- The state of a program is defined by
  - The state of the processor: which value in which register ?
  - The state of the memory: which values in the memory ?
  - The active virtual memory (that is the state of the TLB)
- The **memory** is divided into three parts
  - The **stack**: where are stored the variables local to functions
  - The **heap**: where are stored dynamically allocated variables
  - Other **data segment**: where are stored static variables

# Reminder: The state of a program




Source: http://www.c-jump.com/CIS60/lecture01_2.htm

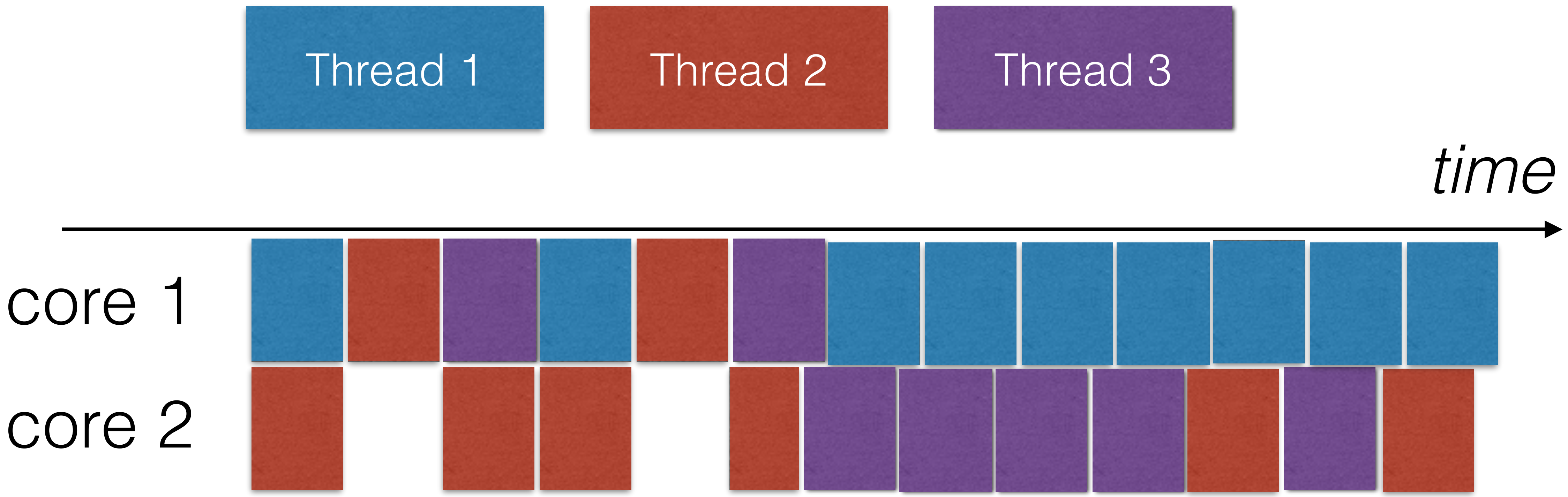
# Mini Test

- **Question:** What does the OS need to store to maintain thread ? Process ?

# Thread Scheduling

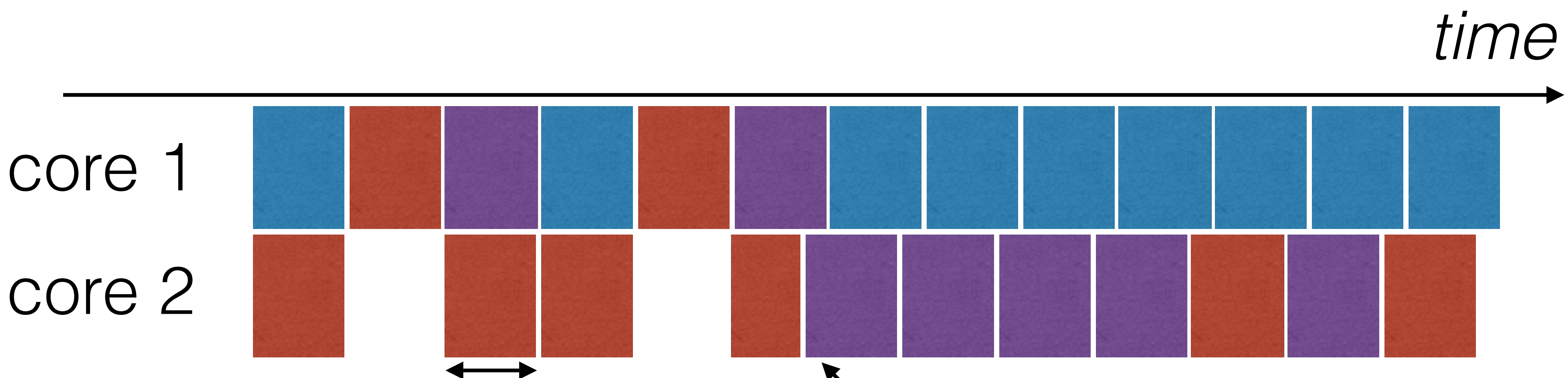
- The OS maintains a list of active threads
- The threads are allocated to computing cores
- When the number of threads is greater that then number of computing cores, threads are re-allocated every fixed amount of time
- This is called **scheduling**

# Example of Thread Scheduling




- The OS executes the scheduling algorithm
- This is an imaginary example of scheduling

# What is a Time Slice ?

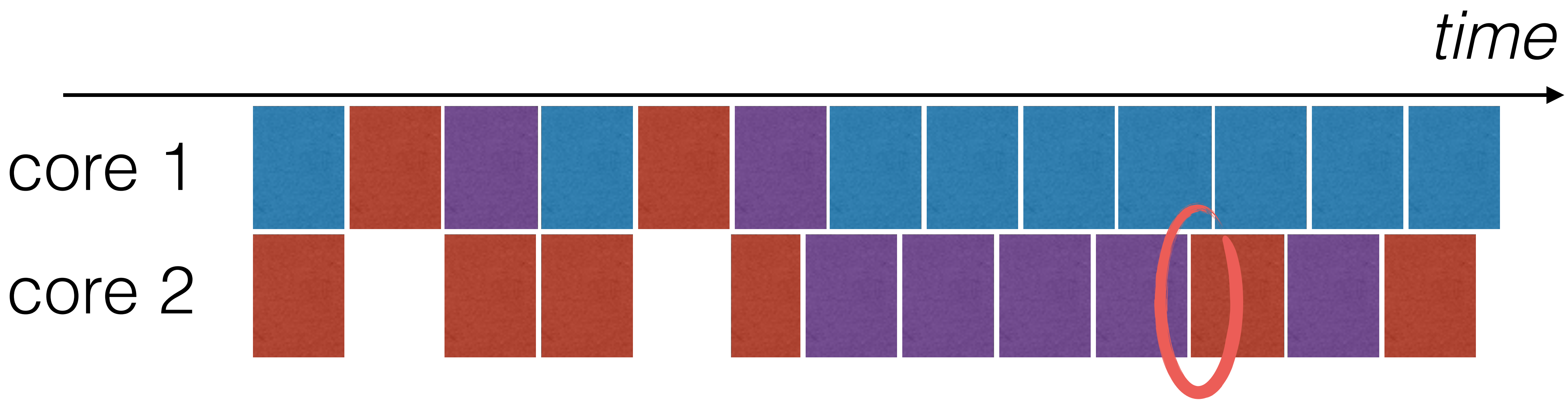



**Time slice**

- The amount of time between each re-scheduling.
- It is usually constant, unless a process waits for I/O

For example thread 1 ends earlier here. This may be because it is waiting for I/O

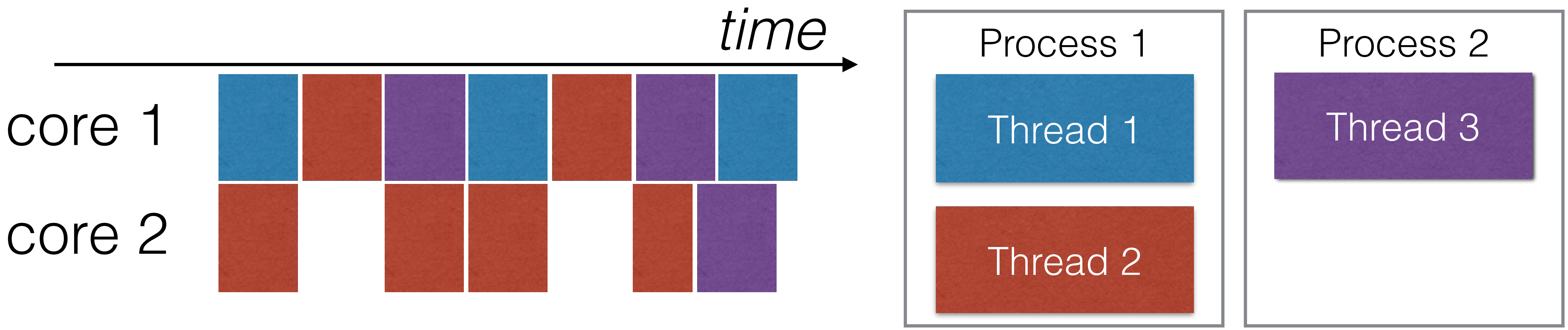
# What is a Context Switch ?




**Context Switch**

- When the scheduler changes the thread active on a core
- Context switch costs CPU time
- The cost depends on the kind of context switch

# Mini-Test




- Threads 1/2/3 are member of processes 1/2 as shown above.
- **Question:** Find 3 types of context switch in the chart below
- **Question:** How are they implemented in the OS, which one is the most expensive ?

# Memory Model

# What is a Memory Model

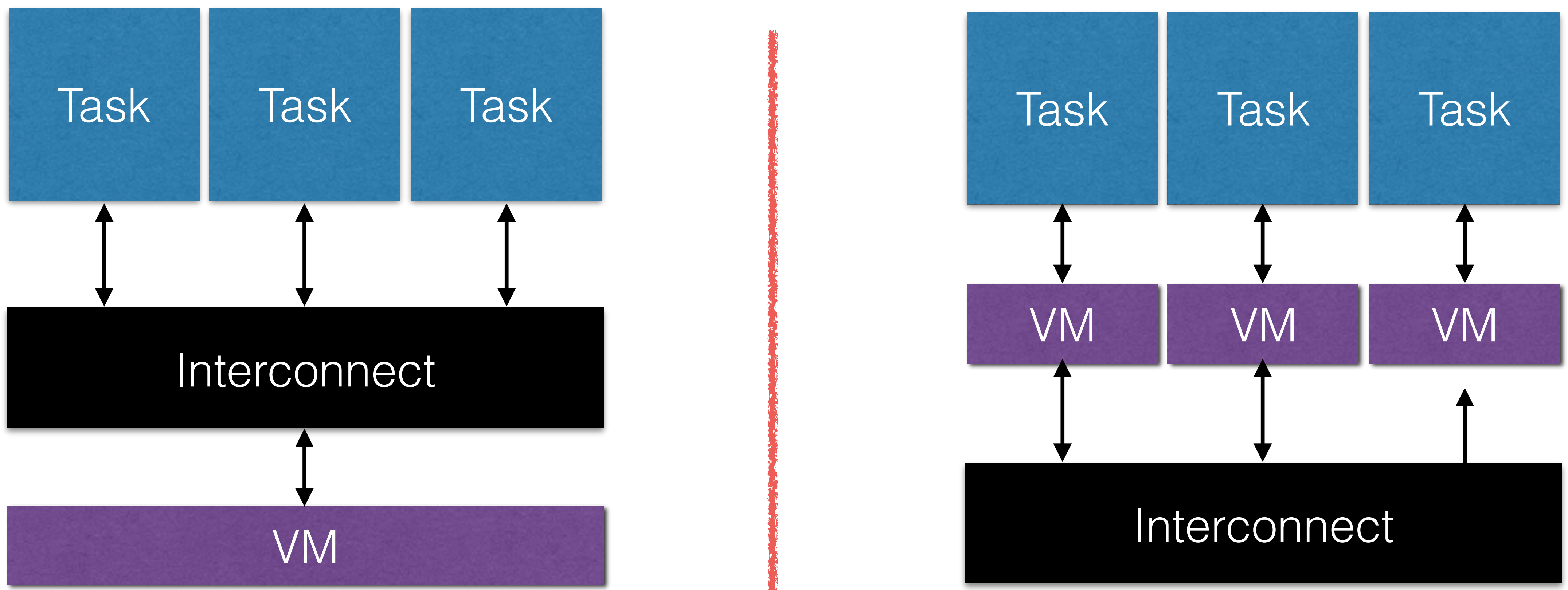
- Modern processors feature complex memory architectures with several levels (e.g. L1 cache, L2 cache, RAM, Scratch-pad Memory, Network)

- But those are not visible from the program

- The **memory model** is the architecture of the memory as exposed by the programming language

- Example: in C, the memory is unified, divided into a global and a local memory

# It is common to classify parallel programming models according to their memory model




Shared Memory

Distributed Memory

- When the memory is distributed, we often use the term distributed programming instead of parallel

# Distributed vs. Parallel Programming

| Type of parallel programming | Parallel | Distributed |
|---|---|---|
| Memory Model | Shared memory | Distributed memory |
| Worker Implementation | Thread | Process |
| Physical Location (typical case) | Same processor (often same core) | Different processor |
| Target Hardware | Single or Multi-core Processor | Many processor systems |
| Inter-task Communication Model | Shared memory | Message passing |
| Major C APIs | POSIX Thread, OpenMP | Fork, MPI, RPC |

# Shared Memory vs. Message Passing

- **Context:** workers want to share data
- When the memory is shared, they can communicate by reading each other memory
- Otherwise, they need a way to send data between each other: this is message passing

| Type of parallel programming | Shared Memory | Message Passing |
|---|---|---|
| On shared memory memory model | ○ | ○ |
| On distributed memory model | × | ○ |
| Cost of communication | Low (need to access a pointer) | High (need to copy data) |

# Exercise

First steps with thread programming with POSIX Thread

Shared memory model

# Before Starting, Let us Setup the Environment

**1.** Configure your virtual machine on Laboratory Cloud

**2.** Install some programs on your personal computer in order to edit the files on Laboratory Cloud

# About Virtual Machine

- We will use **the Cloud** as experimental environment
  - You will have access to your own virtual machine (VM) on Amazon Web Service (AWS), through Laboratory Cloud (LabCloud)
  - It is like having your own computer, but in a remote data center in Tokyo
  - We will connect remotely (ssh) to edit files and execute experiments
- You can think of a virtual machine as a real computer

# Configure your Account to Create a Virtual Machine

- Access to Laboratory Cloud
  - https://www.laboratorycloud.org
- Access to the “toolbox” (ツールボックス)

Login or create a new account



Lab.Cloudツールボックスのトップページ


法人向け：子アカウント登録ページ

**登録を進める：**

① 「法人向け：子アカウント登録」ページで、「カートに追加」をクリック。
② 「購入手続き」をクリックして「カートの内容」ページへ移動。
③ 「カートの内容」ページで、「注文手続きへ」をクリック。
④ 連絡先住所が未登録の場合、「お客様情報」ページで住所を登録し、「続ける」をクリック。

「法人向け：子アカウント登録」ページ　「カートの内容」ページ　「お客様情報」ページ

担当教員の法人コード、および、アカウント名を入力

「注文を確定する」をクリック

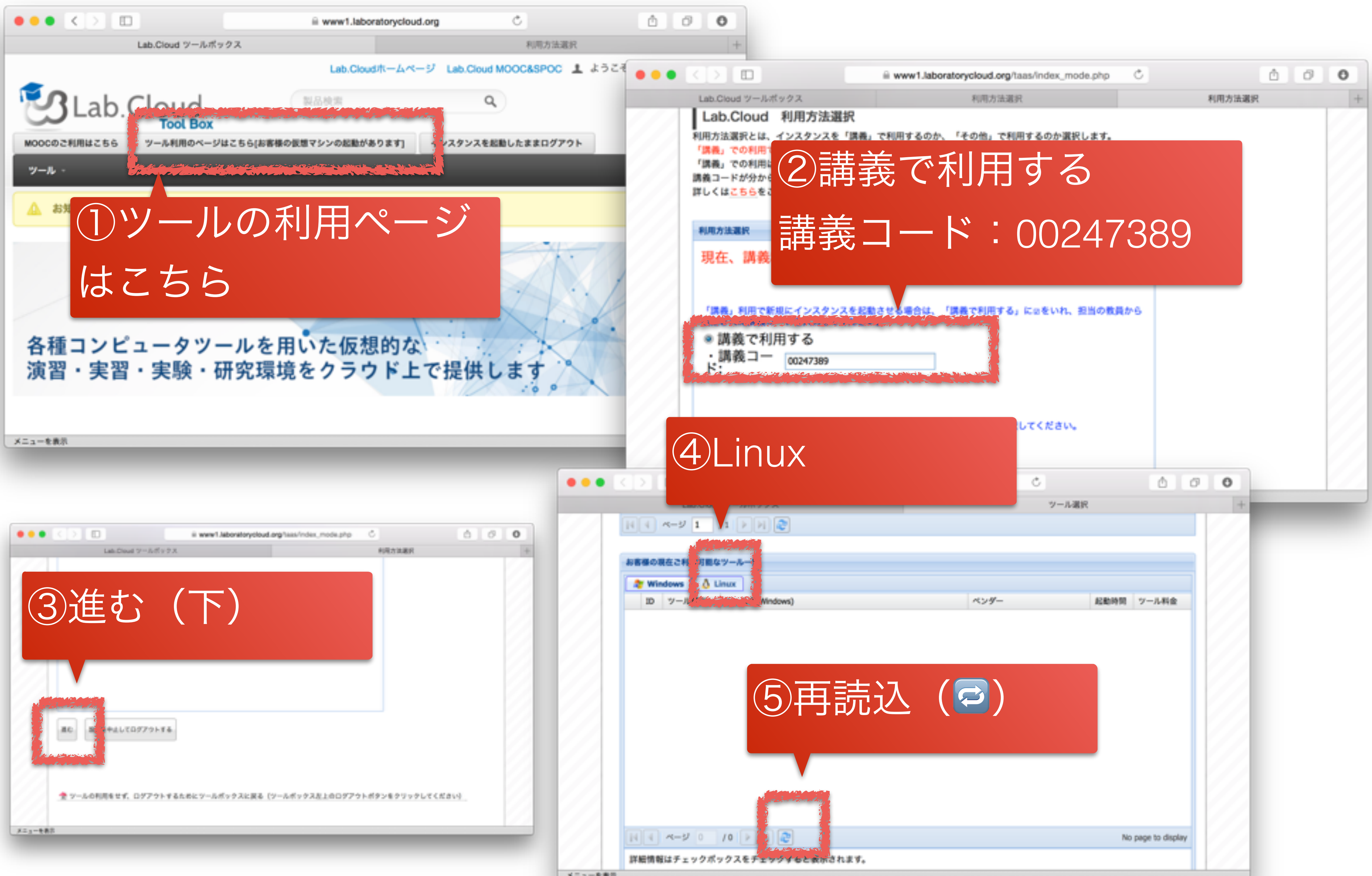
「ご注文手続き」のページ

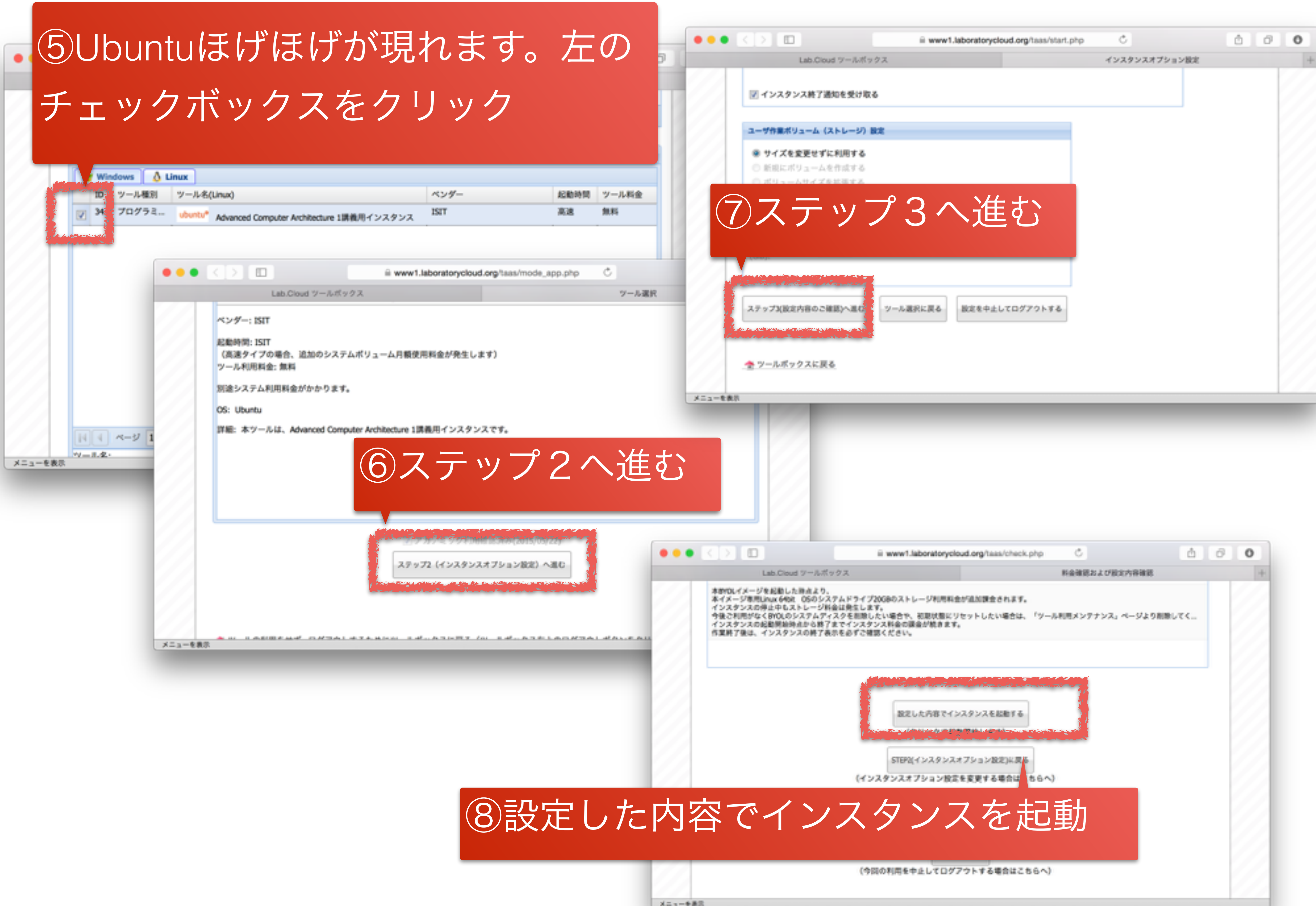
僕のコード：677162933971980

僕のアカウント名：trouve@soc.ait.kyushu-u.ac.jp

**+ Login, Again**

# Create a Virtual Machine




⑤Ubuntuほげほげが現れます。左のチェックボックスをクリック
⑥ステップ２へ進む
⑦ステップ３へ進む
⑧設定した内容でインスタンスを起動

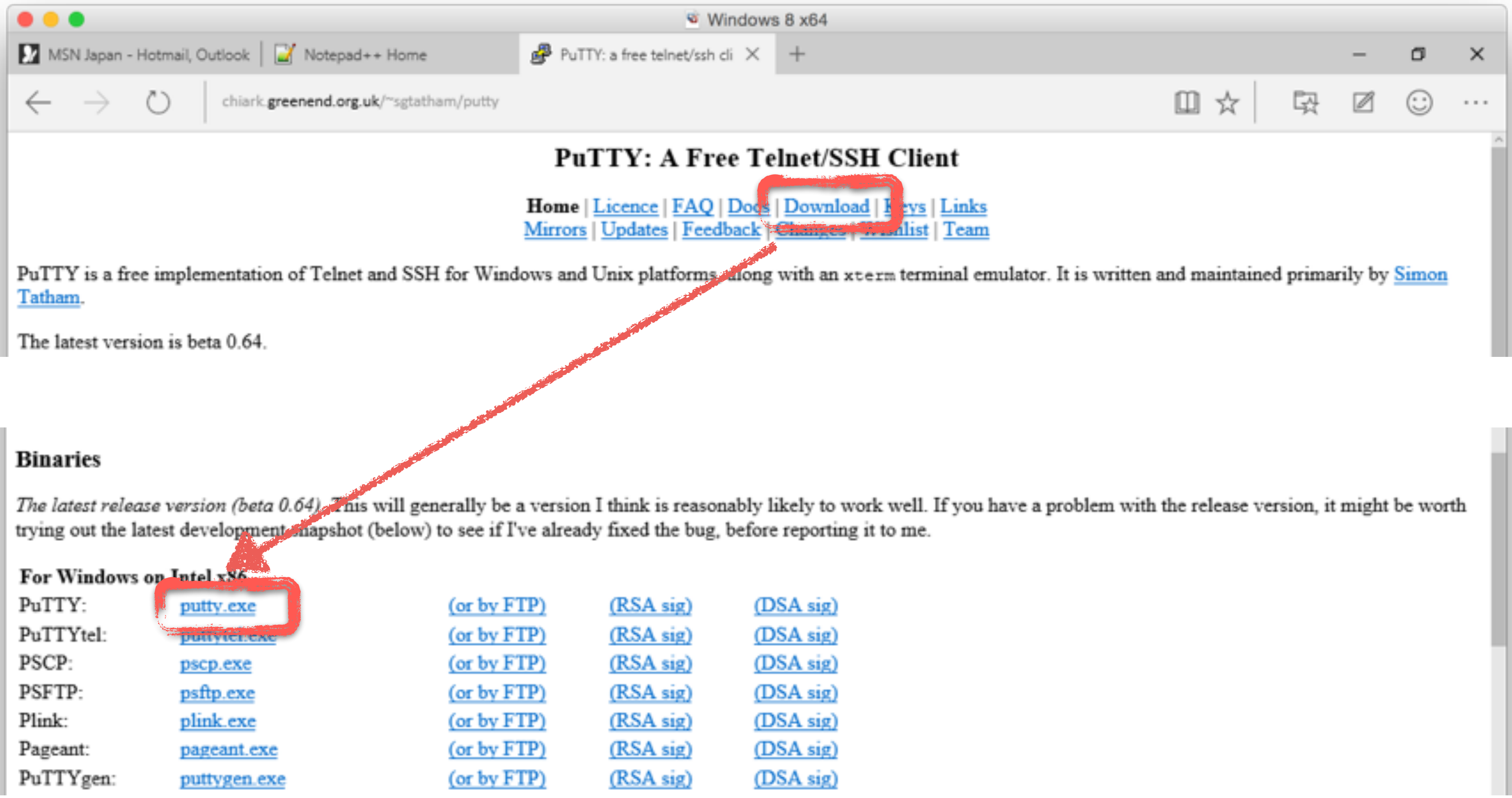
# Connect to your VM

- First you need the IP of the VM so you can connect to it through the Internet

# Access your VM via SSH

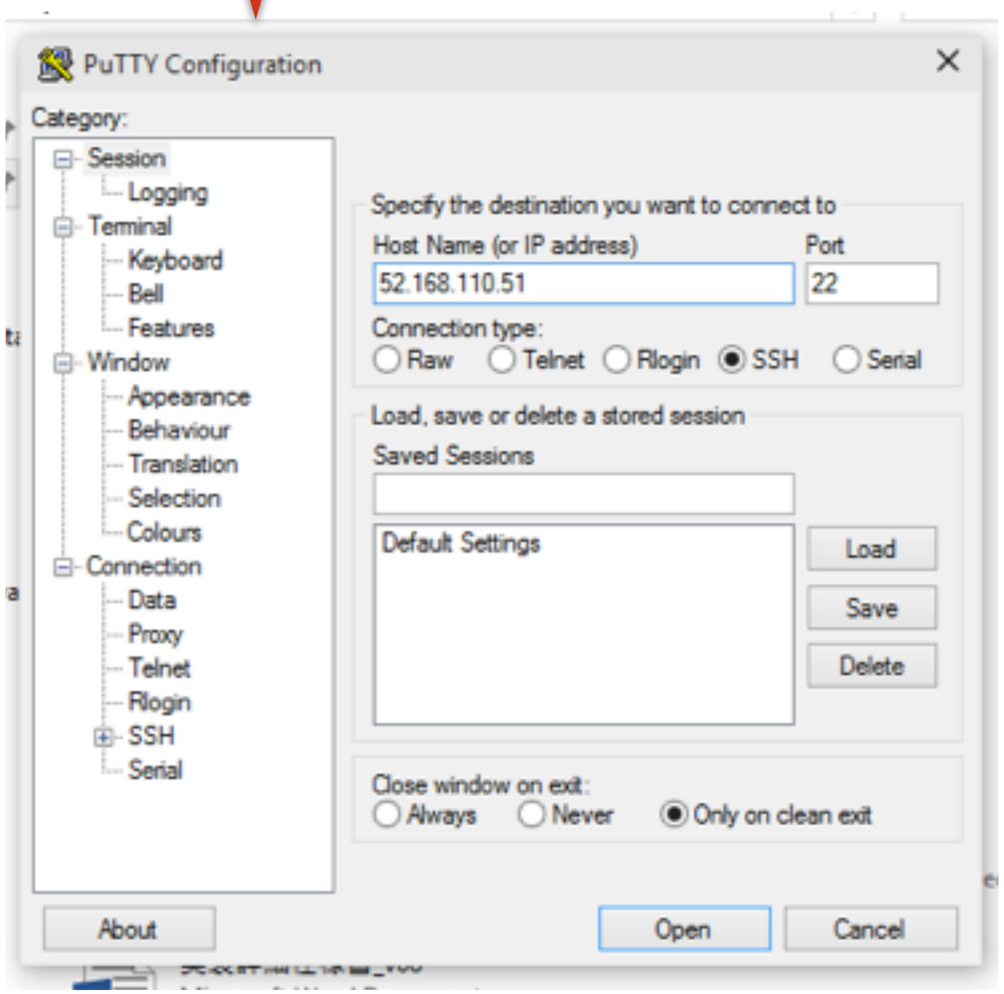
- SSH (Secure SHell)
  - SSH is a protocol to access a distant computer via the network (terminal, file manipulation)
  - Uses encryption
  - Enable to execute command as if your were on the distant computer
- **On Windows**: download Putty
  - Site: http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
  - File “putty.exe“
- **On MacOSX**: use the Terminal
  - In Launchpad, look for “terminal”
- Your **connection information**
  - User name: student
  - Password: I am a student…
  - IP: TBD

# Install Putty

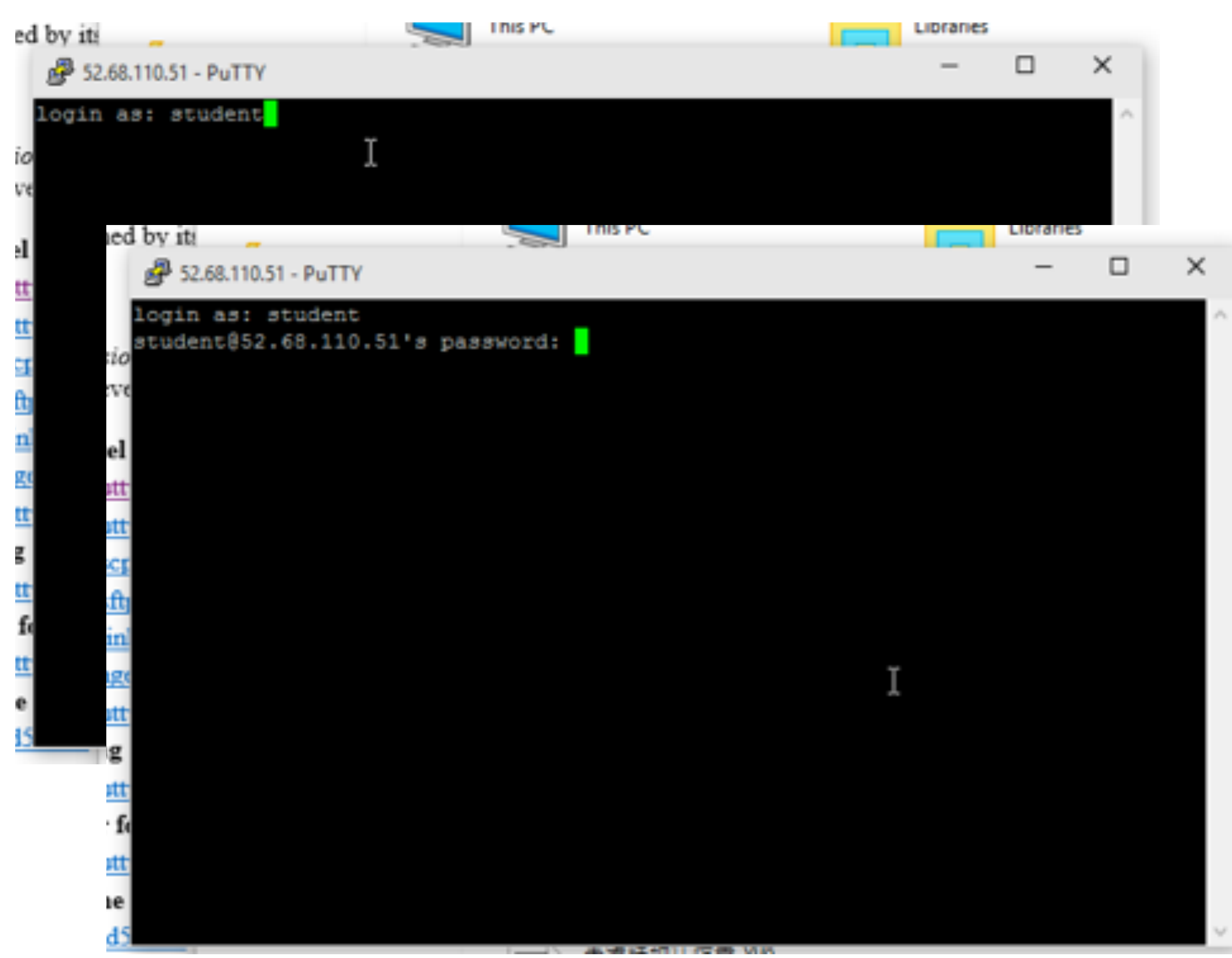


# Access your VM via SSH (Windows)

Enter the IP address of your VM



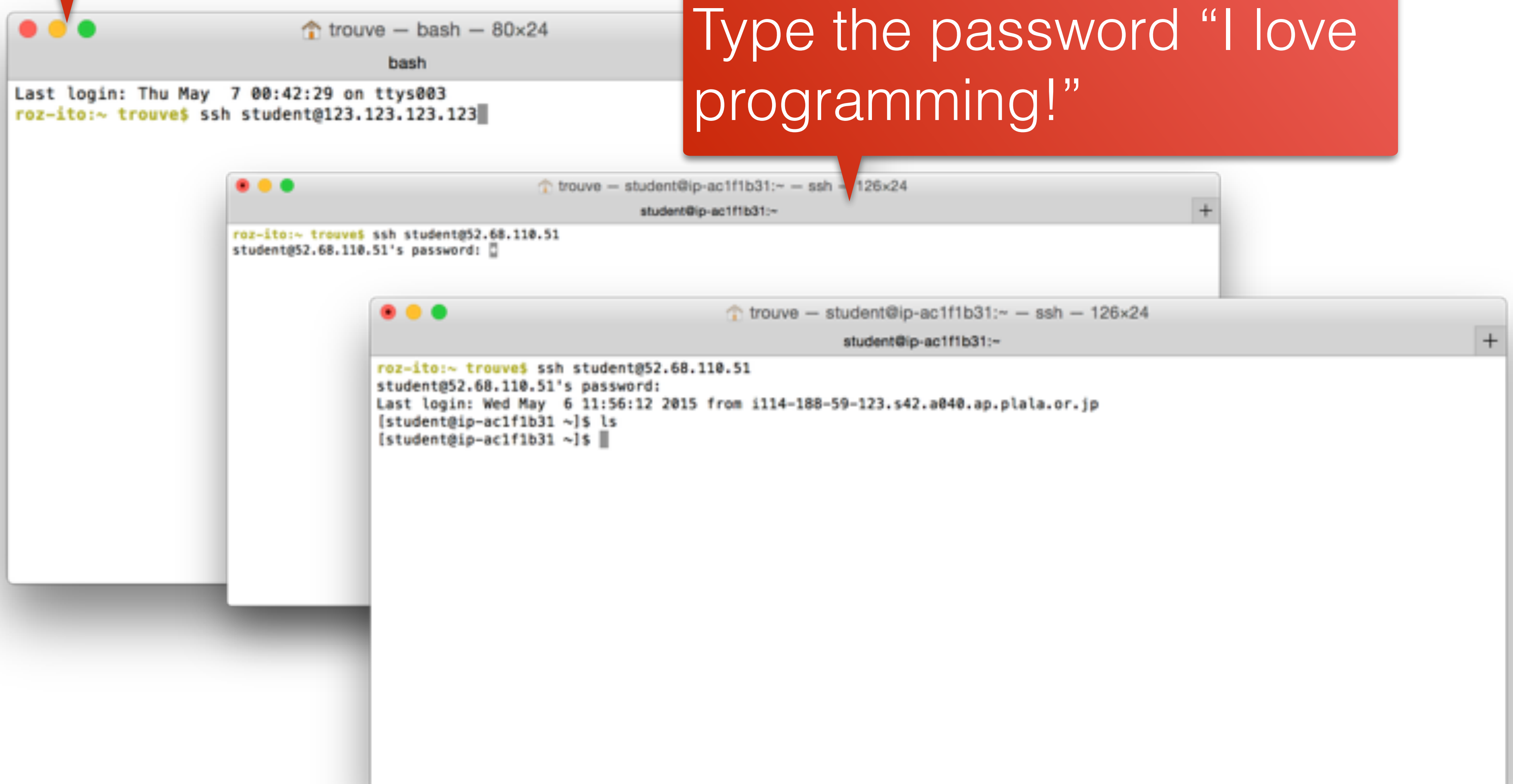

login as: "student"
password: "I love programming!"



# Access your VM via SSH (MacOSX)

Type in the terminal “ssh student@IP”

Type the password “I love programming!”



# Edit Files

- You can edit files with the command line
    - With command “vim” or “emacs” on Putty / Terminal
- But it is **more convenient** to use some remote GUI editing tool
    - Windows: Notepad++ (NppFTP window)
    - MacOSX: Cyberduck “edit” button

# Your very first program in Pthreads

# POSIX Threads in C

- The default way to create threads in Linux is **POSIX threads**, or **pthreads**
- Pthread library is accessible via the library file "**pthread.h**"
- Major functions:
  - Create a thread: `pthread_create(…)`
  - Wait for thread to complete: `pthread_join(…)`
  - Return a value: `pthread_exit(…)`
  - Get the id of the current thread: `pthread_self()`
  - Compare thread ids: `pthread_equal(…)`

# man pthread_create

An address where to store the thread id

**"restrict" keyword**
Says to the compiler that no other pointer points the same object.

$> man pthread_create

```
NAME
     pthread_create -- create a new thread

SYNOPSIS
     #include <pthread.h>

     int
     pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void *),
         void *restrict arg);
```

Argument to pass to the thread function

Some options to create the thread

The function to run in the thread

# Your First Pthread Program

```
#include<stdio.h>    // printf()
#include<unistd.h>   // sleep()
#include<string.h>   // strerror(char*)
#include<pthread.h>

void* doSomeThing(void *arg)
{
  /* The thread id is found, let us switch to some real work */
  printf("Starts thread...\n");

  sleep(3);

  printf("... ends thread.\n");

  return NULL;
}

int main(void)
{
  int i = 0;
  int err;
  pthread_t tid;

  err = pthread_create(&tid, NULL, &doSomeThing, NULL);
  if (err != 0) {
    printf("\ncan't create thread :[%s]\n", strerror(err));
  }

  return 0;
}
```

# Compile / Link / Execute

① Compile the program

```
$> gcc pthread.c -c -o pthread.o
```

② Link

```
$> gcc pthread.o -o pthread.out
/tmp/ccW66lpz.o: In function `main':
pthread.c:(.text+0x57): undefined reference to `pthread_create'
collect2: error: ld returned 1 exit status
```

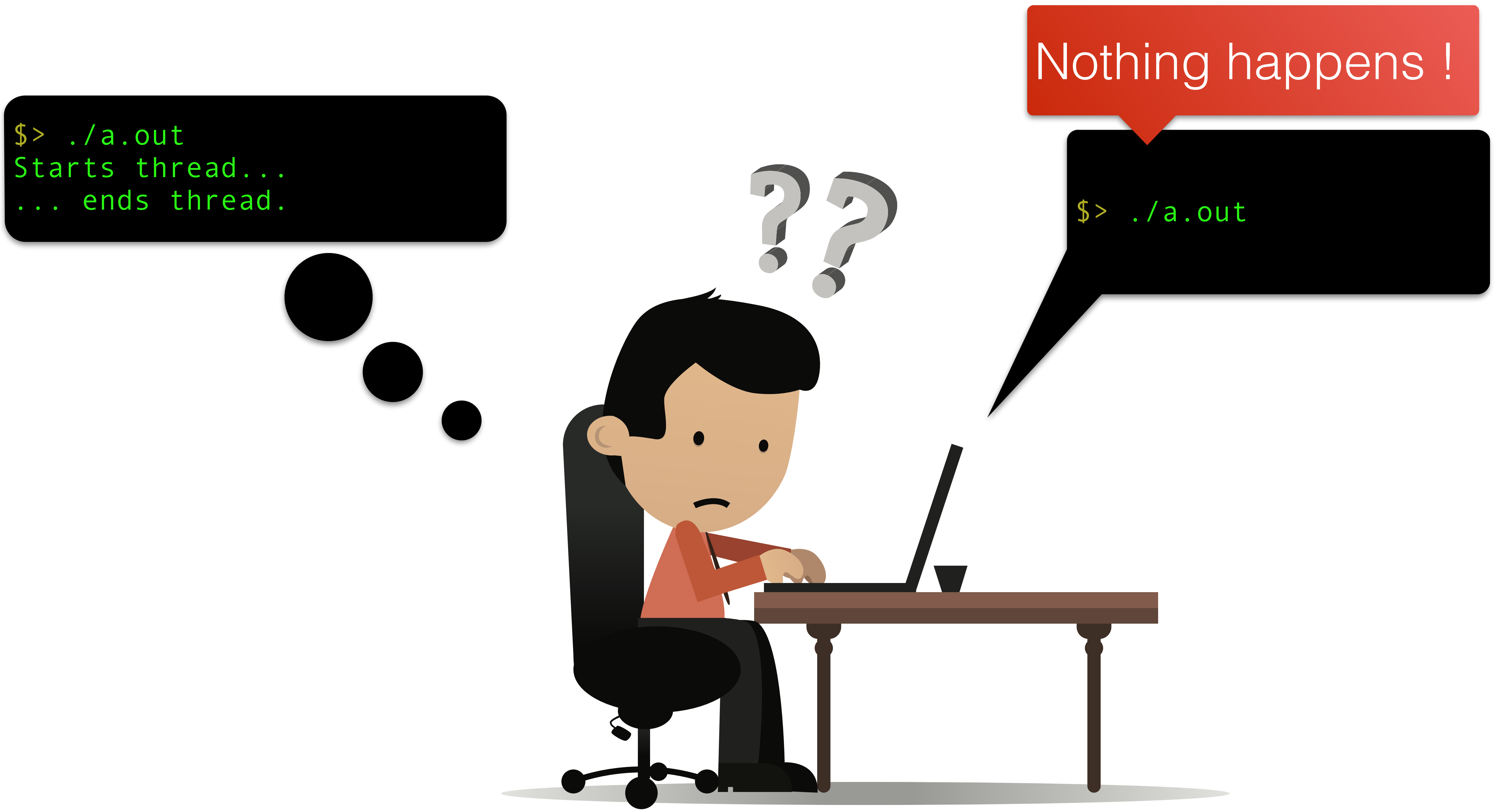
You need to tell gcc to link with libpthread

```
$> gcc pthread.o -lpthread —o pthread.out
```

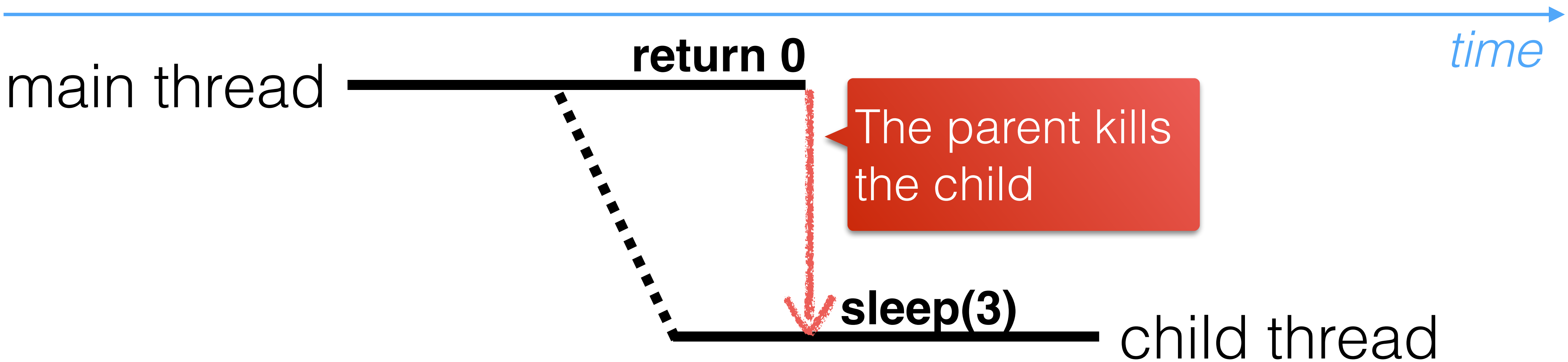
② Execute

```
$> ./pthread.out
```

# Do you get What you Expect ?



# Parent / Child Thread




- The main thread finishes before the other ones
- Because the main thread created the child thread, it is its **parent thread**
- If the parent thread dies or finishes, the child thread is interrupted by the OS

# Question

How would you make the children thread terminate ?

```
$> ./a.out
Starts thread...
Starts thread...
... ends thread.
... ends thread.
```

# How to Make the Child Thread Terminate ?

Answer: make the parent thread **wait** for its children !

# Method 1 (the **bad** one)

```
int main(void)
{
  int i = 0;
  int err;
  pthread_t tid;

  err = pthread_create(&tid, NULL, &doSomeThing, NULL);
  if (err != 0) {
    printf("\ncan't create thread :[%s]\n", strerror(err));
  }

  sleep(3);

  return 0;
}
```

Wait some time for children to finish

# Method 1 (the **bad** one)

```c
int main(void)
{
  int i = 0;
  int err;
  pthread_t tid;

  err = pthread_create(&tid, NULL, &doSomeThing, NULL);
  if (err != 0) {
    printf("\ncan't create thread :[%s]\n", strerror(err));
  }

  sleep(3);

  return 0;
}
```

Wait some time for children to finish

**BAD !**

In general, we don't know how long we have to wait !

# Method 2 (the **good** one)

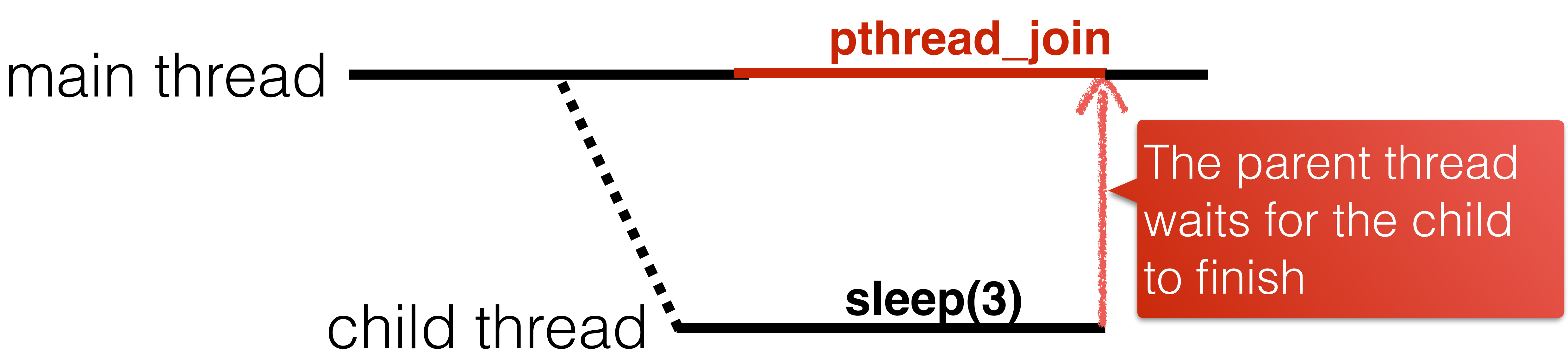
```
int main(void)
{
  int i = 0;
  int err;
  pthread_t tid;

  err = pthread_create(&tid, NULL, &doSomeThing, NULL);
  if (err != 0) {
    printf("\ncan't create thread :[%s]\n", strerror(err));
  }

  pthread_join(tid, NULL);

  return 0;
}
```

Asks the parent to wait for the child



# Your very first **useful** program with Pthreads

# Edge Detection Program

# Edge Detection Program Flow

Read the image file (format bmp)

Copy the image

Apply a convolution matrix (3×3)

Saves the image

# How to Read/Write the Image File

Format “bmp”

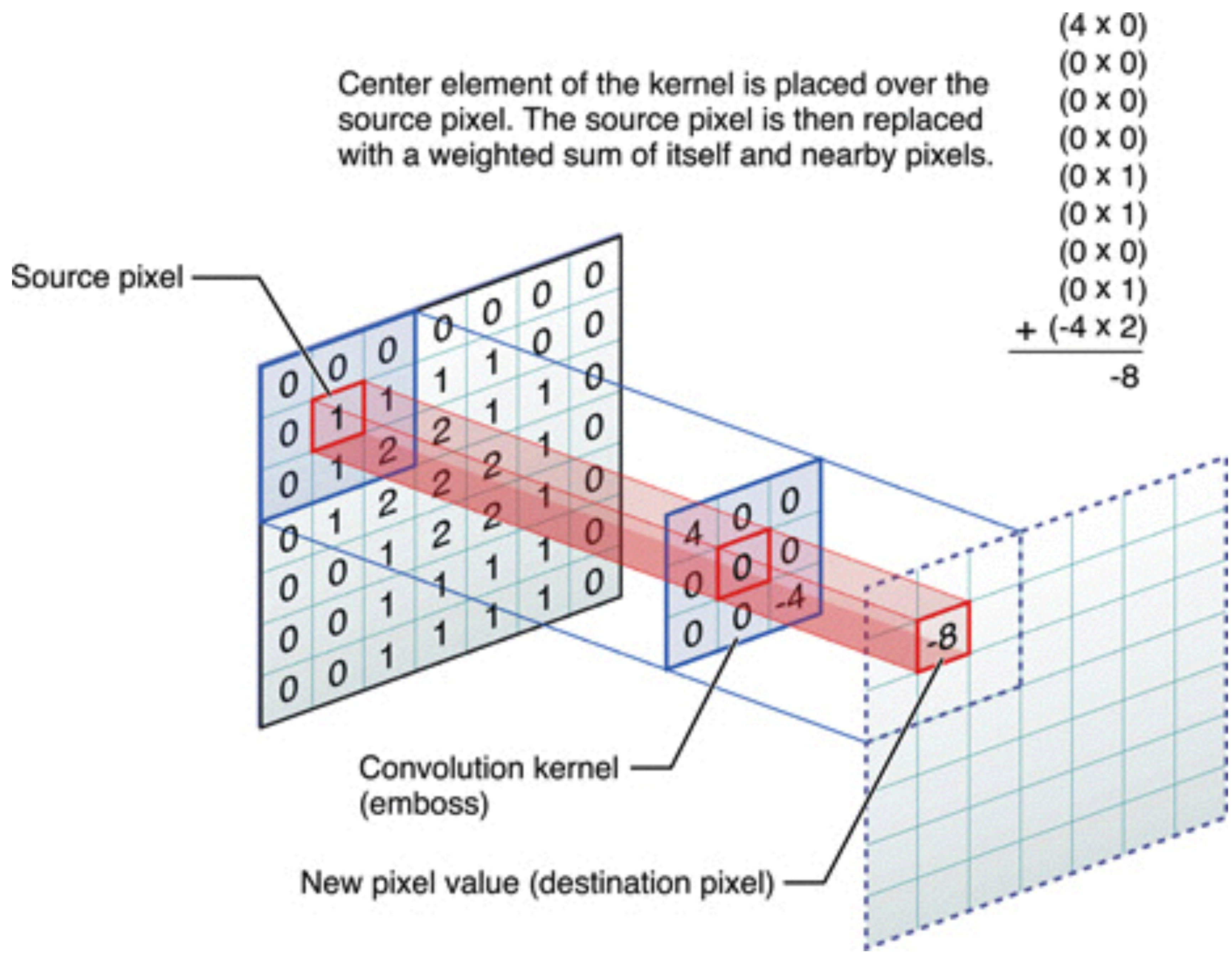
| |
|---|
| Header (54 bytes) |
| Pixels (row major) |

Pixel (32 bits)

| Red | Green | Blue | void |
|---|---|---|---|
| 8 bits | 8 bits | 8 bits | 8 bits |

Always 0

# What is a Convolution Matrix




Source: http://stats.stackexchange.com/questions/114385/what-is-the-difference-between-convolutional-neural-networks-restricted-boltzma

# The Serial Version of the Program

**~/examples/serial/serial.c**

```
int main(int argc, char* argv[]) {
  int x, y, offset;
  int cp, kx, ky, px, py;

  if(argc!=3) { printf("Please specify the names of the input and output files in
parameters:\n\t  %s <input.bmp> <output.bmp>\n", argv[0]); exit(-1); }

  printf("Size of a pixel: %i\n", sizeof(bmp_pixel_t));

  unsigned char info[54];
  /* Reads the file and allocates the data in the heap */
  unsigned char* data = read_BMP(argv[1], info);

  if(data==NULL) { printf("Unable to open the file. Exit...\n"); return -1; }

  /* Does some stuff */
  printf("Start stuffs...\n");

  // extracts image height and width from header
  int width  = BMP_WIDTH(info);
  int height = BMP_HEIGHT(info);

  unsigned char* new_data = malloc(width*height*sizeof(bmp_pixel_t));

  bmp_pixel_t *pixel;
  for(y=1; y<height-1; y++) {
    for(x=1; x<width-1; x++) {
      pixel = BMP_PIXEL(data, x,y);

      /* Applies the kernel matrix */
      for(offset=0; offset<3; offset++) {
        cp=0;

        for(kx=0; kx<3; kx++) {
          for(ky=0; ky<3; ky++) {
            px = (x+kx-1)%(width-1);
            py = (y+ky-1)%(height-1);
            // printf("%d / %d\n", px, py);
            cp += ((int)BMP_PIXEL_COMPONENT(data,px,py, offset)) * kernel_matrix[kx][ky];
          }
        }

        BMP_PIXEL_COMPONENT(new_data,x,y, offset) = (unsigned char)(cp&0xff);
      }
    }
  }

  printf("... end.\n");
  /* Writes the BMP to a file and frees the data from the heap */
  if(write_and_free_BMP(argv[2], new_data, info)==-1) {
    printf("Unable to write the file. Exit...\n"); return -1;
  }

  free(data);

  return 0;
}
```

The main function only

# The Serial Version of the Program

**~/examples/serial/serial.c**

```c
int main(int argc, char* argv[]) {
  int x, y, offset;
  int cp, kx, ky, px, py;

  if(argc!=3) { printf("Please specify the names of the input and output files in parameters:\n\t  %s <input.bmp> <output.bmp>\n", argv[0]); exit(-1); }

  printf("Size of a pixel: %i\n", sizeof(bmp_pixel_t));

  unsigned char info[54];
  /* Reads the file and allocates the data in the heap */
  unsigned char* data = read_BMP(argv[1], info);

  if(data==NULL) { printf("Unable to open the file. Exit...\n"); return -1; }

  /* Does some stuff */
  printf("Start stuffs...\n");

  // extracts image height and width from header
  int width  = BMP_WIDTH(info);
  int height = BMP_HEIGHT(info);

  unsigned char* new_data = malloc(width*height*sizeof(bmp_pixel_t));

  bmp_pixel_t *pixel;
  for(y=1; y<height-1; y++) {
    for(x=1; x<width-1; x++) {
      pixel = BMP_PIXEL(data, x,y);

      /* Applies the kernel matrix */
      for(offset=0; offset<3; offset++) {
        cp=0;

        for(kx=0; kx<3; kx++) {
          for(ky=0; ky<3; ky++) {
            px = (x+kx-1)%(width-1);
            py = (y+ky-1)%(height-1);
            // printf("%d / %d\n", px, py);
            cp += ((int)BMP_PIXEL_COMPONENT(data,px,py, offset)) * kernel_matrix[kx][ky];
          }
        }

        BMP_PIXEL_COMPONENT(new_data,x,y, offset) = (unsigned char)(cp&0xff);
      }
    }
  }
  printf("... end.\n");
  /* Writes the BMP to a file and frees the data from the heap */
  if(write_and_free_BMP(argv[2], new_data, info)==-1) {
    printf("Unable to write the file. Exit...\n"); return -1;
  }

  free(data);

  return 0;
}
```

Loads the bmp file

Applies the convolution matrix

Writes the bmp file

# Compile / Link / Execute

① Compile and link the program

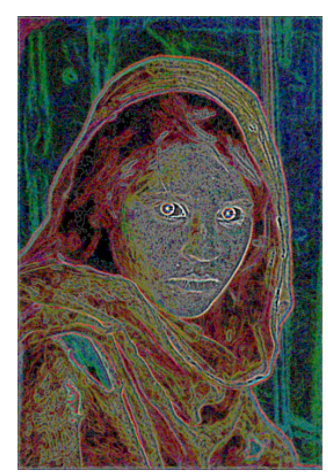
```
$> gcc serial.c -lpthread –o serial.out
```

② Execute

```
$> ./serial ~/examples/img/afghan.bmp afghan.out,bmp
```

afghan.bmp

afghan.out.bmp



# Exercise / Homework

- Execute the serial program. Try with afghan and afghan_blur. Which one looks the best ?
- Try other convolution matrices.

Defined at the top of the file

- Modify the program so that it executes with two worker threads. Use data-parallelism: