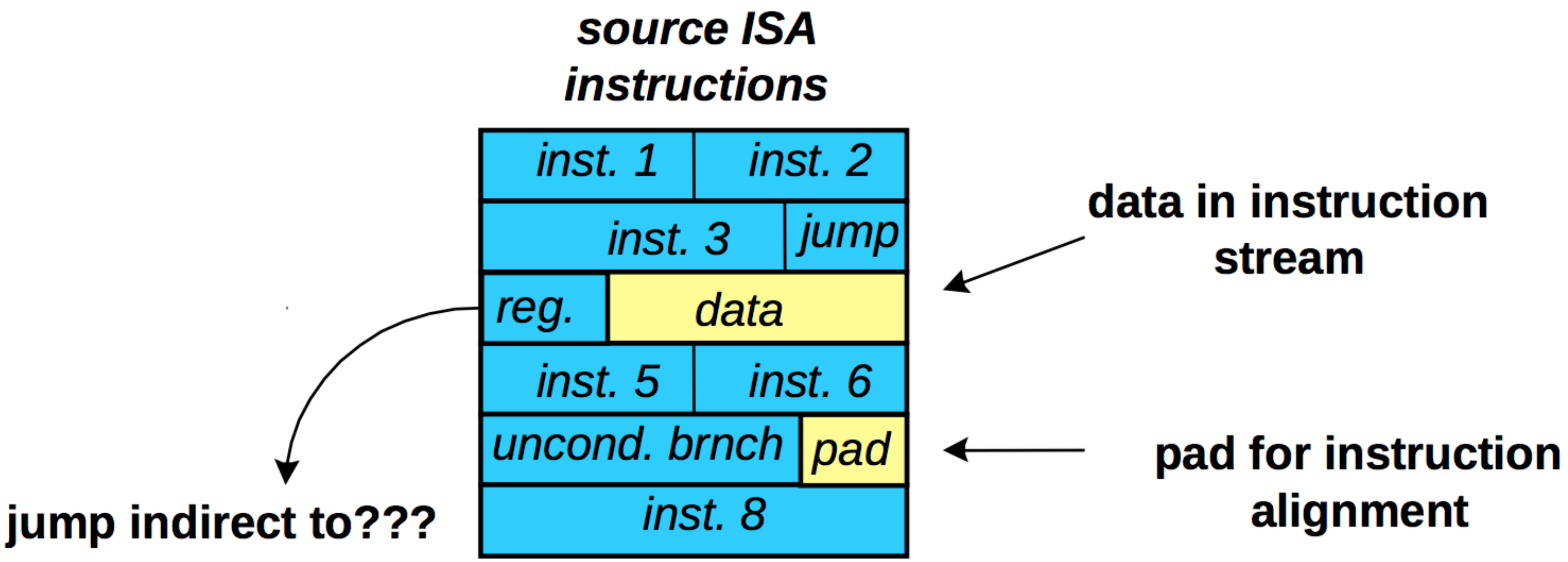
CS 206

Lecture 23 – Runtimes Wrap, Concurrency

Code Location and Code Discovery

- Code Discovery Problem
  - variable-length (CISC) instructions
  - indirect jumps
  - data interspersed with code
  - padding instructions to align branch targets



http://www.ittc.ku.edu/~kulkarni/teaching/EECS768/slides/chapter2.pdf

Code Location and Code Discovery

- Code Location Problem
  - Mapping of the source PC to destination PC for indirect jumps is difficult
    - indirect jump addresses in the translated code still refer to source addresses for indirect jumps

x86 source code

```
movl  %eax, 4(%esp)  ;load jump address from memory
jmp   %eax           ;jump indirect through %eax
```

Static vs Dynamic Binary Translation

- Precise discovery of the legacy code poses a problem, especially for static translation.
  - Given an executable file, it is not always clear what is code and what is data

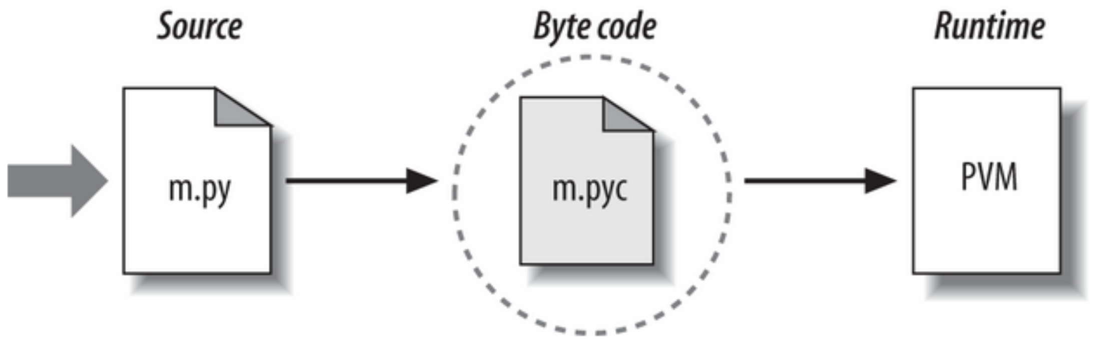
- Another problem : ***self-referential legacy code*** - code that looks at itself, for example, to perform a checksum. Because of the nature of the code, a copy of the legacy program counter must be maintained by the new machine and used whenever the legacy machine references the counter for anything other than an instruction fetch.

- ***Self-modifying code*** presents problems similar to finding legacy code and self-referential legacy code. Handling self-modifying code is not possible with a purely static translator.

Welcome to the Opportunities of Binary Translation (http://people.ac.upc.edu/vmoya/docs/binarytrans.pdf)

Compiling in Python

- Python source code is compiled into **bytecode**
  - bytecode is located in .pyc files
- Note : bytecodes are **NOT** expected to work between different Python virtual machines, nor to be stable between Python releases.



https://docs.python.org/3/glossary.html#term-bytecode

https://www.valeriankinyock.com/A-quick-overview-of-the-python-internals/

Concurrency

- *Sequential* programs: programs with a single active execution context
  - Fundamental to imperative programs
- *Concurrent Programs*: programs with more than one active execution context
  - Mostly used interchangeably with “parallel programs”.
- Why concurrency?
  - Performance: across multiple cores, across multiple *devices* (CPUs, GPUs etc.)
  - Capture logical program structure

Types of Parallelism

- Instruction level
  - Find instructions in a program that can be executed concurrently
- Vector Level
  - Small set of operations being performed repeatedly on all elements of data
- Thread Level
  - Came about due to end of ILP, power considerations
  - Programmer needs to incorporate it in code
- Task Level
  - Program is explicitly divided into “tasks”
  - Tasks can run on one machine or across a cluster
  - Examples - Hadoop and Spark

Examples of Parallelism

- Independent tasks

```
Parallel.For(0, 100, i => { A[i] = foo(A[i]); } );
```

- Not completely independent tasks

```
int zero_count;
public static int foo(int n) {
    int rtn = n - 1;
    if (rtn == 0) zero_count++;
    return rtn;
}
```

Race conditions: Threads are “racing” towards a goal, output of the program depends on who gets there first

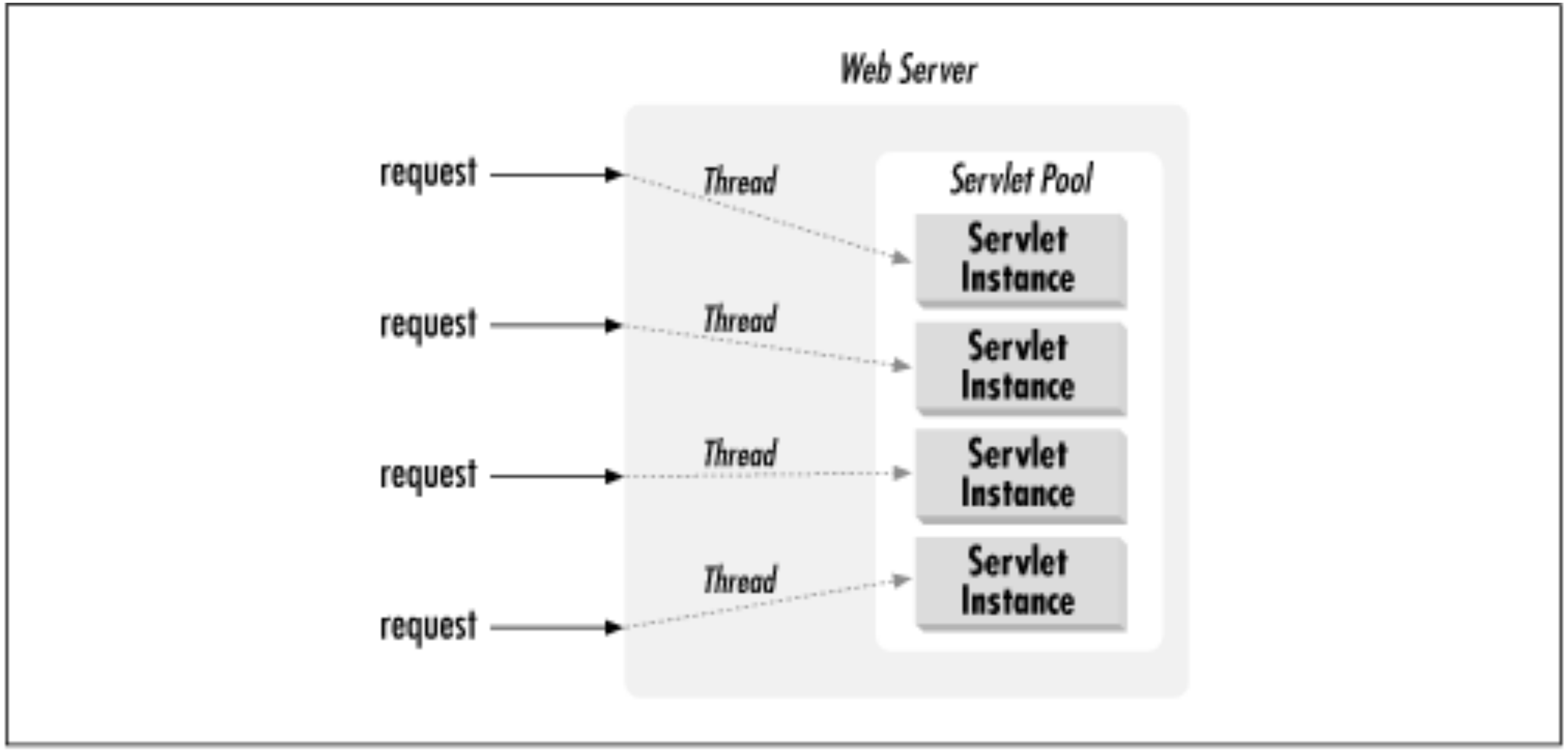
Thread 1

```
...
r1 := zero_count
r1 := r1 + 1
zero_count := r1
...
```

Thread 2

```
...
r1 := zero_count
r1 := r1 + 1
zero_count := r1
```

Example Multithreading: Server Programs

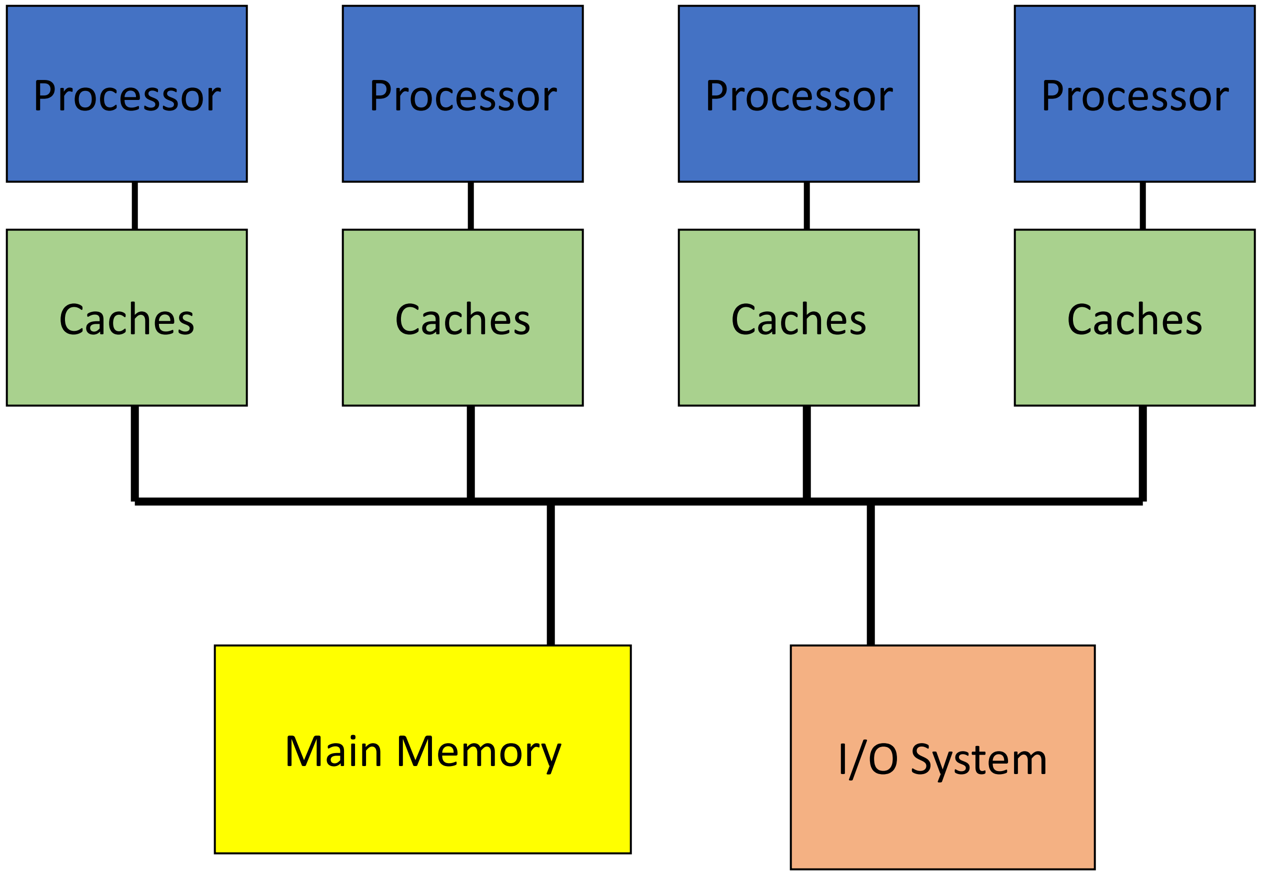


https://stackoverflow.com/questions/27765547/how-does-a-single-servlet-handle-multiple-requests-from-client-side

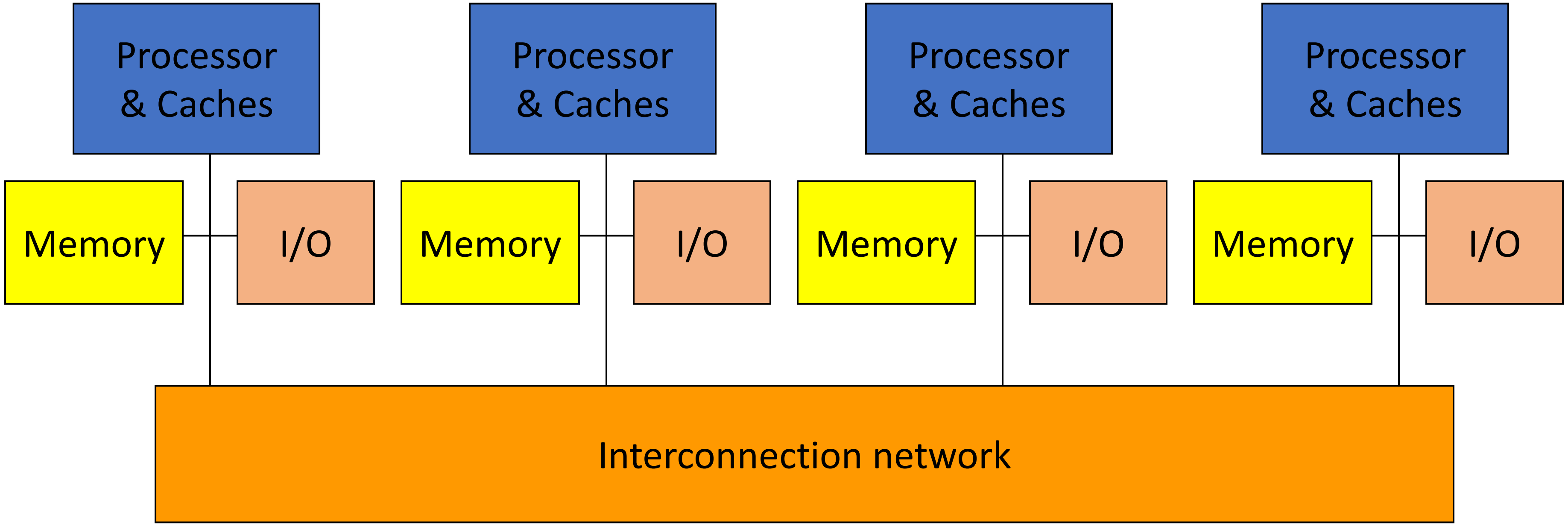
Synchronization

- *critical sections*
  - Code segments that multiple threads can execute at a given point
  - Has access to shared variables
  - Output of the program depends on who gets there first
- *mutual exclusion*
  - guarantee that if one thread is executing within the critical section, the others will be prevented from doing so
- *atomic*
  - Execute a piece of code completely, or not at all
  - Means of executing mutual exclusion

SMP/Centralized Shared Memory



Distributed Memory Multiprocessors



Types of Programming Models

Shared-memory:

- Well-understood programming model
- Communication is implicit and hardware handles protection
- Hardware-controlled caching

Message-passing:

- No cache coherence → simpler hardware
- Explicit communication → easier for the programmer to restructure code
- Sender can initiate data transfer

Concurrent Programming

- *thread* : the active entity that the programmer thinks of as running concurrently with other *threads*.
- Process
  - Sometimes known as *heavyweight* threads.
- Task
  - Typically an entire program, or a collection of them
- Synchronization: any mechanism that allows the programmer to control the relative order in which operations occur in different threads.

Thread Creation

- Algol 68

co-begin

stmt_1

stmt_2

...

stmt_n

end

OpenMP

```
#pragma omp sections
{
#   pragma omp section
    { printf("thread 1 here\n"); }

#   pragma omp section
    { printf("thread 2 here\n"); }
}
```

Parallel For

```
#pragma omp parallel for
for (int i = 0; i < 3; i++) {
    printf("thread %d here\n", i);
}
```

Thread Implementation

- Two level implementation
  - Thread multiplexes threads on top of one or more kernel-level processes
  - Implemented as a library or language run-time package