Lecture 03: Parallel Runtime Systems

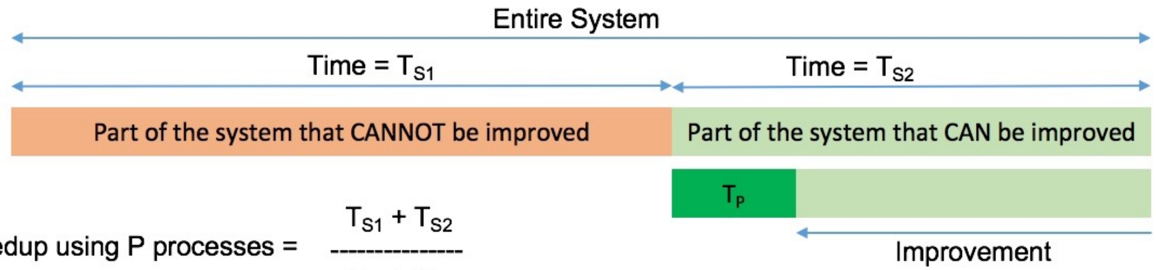
Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture: Recap



$$\text{Speedup using P processes} = \frac{T_{S1} + T_{S2}}{T_{S1} + T_P}$$

- Free lunch is now over!
  - Multicore processors everywhere
- Amdahl's law
- Explicit multithreading
- Thread synchronization
- Functional parallelism

```
int main(int argc, char *argv[]) {
  uint64_t result;
  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    uint64_t x, y;
    std::thread T1([&]() { x = fib(n-1); });
    // main can continue executing
    y = fib(n-2);
    // Wait for the thread to terminate.
    T1.join();
    result = x + y;
  }
  cout<<"Fibonacci of "<<n<<" is "<<result<<endl;
  return 0;
}
```

```
std::mutex _mtx;
std::condition_variable _cv;
bool available=false;
```

Consumer

```
L1: std::unique_lock lck(_mtx);
L2: _cv.wait(lck, [ ]() { return available;});
L3: consume_data();
L4: available=false;
```

Producer

```
L5: std::unique_lock lck(_mtx);
L6: if(!available) {
L7:     create_data();
L8:     available=true;
L9:     _cv.notify_one(lck);
L10: }
```

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f1 = std::async([=](){ return fib(n-1); });
    std::future<uint64_t> f2 = std::async([=](){ return fib(n-2); });
    //get will block until result is ready
    return f1.get() + f2.get();
  }
}
```

Today's Lecture

- Parallel programming landscape
- Linux kernel scheduler
  - Context switching
- Parallel runtime system for task-scheduling
  - Work-sharing
  - Work-stealing

Multicore Parallel Programming Landscape

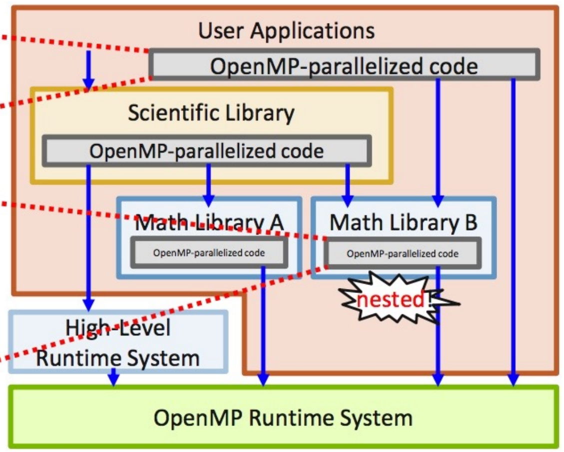
```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f1 = std::async([=](){ return fib(n-1); });
    std::future<uint64_t> f2 = std::async([=](){ return fib(n-2); });
    //get will block until result is ready
    return f1.get() + f2.get();
  }
}
```

- Newer tasks-based programming models
  - OpenMP tasks, HClib, TBB, HPX, etc.
- Thread-based programming model
  - OpenMP work-sharing loops
    - Very popular!
    - Being used extensively in several real world applications
      - Some of them were written really long ago
        - Hard to change to newer programming models
      - Nested **thread** creation
        - Creates threads exponentially

```
#pragma omp parallel for
for (i = 0; i < n; i++)
  dgemv(matrix[n], ...);
```

```
// BLAS library
void dgemv(...) {
  #pragma omp parallel for
  for (i = 0; i < n; i++)
    dgemv_seq(data[n], i);
}
```

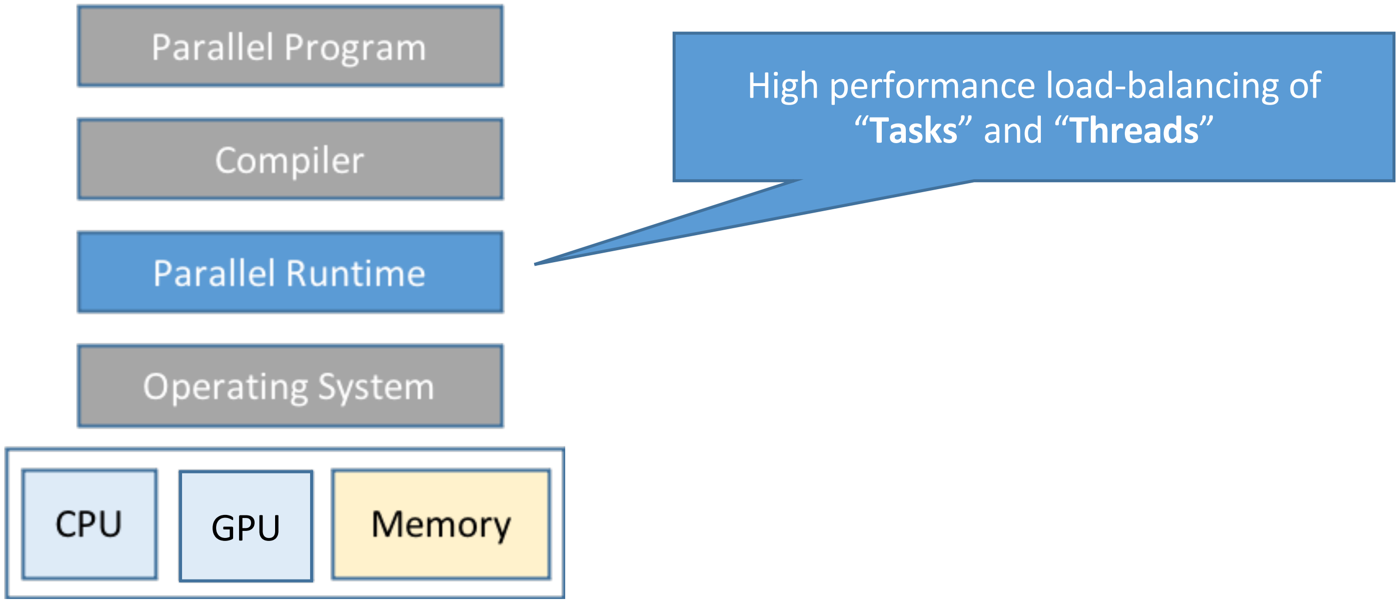
nested!



Code Example

OpenMP picture source: https://www.mcs.anl.gov/~iwasaki/pdfs/papers/PACT2019_slides.pdf

Parallel Runtime System



Multicore Parallel Programming Landscape

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint6
    std::future<uint6
    //get will block
    return f1.get() + f2.get();
  }
}
```

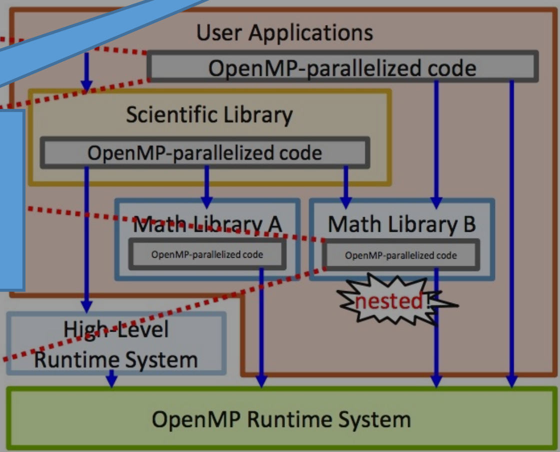
Requirement: Parallel runtime system that could schedule "tasks"

```
#pragma omp parallel for
for (i = 0; i < n; i++)
  dgemv(matrix[
```

```
// BLAS library
void dgemv(...)
  #pragma omp p
  for (i = 0; i < n; i++)
    dgemv_seq(data[n], i);
}
```

Code Example

Requirement: Parallel runtime system that could schedule "threads"



OpenMP picture source: https://www.mcs.anl.gov/~iwasaki/pdfs/papers/PACT2019_slides.pdf

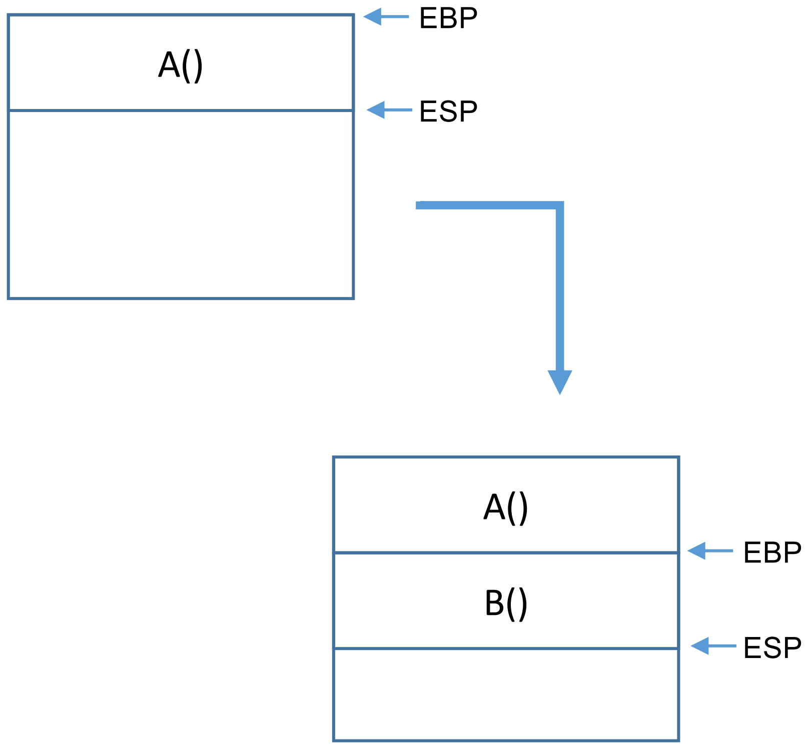
- Newer tasks-based programming models
  - OpenMP tasks, HClib, TBB, HPX, etc.
- Thread-based programming model
  - OpenMP work-sharing loops
    - Most popular
    - Being used extensively in several real world applications
      - Some of them were written really long ago
        - Hard to change to newer programming models
      - Nested **thread** creation
        - Creates threads exponentially

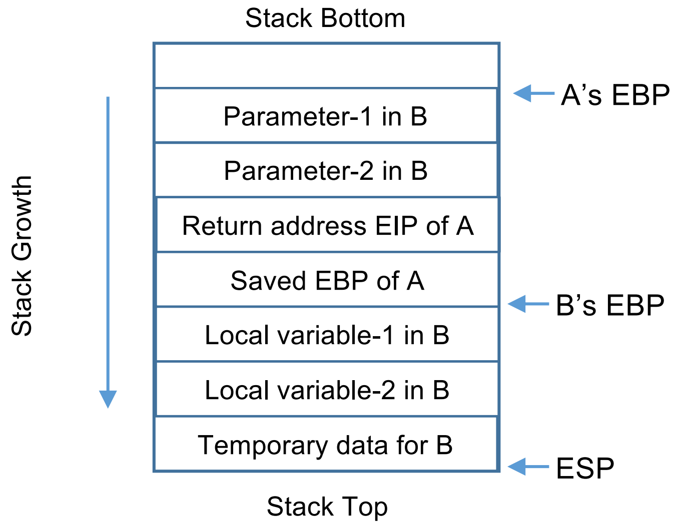
Let us first try to understand the issues with threads

Today’s Lecture

- Parallel programming landscape
- Linux kernel scheduler
  - Context switching
- Parallel runtime system for task-scheduling
  - Work-sharing
  - Work-stealing

Thread Stack






```
int main() {
   /* Create Thread-1 */
   /* Create Thread-2 */
}
void A(…) {
   /* Executed by Thread-1 */
   B(…);
}
void B(….) {
   ….
}
void C(…) {
   /* Executed by Thread-2 */
}
……
```

```
void Code_B() {
   /* method prologue */
   push EBP          // store A's EBP on stack
   move EBP, ESP     // save current stack pointer in EBP
   sub ESP, N        // N bytes reserved for local variable
   /* method body */
   /* method epilogue */
   mov ESP, EBP
   pop EBP
   pop EIP
   jmp EIP
}
```

Assembly for B

Linux Kernel Scheduler

- Several types of scheduling algorithm exists for scheduling processes and threads over the CPUs
- Latest kernel (since 2.6.23) uses Completely Fair Scheduler (CFS) by default
  - Attempts to divide the CPU time fairly (equally) among all the processes
    - Example below shows ideal fairness based scheduling for 4 processes with Burst Time (BT) as 2, 2, 4, and 6 seconds. Assume CPU slice is 4 seconds

Task	EP0	EP1	EP2	EP3
T1 (BT=2)	1	1		
T2 (BT=2)	1	1		
T3 (BT=4)	1	1	2	
T4 (BT=6)	1	1	2	2

- CFS uses virtual runtime (vruntime) variable in PCB to keep track of time a process has executed on the CPU, and it is updated at every **context switch**
  - vruntime += t * weight based on process priority
- Process with the minimum vruntime gets chance to execute earlier (CFS uses red black tree)
- If thread T1 of a process P receives CPU slice “T”, then vruntime of all the threads of P (including T1) incremented by “T” (for fairness with other processes)

Scheduling

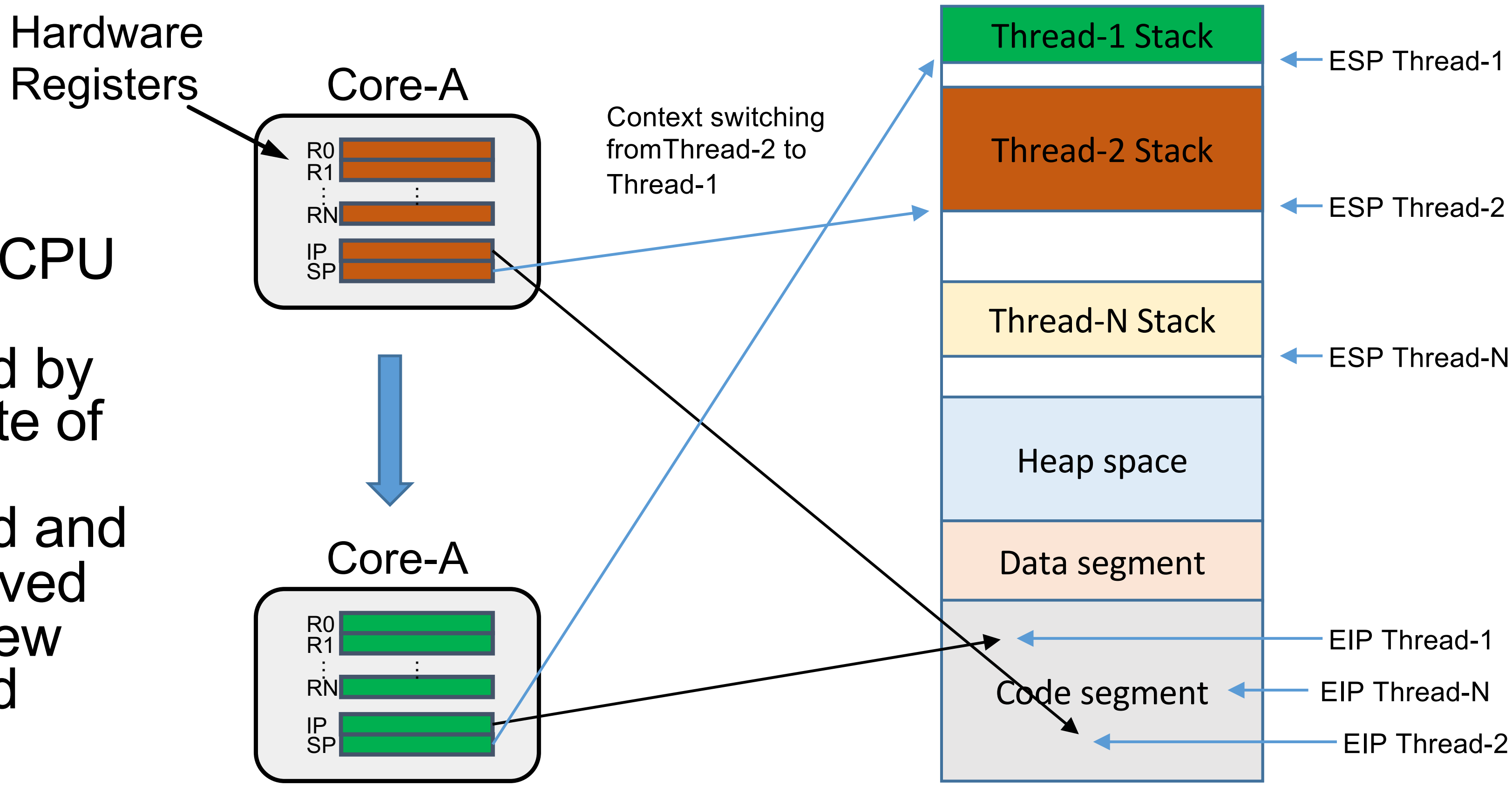
- Cooperative
  - Processes/threads decide when to yield the CPU
- Preemptive (e.g., used by Linux kernel scheduler)
  - Processes/threads preempted at blocking points
    - IO
    - Sleep
    - Wait (locking)
    - Interrupts
- Context switch required in each case!

Context Switch: Why?

- Could happen due to several reasons
  - Blocking operations (IO, synchronizations, etc.)
  - Arrival of a high priority process
  - Process terminating
  - Process has exhausted its allotted CPU slice
    - It would be the primary reason when several processes/threads are being used for running parallel program(s)

Context Switch: What?

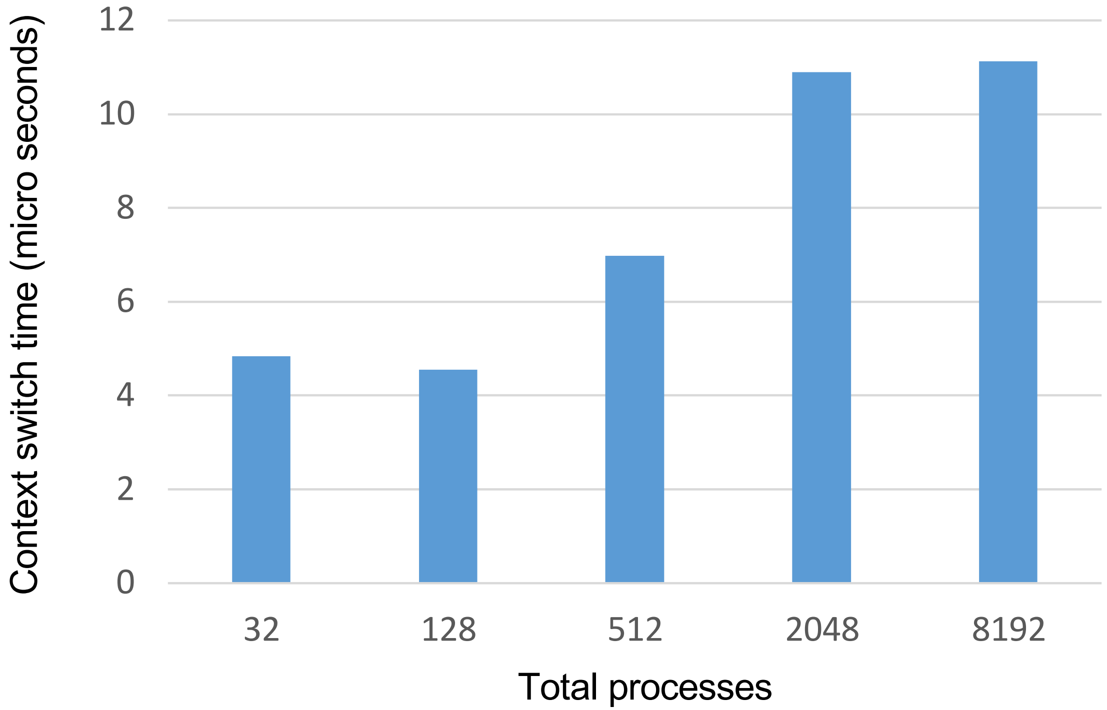
Switching the CPU to another process/thread by saving the state of the old process/thread and loading the saved state for the new process/thread



Context Switch: How?

- CPU C_0 sends timer interrupt
- Process P_A running on CPU C_0 switches from its user stack into its kernel stack
- Key registers (ESP, EIP, etc.) saved automatically by the CPU C_0 in the kernel stack of P_A
- OS saves rest of the registers in the kernel stack of P_A
- Kernel scheduler determines that process P_B will now execute next on the CPU C_0
- Scheduler now points to the kernel stack of P_B
- Reload all registers from the kernel stack of P_B and switch to its user stack

Context Switch: Cost?



- Context switch overhead measured on an AMD EPYC 32-core processor running Ubuntu 18.04.3 LTS
  - Data generated using lmbench benchmark (./lat_ctx –s 0 32 128 512 2048 8192)
- Overheads
  - Timer interrupt latency
  - Saving restoring context
  - Process scheduling
  - Reloading TLB
  - Loss in cache locality

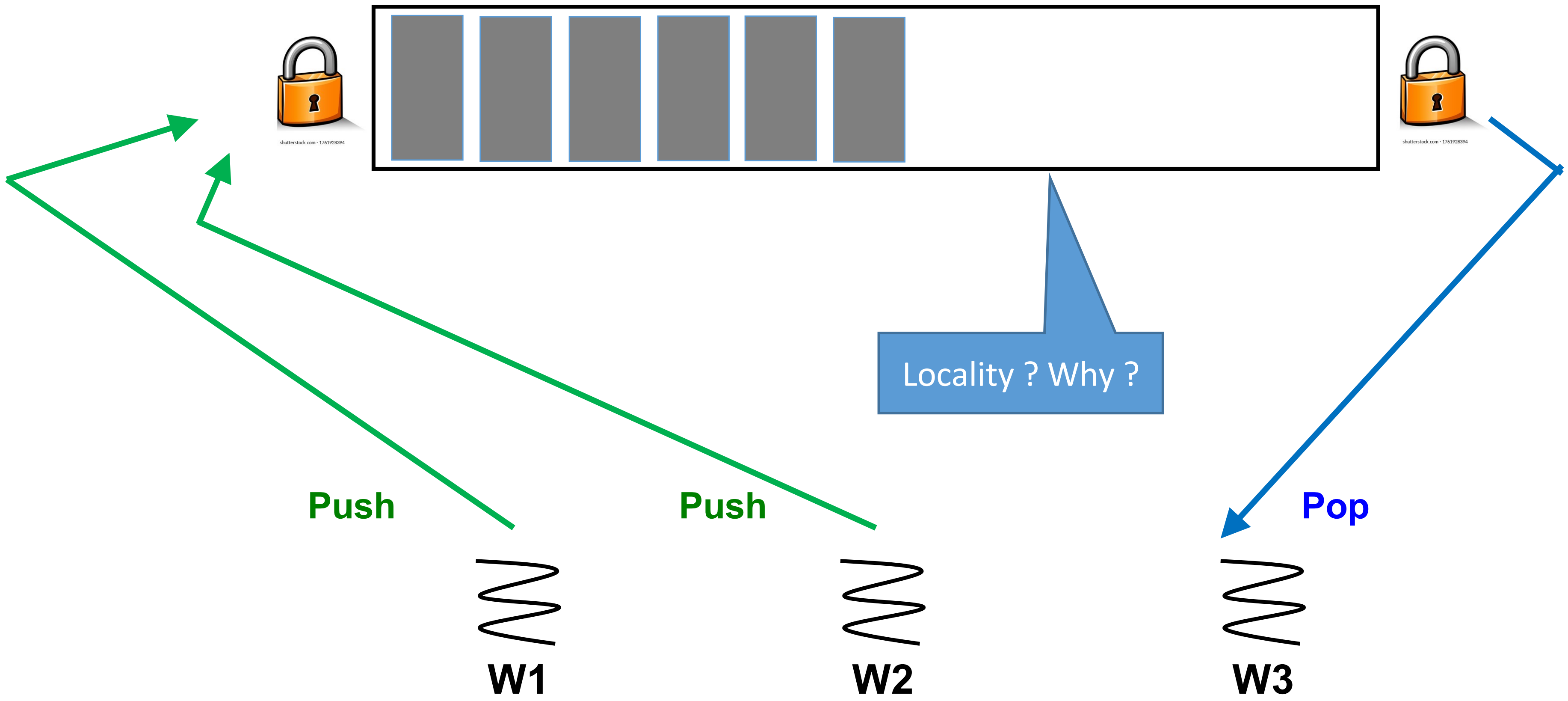
Today's Lecture

- Parallel programming landscape
- Linux kernel scheduler
  - Context switching
- Parallel runtime system for task-scheduling
  - Work-sharing
  - Work-stealing

Parallel Runtime for Task Scheduling

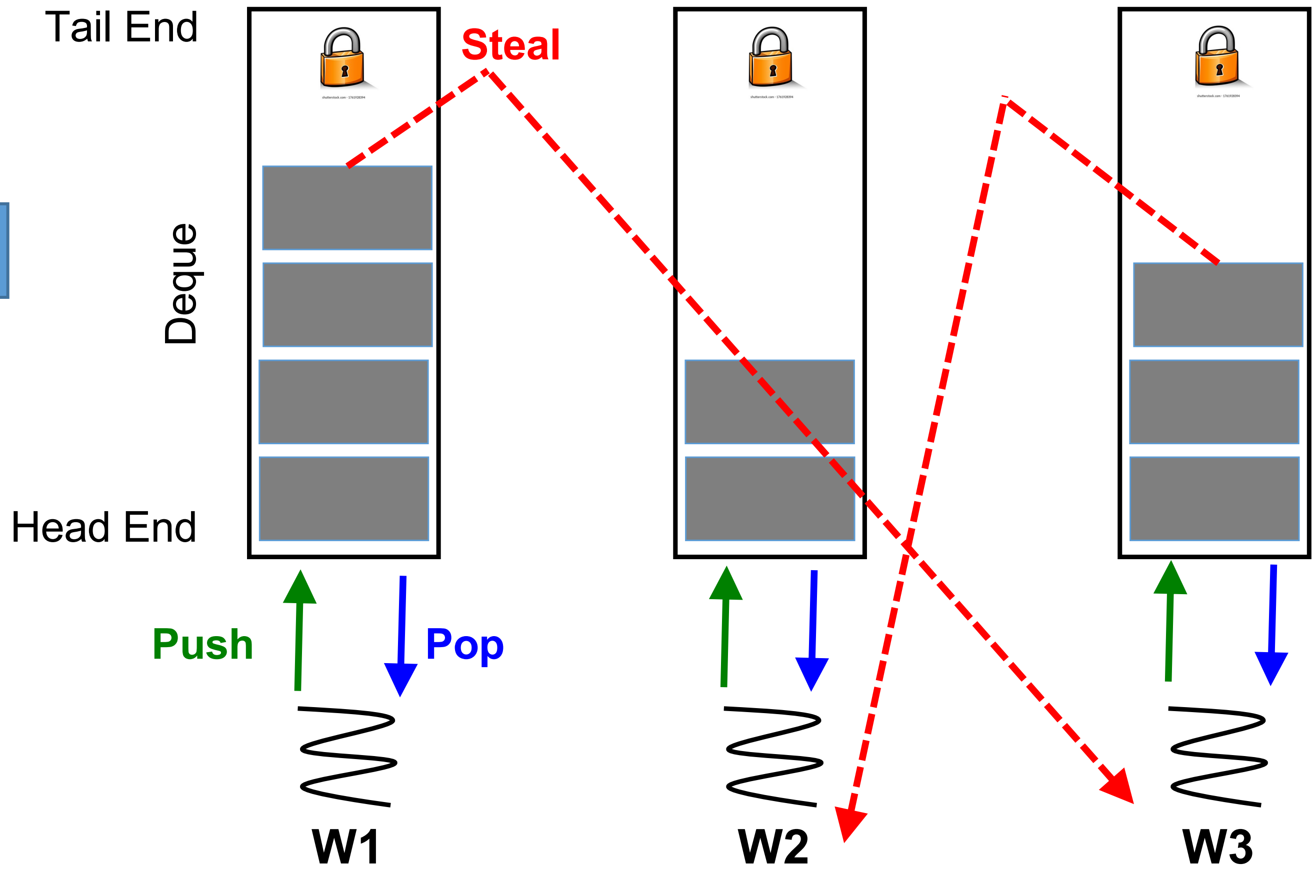
- There are several different implementations of task scheduling runtimes, but at the core all these implementations can be divided into following two categories
  - Work-sharing
  - Work-stealing

Work-Sharing Runtime System



Work-Stealing Runtime System

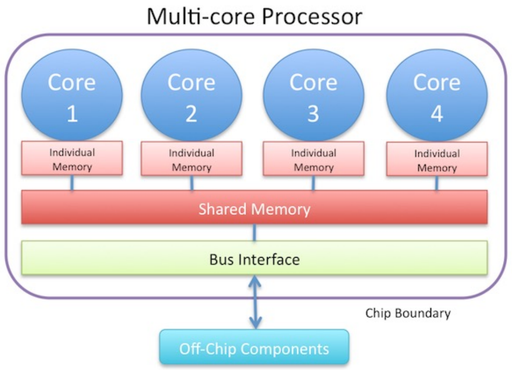
Locality ? Why ?

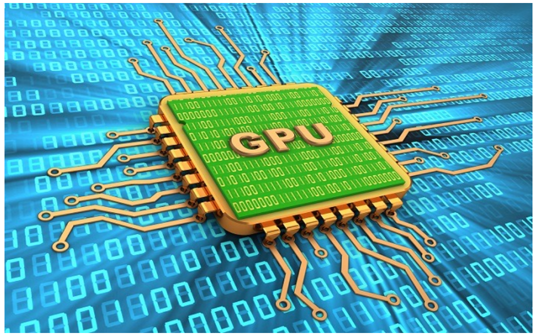


Sharing v/s Stealing

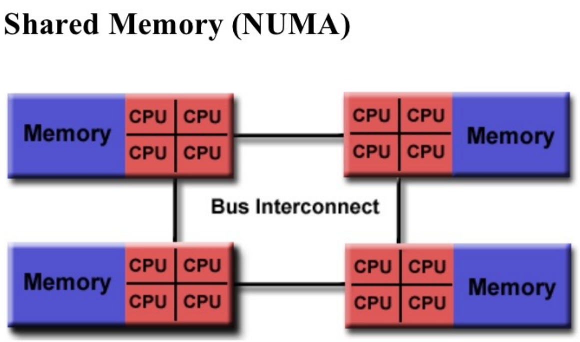
- Work-sharing
  - Busy worker re-distributes the task eagerly
  - Easy implementation through global task pool
  - Access to the global pool needs to be synchronized: scalability bottleneck
- Work-stealing
  - Busy worker pays little overhead to enable stealing
    - A lock is required for pop and steal only in case single task remaining on deque (only feasible by using atomic operations)
    - Idle worker steals the tasks from busy workers
  - Distributed task pools
  - Better scalability

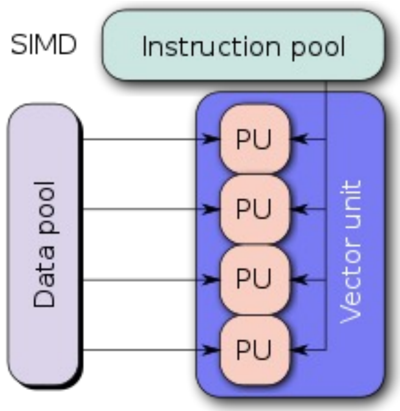
Supported on Wide Range of Architectures





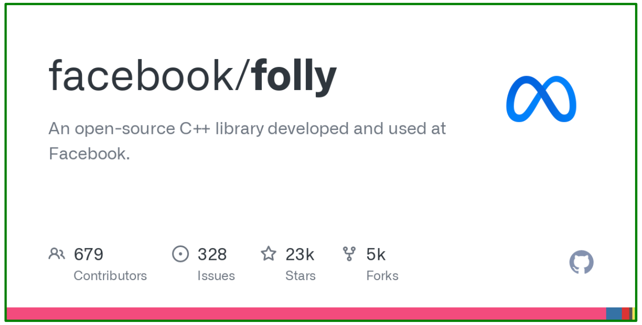
Multiprocessor System-on-Chip

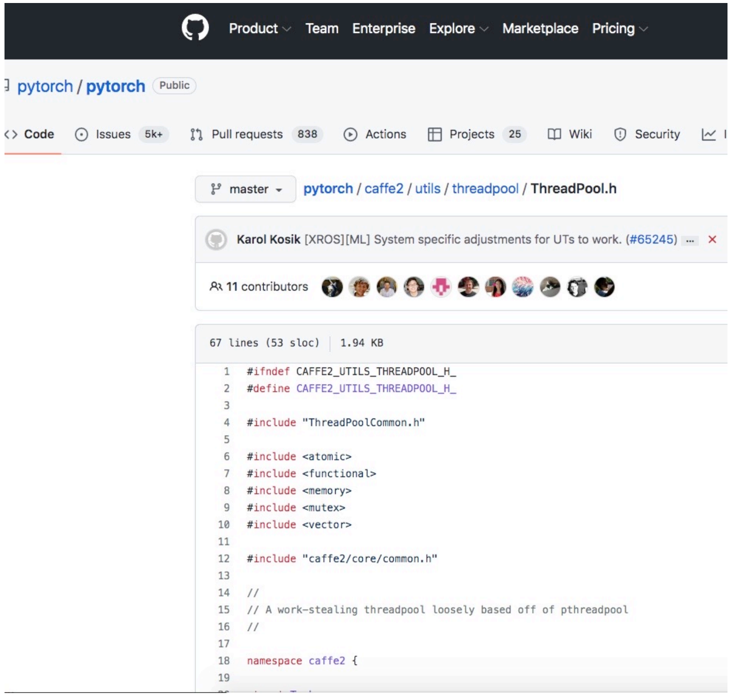


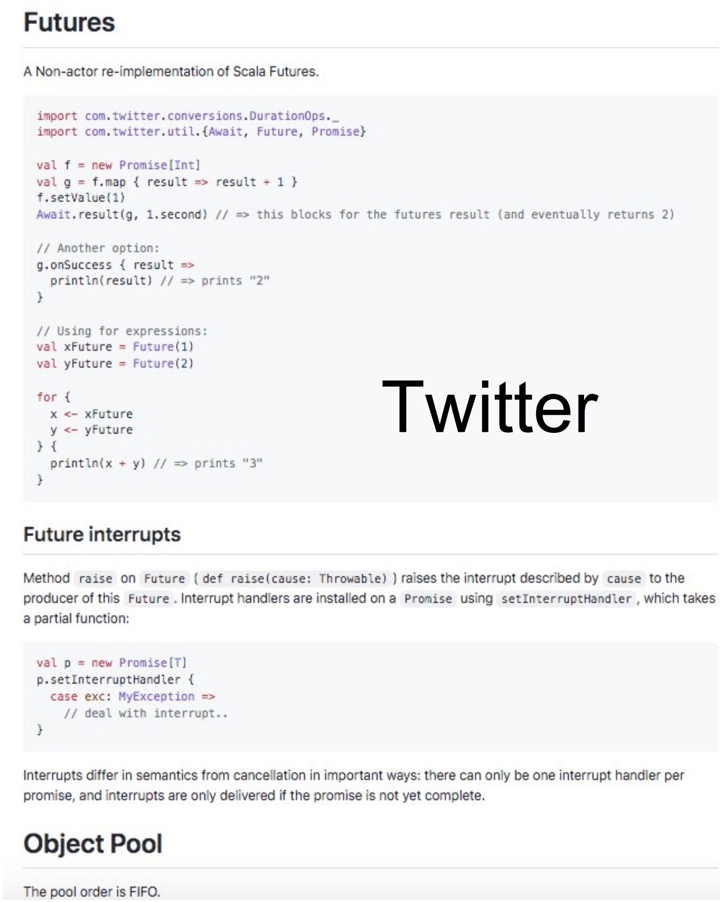


Supercomputers

Supported/Used by Several Companies/Projects







Twitter

PYTORCH

Caffe

Reading Materials

- https://gee.cs.oswego.edu/dl/papers/fj.pdf
- https://docs.kernel.org/scheduler/index.html

Next Lecture (#04)

- Context switching inside the user space
  - Boost library