Threads: putting the pieces together

CS 241

February 24, 2012

Copyright © University of Illinois CS 241 Staff

Goals for today

Pre-lecture quiz

When should you use threads?

Building a parallel application: primality testing

Pre-lecture quiz

1. get_favorites: does it work?

No. get_favorites() returns a pointer to memory which is destroyed before main() gets a chance to use it.

No. main() might try to print out the numbers before the get_favorites thread finishes.

No. 42 and 3.14159... are not, in fact, two of your favorite numbers.

No. In main(), the parameter passed to pthread_join() should just be my_fav instead of &my_fav, because my_fav is already a pointer.

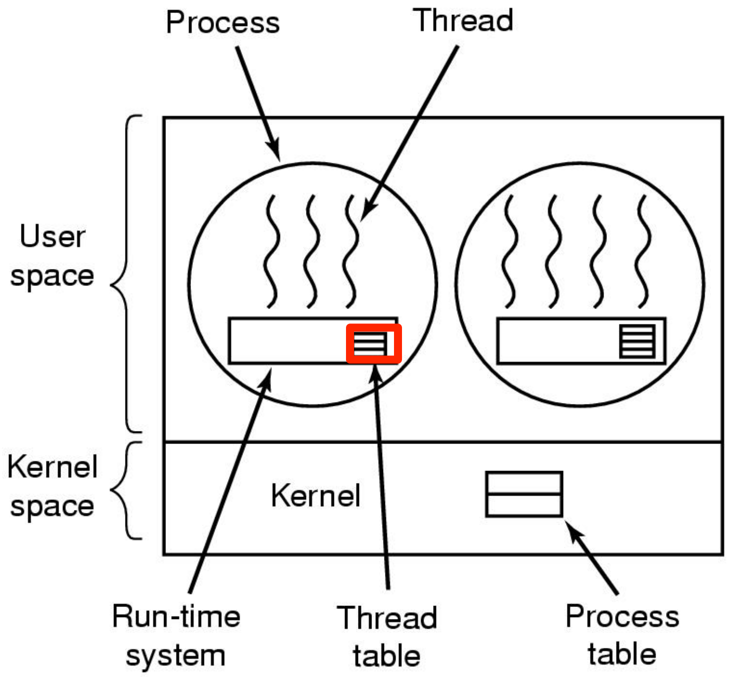
No. main() should not call free(my_fav), because main() did not allocate the memory. Remove the free() and it will work.

Yes.

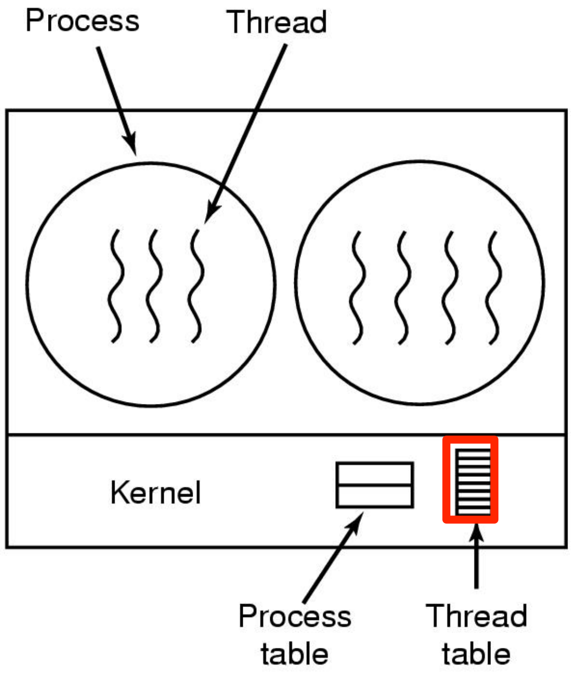
2. What’s the possible output?

3. User-level vs. kernel-level threads

User vs. Kernel Threads



User-level Threads



Kernel-level Threads

Trade-offs?

Kernel thread packages

- Each thread can make blocking I/O calls
- Can run concurrently on multiple processors

Threads in user-level

- Fast context switch
- Customized scheduling
- No need for kernel support

Q: Is kernel thread context-switching faster than process context-switching? Why or why not?

- Both need to switch to kernel mode, swap registers, change program counter, ...
- Kernel threads don't need to change virtual memory spaces

When to use threads

Why threads?

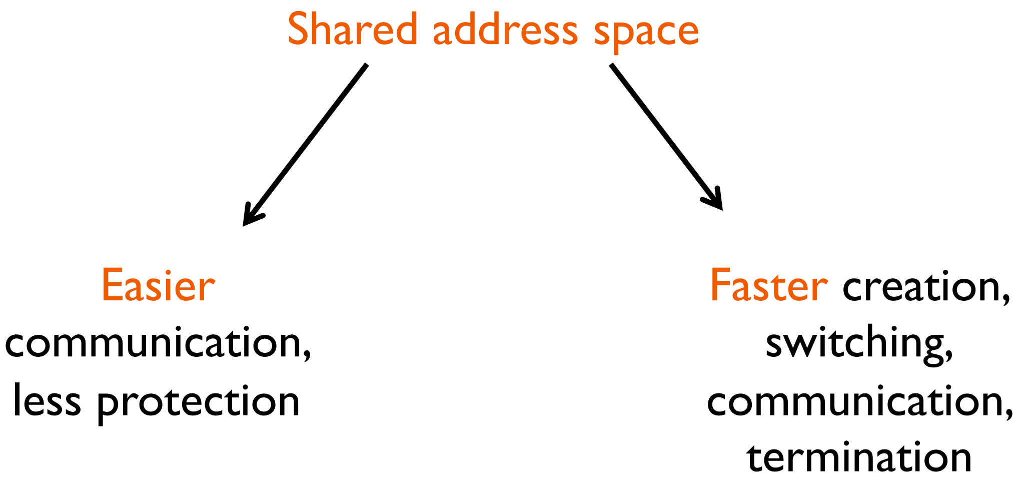
Processes do not share resources well

- Why?

Process context switching cost is high

- Why?

Therefore ... Threads: light-weight processes



Shared address space

Easier communication, less protection

Faster creation, switching, communication, termination

Tasks suitable for threading

Has multiple parallel sub-tasks

Some sub-tasks block for potentially long waits

- Reading off disk
- Waiting for user input
- Waiting for other “asynchronous” events (could arrive at any time)
- Can you implement these without threads?
    - Yes, but threads help modularize

Or, some sub-tasks use many CPU cycles

- Ideas? How about...

Putting it all together: primality testing

Primality testing goals

Decide if an integer is prime

- Input: integer
- Output: prime, or composite with factors

Exploit parallelism

- Testing primality can be slow
- My laptop has multiple cores

the primes (x axis) in binary (y axis) [MathWorld]

Attacking the problem

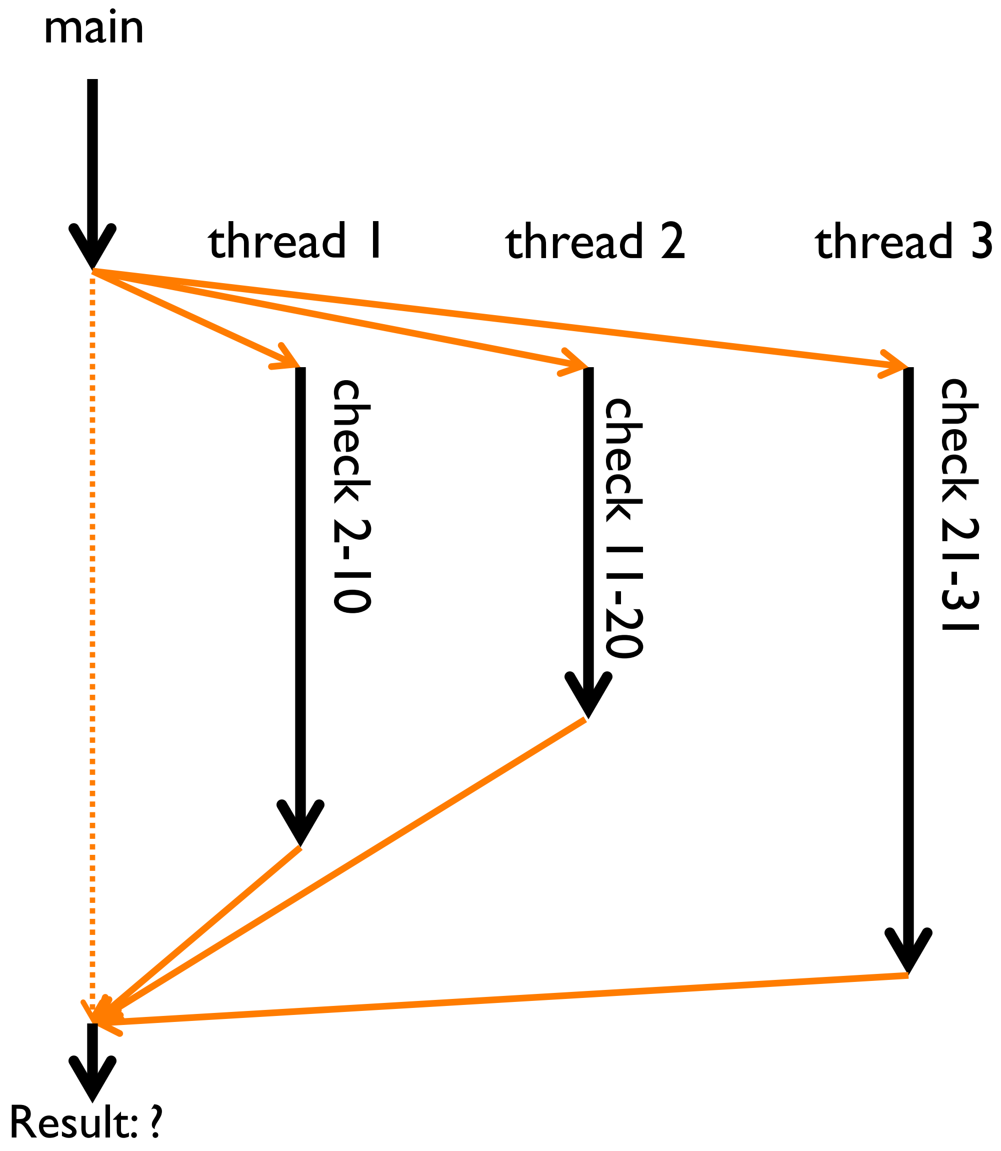
Serial algorithm

- Iterate through possible factors f, testing if f divides x

Easy to parallelize

- Lots of very small chunks of independent work
- Technical term: embarrassingly parallel

input: is 901 prime?



main

thread 1

thread 2

thread 3

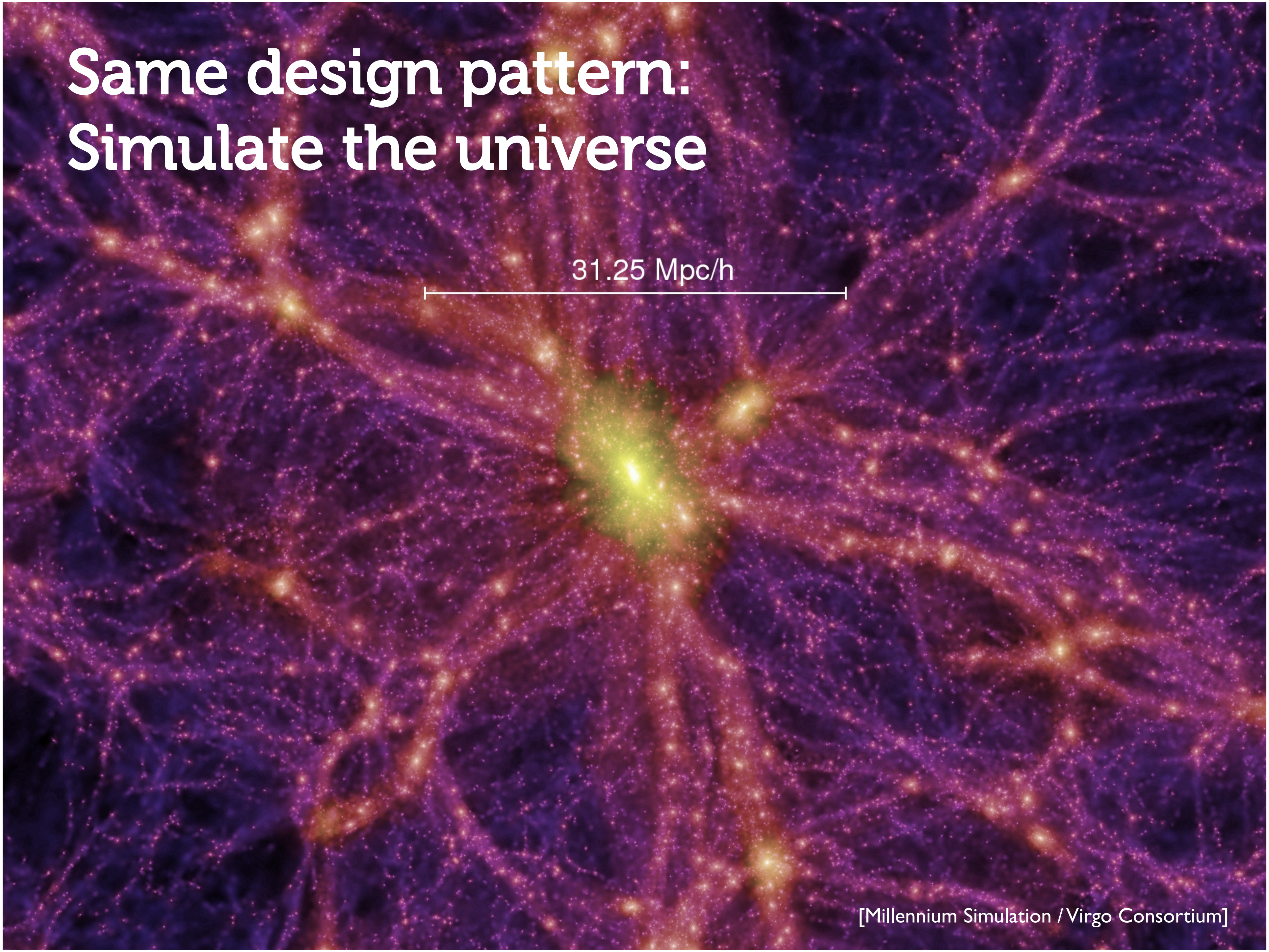
check 2-10

check 11-20

check 21-31

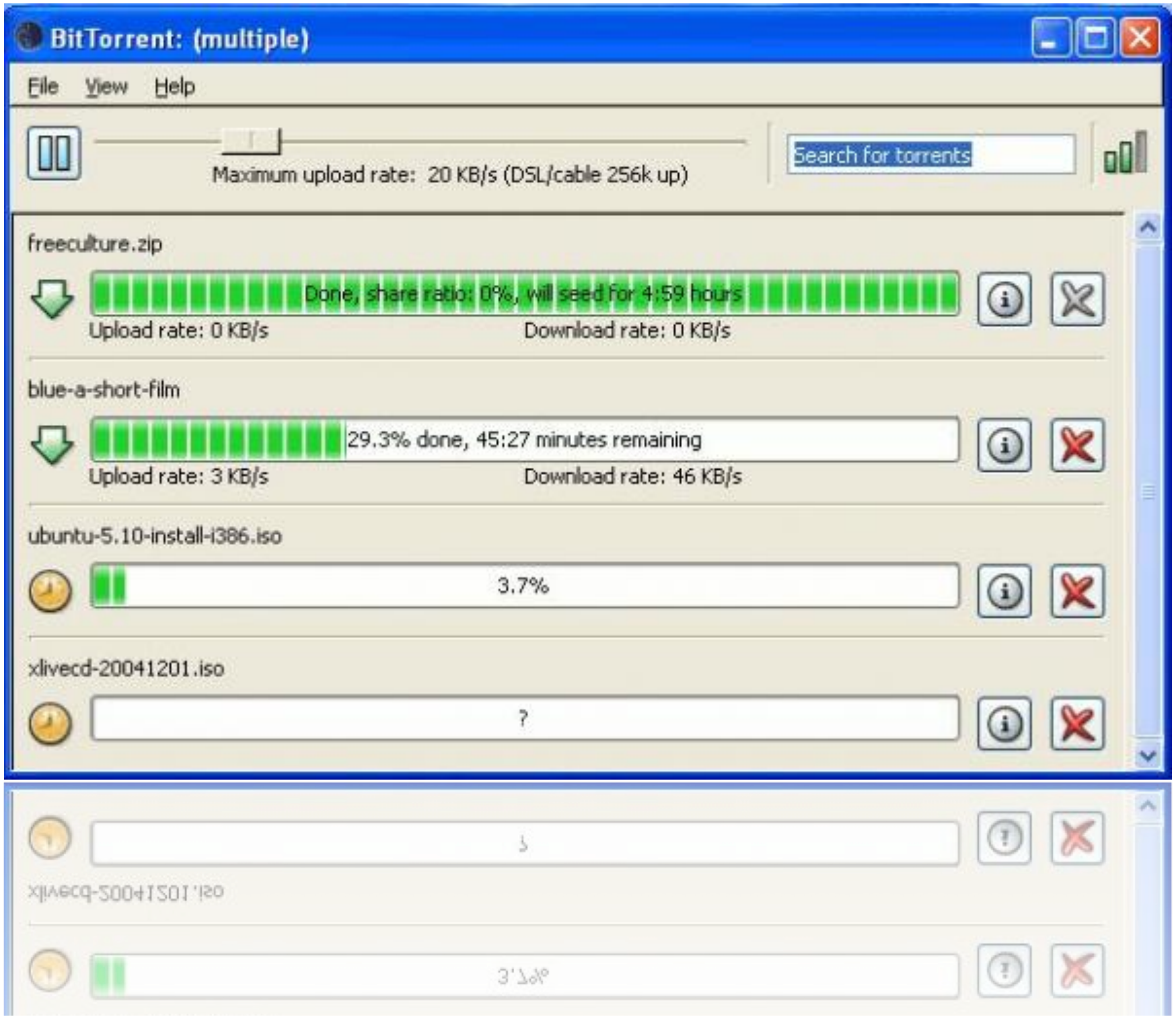
Result: ?

Same design pattern:
Simulate the universe



[Millennium Simulation / Virgo Consortium]

Same design pattern: Download movies



Maximum upload rate: 20 KB/s (DSL/cable 256k up)

freeculture.zip

Done, share ratio: 0%, will seed for 4:59 hours

Upload rate: 0 KB/s Download rate: 0 KB/s

blue-a-short-film

29.3% done, 45:27 minutes remaining

Upload rate: 3 KB/s Download rate: 46 KB/s

ubuntu-5.10-install-i386.iso

3.7%

xlivecd-20041201.iso

?

Same design pattern: Render movies

Lucasfilm data center

Decision: processes or threads?

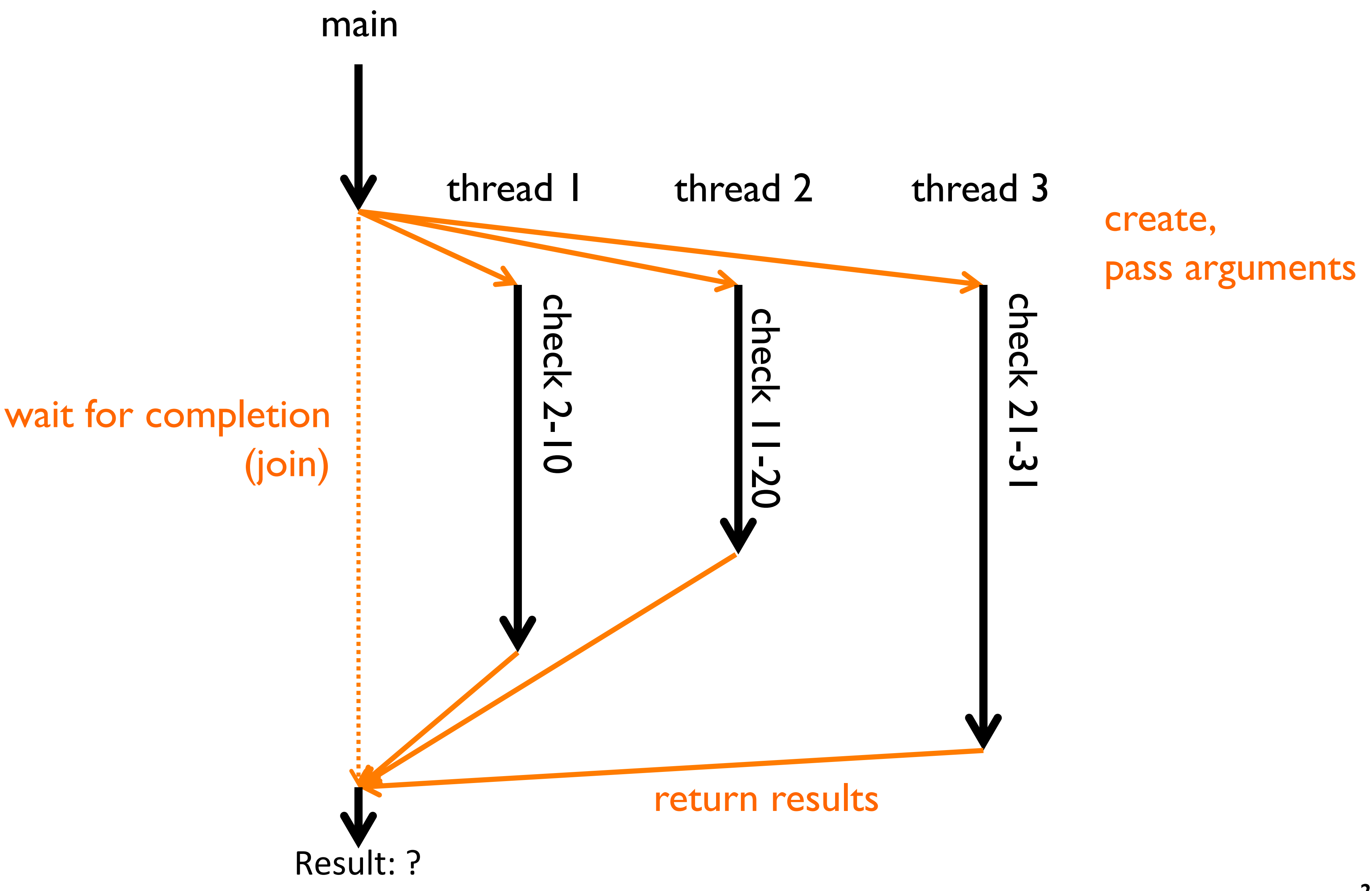
Processes

- Exploit parallelism successfully
- Separate memory space: good for protection

Threads

- Exploit parallelism successfully
- Shared memory space: good for working together

Planning the thread operations



main

thread 1

thread 2

thread 3

create,
pass arguments

check 2-10

check 11-20

check 21-31

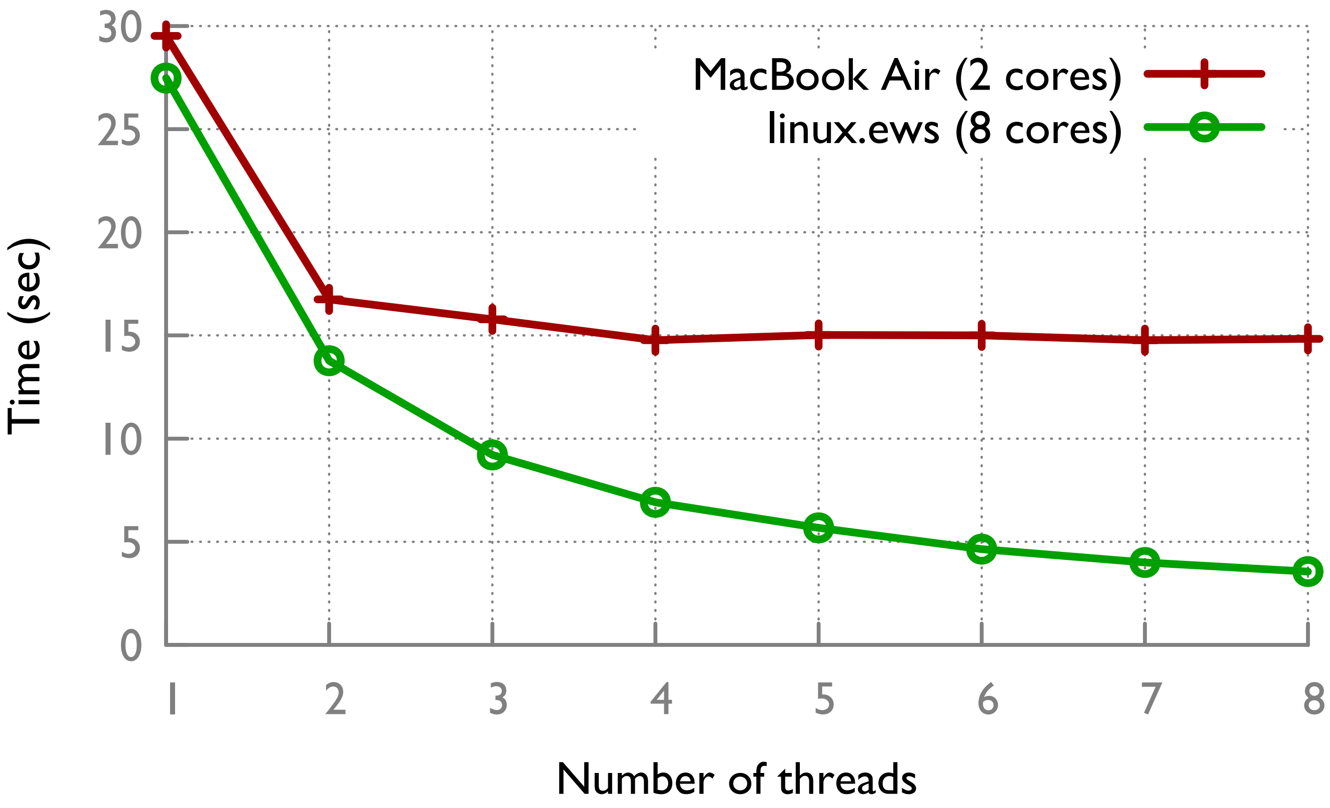
wait for completion
(join)

return results

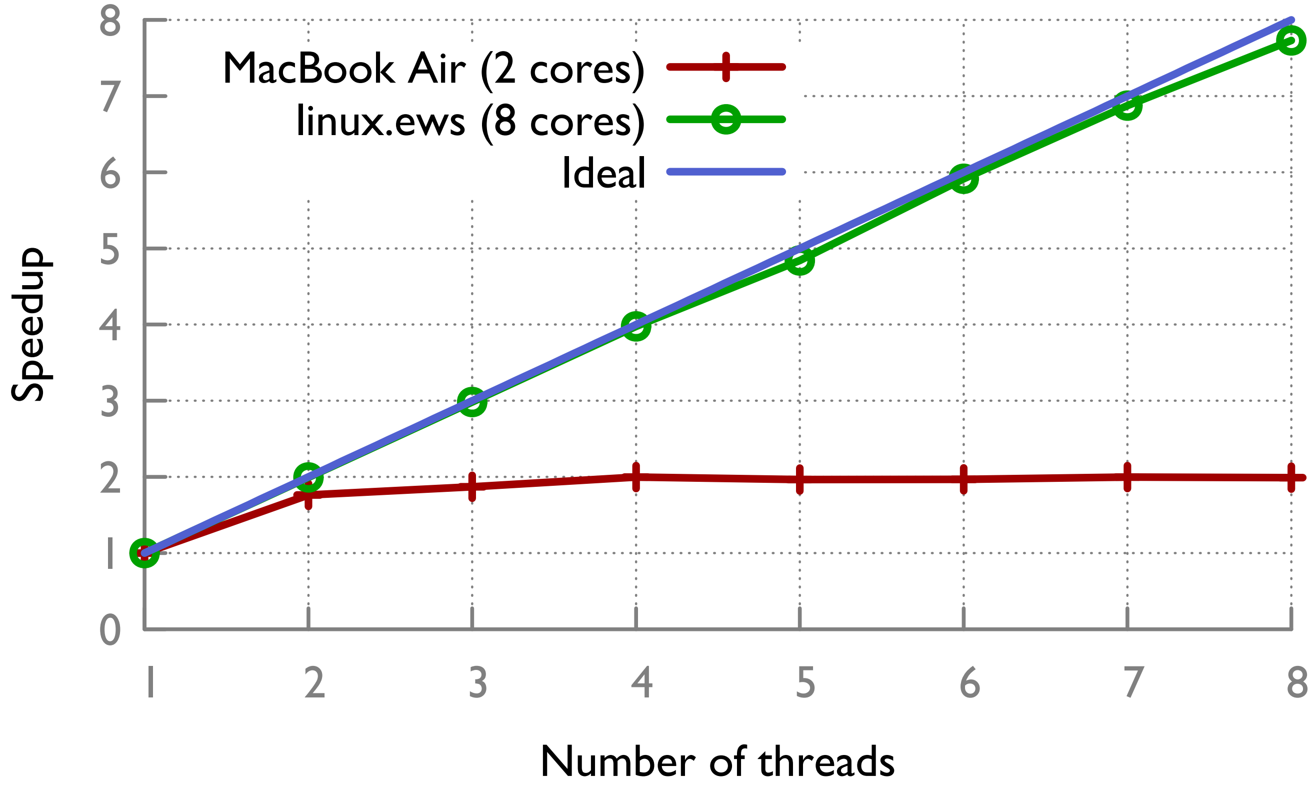
Result: ?

Away we go…

Parallel performance



Parallel performance



Next time: Scheduling

For real this time…