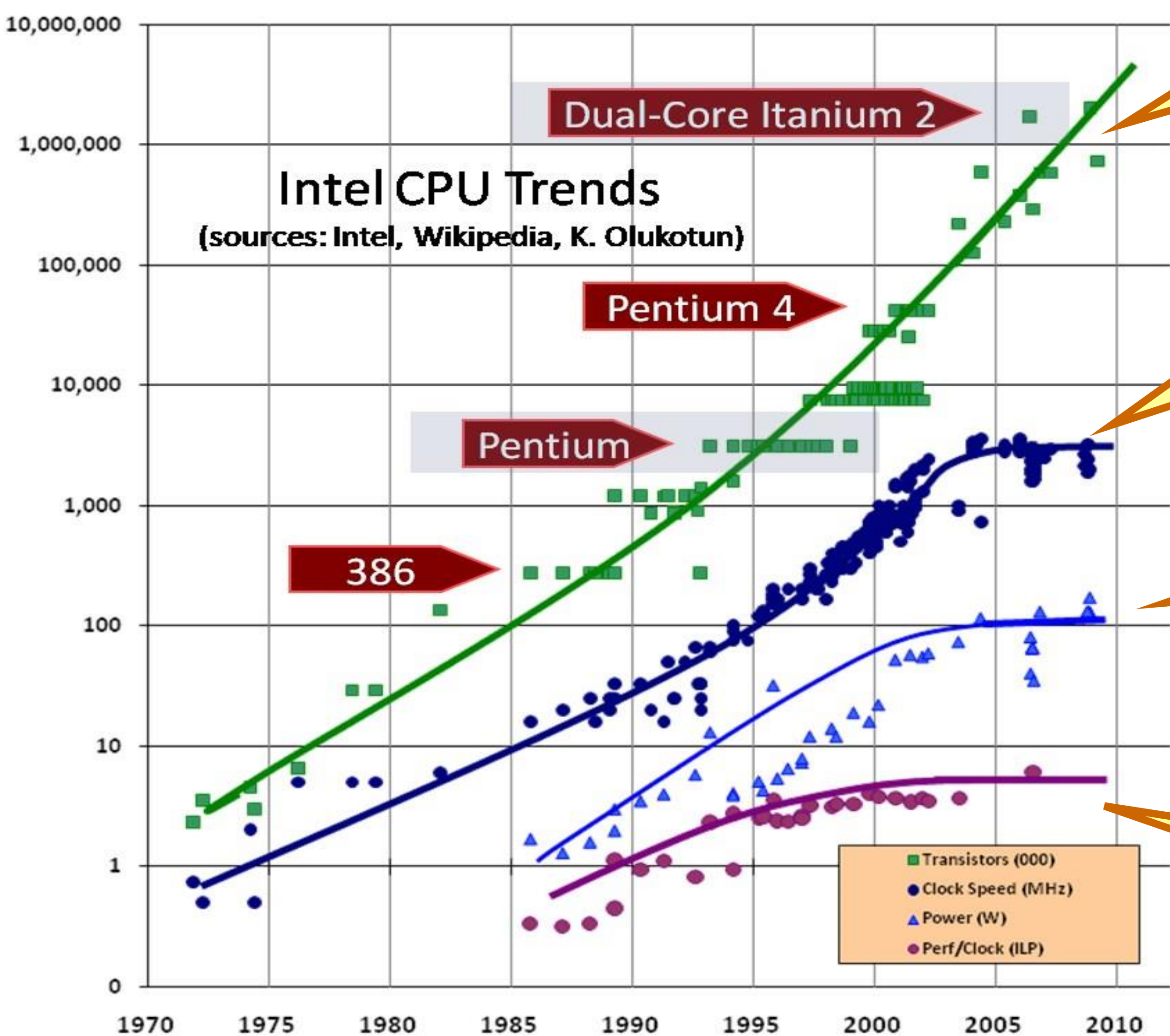
Operating Systems

Recitation 7

Plan

- Threads
  - Virtual Memory
  - Threads API
  - Examples

Moore’s Law



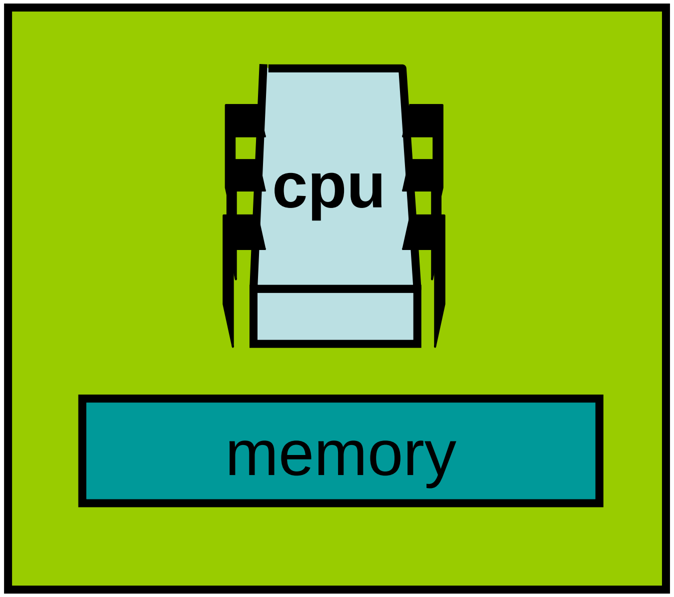
Transistor count still rising

Clock speed flattening sharply

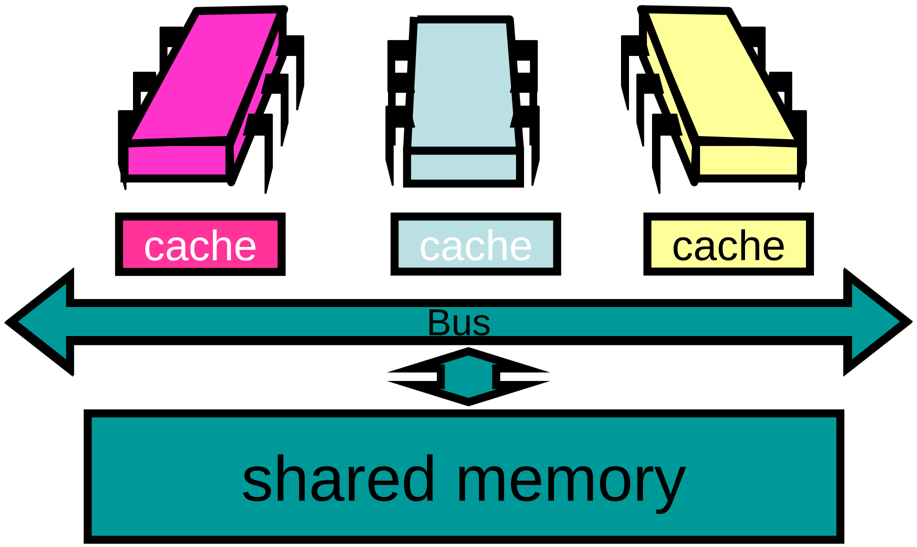
Power (Watt)

Performance

Nearly Extinct: the Uniprocesor

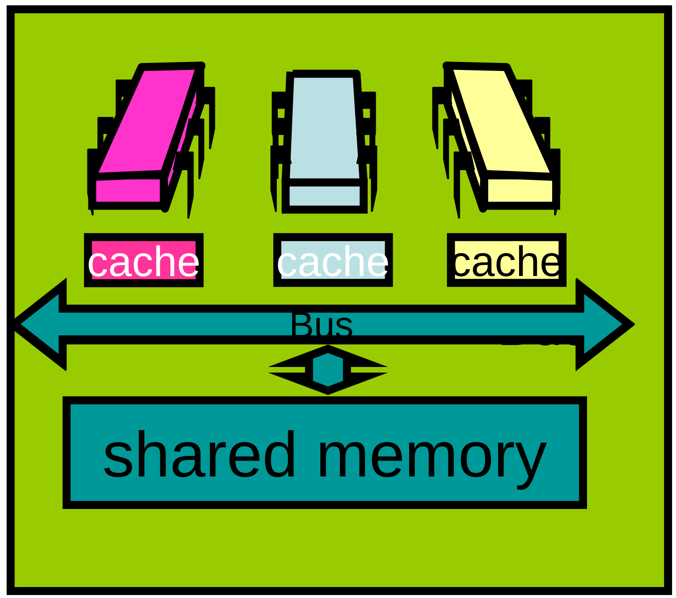


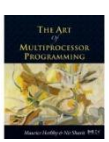
Endangered: The Shared Memory Multiprocessor (SMP)



The New Boss: The Multicore Processor (CMP)

All on the same chip





Virtual Memory

- Each process has its own heap, stack, data, code, **point of execution**

- On context switch between processes, virtual address space is changed

```
|---------------------|
|    program code     |
|                     |
|                     |
|---------------------|
|        heap         |
|                     |
|                     |
|---------------------|
|          |          |
|          v          |
|                     |
|                     |
|                     |
|                     |
|                     |
|        free         |
|                     |
|                     |
|                     |
|                     |
|                     |
|                     |
|          ^          |
|          |          |
|---------------------|
|        stack        |
|                     |
|---------------------|
```

Threads

- "Light-weight" processes
- Share same address space
  - Code
  - Heap
- Have private PC (Program Counter), and stack

```
|-----------------------|
|     program code      |
|                       |
|                       |
|-----------------------|
|         heap          |
|                       |
|                       |
|-----------------------|
|           |           |
|           v           |
|                       |
|         free          |
|                       |
|           ^           |
|           |           |
|-----------------------|
|        stack 1        |
|                       |
|-----------------------|
|         free          |
|                       |
|           ^           |
|           |           |
|-----------------------|
|        stack 2        |
|                       |
|-----------------------|
|         free          |
|                       |
|           ^           |
|           |           |
|-----------------------|
|        stack 3        |
|                       |
|-----------------------|
|         free          |
|                       |
|           ^           |
|           |           |
|-----------------------|
|        stack 0        |
|                       |
|-----------------------|
```

Threads

- What does this mean?
  - Can access all functions of the code
  - Share all allocated memory
  - Threads don’t know about locally allocated variables (int i) of other threads
    - Not entirely true…why?

```
|-----------------------|
|     program code      |
|                       |
|                       |
|-----------------------|
|         heap          |
|                       |
|                       |
|-----------------------|
|           |           |
|           v           |
|                       |
|         free          |
|                       |
|           ^           |
|           |           |
|-----------------------|
|        stack 1        |
|                       |
|-----------------------|
|         free          |
|                       |
|           ^           |
|           |           |
|-----------------------|
|        stack 2        |
|                       |
|-----------------------|
|         free          |
|                       |
|           ^           |
|           |           |
|-----------------------|
|        stack 3        |
|                       |
|-----------------------|
|         free          |
|                       |
|           ^           |
|           |           |
|-----------------------|
|        stack 0        |
|                       |
|-----------------------|
```

Scheduling

- Execution scheduled by OS
- Scheduled as processes with preemptive multitasking at kernel level
  - Not always the case!
  - Some OS/thread library implementations are user-level

Linux threads in practice

- Process starts with 1 “primary” thread
- Can add threads during execution
- When primary finishes, all threads terminate immediately
  - What happened in processes?

Threads API

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

- **thread** – pointer to store new thread id
- **start_routine** – routine for thread to run
- **arg** – arguments to pass to routine
- **attr** – various config attributes (NULL)

Threads API

```c
void pthread_exit(void *retval);
```

- Calling thread will exit and return value pointed by **`retval`**
- If called by main thread – will wait for all threads to finish**!**
  - Note - `exit()` on any thread, will exit all threads!

```c
pthread_t pthread_self(void);
```

- Returns unique thread id

pthread Library

- POSIX threads – part of IEEE POSIX standard
- Linux threads library
  - (also available in Windows)
- Need to link code to it
  - `gcc ex1.c –o ex1` ***–lpthread***

    or ***-pthread***

Code example 1

Threads API

```
int pthread_cancel(
            pthread_t th);
```

- Cancel run of thread
  - `thread` - id of cancelled thread

Threads API

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

- Wait to thread to end
  - `th` - id of waited thread
  - `thread_return` – store pointer to returned value

- Any thread can join any thread within process

- ***Code example 2***

Simple Example

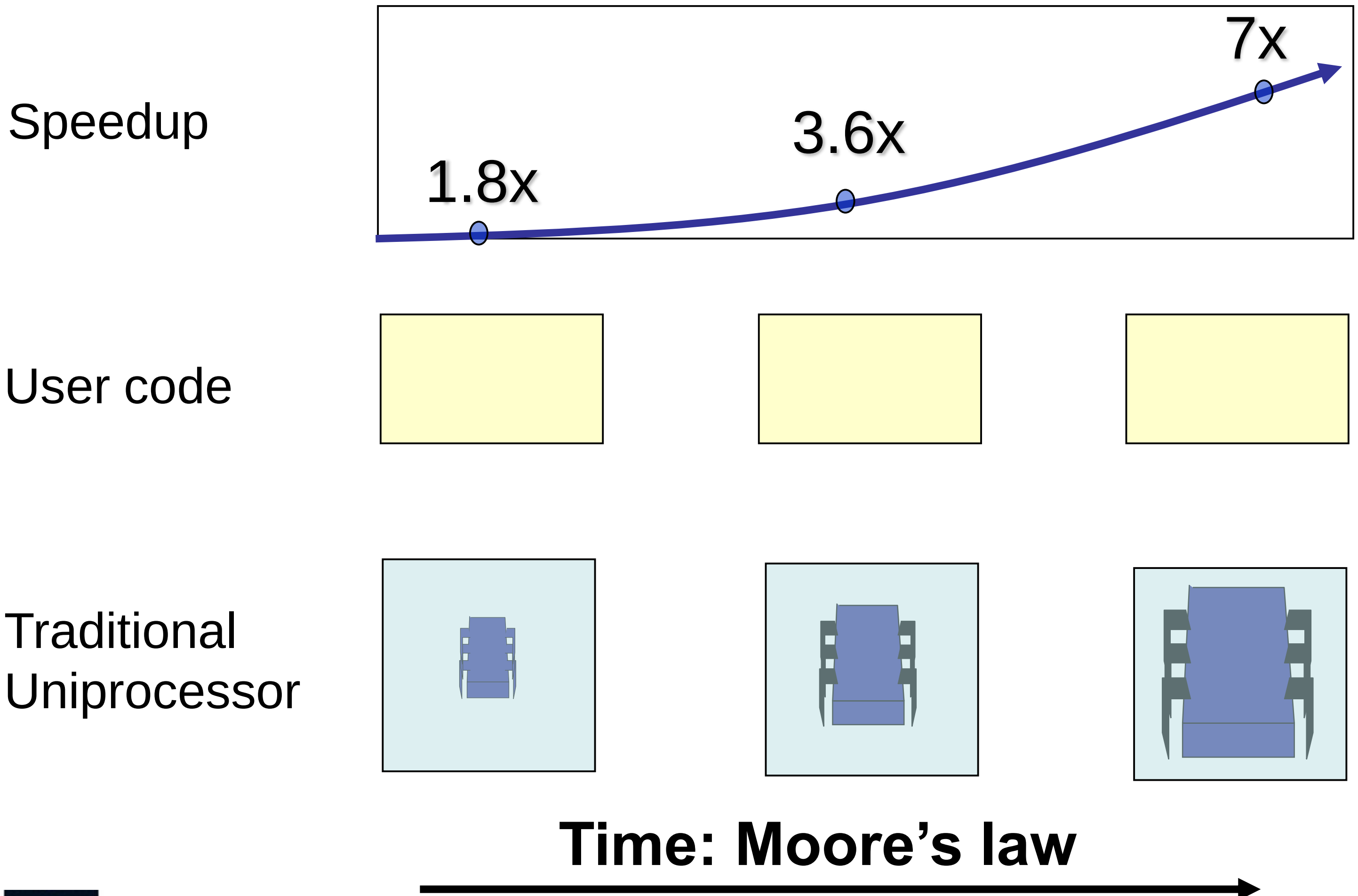
What will this do?

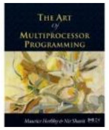
```
void* PrintVar(void* arg) {
    int* a = (int*) arg;
    printf("%d\n", *a);
}

int main(int argc, char *argv[]) {
    int a;
    a = 10;
    pthread_t thr;
    pthread_create(&thr, NULL, PrintVar, &a);
}
```

Simple Example

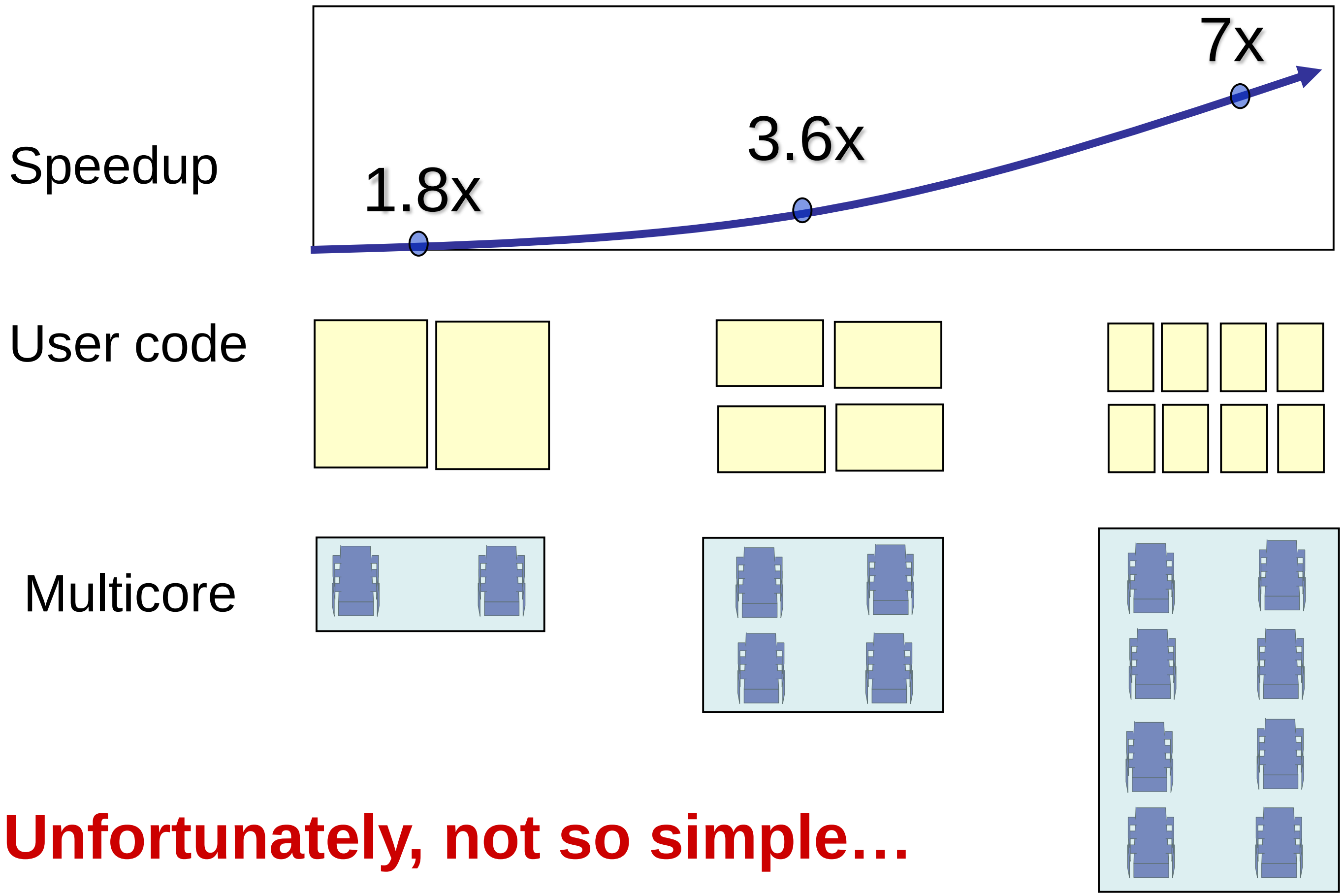
```
void* PrintVar(void* arg) {
    int* a = (int*) arg;
    printf("%d\n", *a);
}

int main(int argc, char *argv[]) {
    int a, rc;
    a = 10;
    pthread_t thr;
    pthread_create(&thr, NULL, PrintVar, &a);
    pthread_exit(&rc);
}
```

Traditional Scaling Process





Ideal Multicore Scaling Process



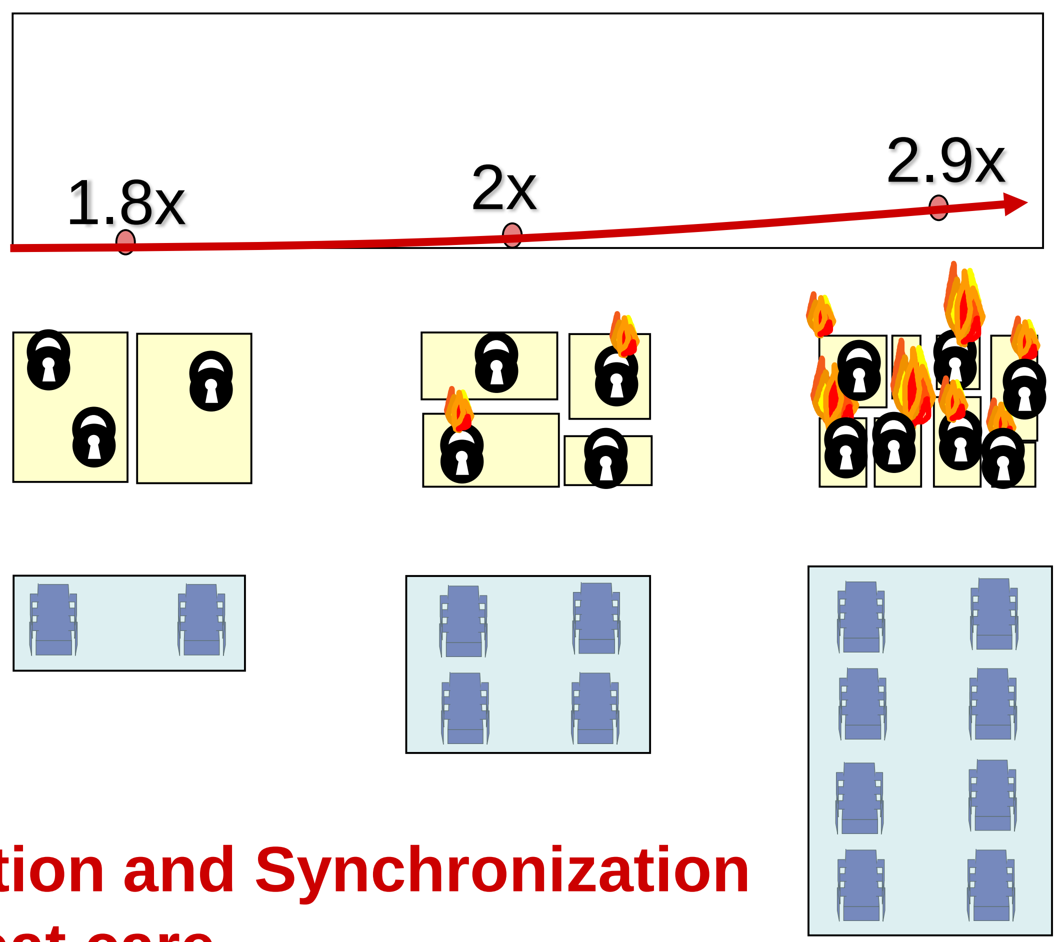
Unfortunately, not so simple…

Actual Multicore Scaling Process

Speedup

User code

Multicore



Parallelization and Synchronization require great care…

Why?

Amdahl's Law:

Speedup = 1/(ParallelPart/N + SequentialPart)

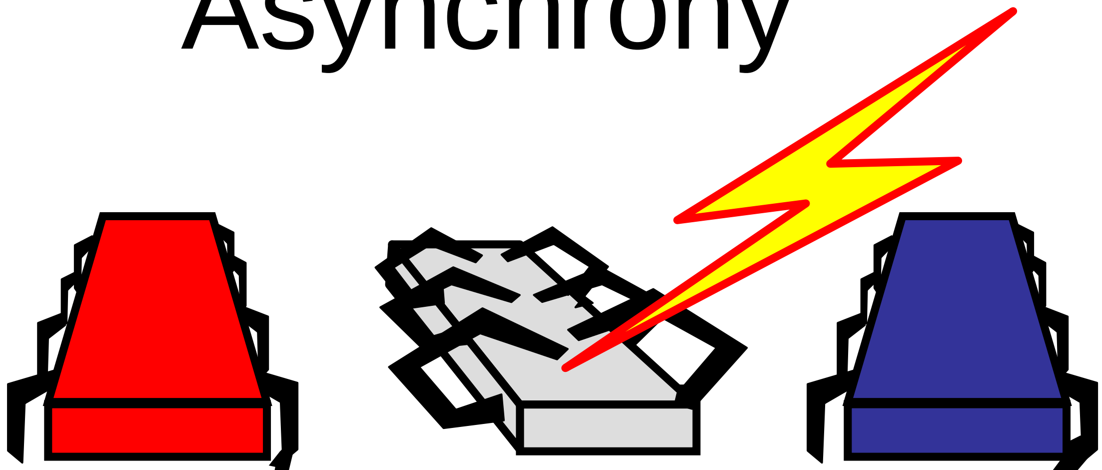
Pay for N = 8 cores

SequentialPart = 25%

As num cores grows the effect of 25% becomes more accute

2.3/4, 2.9/8, 3.4/16, 3.7/32….

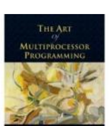
Asynchrony



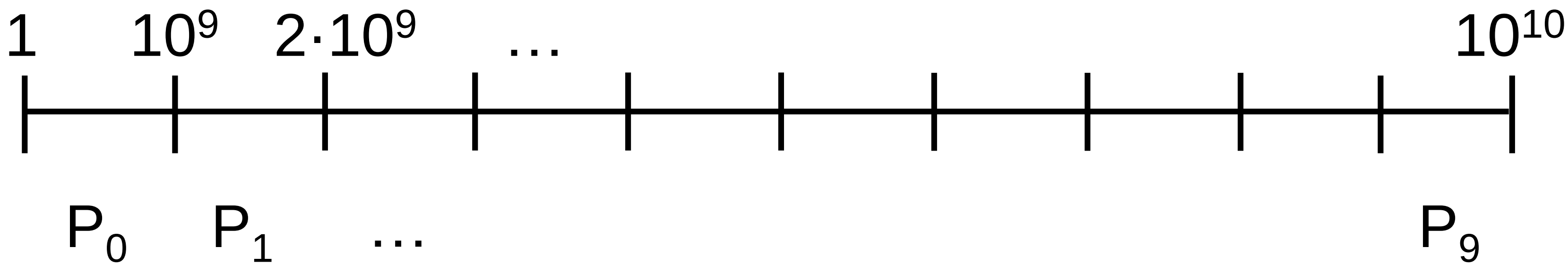
- Sudden unpredictable delays
  - Cache misses (*short*)
  - Page faults (*long*)
  - Scheduling quantum used up (*really long*)

Parallel Primality Testing

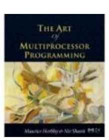
- Challenge
  - Print primes from 1 to 10^{10}
- Given
  - Ten-processor multiprocessor
  - One thread per processor
- Goal
  - Get ten-fold speedup (or close)



Load Balancing

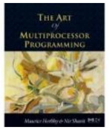


- Split the work evenly
- Each thread tests range of 10^9



Procedure for Thread *i*

```
void prime_print(void* arg) {
  long i = (long) arg; // IDs in {0..9}
  for (int j = i*10^9+1, j<(i+1)*10^9; ++j)
  {
     if (is_prime(j))
        print(j);
  }
}
```



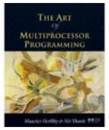
Issues

- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict

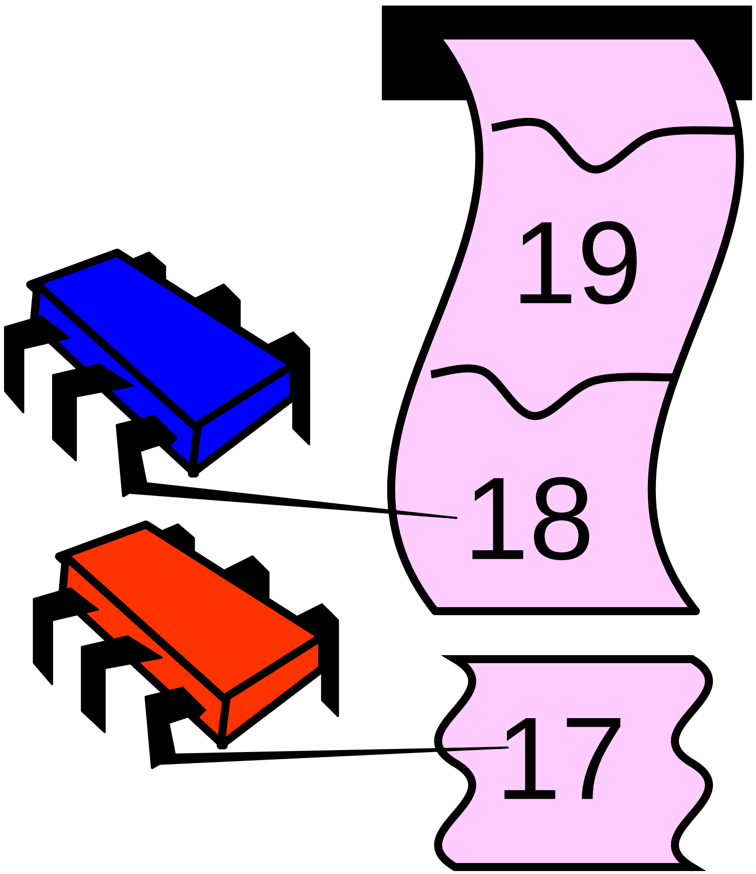
Issues

- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict
- Need *dynamic* load balancing

rejected



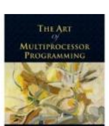
Shared Counter



each thread takes a number

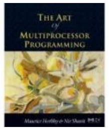
Procedure for Thread *i*

```
void prime_print(void* arg) {
      long j = 0;
      while (j < 10^10) {
            j = next_counter();
            if (is_prime(j))
               print(j);
      }
}
```



Counter Implementation

```
static int counter = 1;

int next_counter(void) {
  return counter++;
}
```



Counter Implementation

```
static int counter = 1;

int next_counter(void) {
  return counter++;
}
```

OK for single thread, not for concurrent threads

Code example 3