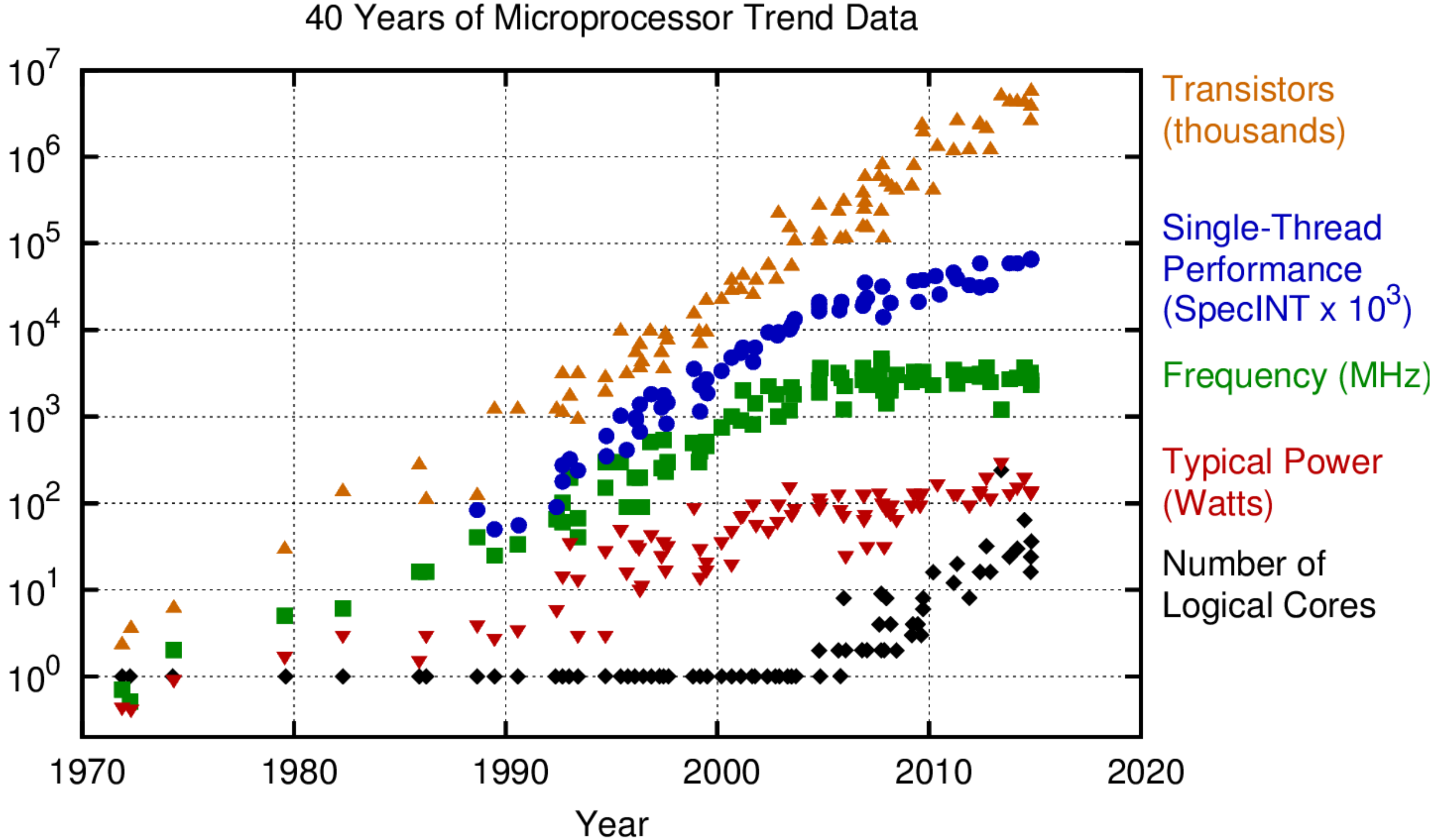
CSCI-UA.0201

Computer Systems Organization

Concurrency - Multithreading

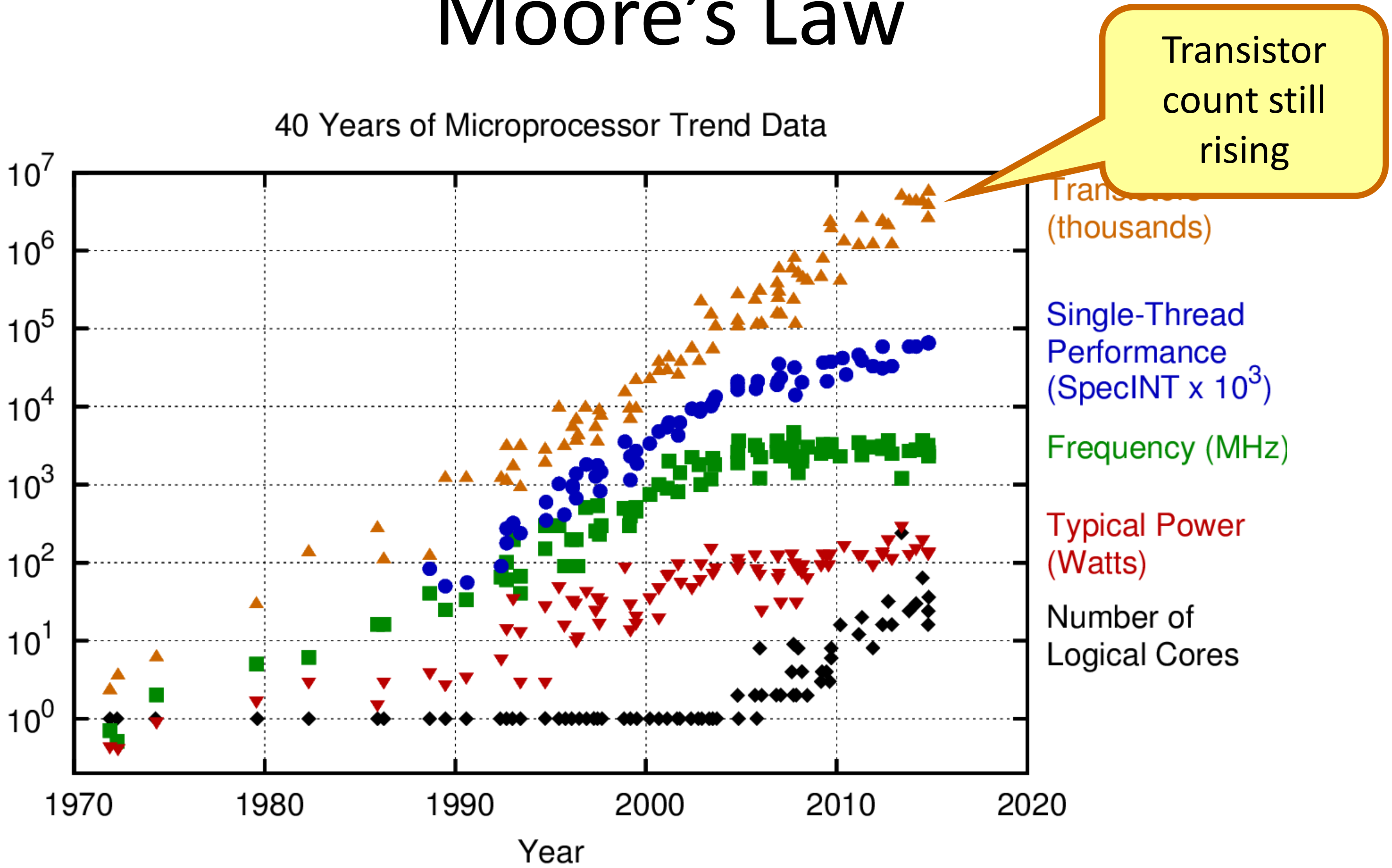
Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies

Moore's Law



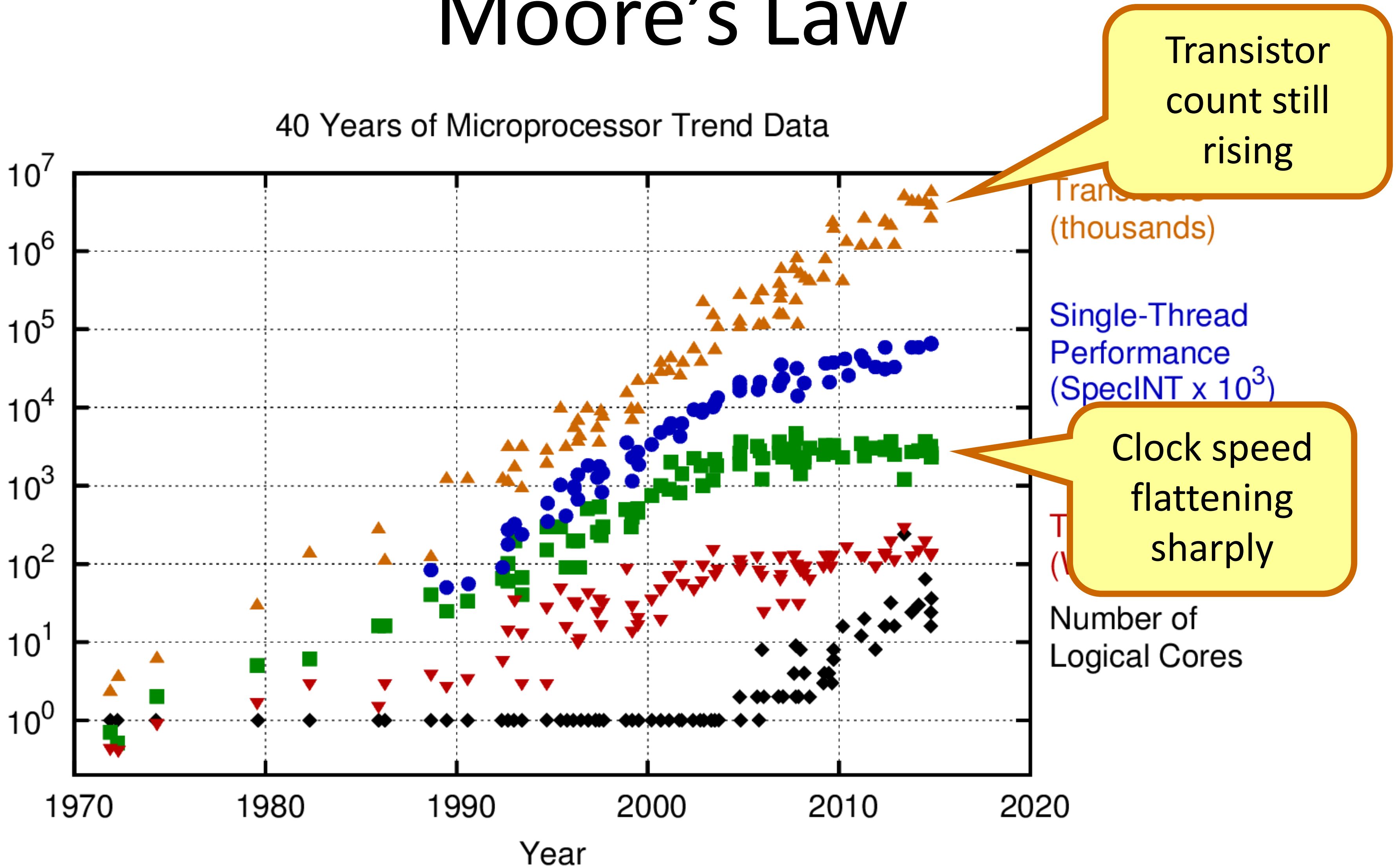
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Moore's Law



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

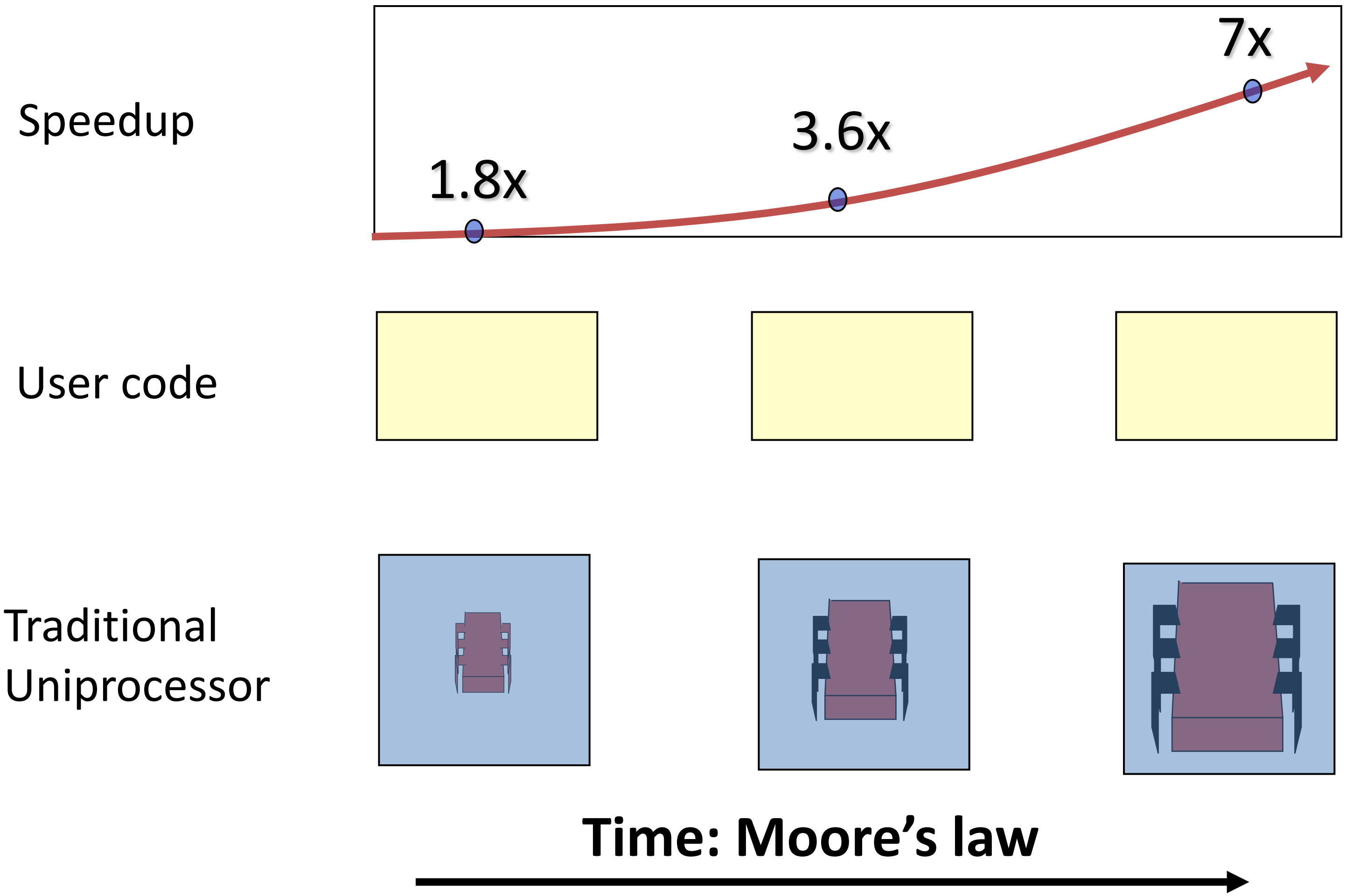
Moore's Law



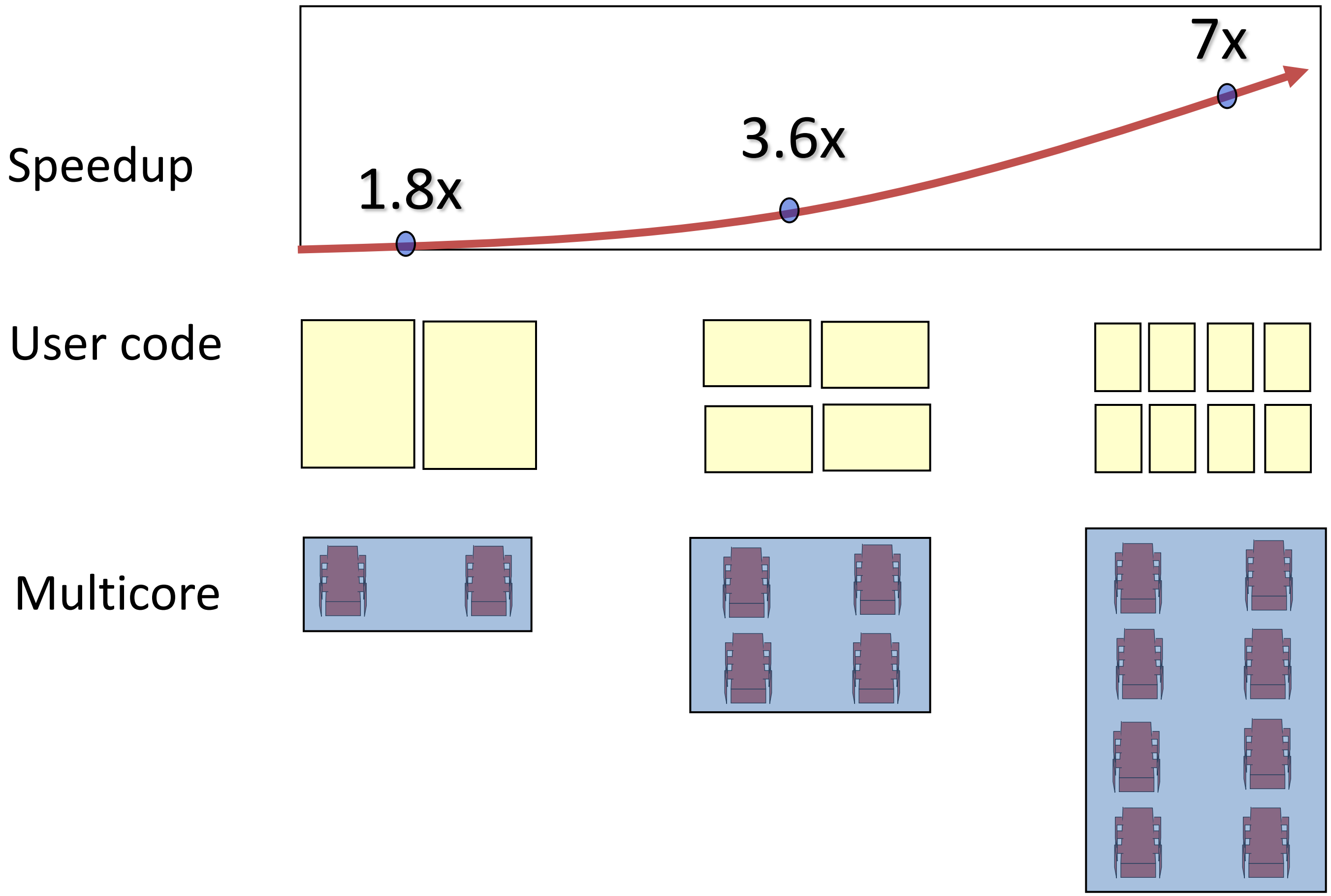
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Moore’s Law (in practice)

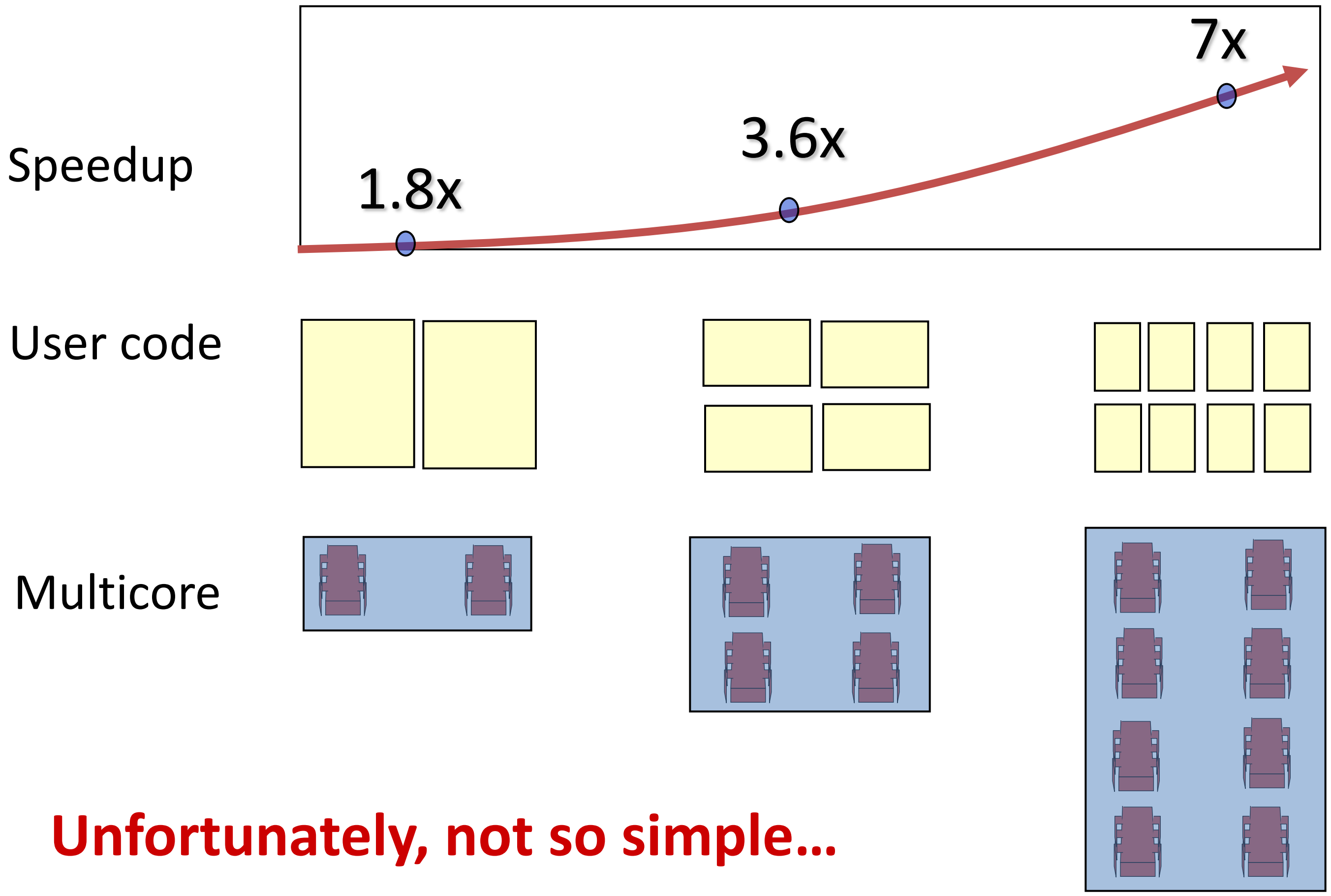
Traditional Scaling Process



Ideal Scaling Process

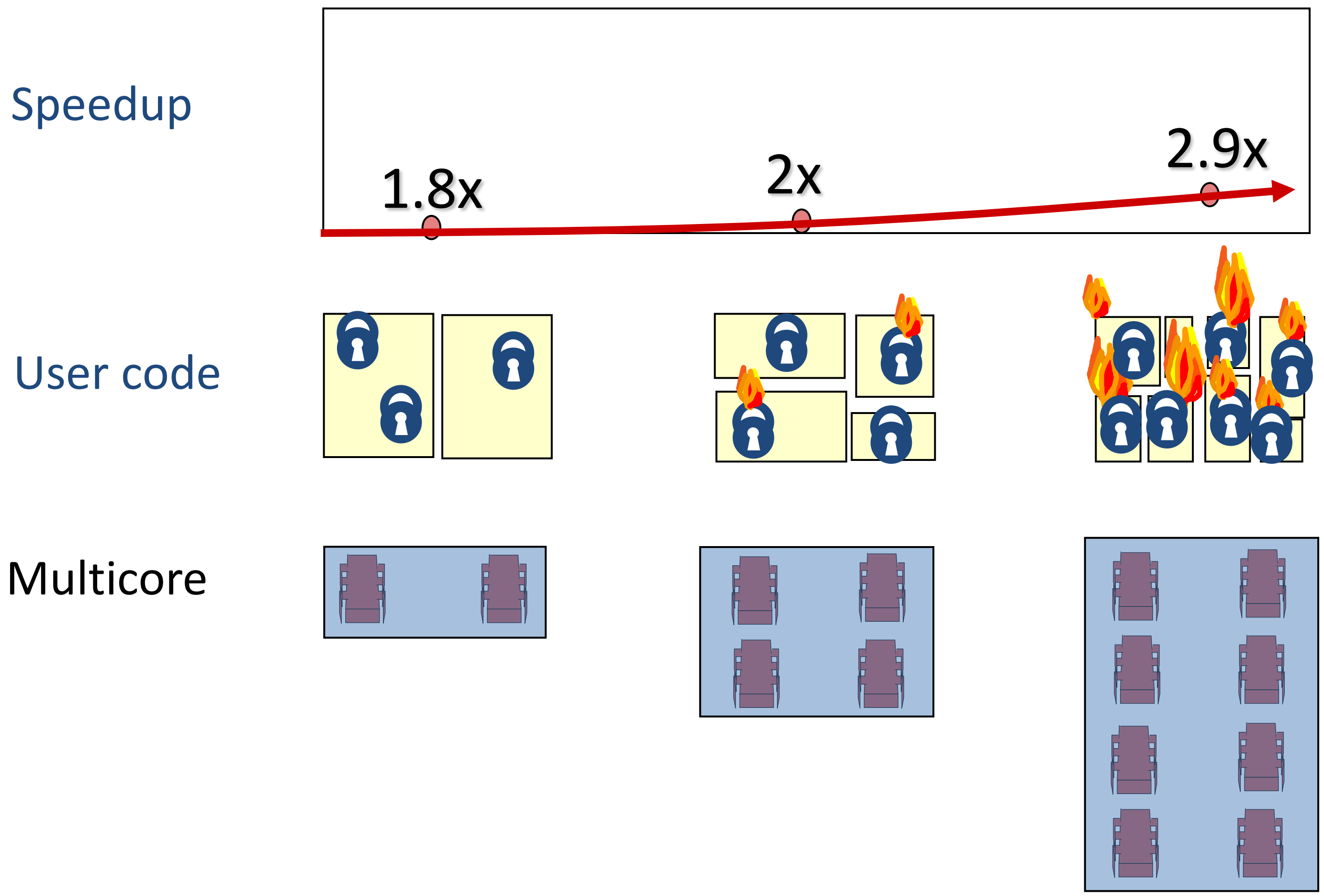


Ideal Scaling Process

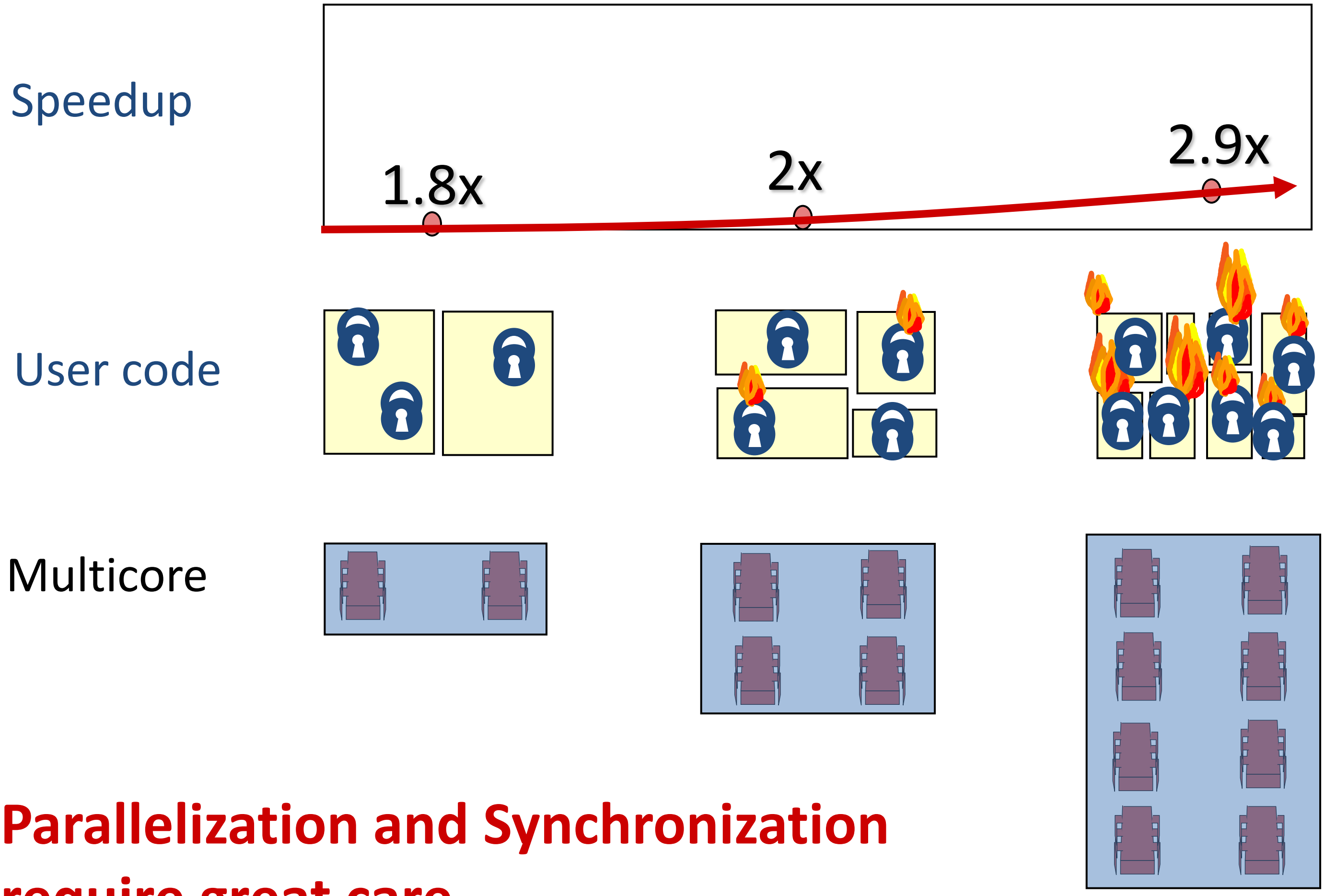


Unfortunately, not so simple…

Actual Scaling Process



Actual Scaling Process



Parallelization and Synchronization require great care…

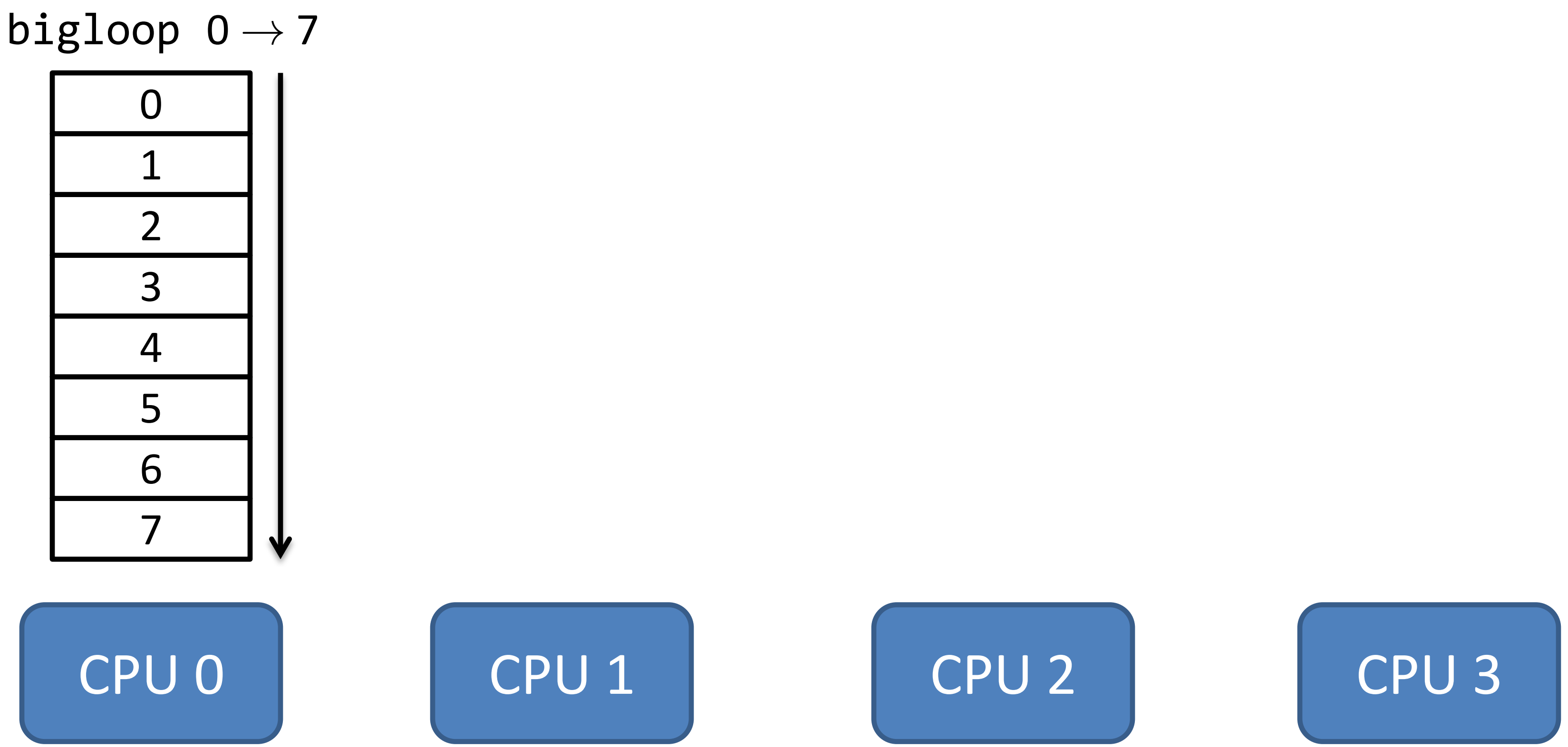
Multithreading Basics

Example

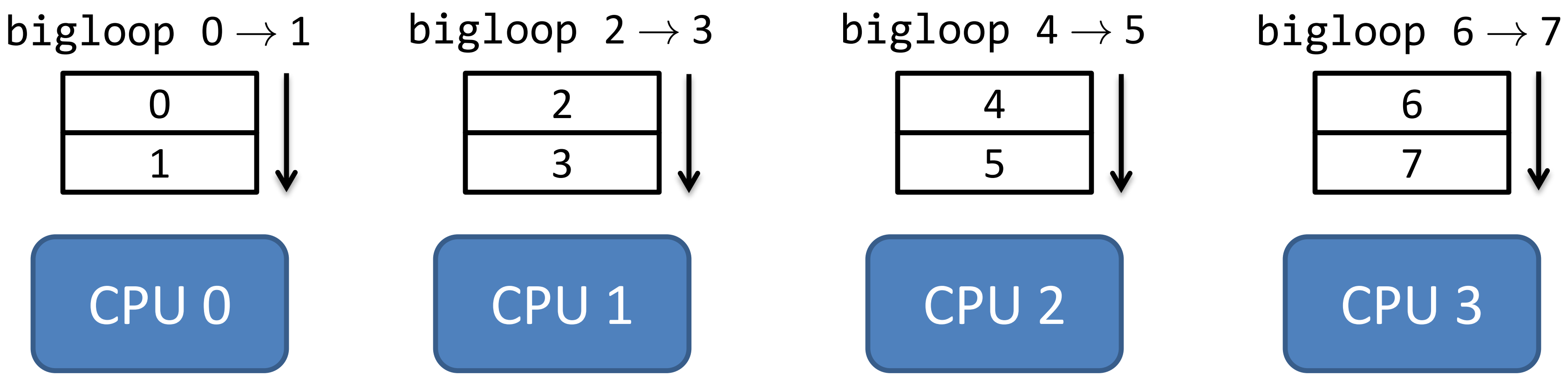
```
long bigloop(int *arr, int len) {
  long r = 0;
  for(int i = 0; i < len; i++)
    r += arr[i];
  return r;
}

int main() {
  int *arr = malloc(8 * sizeof(int));
  ...
  long r = bigloop(arr, 8);
  ...
}
```

How to improve the performance with multithreading?

Parallelization



Parallelization

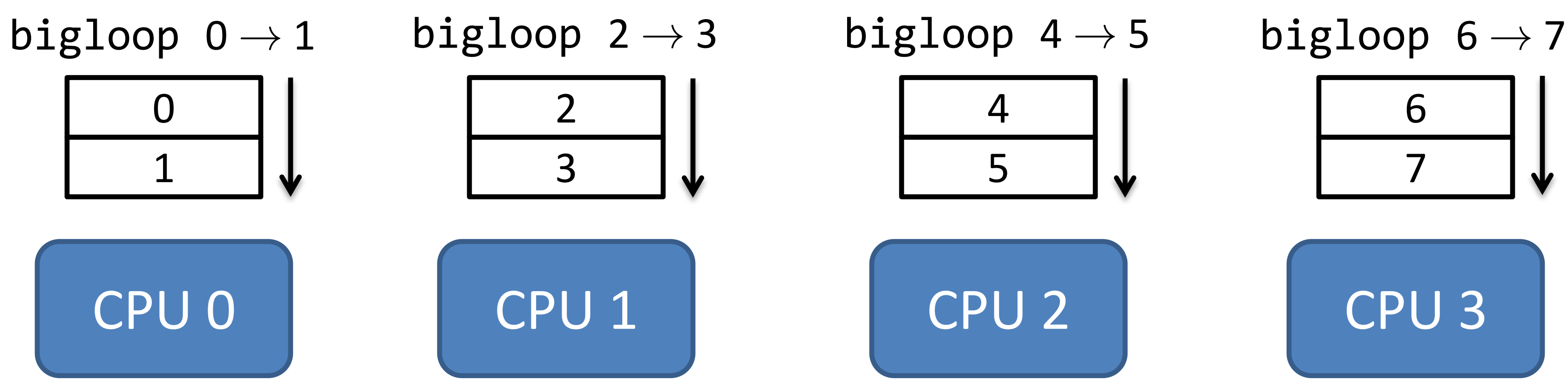


Performance can be improved by 4X

Parallelization

What is concurrency?

- things happening "simultaneously"
  - multiple CPU cores concurrently executing instructions
  - CPU and I/O devices concurrently doing processing



Performance can be improved by 4X

Concurrency

What is concurrency?

- multiple CPU cores concurrently executing instructions
- CPU and I/O devices concurrently doing processing

- Why write concurrent programs?
  - speed up programs using multiple CPUs
  - speed up programs by interleaving CPU processing and I/O.

How to write concurrent programs?

- Use multiple processes
  - Each process uses a different CPU
  - Different processes runs different tasks
  - They have separate address spaces
  - It is difficult to communicate with each other

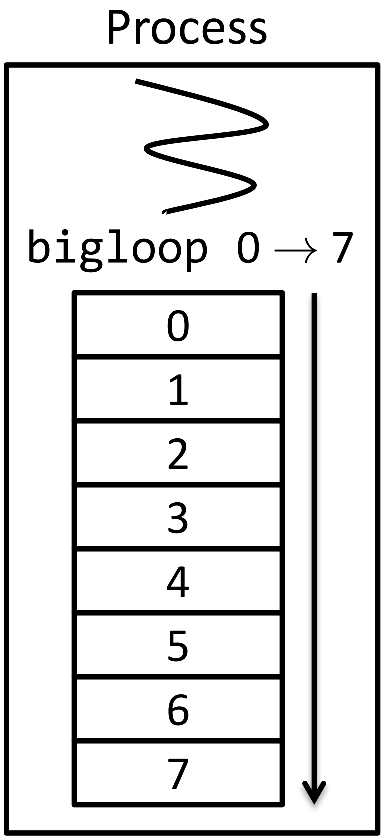
- Use multiple threads

How to write concurrent programs?

- Use multiple processes
  - Each process uses a different CPU
  - Different processes runs different tasks
  - They have separate address spaces
  - It is difficult to communicate with each other

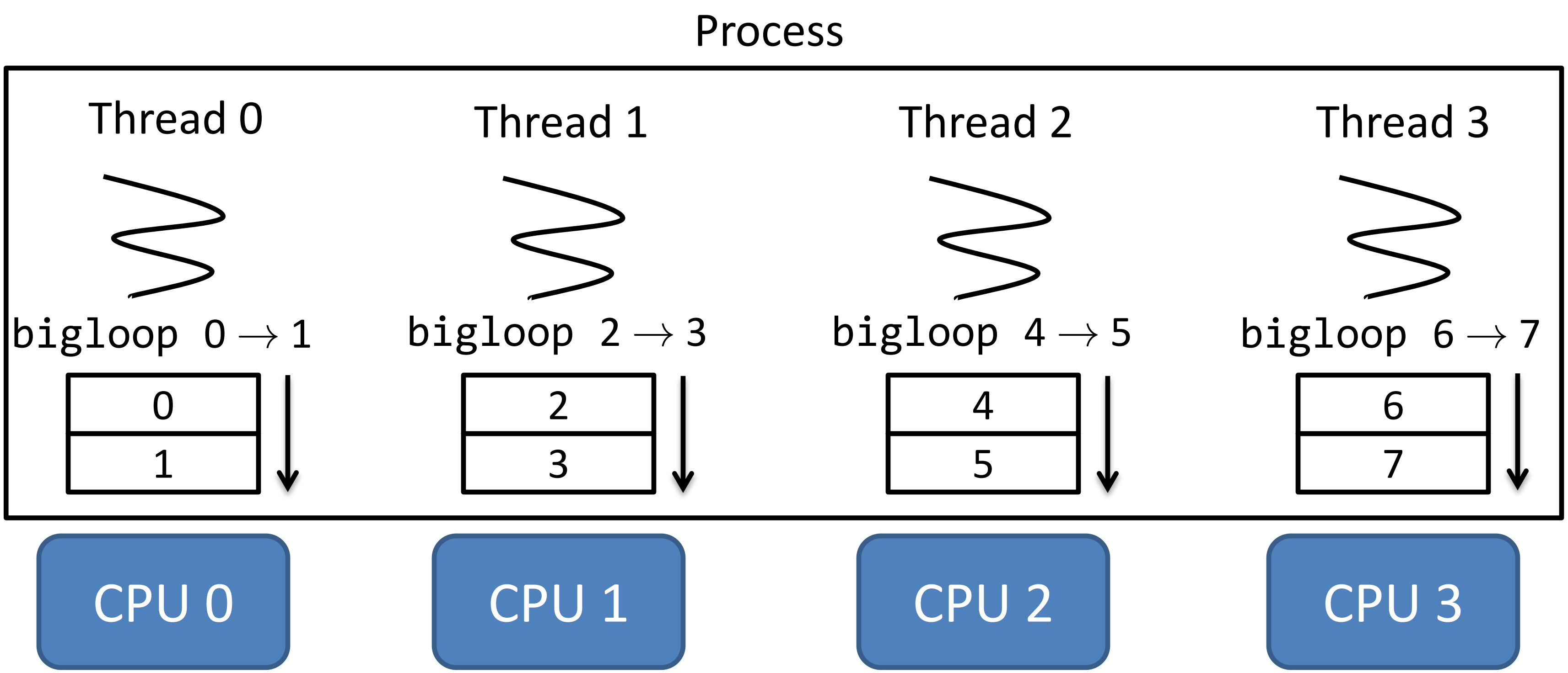
- Use multiple threads

Multiple threads (Multithreading)




```
long bigloop(int *arr, int len) {
  long r = 0;
  for(int i = 0; i < len; i++)
    r += arr[i];
  return r;
}

int main() {
  int *arr = malloc(8 * sizeof(int));
  ...
  long r = bigloop(arr, 8);
  ...
}
```

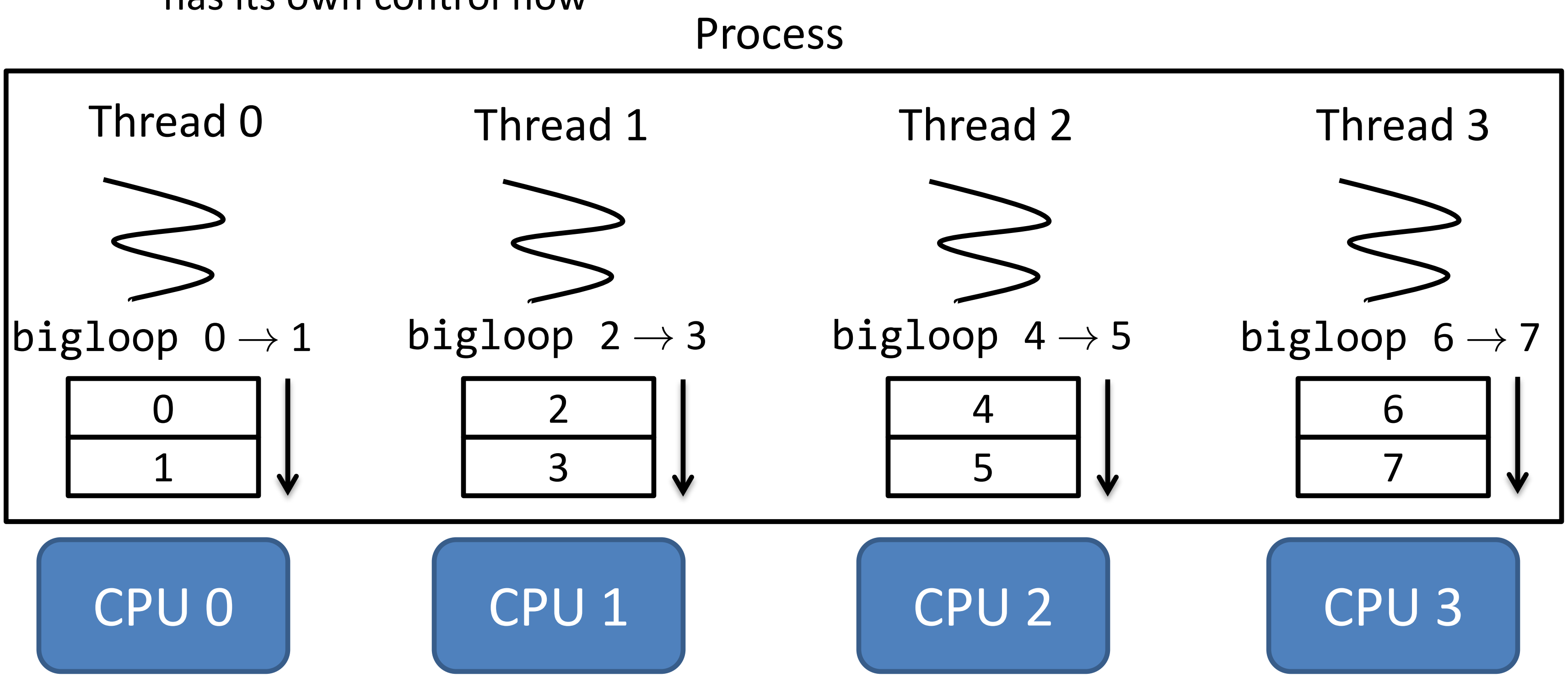
Multiple threads (Multithreading)



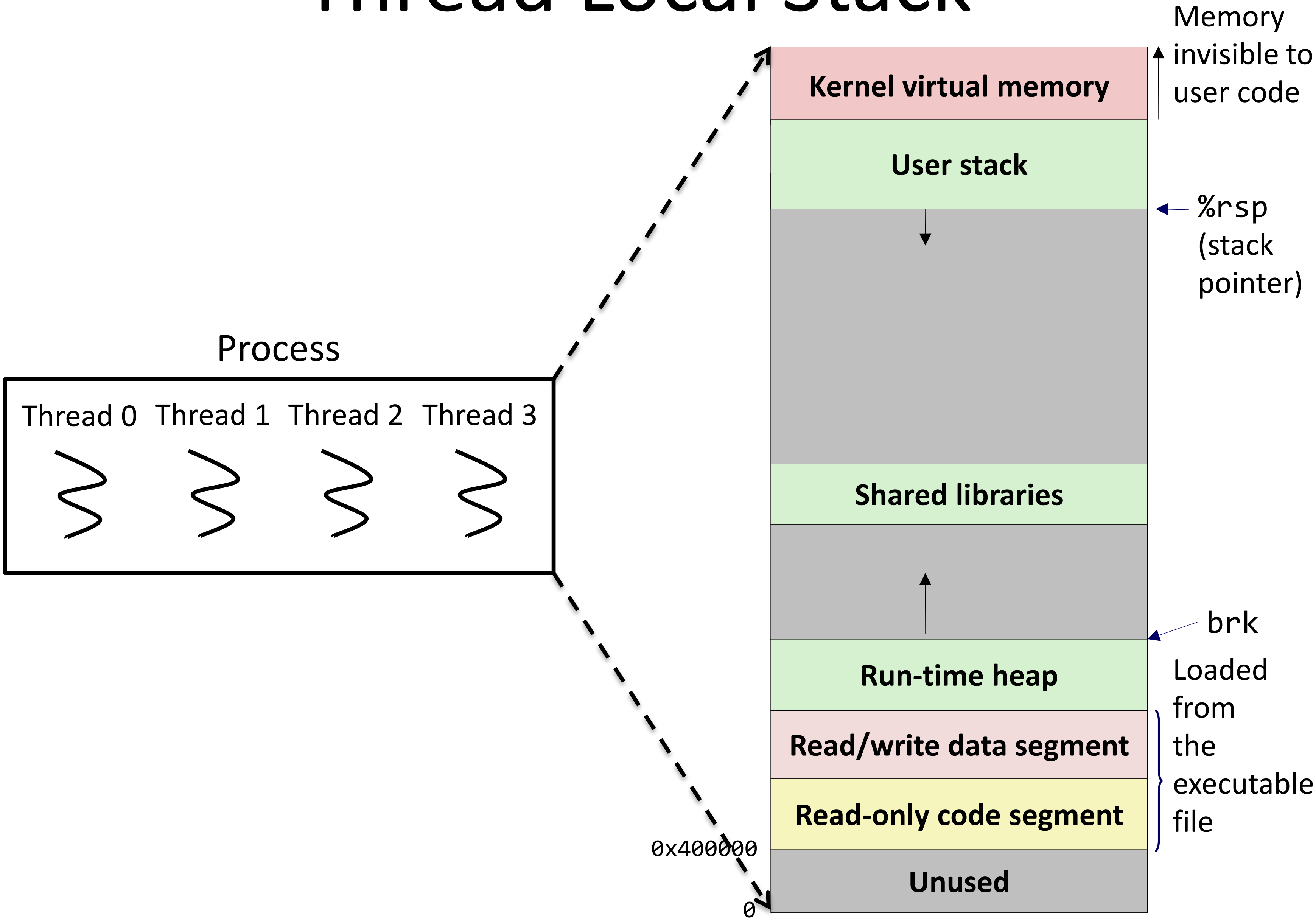
Multiple threads (Multithreading)

Single process, multiple threads

- Threads Share the same virtual memory space
- Each thread
  - has its own stack
  - has its own control flow



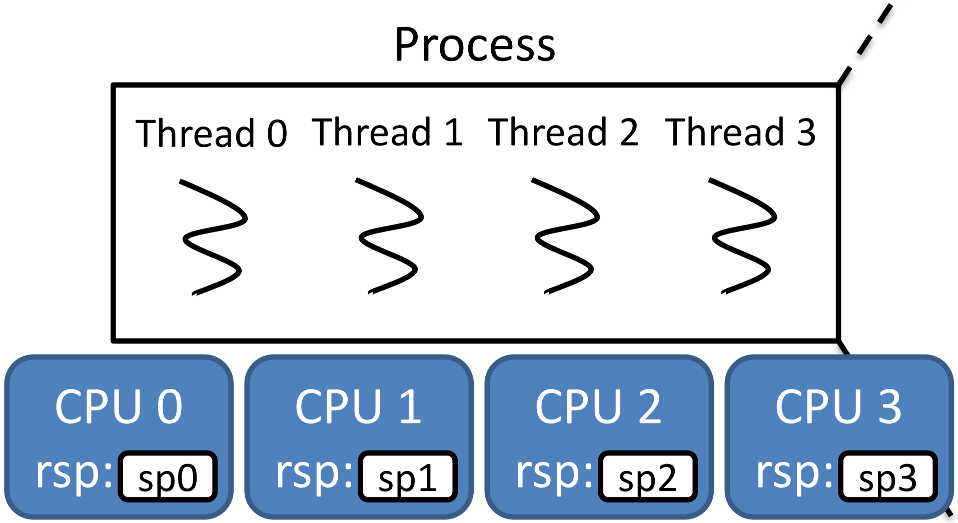
Thread Local Stack

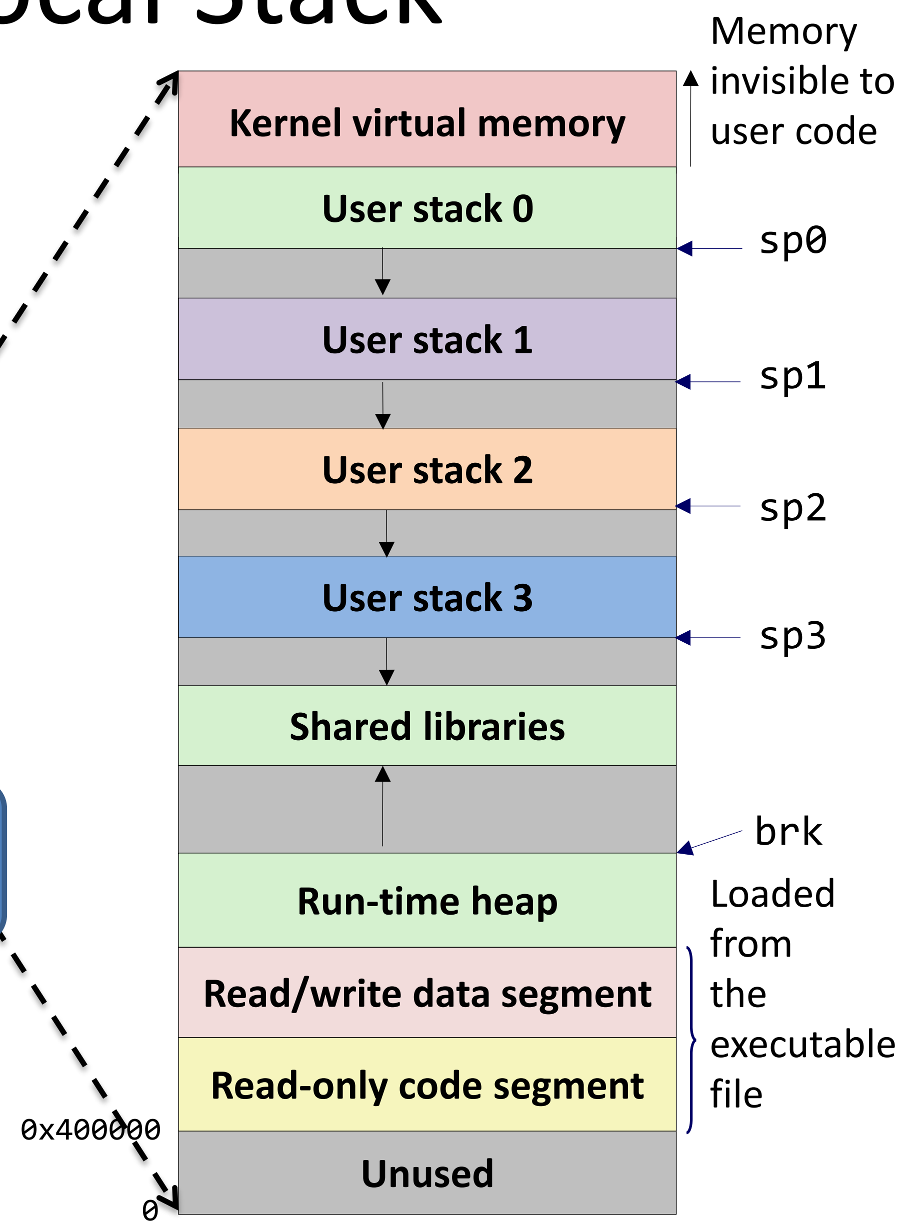


Thread Local Stack

Each thread has its own stack segment

- Each thread has its own stack pointer
- Store the stack pointer into %rsp before running





POSIX Thread Interface

- POSIX: Portable Operating System Interface
  - POSIX defines the API for variants of Unix

- Thread interface defined by POSIX
  - `pthread_create`: create a new thread
  - `pthread_join`: wait until the target thread has terminated

pthread_create

```
#include <pthread.h>
int pthread_create(pthread_t *thread_id,
             const pthread_attr_t *attr,
             void *(*start_routine)(void*),
             void *arg);
```

- Create a new thread
  - It executes `start_routine` with `arg` as its sole argument.
  - Its attribute is specified by `attr`
  - Upon successful completion, it will store the ID of the created thread in the location referenced by `thread_id`.
- Return value
  - zero: success
  - non-zero (error number): fail

Example 1 - Create

```
void* func(void* arg) {
  printf("This is the created thread\n");
  return NULL;
}

int main(int argc, char* argv[]) {
  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, NULL);
  if(r != 0) {
    printf("create thread failed");
    return 1;
  }
  return 0;
}

$ gcc create.c -lpthread
```

Example 1 - Create

```
void* func(void* arg) {
  printf("This is the created thread\n");
  return NULL;
}

int main(int argc, char* argv[]) {
  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, NULL);
  if(r != 0) {
    printf("create thread failed");
    return 1;
  }
  return 0;
}

$ gcc create.c -lpthread
```

Main thread returns before the created thread finishes.

- Automatically terminate and reclaim the created thread.

pthread_join

```
#include <pthread.h>
int pthread_join(pthread_t thread_id, void **ret_ptr);
```

- Wait for the target thread to finish
  - The target thread is specified by `thread_id`
  - Upon success, the return value of the created thread will be available in the location referenced by `ret_ptr`.
- Return value
  - zero: success
  - non-zero (error number): fail

Example 2 - Join

```
void* func(void* arg) {
  printf("This is the created thread\n");
  return NULL;
}

int main(int argc, char* argv[]) {
  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, NULL);
  if(r != 0) {
    ...
  r = pthread_join(tid, NULL);
  if(r != 0)
    ...
  return 0;
}
```

Example 3 - Parameter

```
void* func(void* arg) {
  int p = *(int *) arg;
  p = p + 1;
  return &p;
}

int main(int argc, char* argv[]) {
  int param = 100;
  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, (void *) &param);
  ...
  int *res = NULL;
  r = pthread_join(tid, &res);
  ...
  printf("result: addr %lx, val %d\n", res, *res);
  return 0;
}
```

Example 3 - Parameter

```
void* func(void* arg) {
  int p = *(int *) arg;
  p = p + 1;
  return &p;
}

int main(int argc, char* argv[]) {
  int param = 100;
  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, (void *) &param);
  ...
  int *res = NULL;
  r = pthread_join(tid, &res);
  ...
  printf("result: addr %lx, val %d\n", res, *res);
  return 0;
}
```

Question – what is the expected output?

Example 3 - Parameter

```
void* func(void* arg) {
  int p = *(int *) arg;
  p = p + 1;
  return &p;
}

int main(int argc, char* argv[]) {
  int param = 100;
  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, (void *) &param);
  ...
  int *res = NULL;
  r = pthread_join(tid, &res);
  ...
  printf("result: addr %lx, val %d\n", res, *res);
  return 0;
}
```

p is on the stack of the created thread
-- it is no longer valid when the thread terminates

Question – what is the expected output?

Example 3 - Parameter

```
void* func(void* arg) {
  int p = *(int *) arg;
  p = p + 1;
  int *r = (int *) malloc(sizeof(int));
  *r = p
  return (void *) r;
}

int main(int argc, char* argv[]) {
  int param = 100;
  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, (void *) &param);
  ...
  int *res = NULL;
  r = pthread_join(tid, &res);
  ...
  printf("result: addr %lx, val %d\n", res, *res);
  return 0;
}
```

Example 3 - Parameter

```
void* func(void* arg) {
  int p = *(int *) arg;
  p = p + 1;
  int *r = (int *) malloc(sizeof(int));
  *r = p
  return (void *) r;
}

int main(int argc, char* argv[]) {
  int param = 100;
  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, (void *) &param);
  ...
  int *res = NULL;
  r = pthread_join(tid, &res);
  ...
  printf("result: addr %lx, val %d\n", res, *res);
  free(res);
  return 0;
}
```

Example 4 - Interleaving

Question – what is the expected output?

```
void* func(void* arg) {
  printf(“1”);
}

int main(int argc, char* argv[]) {
  printf(“0”);

  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, NULL);
  ...
  printf(“2”);
  ...
  return 0;
}
```

Example 4 - Interleaving

```
void* func(void* arg) {
  printf("1");
}

int main(int argc, char* argv[]) {
  printf("0");

  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, NULL);
  ...
  printf("2");
  ...
  return 0;
}
```

Question – what is the expected output?

Answer: 012 or 021

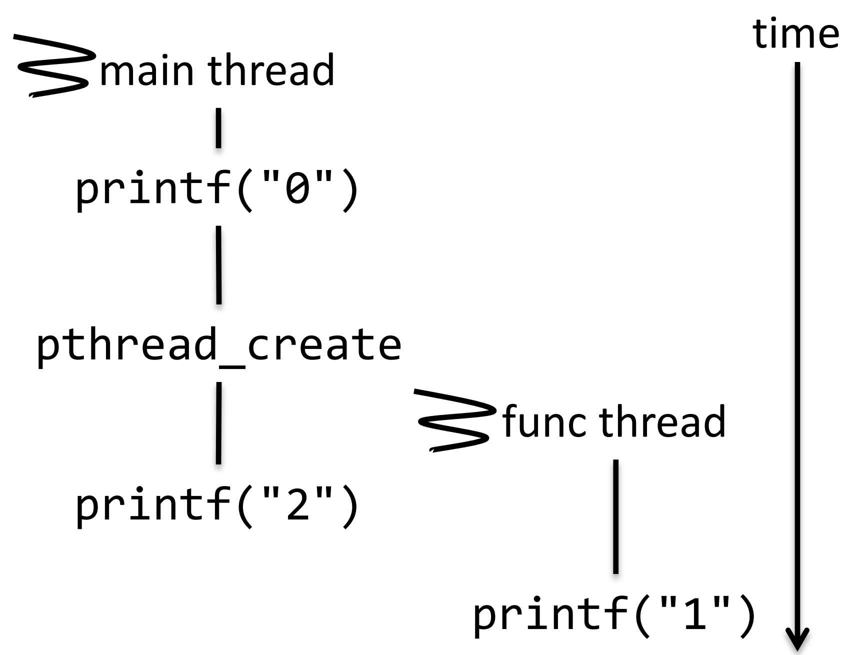
Example 4 - Interleaving

```
void* func(void* arg) {
  printf(“1”);
}

int main(int argc, char* argv[]) {
  printf(“0”);

  pthread_t tid;
  int r = pthread_create(&tid, NULL,
                         &func, NULL);
  ...
  printf(“2”);
  ...
  return 0;
}
```

Question – what is the expected output?

Answer: 012 or 021

012



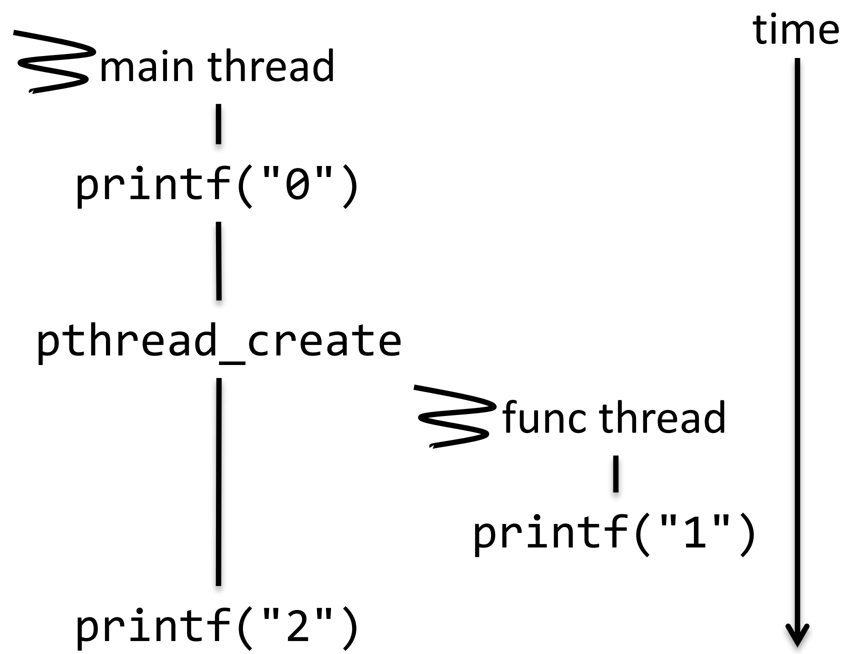
Example 4 - Interleaving

```
void* func(void* arg) {
  printf(“1”);
}

int main(int argc, char* argv[]) {
  printf(“0”);

  pthread_t tid;
  int r = pthread_create(&tid, NULL,
                         &func, NULL);
  ...
  printf(“2”);
  ...
  return 0;
}
```

Question – what is the expected output?

Answer: 012 or 021

021



Example 3 Revisited

Question – can we get rid of r in func?

```
void* func(void* arg) {
  int p = *(int *) arg;
  p = p + 1;
  int *r = (int *) malloc(sizeof(int));
  *r = p
  return (void *) r;
}

int main(int argc, char* argv[]) {
  int param = 100;
  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, (void *) &param);
  ...
  int *res = NULL;
  r = pthread_join(tid, &res);
  ...
  printf("result: addr %lx, val %d\n", res, *res);
  free(res);
  return 0;
}
```

Example 3 Revisited

Question – can we get rid of r in func?

```
void* func(void* arg) {
  int *p = (int *) arg;
  *p = *p + 1;
  return NULL;
}

int main(int argc, char* argv[]) {
  int param = 100;
  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, (void *) &param);
  ...
  int *res = NULL;
  r = pthread_join(tid, &res);
  ...
  printf("result: %d\n", param);
  return 0;
}
```

Example 5 – Stack, Heap, Global

```
int global = 0;

void* write(void* arg) {
  int local = 0;
  local = 10;
  global = 10;
  int *ptr = (int *)arg;
  (*ptr) = 10;
}
```

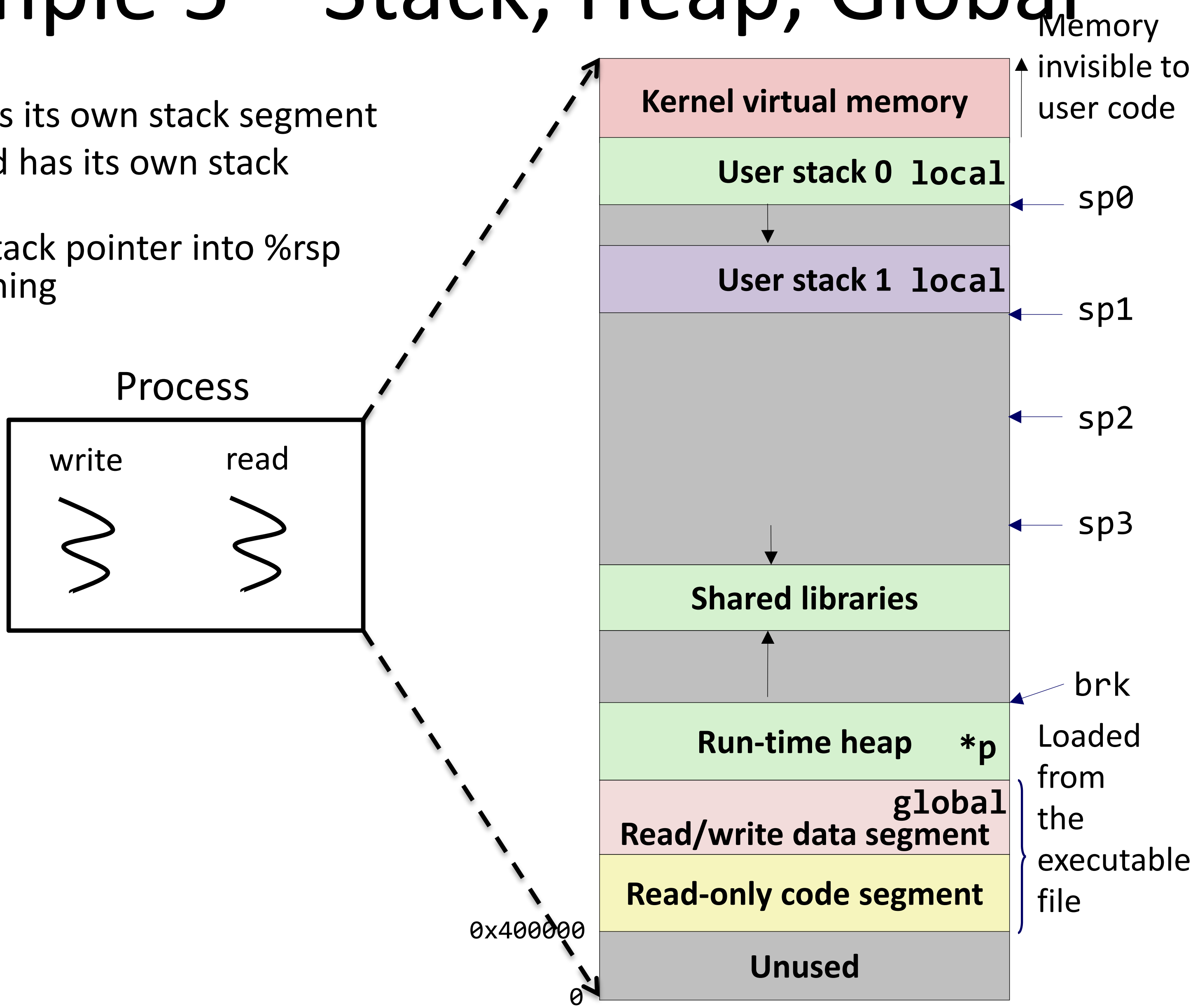
```
void* read(void* arg) {
  int local = 0;
  printf("local %d global %d heap %d\n",
         local, global, *(int *)arg);
  return NULL;
}
```

```
int main() {
  int *p = (int *) malloc(sizeof(int));
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, &write, (void *)p);
  ...
  pthread_join(tid1, NULL);
  pthread_create(&tid2, NULL, &read, (void *)p);
  ...
  return 0;
}
```

Example 5 – Stack, Heap, Global

Each thread has its own stack segment

- Each thread has its own stack pointer
- Store the stack pointer into %rsp before running



Example 5 – Stack, Heap, Global

```
int global = 0;

void* write(void* arg) {
  int local = 0;
  local = 10;
  global = 10;
  int *ptr = (int *)arg;
  (*ptr) = 10;
}
```

```
void* read(void* arg) {
  int local = 0;
  printf("local %d global %d heap %d\n",
          local, global, *(int *)arg);
  return NULL;
}
```

```
int main() {
  int *p = (int *) malloc(sizeof(int));
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, &write, (void *)p);
  ...
  pthread_join(tid1, NULL);
  pthread_create(&tid2, NULL, &read, (void *)p);
  ...
  return 0;
}
```

What are the outputs?

local 0 global 10 heap 10

Example 5 – Stack, Heap, Global

```
int global = 0;

void* write(void* arg) {
  int local = 0;
  local = 10;
  global = 10;
  int *ptr = (int *)arg;
  (*ptr) = 10;
}
```

```
void* read(void* arg) {
  int local = 0;
  printf("local %d global %d heap %d\n",
         local, global, *(int *)arg);
  return NULL;
}
```

```
int main() {
  int *p = (int *) malloc(sizeof(int));
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, &write, (void *)p);
  ...
  pthread_join(tid1, NULL);
  pthread_create(&tid2, NULL, &read, (void *)p);
  ...
  return 0;
}
```

What are the outputs?

local 0 global 10 heap 10

Example 5 – Stack, Heap, Global

```
int global = 0;

void* write(void* arg) {
  int local = 0;
  local = 10;
  global = 10;
  int *ptr = (int *)arg;
  (*ptr) = 10;
}
```

```
void* read(void* arg) {
  int local = 0;
  printf("local %d global %d heap %d\n",
          local, global, *(int *)arg);
  return NULL;
}
```

```
int main() {
  int *p = (int *) malloc(sizeof(int));
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, &write, (void *)p);
  ...
  pthread_join(tid1, NULL);
  pthread_create(&tid2, NULL, &read, (void *)p);
  ...
  return 0;
}
```

What are the outputs?

local 0 global 10 heap 10

local 0 global 10 heap 0

local 0 global 0 heap 0