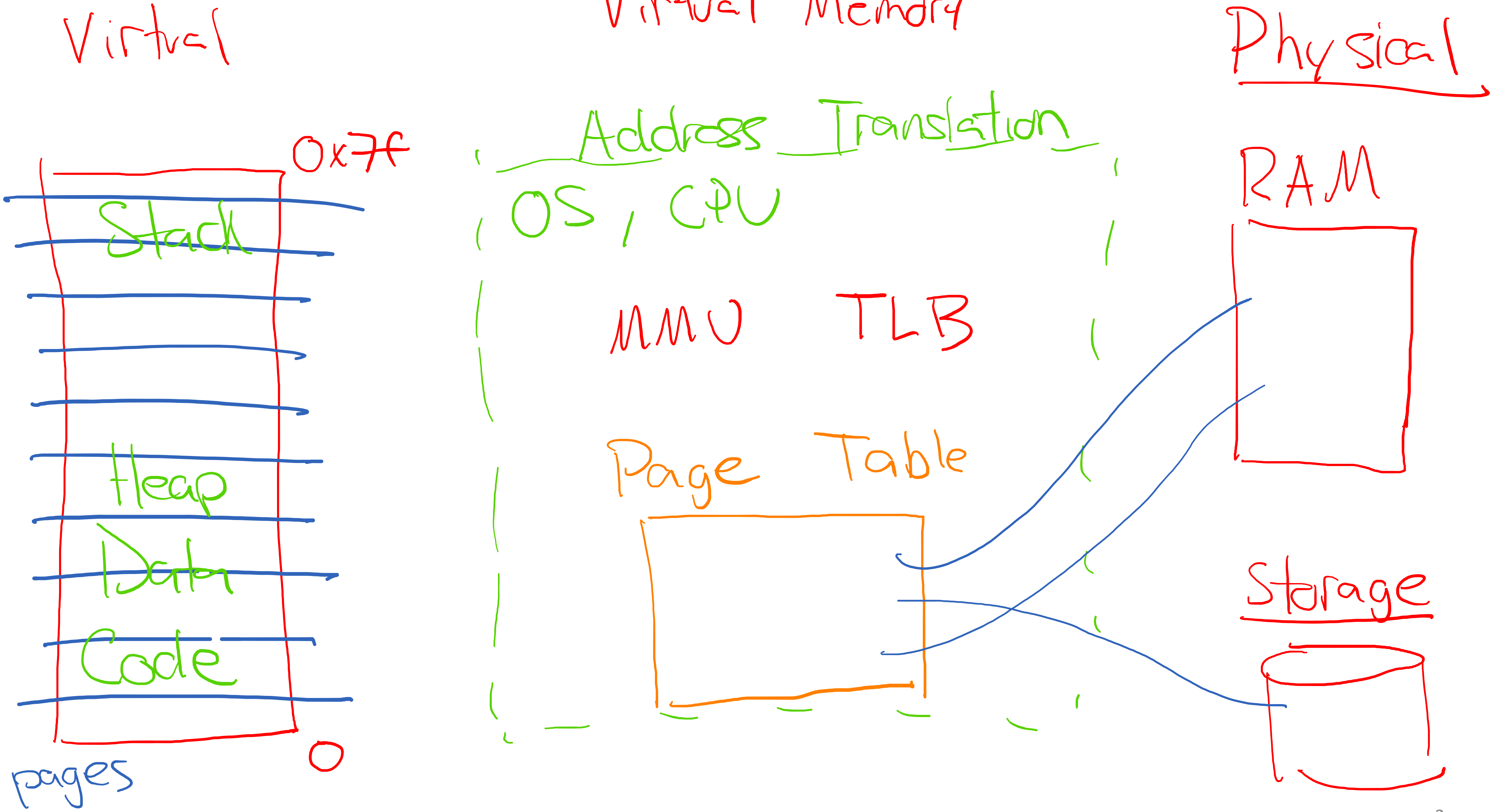
Threads

Drawing: Virtual Memory

- Take three minutes to draw “virtual memory”

- Some reminders
  - Hardware (MMU, TLB)
  - Virtual memory, physical memory, secondary storage
  - Operating Systems
  - Virtual Address Spaces
  - Page tables, page directories
  - Multi-level page tables
  - Address translation
  - Page faults

Virtual Memory



Threading

- One of my favorite subjects in CS and programming
- Challenging, but very rewarding when you get something implemented correctly

Pedagogically Annoying Note

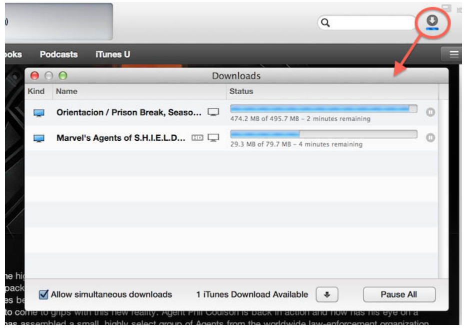
- I could teach one of (or both) two thread libraries:
  - The C11 standard implementation for concurrency
  - Or POSIX threads (pthreads)

- There is not currently a correct answer for this
- I am only going to teach one to reduce confusion
- C11 will be the library to use for all new C projects in which you can use a modern compiler
- But you are more likely to see legacy C code than write new C code
- So, we’ll go with pthreads

Single Process, Multiple Threads

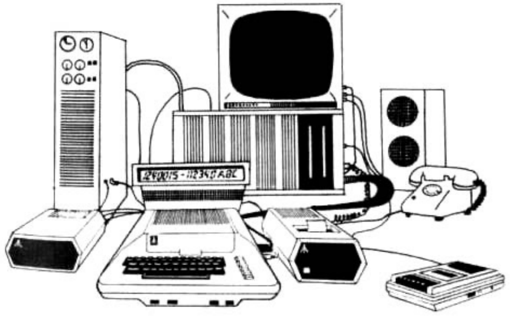
Program Structure: expressing logically concurrent programs

- Spell-check
- Autosaving
- Etc.

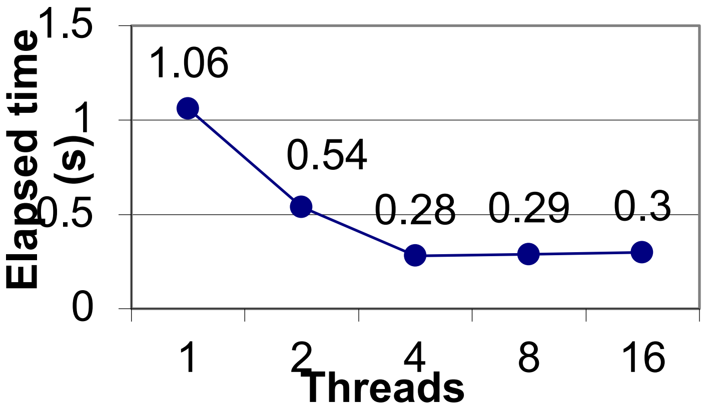


Kind	Name	Status
	Orientacion / Prison Break, Seaso...	474.2 MB of 495.7 MB – 2 minutes remaining
	Marvel's Agents of S.H.I.E.L.D...	29.3 MB of 79.7 MB – 4 minutes remaining

Responsiveness: shifting work to run in the background

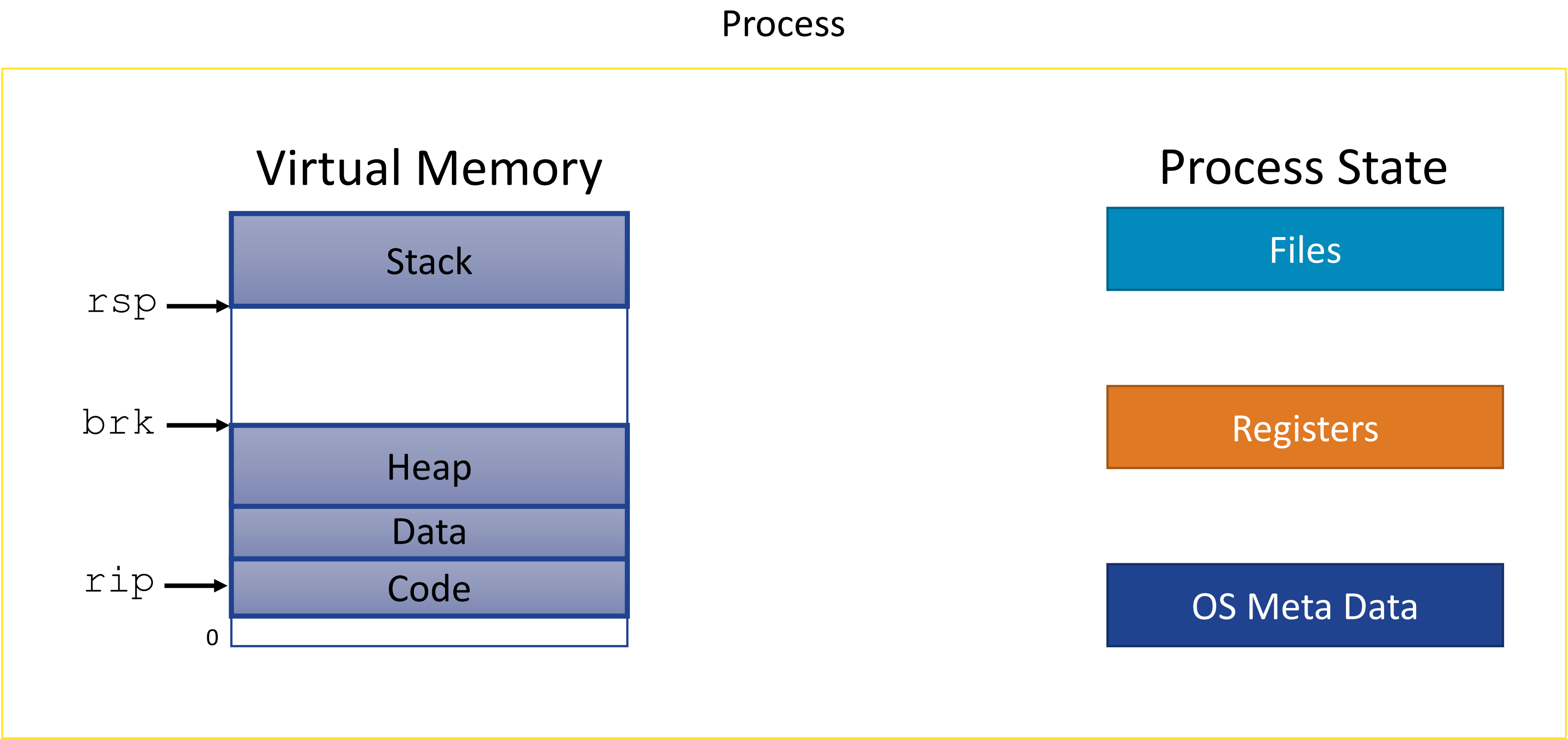


Responsiveness: managing I/O devices

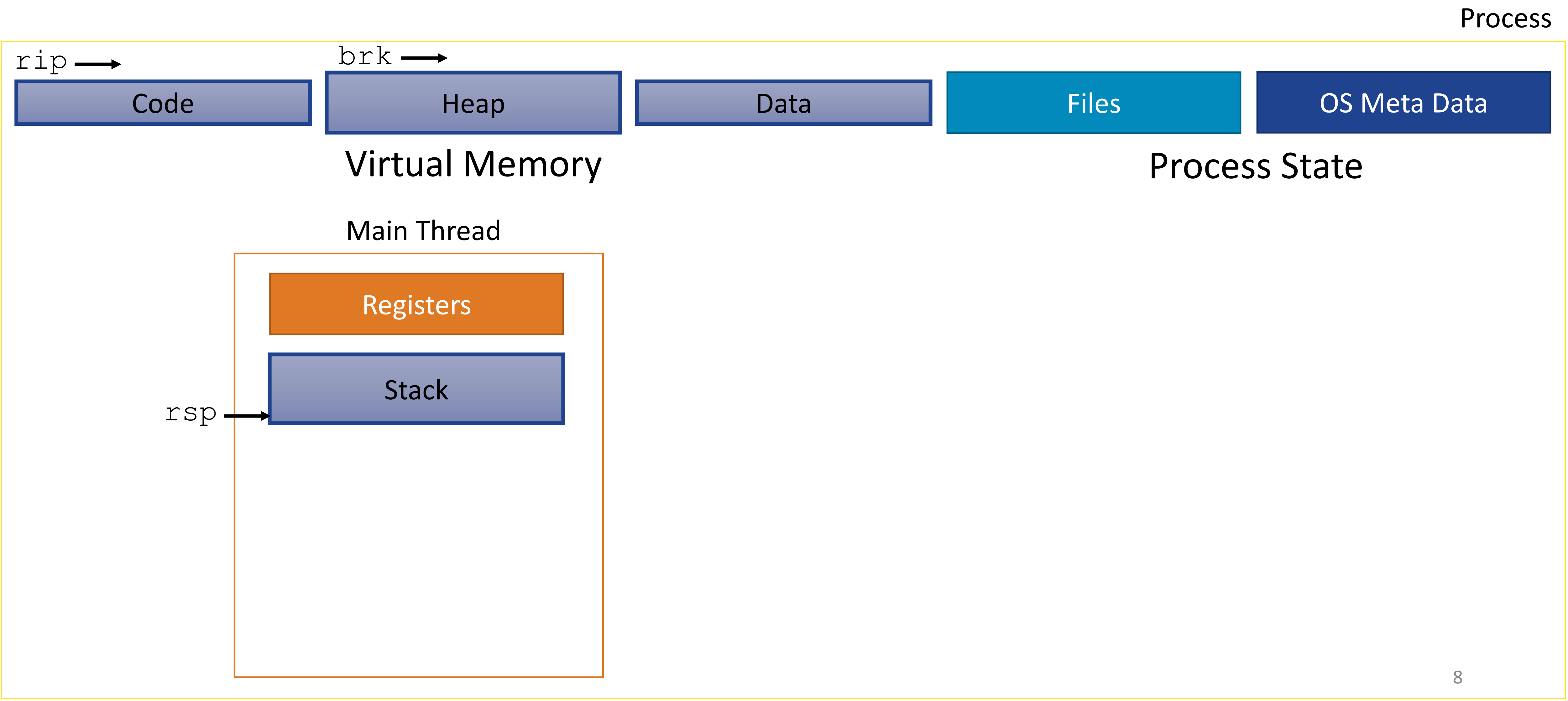


Performance: exploiting multiprocessors

Current View of a Process



Threaded View of a Process



Process

`rip`

`brk`

Code

Heap

Data

Files

OS Meta Data

Virtual Memory

Process State

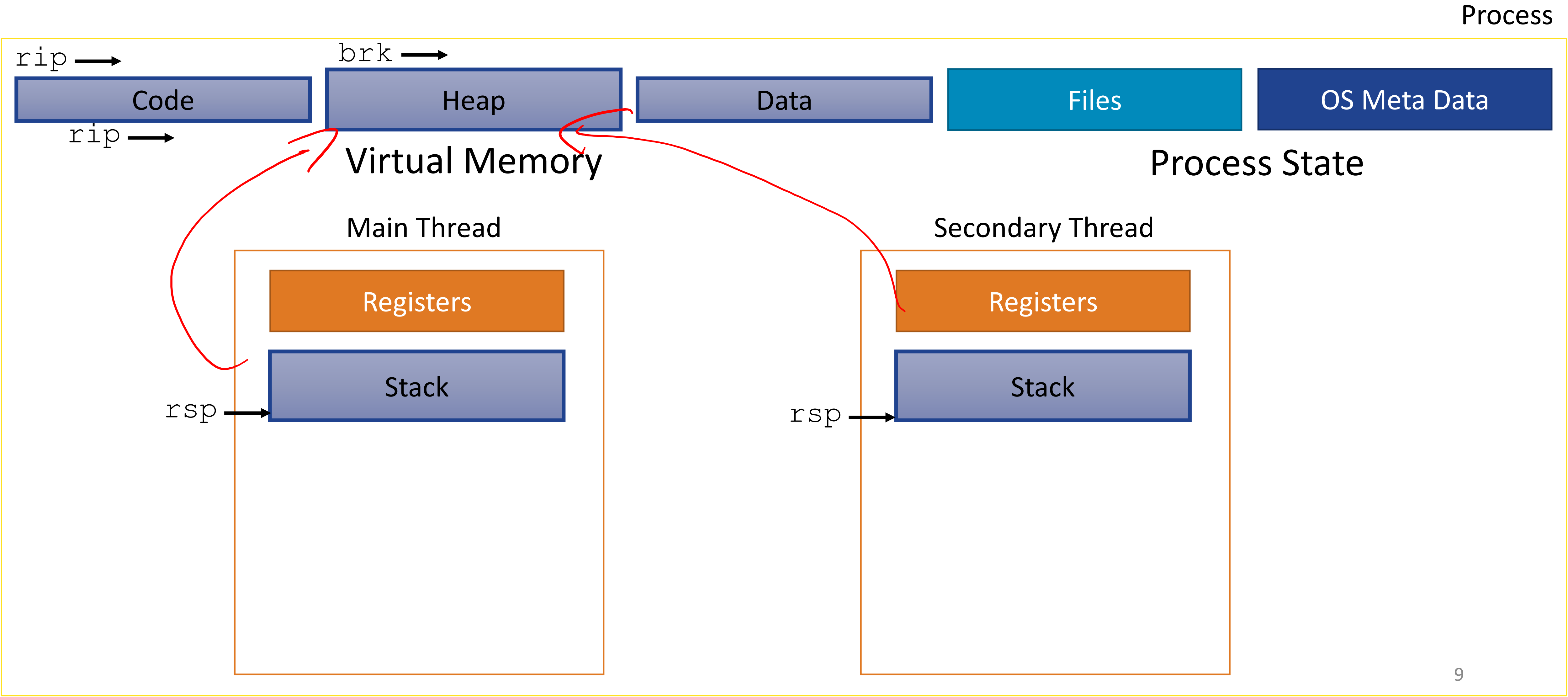
Main Thread

Registers

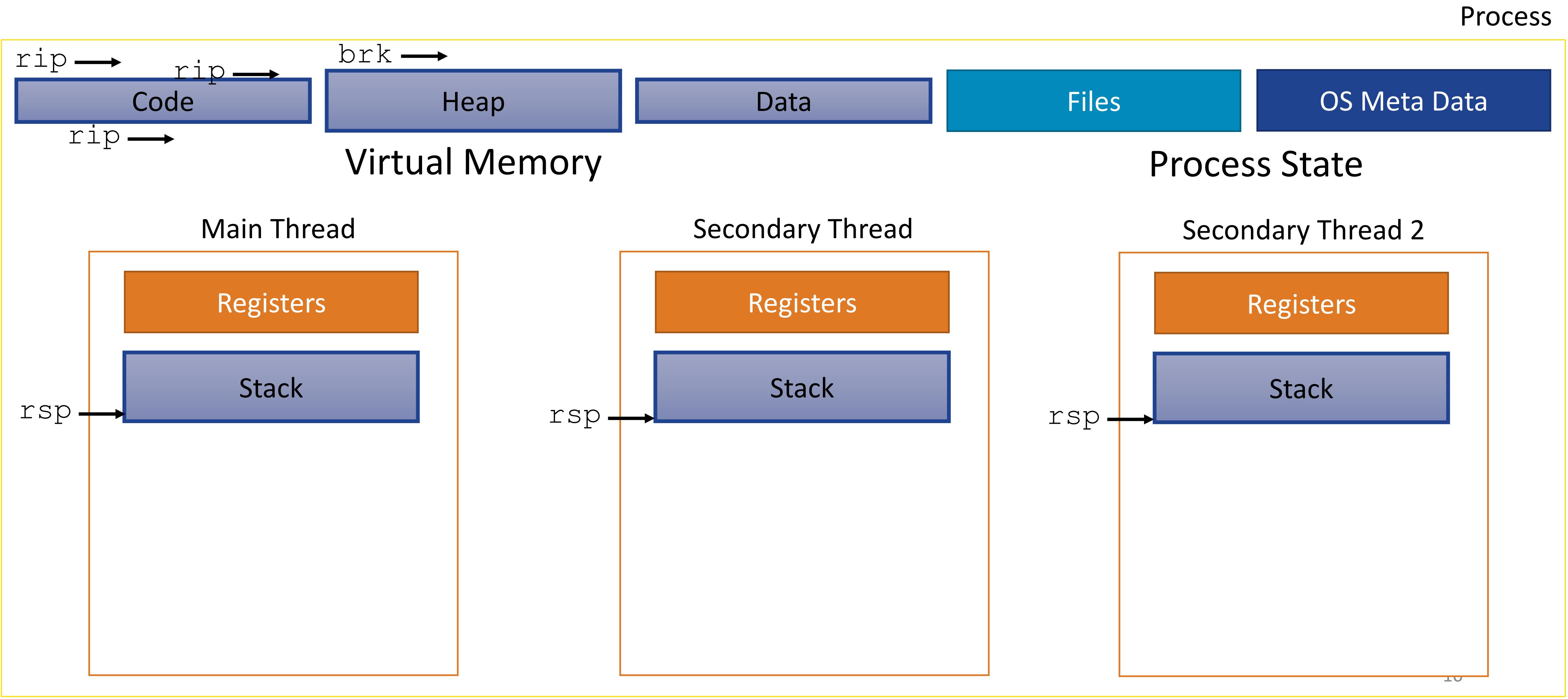
Stack

`rsp`

Threaded View of a Process



Threaded View of a Process



Process

rip →

rip →

rip →

Code

brk →

Heap

Data

Files

OS Meta Data

Virtual Memory

Process State

Main Thread

Registers

Stack

rsp →

Secondary Thread

Registers

Stack

rsp →

Secondary Thread 2

Registers

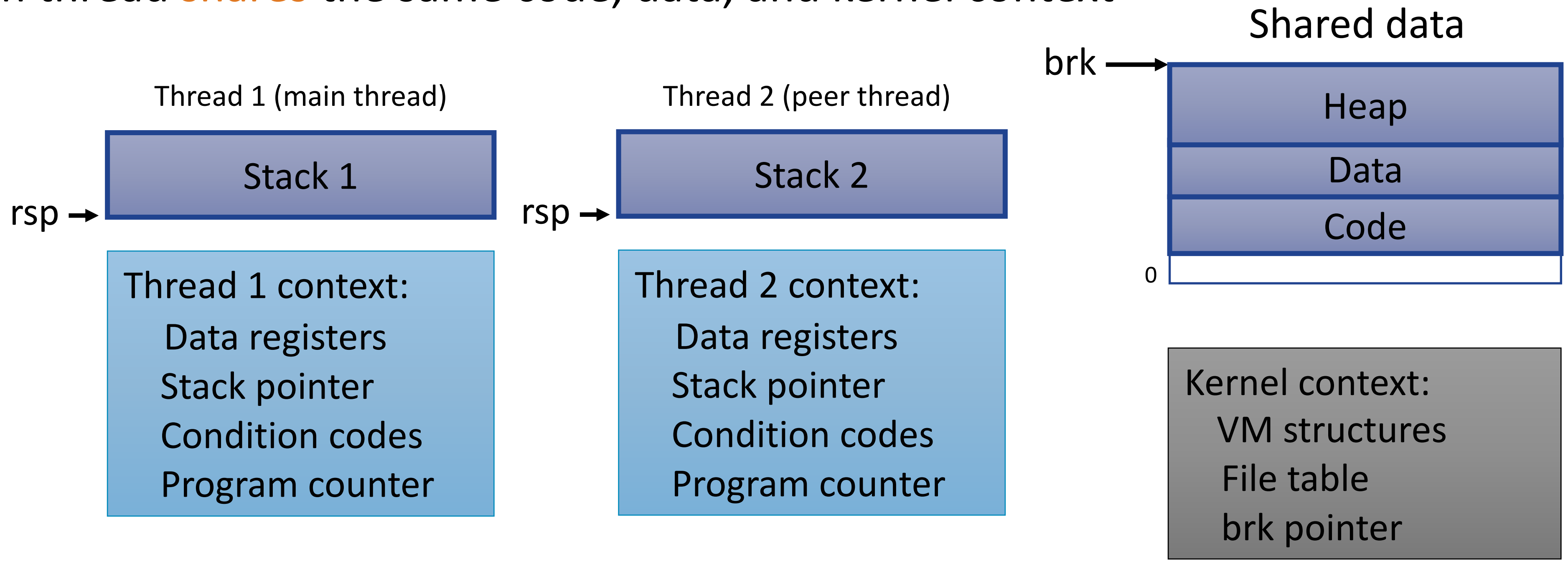
Stack

rsp →

A Process With Multiple Threads

Multiple threads can be associated with a single process

- Each thread has its own logical control flow
- Each thread has its own stack for local variables
- Each thread has its own thread id (`TID`)
- Each thread shares the same code, data, and kernel context



Shared data

brk

Heap

Data

Code

0

Kernel context:
VM structures
File table
brk pointer

Thread 1 (main thread)

rsp

Stack 1

Thread 1 context:
Data registers
Stack pointer
Condition codes
Program counter

Thread 2 (peer thread)

rsp

Stack 2

Thread 2 context:
Data registers
Stack pointer
Condition codes
Program counter

Threads vs. Processes

How threads and processes are similar

- Each has its own logical control flow
- Each can run concurrently with others (possibly on different cores)
- Each is scheduled and context switched

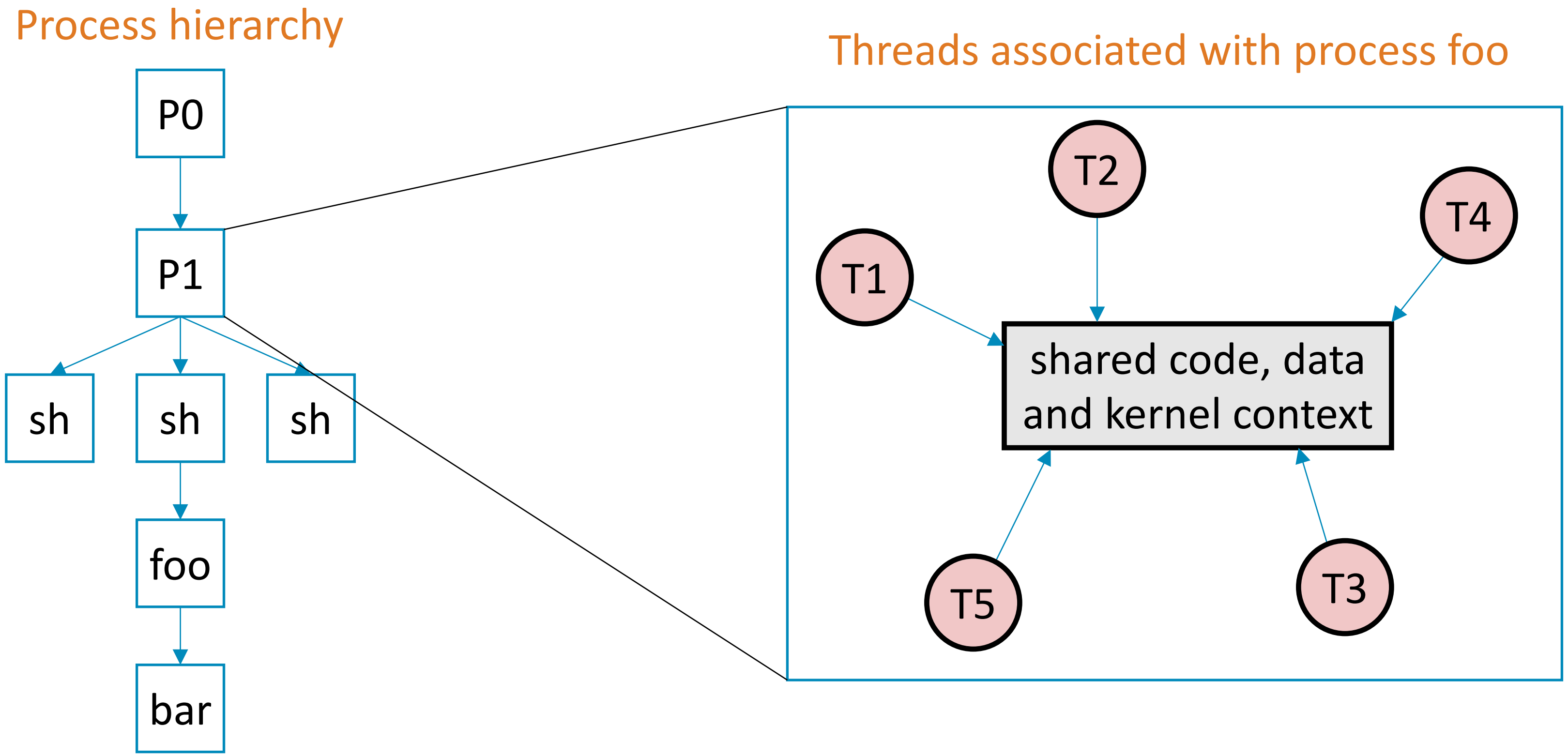
How threads and processes are different

- Threads share all code and data (except local stacks)
- Threads are somewhat less expensive than processes
  - Thread control (creating and reaping) is half as expensive as process control
    - ~20K cycles to create and reap a process
    - ~10K cycles (or less) to create and reap a thread
  - Thread context switches are less expensive (e.g., don't flush TLB)

Logical View of Threads

- Threads associated with process form a pool of peers
- Unlike processes which form a tree hierarchy

How does P0 create P1?



Posix Threads Interface (Pthreads)

- Creating and reaping threads
  - `pthread_create()`
  - `pthread_join()`

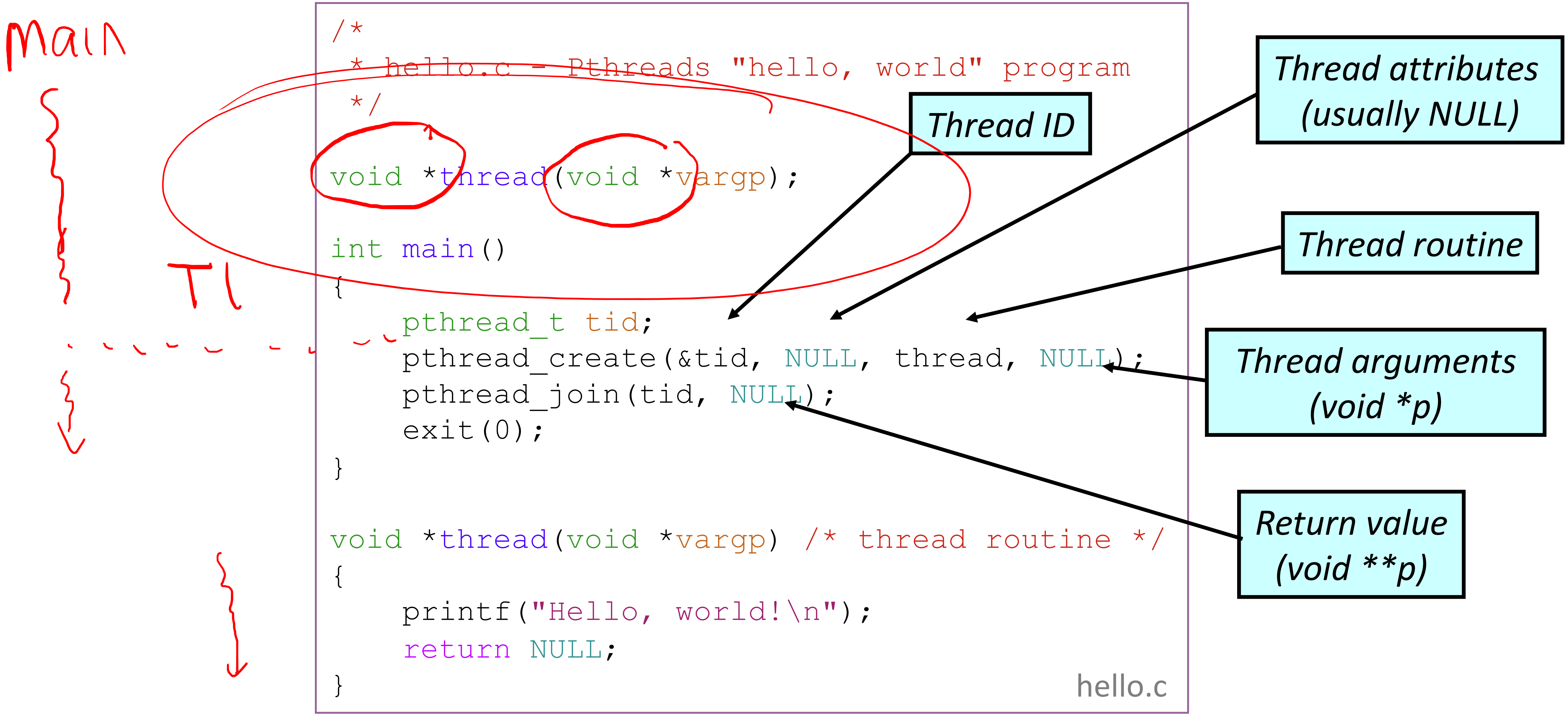
- Determining your thread ID
  - `pthread_self()`

- Terminating threads
  - `pthread_cancel()`
  - `pthread_exit()`
  - `exit()` [terminates all threads]
  - `RET` [terminates current thread]

The Pthreads "hello, world" Program

main

T1



```
/*
 * hello.c - Pthreads "hello, world" program
 */

void *thread(void *vargp);

int main()
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    exit(0);
}

void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

Thread ID

Thread attributes (usually NULL)

Thread routine

*Thread arguments (void *p)*

*Return value (void **p)*

Example Program to Illustrate Sharing

Where is ptr stored?

Where is msgs stored?

```
char **ptr;  /* global var */

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(
          &tid,
          NULL,
          thread,
          (void *)i
        );
    pthread_exit(NULL);
}
```

race condition

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
    return NULL;
}
```

Where is myid stored?

Where is cnt stored?

Peer threads reference main thread’s stack indirectly through global ptr variable

0 : Hello from foo (cnt=1)

1 : Hello from bar (cnt=2)

Mapping Variable Instances to Memory

- Global variables
  - *Def:* Variable declared outside of a function
  - **Virtual memory contains exactly one instance of any global variable**
- Local variables
  - *Def:* Variable declared inside function without `static` attribute
  - **Each stack frame contains one instance of each local variable**
- Local static variables
  - *Def:* Variable declared inside function with the `static` attribute
  - **Virtual memory contains exactly one instance of any local static variable.**

Mapping Variable Instances to Memory

```
char **ptr;   /* global var */

int main(){
  long i;
  pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar"};

  ptr = msgs;
  for (int i = 0; i < 2; i++)
    pthread_create(&tid, NULL,
                   thread, (void *)i);
  pthread_exit(NULL);
}
```

```
void *thread(void *vargp){
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]: %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
  return NULL;
}
```

Global var: 1 instance (`ptr` [data])

Local vars: 1 instance (`i.m`, `msgs.m`)

Local var: 2 instances (
`myid.p0` [peer thread 0's stack],
`myid.p1` [peer thread 1's stack]
)

Local static var: 1 instance (`cnt` [data])

Practice with Shared Variables

```
char **ptr;  /* global var */

int main(){
  long i;
  pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar"};
  ptr = msgs;
  for (int i = 0; i < 2; i++)
    pthread_create(&tid, NULL,
                   thread, (void *)i);
  pthread_exit(NULL);
}
```

```
void *thread(void *vargp){
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]: %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
  return NULL;
}
```

Which variables are shared?

- `ptr`
- `cnt`
- `i.main`
- `msgs.main`
- `myid.thread0`
- `myid.thread1`

Practice with Shared Variables

```
char **ptr;  /* global var */

int main(){
  long i;
  pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar"};
  ptr = msgs;
  for (int i = 0; i < 2; i++)
    pthread_create(&tid, NULL,
                   thread, (void *)i);
  pthread_exit(NULL);
}
```

```
void *thread(void *vargp){
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]: %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
  return NULL;
}
```

Which variables are shared?

- `ptr`
- `cnt`
- `i.main`
- `msgs.main`
- `myid.thread0`
- `myid.thread1`

A variable `x` is shared iff multiple threads reference at least one instance of `x`.

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.main`	yes	no	no
`msgs.main`	yes	yes	yes
`myid.thread0`	no	yes	no
`myid.thread1`	no	no	yes

What can go wrong?

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv){
    long num_incs;
    pthread_t tid1, tid2;

    num_incs = atoi(argv[1]);
    pthread_create(&tid1, NULL, thread, &num_incs);
    pthread_create(&tid2, NULL, thread, &num_incs);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * num_incs))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

```c
/* Thread routine */
void *thread(void *vargp){
  long i, num_incs;
  num_incs =  *((long *)vargp);

  for (i = 0; i < num_incs; i++){
    cnt++;
  }

  return NULL;
}
```

```
linux> ./badcnt 10000
OK cnt=20000

linux> ./badcnt 10000
BOOM! cnt=13051

linux>
```

```
/* Thread routine */
void *thread(void *vargp){
  long i, num_incs;
  num_incs =  *((long *)vargp);

  for (i = 0; i < num_incs; i++){
    cnt++;
  }

  return NULL;
}
```

```
thread:
        mov     rcx, QWORD PTR [rdi]
        test    rcx, rcx
        jle     .L2
        mov     edx, 0
.L3:
        mov     rax, QWORD PTR cnt[rip]
        add     rax, 1
        mov     QWORD PTR cnt[rip], rax
        add     rdx, 1
        cmp     rcx, rdx
        jne     .L3
.L2:
        mov     eax, 0
        ret
cnt:
        .zero   8
```

Load `cnt`

Increment register holding `cnt`

Store new value back to `cnt`

```c
/* Thread routine */
void *thread(void *vargp){
  long i, num_incs;
  num_incs =  *((long *)vargp);

  for (i = 0; i < num_incs; i++){
    cnt++;
  }

  return NULL;
}
```

What happens if the thread is preempted in the middle?

```
thread:
        mov     rcx, QWORD PTR [rdi]
        test    rcx, rcx
        jle     .L2
        mov     edx, 0
.L3:
        mov     rax, QWORD PTR cnt[rip]
        add     rax, 1
        mov     QWORD PTR cnt[rip], rax
        add     rdx, 1
        cmp     rcx, rdx
        jne     .L3
.L2:
        mov     eax, 0
        ret
cnt:
        .zero   8
```

Load `cnt`

Increment register holding `cnt`

Store new value back to `cnt`

Thread 1	Thread 2
1. Load `cnt`	1.
2.	2. Load `cnt`
3.	3. Increment register
4.	4. Store `cnt`
5. Increment register	5.
6. Store `cnt`	6.

Race conditions

- A race condition is a timing-dependent error involving shared state
    - Whether the error occurs depends on thread schedule
- Program execution/schedule can be non-deterministic
- Compilers and processors can re-order instructions

A concrete example…

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.
- **Liveness:** if you are out of milk, someone buys milk
- **Safety:** you never have more than one quart of milk

Algorithm 1:

```
Look in fridge.
If out of milk:
    go to store,
    buy milk,
    go home
    put milk in fridge
```

Algorithm 1:

```
if (milk == 0) {      // no milk
  milk++;             // buy milk
}
```

A problematic schedule

You		**Your Roommate**	
3:00	Look in fridge; out of milk		
3:05	Leave for store		
3:10	Arrive at store	3:10	Look in fridge; out of milk
3:15	Buy milk	3:15	Leave for store
3:20	Arrive home	3:20	Arrive at store
3:21	Put milk in fridge	3:25	Buy milk
		3:30	Arrive home
		3:31	Put milk in fridge

Safety violation:
You have too much milk and it spoils

Solution 1: Leave a note

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Algorithm 2:

```
if (milk == 0) {    // no milk
  if (note == 0) {  // no note
    note = 1;       // leave note
    milk++;         // buy milk
    note = 0;       // remove note
  }
}
```

Algorithm 2:

```
if (milk == 0) {    // no milk
  if (note == 0) {  // no note
    note = 1;       // leave note
    milk++;         // buy milk
    note = 0;       // remove note
  }
}
```

Solution 1: Leave a note

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Safety violation: you've introduced a Heisenbug!

Algorithm 2:

```
if (milk == 0) {     // no milk
  if (note == 0) {   // no note
    note = 1;        // leave note
    milk++;          // buy milk
    note = 0;        // remove note
  }
}
```

Algorithm 2:

```
if (milk == 0) {     // no milk
  if (note == 0) {   // no note
    note = 1;        // leave note
    milk++;          // buy milk
    note = 0;        // remove note
  }
}
```

Solution 2: Leave note before check note

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Algorithm 3:

```
note1 = 1
if (note2 == 0) {
if (milk == 0) {
    milk++;
  }
}
note1 = 0
```

Algorithm 3:

```
note2 = 1
if (note1 == 0) {
if (milk == 0) {
    milk++;
  }
}
note2 = 0
```

Solution 2: Leave note before check note

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Liveness violation: No one buys milk

Algorithm 3:

```
note1 = 1
if (note2 == 0) {
if (milk == 0) {
    milk++;
  }
}
note1 = 0
```

Algorithm 3:

```
note2 = 1
if (note1 == 0) {
if (milk == 0) {
    milk++;
  }
}
note2 = 0
```

Solution 3: Keep checking for note

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Algorithm 4:

```
note1 = 1
while (note2 == 1) {
  ;
}
if (milk == 0) {
  milk++;
}
note1 = 0
```

Algorithm 4:

```
note2 = 1
while (note1 == 1) {
  ;
}
if (milk == 0) {
  milk++;
}
note2 = 0
```

Solution 3: Keep checking for note

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Liveness violation: You've introduced deadlock

Algorithm 4:

```
note1 = 1
while (note2 == 1) {
   ;
}
if (milk == 0) {
  milk++;
}
note1 = 0
```

Algorithm 4:

```
note2 = 1
while (note1 == 1) {
   ;
}
if (milk == 0) {
  milk++;
}
note2 = 0
```

Solution 4: Take turns

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Algorithm 5:

```
note1 = 1
turn = 2
while (note2 == 1 and turn == 2){
  ;
}
if (milk == 0) {
  milk++;
}
note1 = 0
```

Algorithm 5:

```
note2 = 1
turn = 2
while (note1 == 1 and turn == 2){
  ;
}
if (milk == 0) {
  milk++;
}
note2 = 0
```

Solution 4: Take turns

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

(probably) correct, but complicated and inefficient

Algorithm 5:

```
note1 = 1
turn = 2
while (note2 == 1 and turn == 2){
   ;
}
if (milk == 0) {
   milk++;
}
note1 = 0
```

Algorithm 5:

```
note2 = 1
turn = 2
while (note1 == 1 and turn == 2){
   ;
}
if (milk == 0) {
   milk++;
}
note2 = 0
```

Locks

A **lock** (aka a mutex) is a synchronization primitive that provides mutual exclusion. When one thread holds a lock, no other thread can hold it.

- A lock can be in one of two states: locked or unlocked
- A lock is initially unlocked

- Function `acquire(&lock)` waits until the lock is unlocked, then atomically sets it to locked

- Function `release(&lock)` sets the lock to unlocked

Solution 5: use a lock

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Algorithm 6:

```
acquire(&milk_lock)
if (milk == 0) {
  milk++;
}
release(&milk_lock)
```

Algorithm 6:

```
acquire(&milk_lock)
if (milk == 0) {
  milk++;
}
release(&milk_lock)
```

Solution 5: use a lock

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Simpler and Correct!

Algorithm 6:

```
acquire(&milk_lock)
if (milk == 0) {
  milk++;
}
release(&milk_lock)
```

Algorithm 6:

```
acquire(&milk_lock)
if (milk == 0) {
  milk++;
}
release(&milk_lock)
```

Atomic Operations

- Solution: hardware primitives to support synchronization
- A machine instruction that (atomically!) reads and updates a memory location
- Example: `xchg DEST, SRC`
  - one instruction
  - semantics: `TEMP ← DEST; DEST ← SRC; SRC ← TEMP;`

Spinlocks

```
acquire:
    mov  eax, 1         ; Set EAX to 1
    xchg [rdi], eax     ; Atomically swap EAX w/ lock val
    test eax, eax       ; Check if EAX is 0 (lock unlocked)
    jnz  acquire        ; If was locked, loop
    ret                 ; Lock has been acquired, return

release:
    xor  eax, eax       ; Set EAX to 0
    xchg [rdi], eax     ; Atomically swap EAX w/ lock val
    ret                 ; Lock has been released, return
```

Programming with Locks (Pthreads)

- Defines lock type `pthread_mutex_t`

- Functions to create/destroy locks:
  - `int pthread_mutex_init(&lock, attr);`
  - `int pthread_mutex_destroy(&lock);`

- Functions to acquire/release lock:
  - `int pthread_mutex_lock(&lock);`
  - `int pthread_mutex_unlock(&lock);`

Practice with Locks

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv){
    long num_incs;
    pthread_t tid1, tid2;

    num_incs = atoi(argv[1]);
    pthread_create(&tid1, NULL, thread, &num_incs);
    pthread_create(&tid2, NULL, thread, &num_incs);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * num_incs))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

```
/* Thread routine */
void *thread(void *vargp){
  long i, num_incs;
  num_incs =  *((long *)vargp);

  for (i = 0; i < num_incs; i++){
    cnt++;
  }

  return NULL;
}
```

Modify this example to guarantee correctness

Practice with Locks

Create lock

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv){
    long num_incs;
    pthread_t tid1, tid2;

    num_incs = atoi(argv[1]);
    pthread_create(&tid1, NULL, thread, &num_
    pthread_create(&tid2, NULL, thread, &num_
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * num_incs))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

Acquire lock

Release lock

```
/* Thread routine */
void *thread(void *vargp){
  long i, num_incs;
  num_incs =  *((long *)vargp);

  for (i = 0; i < num_incs; i++){
    cnt++;
  }

  return NULL;
}
```

Modify this example to guarantee correctness

Problems with Locks

1. Locks are slow
   - Threads that fail to acquire a lock on the first attempt must "spin", which wastes CPU cycles
   - Threads get scheduled and de-scheduled while the lock is still locked

2. Using locks correctly is (surprisingly) hard
   - Hard to ensure all race conditions are eliminated
   - Easy to introduce synchronization bugs (deadlock, livelock)
   - Gets much harder when you have multiple needed resources

Better Synchronization Primitives

- Semaphores
    - Stateful synchronization primitive
- Condition variables
    - Event-based synchronization primitive
- These are the topic of our next class period