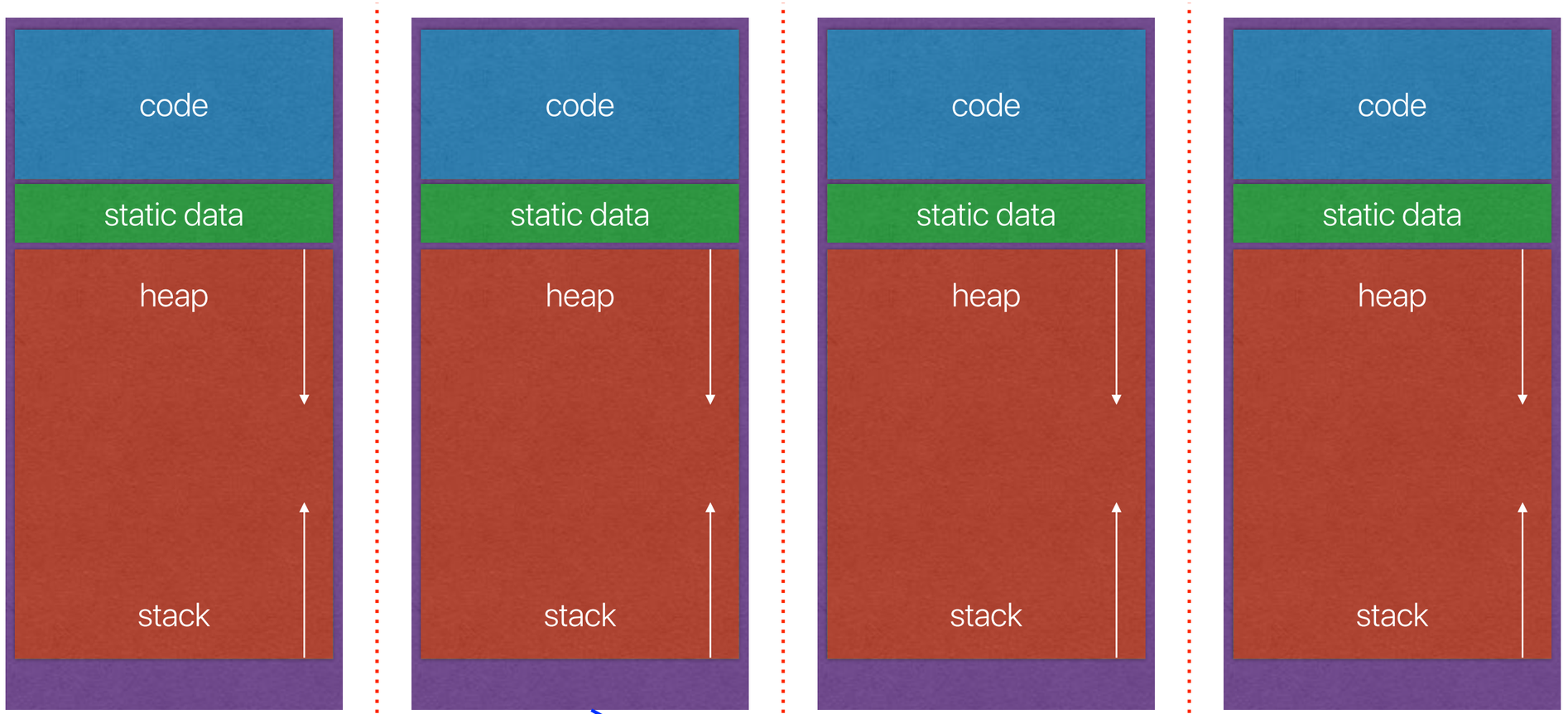
Design philosophy of operating systems (IV)

Hung-Wei Tseng

Recap: Each process has a separate virtual memory space



They are isolated from one another. Each of them is not supposed to know what happens to another one

Processor

Virtually, every process seems to have a processor, but only a few of them are physically executing.

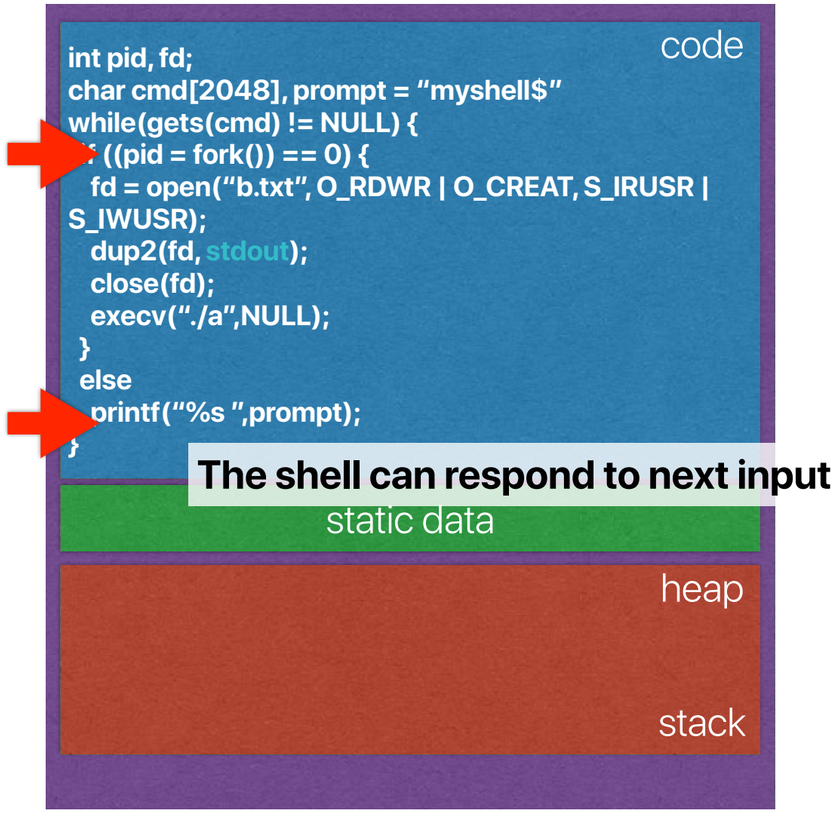
Recap: The basic process API of UNIX

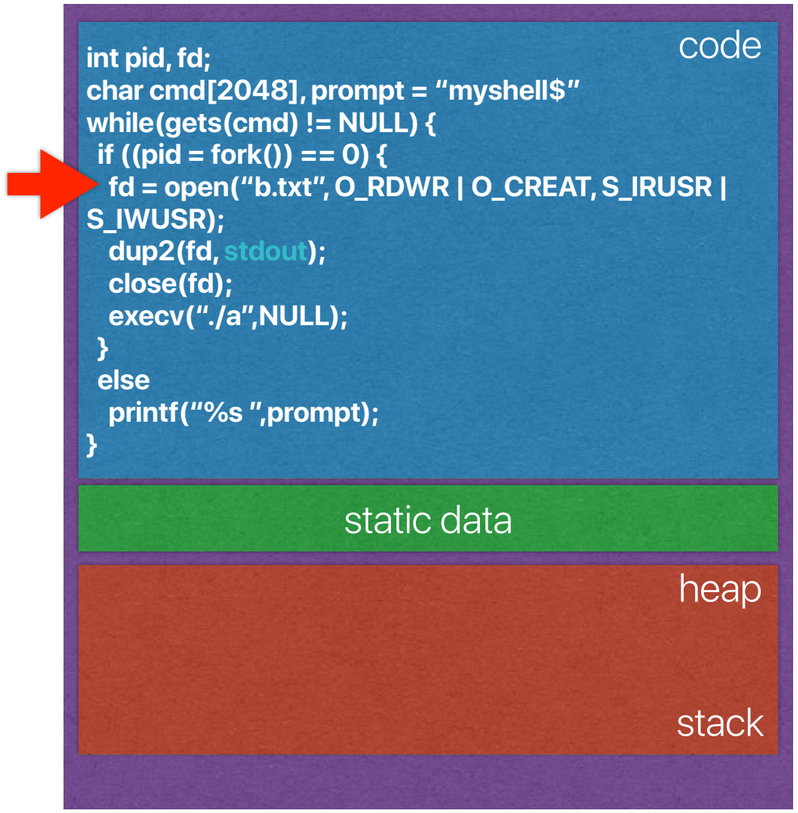
- `fork`
- `wait`
- `exec`
- `exit`

Recap: How to implement redirection in shell

- Say, we want to do ./a > b.txt
- fork
- The forked code opens b.txt
- The forked code dup the file descriptor to stdin/stdout
- The forked code closes b.txt
- exec("./a", NULL)

Homework for you:
Think about the case when
your `fork` is equivalent to `fork+exec()`





Why "Mach"?

- The hardware is changing
  - Multiprocessors
  - Networked computing
- The software
  - The demand of extending an OS easily
  - Repetitive but confusing mechanisms for similar stuffs

> be built and future development of UNIX-like systems for new architectures can continue. The computing environment for which Mach is targeted spans a wide class of systems, providing basic support for large, general purpose multiprocessors, smaller multiprocessor networks and individual workstations (see

> As the complexity of distributed environments and multiprocessor architectures increases, it becomes increasingly important to return to the original UNIX model of consistent interfaces to system facilities. Moreover, there is a clear need to allow the underlying system to be transparently extended to allow user-state processes to provide services which in the past could only be fully integrated into UNIX by adding code to the operating system kernel.

Make UNIX great again!

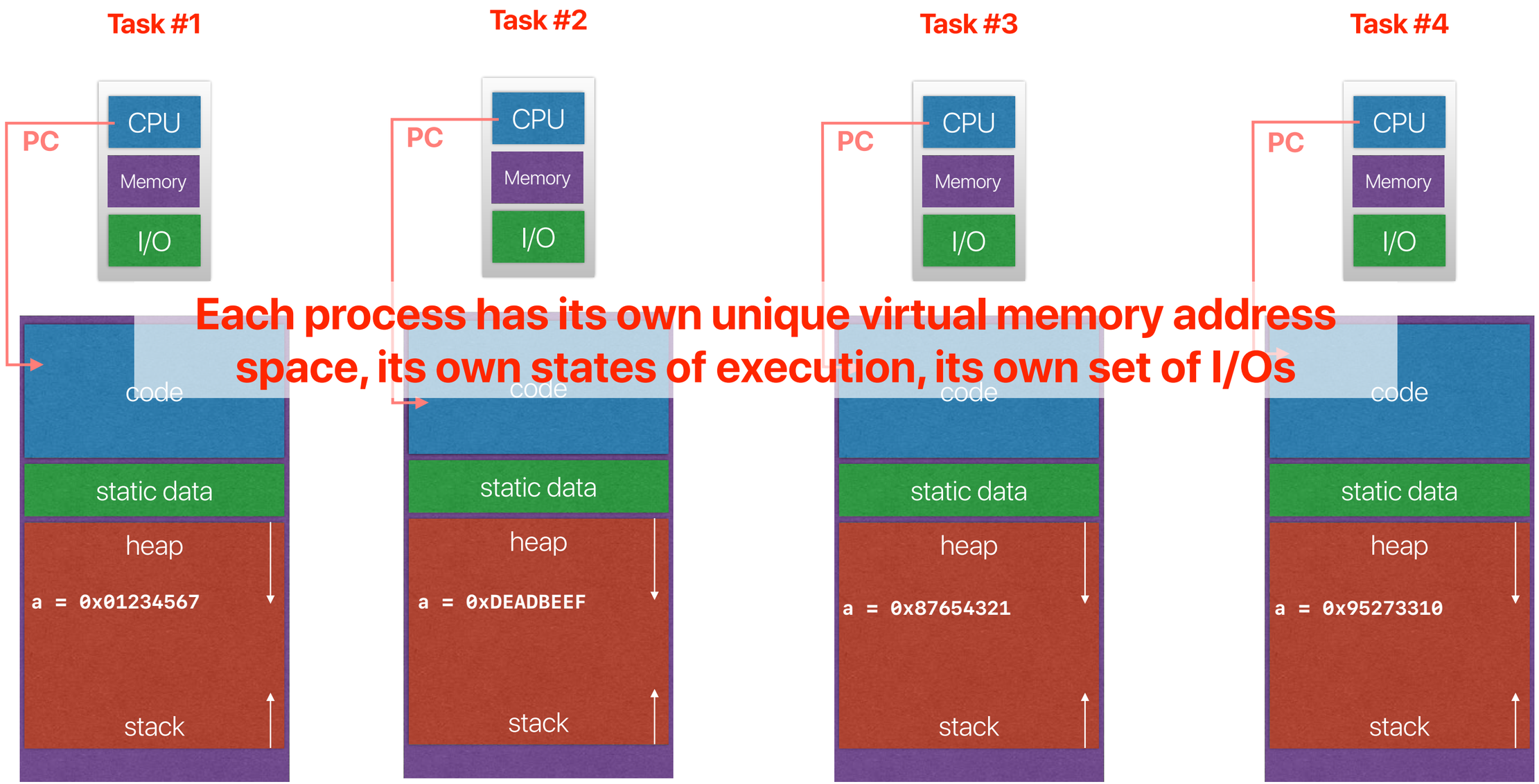
Outline

- Mach: A New Kernel Foundation For UNIX Development (cont.)
- Taxonomy of Kernels
- Synchronization

Mach: A New Kernel Foundation For UNIX Development

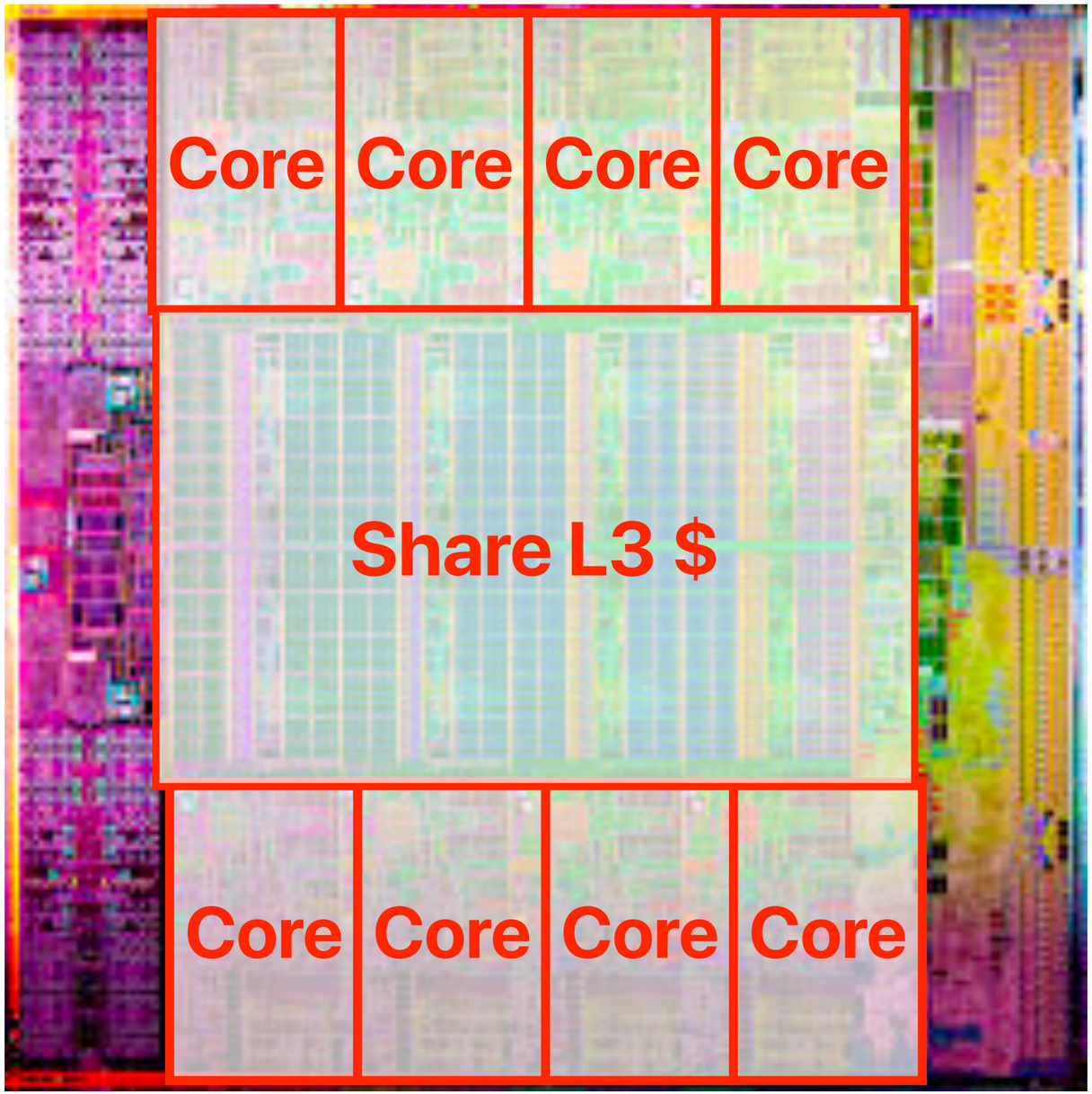
Mike Accetta , Robert Baron , William Bolosky , David Golub , Richard Rashid , Avadis Tevanian , Michael Young
Computer Science Department, Carnegie Mellon University

Tasks/processes

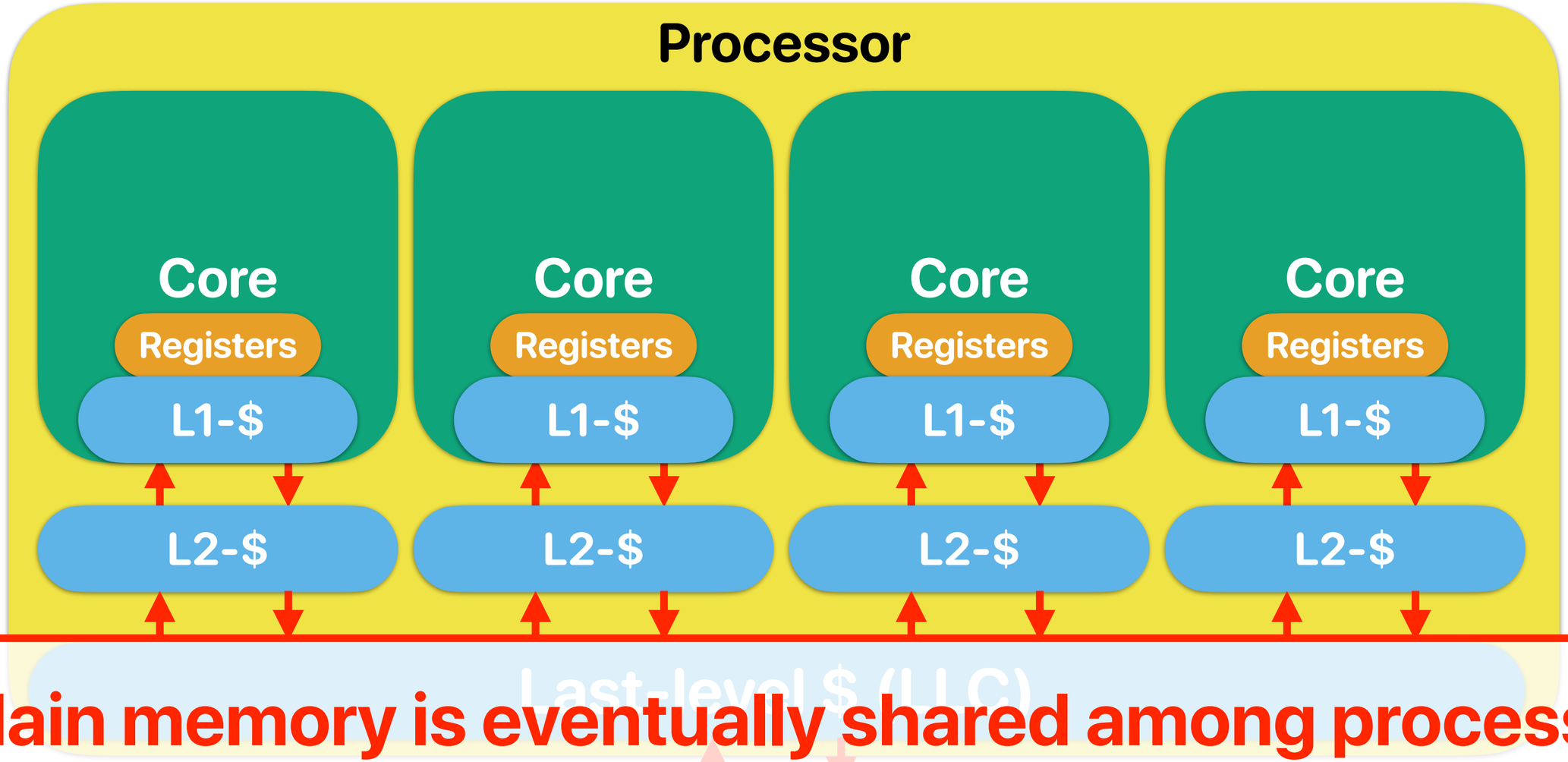


Each process has its own unique virtual memory address space, its own states of execution, its own set of I/Os

Intel Sandy Bridge

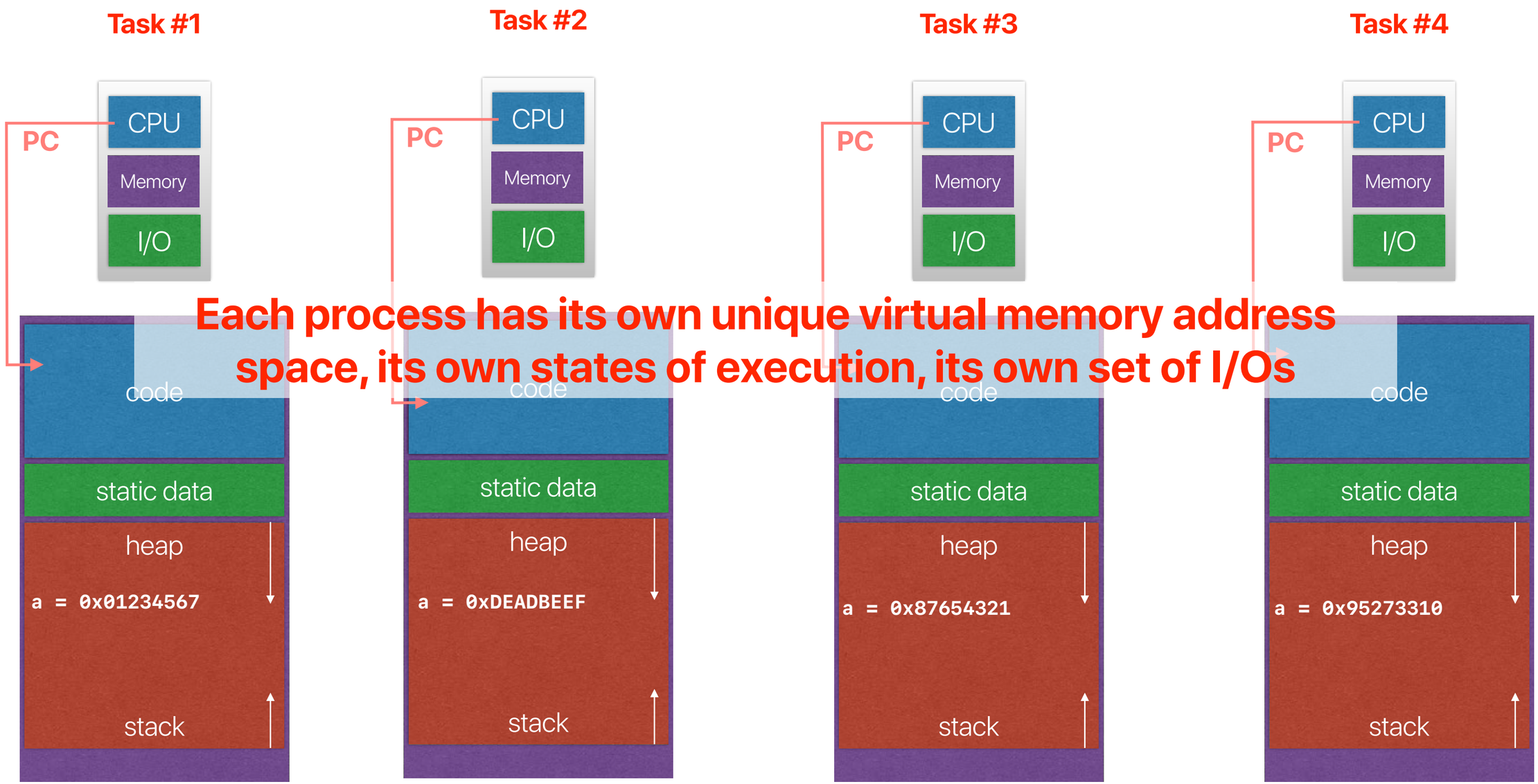


Concept of chip multiprocessors



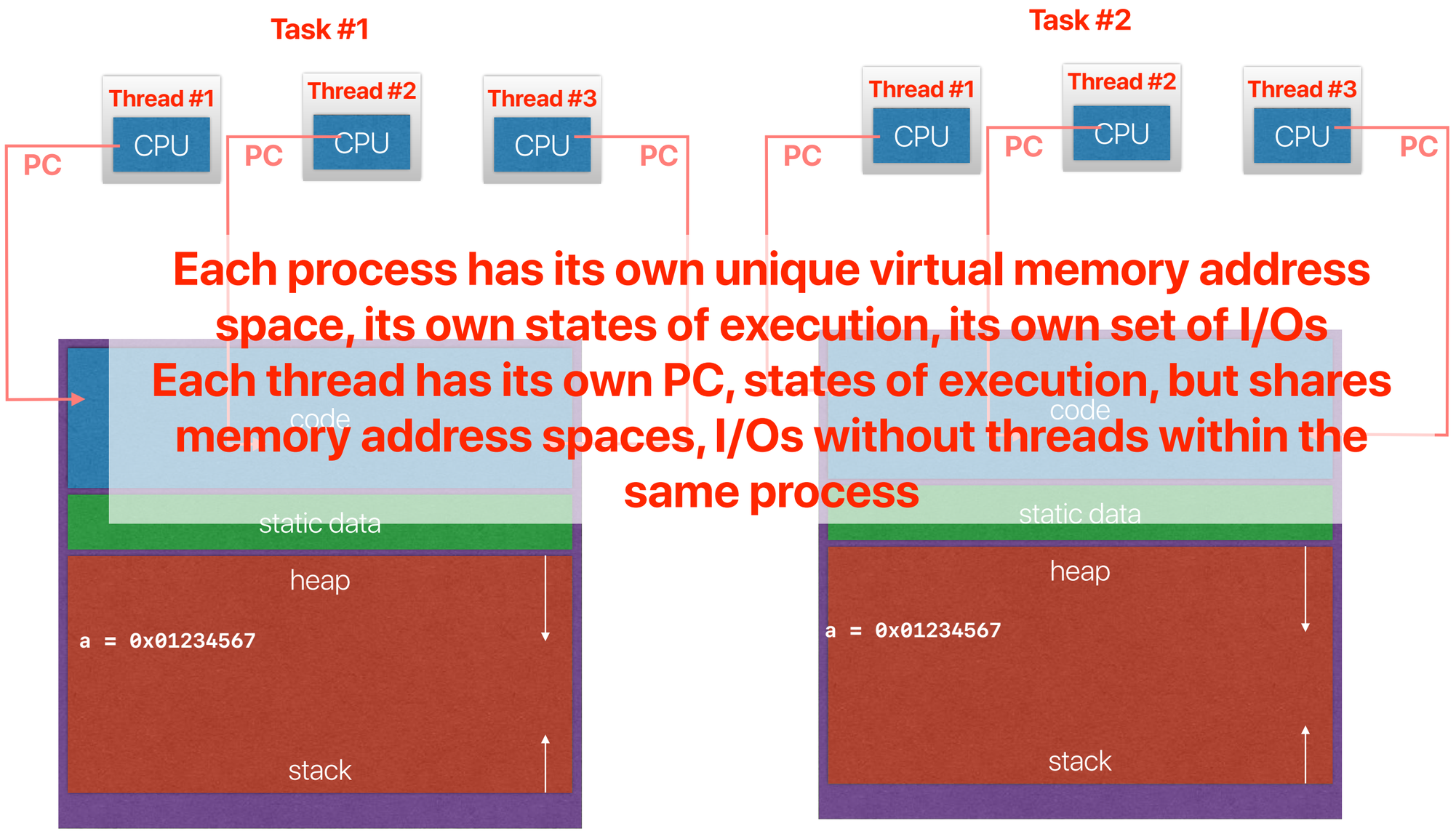
Main memory is eventually shared among processor cores

Tasks/processes



Each process has its own unique virtual memory address space, its own states of execution, its own set of I/Os

Threads



Each process has its own unique virtual memory address space, its own states of execution, its own set of I/Os

Each thread has its own PC, states of execution, but shares memory address spaces, I/Os without threads within the same process

Tasks/Processes and threads

- How many of the following regarding the comparison of parallelizing computation tasks using processes and threads is/are correct?
  - ① The context switch and creation overhead of processes is higher
  - ② The overhead of exchanging data among different computing tasks for the same applications is higher in process model
  - ③ The demand of memory usage is higher when using processes
  - ④ The security and isolation guarantees are better achieved using processes

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

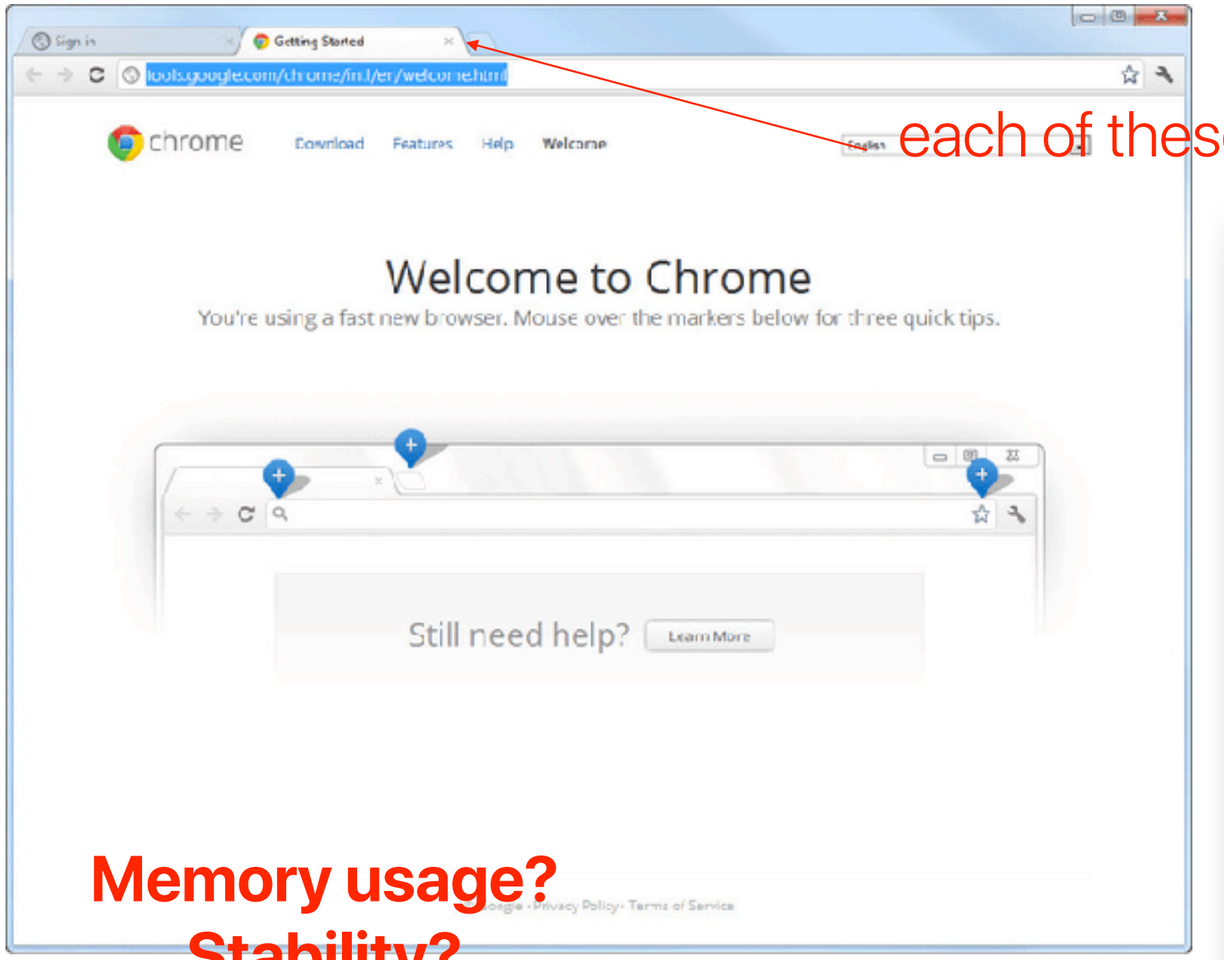
— you have to change page tables, warm up TLBs, warm up caches, create a new memory space …

— you cannot directly share data without leveraging other mechanisms

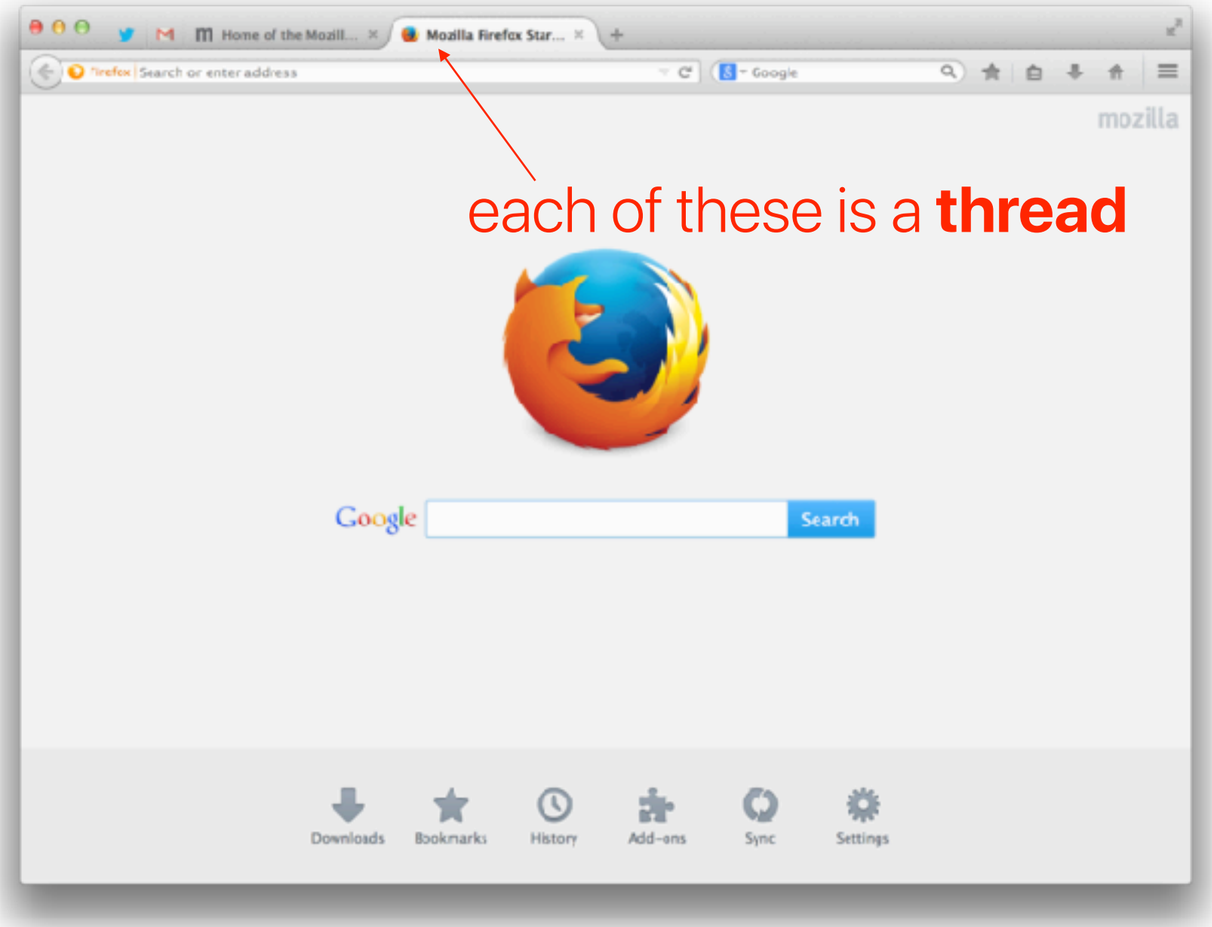
— each process needs its own address space even if most data are potentially identical

— separate address, it's not easy to access data from another process

Case study: Chrome v.s. Firefox



each of these is a **process**



each of these is a **thread**

Memory usage?

Stability?

Security?

Latency?

Poll close in 1:30

What's in the kernel?

- How many of the following Mach features/functions are implemented in the kernel?
  - ① I/O device drivers
  - ② File system
  - ③ Shell
  - ④ Virtual memory management
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

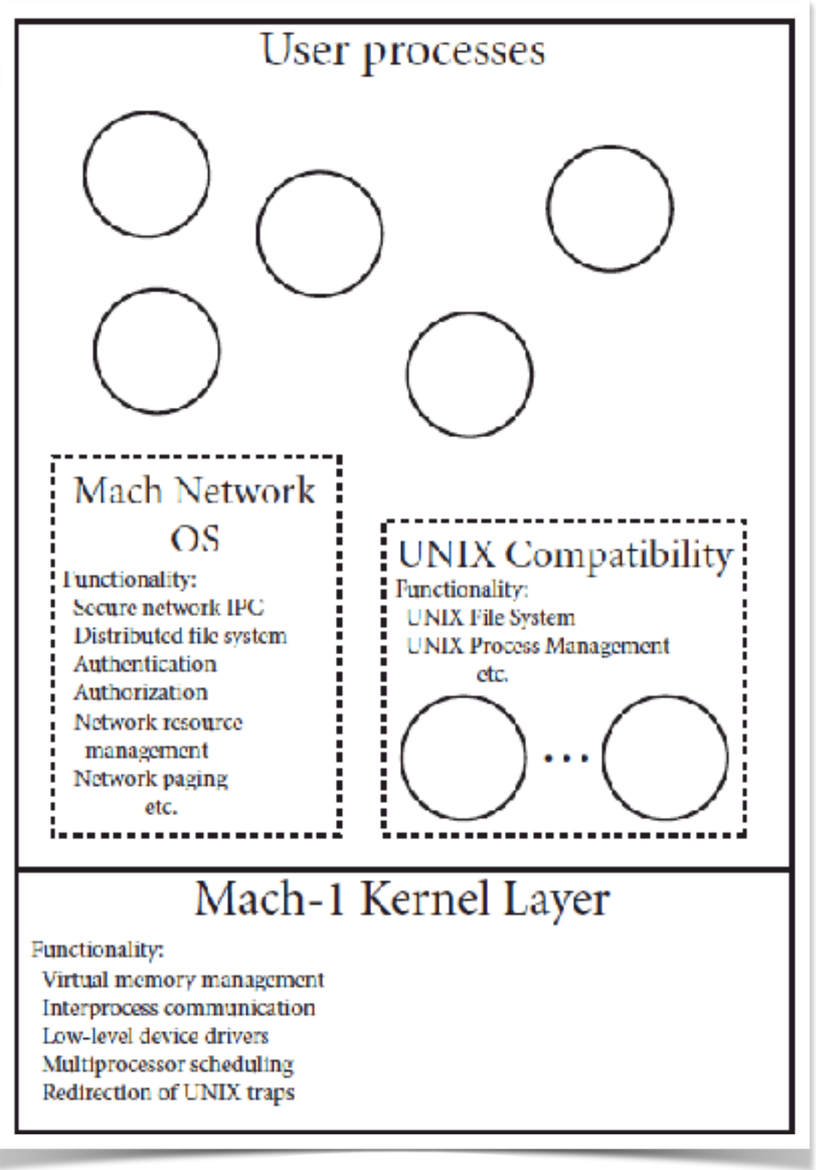
Poll close in 1:30

What's in the kernel?

- How many of the following Mach features/functions are implemented in the kernel?
  - ① I/O device drivers
  - ② File system
  - ③ Shell
  - ④ Virtual memory management
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

What's in the kernel?

- How many of the following Mach features/fun
  implemented in the kernel?
  - ① I/O device drivers
  - ② File system
  - ③ Shell
  - ④ Virtual memory management
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4



Policies

Mechanisms

Poll close in 1:30

Policy? Mechanisms?

- How many of the following terms belongs to “policies”?
  - ① Least-recently used (LRU)
  - ② First-in, first-out
  - ③ Paging
  - ④ Preemptive scheduling
  - ⑤ Capability
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Poll close in 1:30

Policy? Mechanisms?

- How many of the following terms belongs to “policies”?
  - ① Least-recently used (LRU)
  - ② First-in, first-out
  - ③ Paging
  - ④ Preemptive scheduling
  - ⑤ Capability

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Policy? Mechanisms?

- How many of the following terms belongs to "policies"?
  - ① Least-recently used (LRU) — **Policy**
  - ② First-in, first-out — **Policy**
  - ③ Paging — **Mechanism**
  - ④ Preemptive scheduling — **Mechanism**
  - ⑤ Capability — **Mechanism**
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Whys v.s. whats

- How many pairs of the “why” and the “what” in Mach are correct?

	Why	What
(1)	Support for multiprocessors	**Threads**
(2)	Networked computing	**Messages/Ports**
(3)	OS Extensibility	**Microkernel/Object-oriented design**
(4)	Repetitive but confusing mechanisms	**Messages/Ports**

A. 0
B. 1
C. 2
D. 3
E. 4

Poll close in 1:30

Types of kernels

- What type of kernels does the UNIX described in Dennis M. Ritchie's paper belong to?
  A. Microkernel — the kernel only provides a minimal set of services/ mechanisms including memory management, multitasking and inter-process communication
  B. Monolithic — the kernel implements every function that cannot be in a user-space library: device drivers, scheduler, memory handling, file systems, network stacks
  C. Modular — the kernel provides a basic set of functions like microkernels, but allows load/unload kernel modules if necessary
  D. Layered kernel — the kernel follows strict layered design that lower-order module cannot interact with higher-order modules

Poll close in 1:30

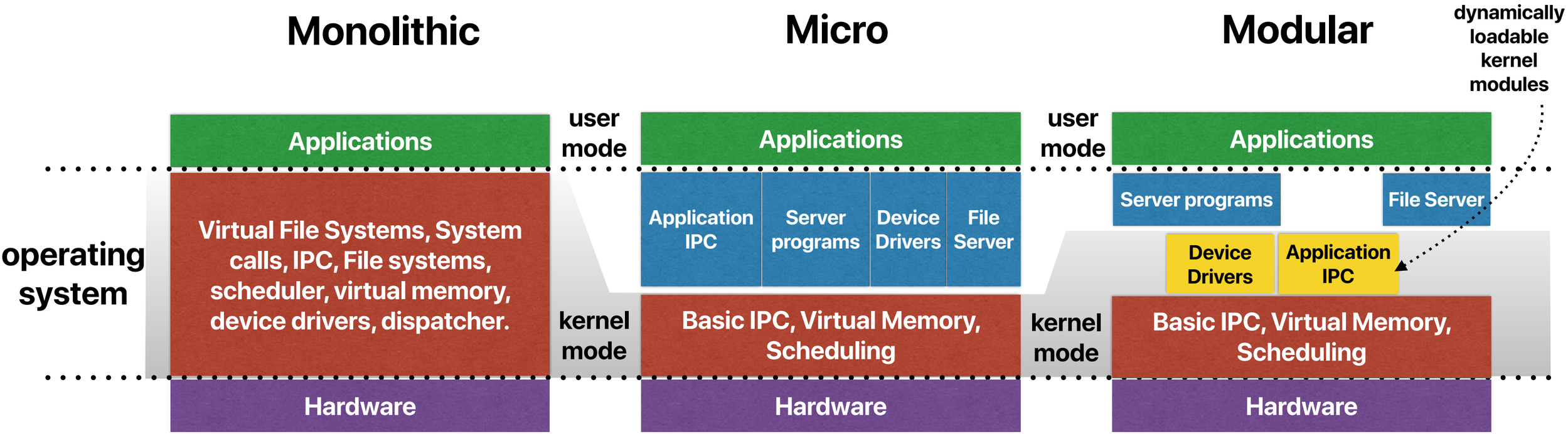
Types of kernels

- What type of kernels does the UNIX described in Dennis M. Ritchie's paper belong to?
  A. Microkernel — the kernel only provides a minimal set of services/ mechanisms including memory management, multitasking and inter-process communication
  B. Monolithic — the kernel implements every function that cannot be in a user-space library: device drivers, scheduler, memory handling, file systems, network stacks
  C. Modular — the kernel provides a basic set of functions like microkernels, but allows load/unload kernel modules if necessary
  D. Layered kernel — the kernel follows strict layered design that lower-order module cannot interact with higher-order modules

Types of kernels

- What type of kernels does the UNIX described in Dennis M. Ritchie's paper belong to?
  A. Microkernel — the kernel only provides a minimal set of services/ mechanisms including memory management, multitasking and inter-process communication **Mach, Nucleus**
  B. Monolithic — the kernel implements every function that cannot be in a user-space library: device drivers, scheduler, memory handling, file systems, network stacks **Old UNIX**
  C. Modular — the kernel provides a basic set of functions like microkernels, but allows load/unload kernel modules if necessary **Linux, Windows, MacOS, FreeBSD**
  D. Layered kernel — the kernel follows strict layered design that lower-order module cannot interact with higher-order modules **THE**

Types of Kernels



Only mechanisms are in the kernel

Original UNIX

Nucleus, Mach

Linux, Windows, MacOS

Poll close in 1:30

Why not microkernels?

- Although Mach's design strongly influenced modern operating systems, why most modern operating systems do not adopt the design of microkernels?
  A. Microkernels are more difficult to extend than monolithic kernels
  B. Microkernels are more difficult to maintain than monolithic kernels
  C. Microkernels are less stable than monolithic kernels
  D. Microkernels are not as competitive as monolithic kernels in terms of application performance
  E. Microkernels are less flexible than monolithic kernels

Poll close in 1:30

Why not microkernels?

- Although Mach's design strongly influenced modern operating systems, why most modern operating systems do not adopt the design of microkernels?
  A. Microkernels are more difficult to extend than monolithic kernels
  B. Microkernels are more difficult to maintain than monolithic kernels
  C. Microkernels are less stable than monolithic kernels
  D. Microkernels are not as competitive as monolithic kernels in terms of application performance
  E. Microkernels are less flexible than monolithic kernels

Why not microkernels?

- Although Mach's design strongly influenced modern operating systems, why most modern operating systems do not adopt the design of microkernels?
  A. Microkernels are more difficult to extend than monolithic kernels
  B. Microkernels are more difficult to maintain than monolithic kernels
  C. Microkernels are less stable than monolithic kernels
  D. Microkernels are not as competitive as monolithic kernels in terms of application performance **Context switches!**
  E. Microkernels are less flexible than monolithic kernels

The impact of Mach

- Threads
- Extensible operating system kernel design
- Strongly influenced modern operating systems
  - Windows NT/2000/XP/7/8/10
  - MacOS

Mach Overview

The fundamental services and primitives of the OS X kernel are based on Mach 3.0. Apple has modified and extended Mach to better meet OS X functional and p

Mach 3.0 was originally conceived as a simple, extensible, communications microkernel. It is capable of running as a stand-alone kernel, with other traditional o networking stacks running as user-mode servers.

However, in OS X, Mach is linked with other kernel components into a single kernel address space. This is primarily for performance; it is much faster to make a messages or do remote procedure calls (*RPC*) between separate tasks. This modular structure results in a more robust and extensible system than a monolithic k microkernel.

Thus in OS X, Mach is not primarily a communication hub between clients and servers. Instead, its value consists of its abstractions, its extensibility, and its flex

- object-based APIs with communication channels (for example, ports) as object references
- highly parallel execution, including preemptively scheduled threads and support for *SMP*
- a flexible scheduling framework, with support for real-time usage
- a complete set of *IPC* primitives, including messaging, *RPC*, synchronization, and notification
- support for large virtual address spaces, shared memory regions, and memory objects backed by persistent store
- proven extensibility and portability, for example across instruction set architectures and in distributed environments
- security and resource management as a fundamental principle of design; all resources are virtualized

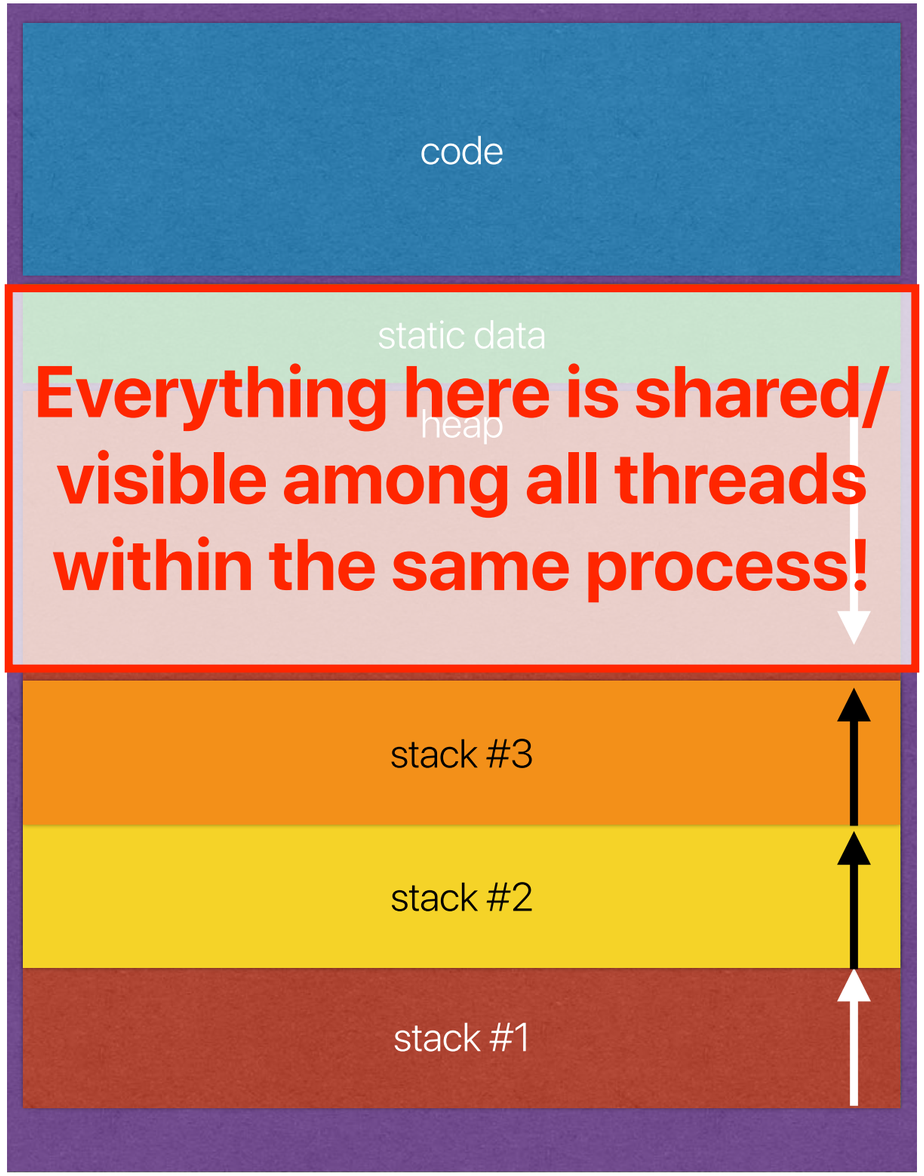
Mach Kernel Abstractions

Mach provides a small set of abstractions that have been designed to be both simple and powerful. These are the main kernel abstractions:

- *Tasks*. The units of resource ownership; each task consists of a virtual address space, a *port right namespace*, and one or more *threads*. (Similar to a process.
- *Threads*. The units of CPU execution within a task.
- *Address space*. In conjunction with memory managers, Mach implements the notion of a sparse virtual address space and shared memory.
- *Memory objects*. The internal units of memory management. Memory objects include named entries and regions; they are representations of potentially persi
- *Ports*. Secure, simplex communication channels, accessible only via send and receive capabilities (known as port rights).
- *IPC*. Message queues, remote procedure calls, notifications, semaphores, and lock sets.
- *Time*. Clocks, timers, and waiting.

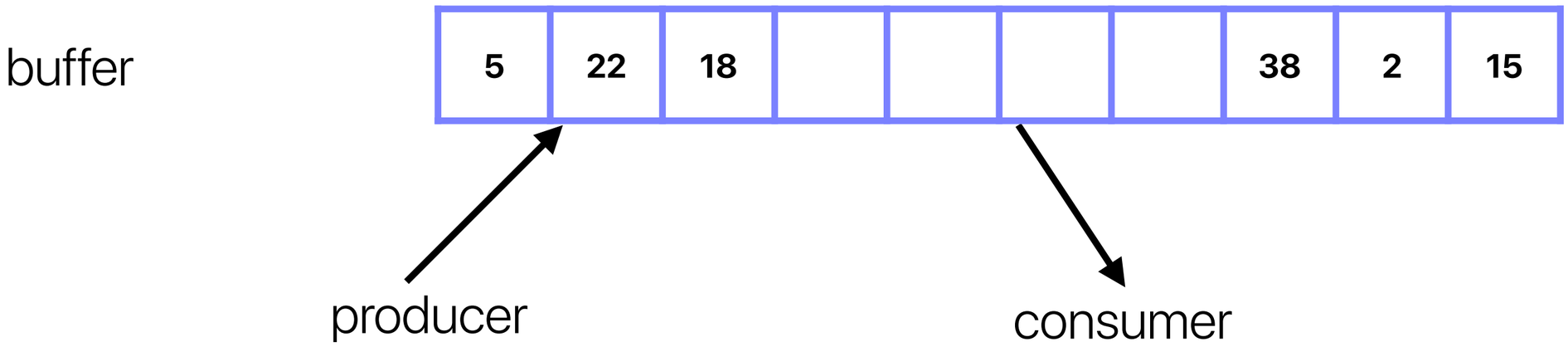
Thread programming & synchronization

The virtual memory of multithreaded applications



Bounded-Buffer Problem

- Also referred to as "producer-consumer" problem
- Producer places items in shared buffer
- Consumer removes items from shared buffer



We need to control accesses to the buffer!

```
int buffer[BUFF_SIZE]; // shared global
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;

        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;

    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {

        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;

        // do something w/ item
    }
    return NULL;
}
```

Solving the "Critical Section Problem"

1. Mutual exclusion — at most one process/thread in its critical section
2. Progress — a thread outside of its critical section cannot block another thread from entering its critical section
3. Fairness — a thread cannot be postponed indefinitely from entering its critical section
4. Accommodate nondeterminism — the solution should work regardless the speed of executing threads and the number of processors

Use locks

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

How to implement lock/unlock

Naive implementation

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
  ; // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock)
{
    *lock = 0;
}
```

Poll close in 1:30

Naive implementation

- How many of the following can the naive implementation guarantee for the producer-consumer problem?
  - ① At most one process/thread in its critical section
  - ② A thread outside of its critical section cannot block another thread from entering its critical section
  - ③ A thread cannot be postponed indefinitely from entering its critical section
  - ④ The solution should work regardless the speed of executing threads and the number of processors

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
  ; // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

Poll close in 1:30

Naive implementation

- How many of the following can the naive implementation guarantee for the producer-consumer problem?
  - ① At most one process/thread in its critical section
  - ② A thread outside of its critical section cannot block another thread from entering its critical section
  - ③ A thread cannot be postponed indefinitely from entering its critical section
  - ④ The solution should work regardless the speed of executing threads and the number of processors

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
  ; // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

Naive implementation

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

what if context switch happens here?

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock)
{
    *lock = 0;
}
```

Naive implementation

- How many of the following can the naive implementation guarantee for the producer-consumer problem?
  - ~~①~~ At most one process/thread in its critical section
  - ② A thread outside of its critical section cannot block another thread from entering its critical section
  - ③ A thread cannot be postponed indefinitely from entering its critical section
  - ④ The solution should work regardless the speed of executing threads and the number of processors

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
  ; // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

Naive implementation

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

what if context switch happens here?

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

what if the thread crashes/halts here?

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock)
{
    *lock = 0;
}
```

Naive implementation

- How many of the following can the naive implementation guarantee for the producer-consumer problem?
  - ~~①~~ At most one process/thread in its critical section
  - ② A thread outside of its critical section cannot block another thread from entering its critical section
  - ~~③~~ A thread cannot be postponed indefinitely from entering its critical section
  - ④ The solution should work regardless the speed of executing threads and the number of processors

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
  ; // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

Naive implementation

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

what if context switch happens here?

what if the thread crashes/halts here?

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;         // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock)
{
    *lock = 0;
}
```

all threads can see lock as 0 at this point

Naive implementation

- How many of the following can the naive implementation guarantee for the producer-consumer problem?
  - ~~①~~ At most one process/thread in its critical section
  - ② A thread outside of its critical section cannot block another thread from entering its critical section
  - ~~③~~ A thread cannot be postponed indefinitely from entering its critical section
  - ~~④~~ The solution should work regardless the speed of executing threads and the number of processors

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
  ; // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

Naive implementation

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

what if context switch happens here?

what if the thread crashes/halts here?

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;         // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock)
{
    *lock = 0;
}
```

all threads can see lock as 0 at this point

coherence cache misses? page fault?

Naive implementation

- How many of the following can the naive implementation guarantee for the producer-consumer problem?
  - ✗ At most one process/thread in its critical section
  - ✗ A thread outside of its critical section cannot block another thread from entering its critical section
  - ✗ A thread cannot be postponed indefinitely from entering its critical section
  - ✗ The solution should work regardless the speed of executing threads and the number of processors

A. 0
B. 1
C. 2
D. 3
E. 4

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
  ; // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

Announcement

- Reading quizzes due next Tuesday
  - Welcome new friends! — will drop a total of 6 reading quizzes for the quarter
  - Attendance count as 4 reading quizzes
  - We plan to have a total of 11 reading quizzes
- Office Hour links are inside Google Calendar events
  - https://calendar.google.com/calendar/u/0/r?cid=ucr.edu_b8u6dvkretn6kq6igunlc6bldg@group.calendar.google.com
  - Different links from lecture ones
  - We cannot share through any public channels so that we can better avoid Zoom bombing
- We will make both midterm and final exams online this quarter
  - Avoid the uncertainty of COVID-19
  - Avoid high-density in the classroom (only sits 60 and we have 59 for now) during examines

Computer
Science &
Engineering

202

つづく