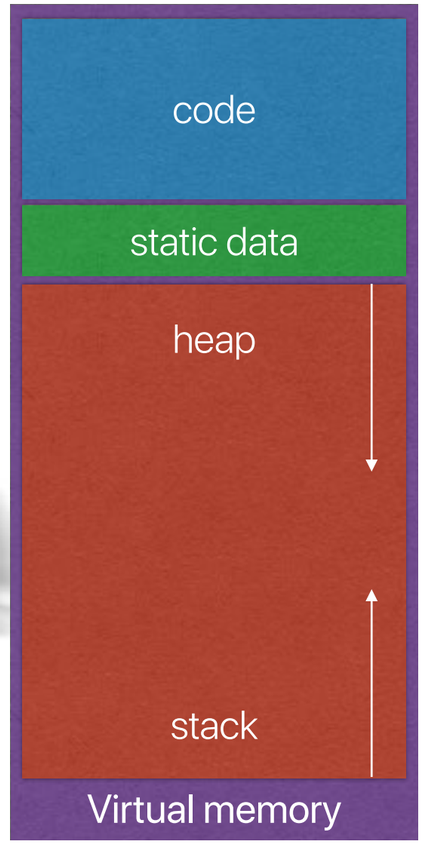
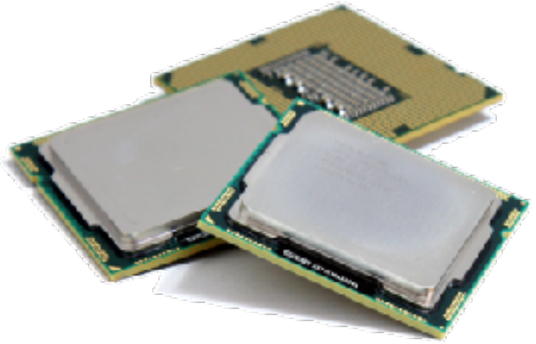
Thread scheduling & Virtual memory (I)

Hung-Wei Tseng

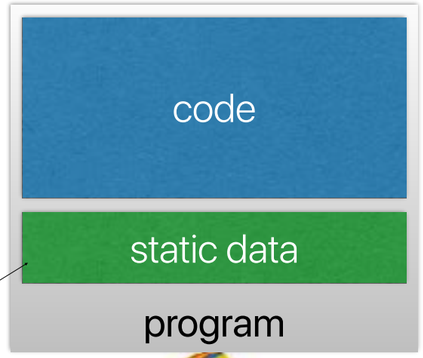
Recap: What happens when creating a process





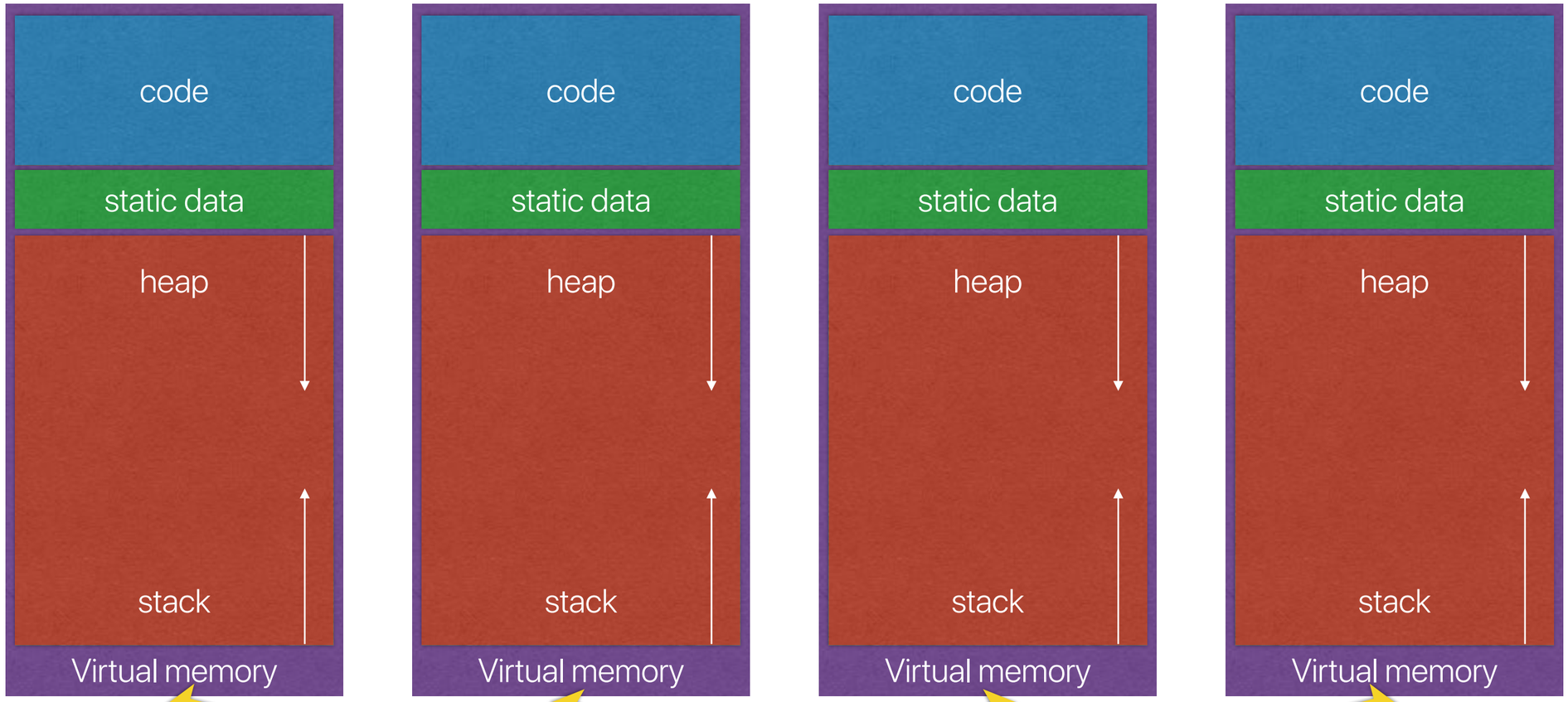
Dynamic allocated data: `malloc()`

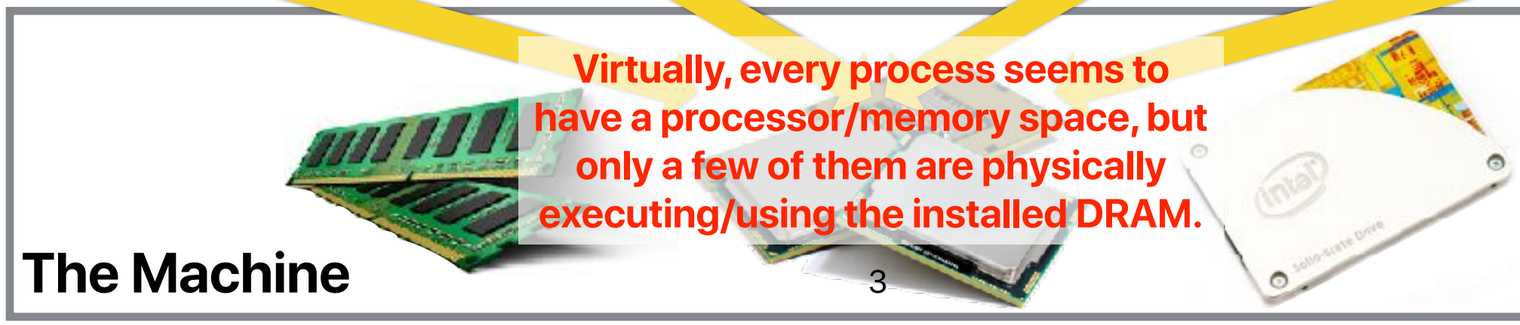
Local variables, arguments



Linux contains a .bss section for uninitialized global variables

Previously, we talked about virtualization





Recap: Virtualizing the processor

- The mechanism
  - Non-preemptive/cooperative: the run process itself initiate context switches — by using system calls
  - Preemptive: the OS kernel can actively incur context switches — by using hardware (timer) interrupts
- The policy
  - Non-preemptive
    - First Come First Serve
    - ~~Shortest job first: SJF~~
  - Preemptive
    - Round robin
    - ~~Shortest Time-to-completion~~
    - Multi-level scheduling algorithm

Recap: The Multilevel Scheduling Algorithm

- Place new process in the one of the queue
  - Depending on the program size

$$\ell_o = \left\lceil \log_2 \left(\left[\frac{w_p}{w_q} \right] + 1 \right) \right\rceil$$

w_p is the program memory size — smaller ones are assigned to lower numbered queues

Why?

  - **Smaller tasks are given higher priority in the beginning**
- Schedule processes in one of N queues
  - Start in initially assigned queue n
  - Run for 2^n quanta (where n is current depth)
  - If not complete, move to a higher queue (e.g. n +1)
  - **Larger process will execute longer before switch**
- Level m is run only when levels 0 to m-1 are empty
- **Smaller process, newer process are given higher priority**

Why Lottery

enormous impact on throughput and response time. Accurate control over the quality of service provided to users and applications requires support for specifying relative computation rates. Such control is desirable across a wide spectrum of systems. For long-running computations such as scientific applications and simulations, the consumption of computing resources that are shared among users and applications of varying importance must be regulated [Hel93]. For interactive computations such as databases and media-based applications, programmers and users need the ability

We want Quality of Service

Few general-purpose schemes even come close to supporting flexible, responsive control over service rates.

Most approaches are not flexible, responsive

Existing *fair share* schedulers [Hen84, Kay88] and *microeconomic* schedulers [Fer88, Wal92] successfully address some of the problems with absolute priority schemes. However, the assumptions and overheads associated with these systems limit them to relatively coarse control over long-running computations. Interactive systems require

The overhead of running those algorithms are high!

ware systems. In fact, with the exception of hard real-time systems, it has been observed that the assignment of priorities and dynamic priority adjustment schemes are often ad-hoc [Dei90]. Even popular priority-based schemes for CPU allocation such as *decay-usage scheduling* are poorly understood, despite the fact that they are employed by numerous operating systems, including Unix [Hel93].

No body knows how they work...

Recap: How does lottery work?

- Each process hold a certain number of lottery tickets
- Ticket
  - Each ticket represent a chance to win a CPU/resource quanta
  - Each ticket has equal chance to win/use a resource
- Randomize to generate a lottery
- If a process wants to have higher priority
  - Obtain more tickets!

Outline

- Thread scheduling
- When thread programing meets scheduling
- Why virtualize your memory
- Start with the basic proposal — segmentation
- Demand paging

Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska and Henry M. Levy
University of Washington

Poll close in 1:30

User-level v.s kernel threads

- Comparing user-level threads and kernel threads, please identify how many of the following statements are correct.
  - ① The overhead of switching threads is smaller for user-level threads
  - ② The OS scheduler can directly control the scheduling of kernel thread, but not for user-level threads
  - ③ A user-level thread can potentially block all other threads in the same process
  - ④ Implementing the user-level thread library can be achieved without modifying the OS kernel

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

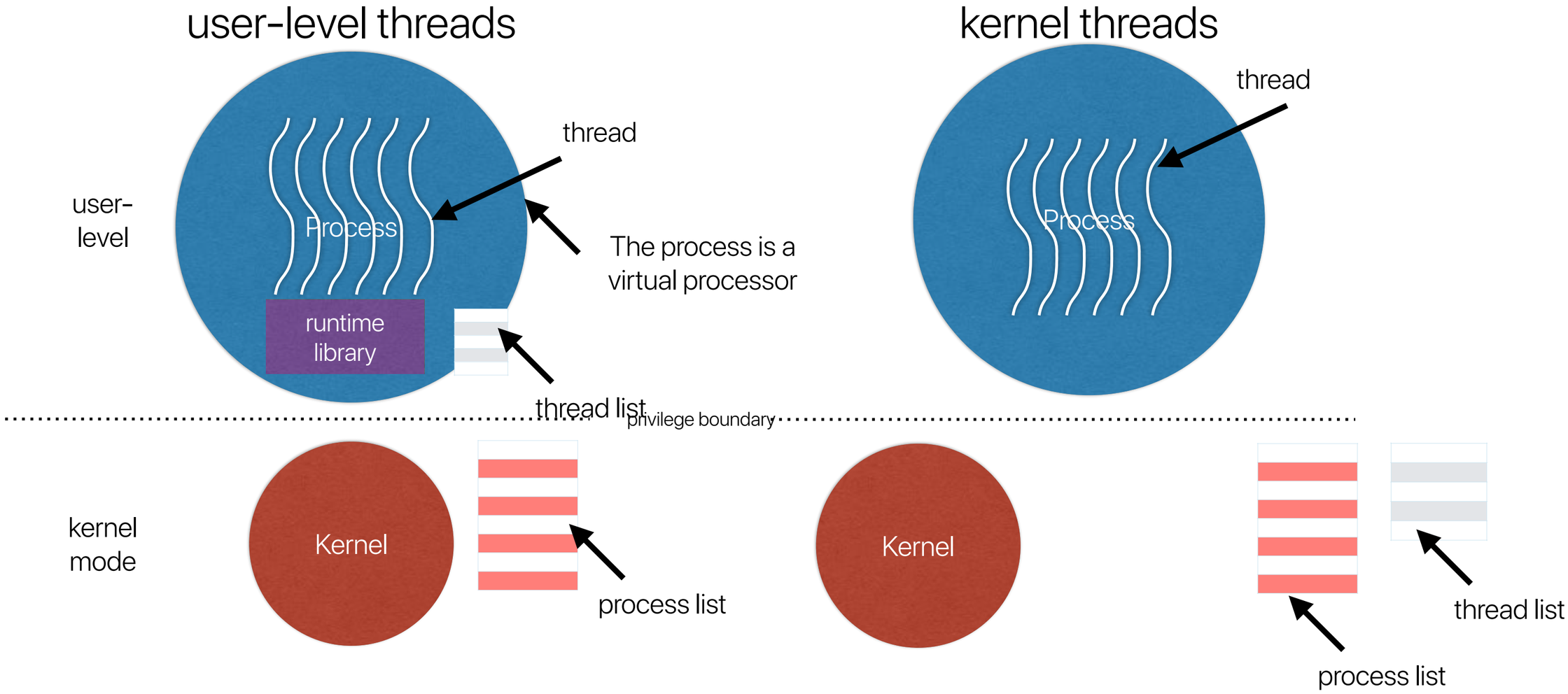
Poll close in 1:30

User-level v.s kernel threads

- Comparing user-level threads and kernel threads, please identify how many of the following statements are correct.
  - ① The overhead of switching threads is smaller for user-level threads
  - ② The OS scheduler can directly control the scheduling of kernel thread, but not for user-level threads
  - ③ A user-level thread can potentially block all other threads in the same process
  - ④ Implementing the user-level thread library can be achieved without modifying the OS kernel

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

User-level v.s kernel threads



- **The OS kernel is unaware of user-level threads**
- **Switching threads does not require kernel mode operations**
- **A thread can block other threads within the same process**

- **The kernel can control threads directly**
- **Thread switch requires kernel/user mode switch and system calls**
- **Thread works individually**

User-level v.s kernel threads

- Comparing user-level threads and kernel threads, please identify how many of the following statements are correct.
  - ① The overhead of switching threads is smaller for user-level threads
  - ② The OS scheduler can directly control the scheduling of kernel thread, but not for user-level threads
  - ③ A user-level thread can potentially block all other threads in the same process
  - ④ Implementing the user-level thread library can be achieved without modifying the OS kernel

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

— kernel threads requires kernel switch!!!

— user threads are not visible from kernel!!!

— because the OS scheduler treat all threads as the same scheduling identity, if one is doing I/O, whole process maybe switched out

— how do you implement "locks" in user-level threads and kernel-level threads?

Why — the "dilemma" of thread implementations

- User-level threads
  - Efficient, flexible, safer, customizable
- Kernel threads
  - Slower, more powerful
  - Better matches the multiprocessor hardware
- Problems
  - OS is only aware of kernel threads
  - OS is unaware of user-level threads as they are hidden behind each process

What does “Scheduler Activations” propose?

- The OS kernel provides each user-level thread system with its own virtual multiprocessor
- Communication mechanism between kernel and user-level

The virtual multiprocessor abstraction

- The kernel allocates processors to an address space/process
  - An address space is shared by all threads within the same process
  - The kernel controls the number of processors to an address space
- Each process has complete control over the processor-thread allocation
- The kernel notifies the address space when the allocated number of processors changes
- The process notifies the kernel when it needs more or fewer processors
- Transparent to users/programmers

How scheduler activation works?

- Create a scheduler activation when the system create a process on a processor
- Create a scheduler activation when the kernel needs to perform an “upcall” user-level
  - Add a processor
  - Processor has been preempted
  - Scheduler activation has blocked
  - Scheduler activation has unblocked
- Downcalls — hints for kernel to perform resource management
  - Add more processors
  - This processor is idle
- Key difference from a kernel thread
  - Kernel never restarts user thread after it is blocked

Will you use Scheduler activation?

- Once been implemented in NetBSD, FreeBSD, Linux
- A user-level thread gets preempted whenever there is scheduling-related event
  - Overhead
  - You may preempt a performance critical thread
- Blocking system call

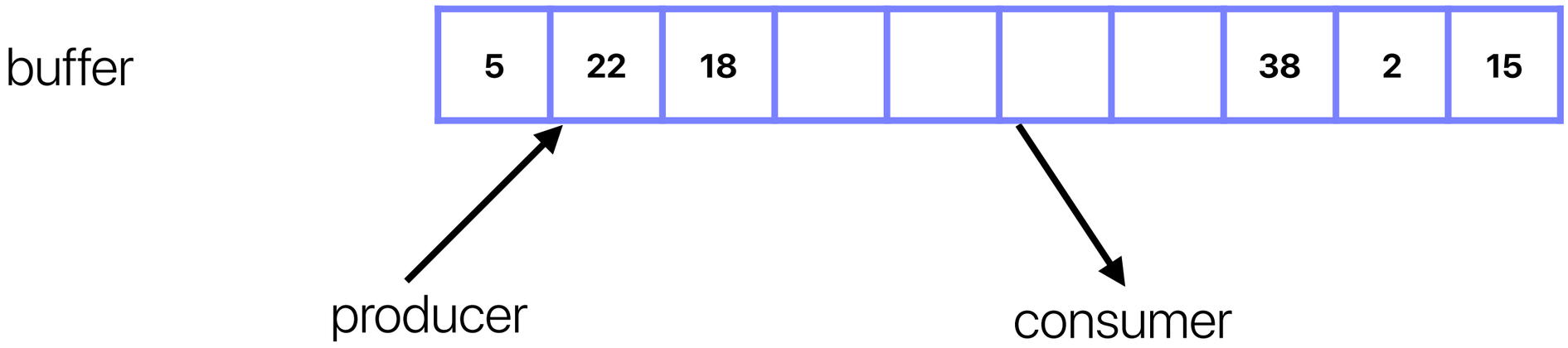
Linux's thread implementation

- Linux treat all schedule identities as "tasks" — context of executions
- COEs can share parts of their contexts with each
  - Processes share nothing
  - Threads share everything but the CPU states
- http://www.evanjones.ca/software/threading-linus-msg.html

When threading meets scheduling

Bounded-Buffer Problem

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



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;
}
```

Use locks

```
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;
}
```

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

Spin locks?

- Which of the following can the spin lock implementation guarantee for the bounded-buffer example?
  A. At most one process/thread in its critical section
  B. A thread outside of its critical section cannot block another thread from entering its critical section
  C. A thread cannot be postponed indefinitely from entering its critical section
  D. The solution should work regardless the speed of executing threads and the number of processors
  E. None of the above

Poll close in 1:30

Spin locks?

- Which of the following can the spin lock implementation guarantee for the bounded-buffer example?
  A. At most one process/thread in its critical section
  B. A thread outside of its critical section cannot block another thread from entering its critical section
  C. A thread cannot be postponed indefinitely from entering its critical section
  D. The solution should work regardless the speed of executing threads and the number of processors
  E. None of the above

Use locks

```
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;
}
```

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

Both threads can grab the lock if context switches occurs in the middle

what if context switch happens here?

what if context switch happens here?

The thread may not be able to enter its critical section because context switched out!

```
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;
}
```

Spin locks?

- Which of the following can the spin lock implementation guarantee for the bounded-buffer example?
  A. At most one process/thread in its critical section
  B. A thread outside of its critical section cannot block another thread from entering its critical section
  C. A thread cannot be postponed indefinitely from entering its critical section
  D. The solution should work regardless the speed of executing threads and the number of processors
  E. None of the above

Both threads can grab the lock if context switches occurs in the middle

What if we have multiple processors?

The thread with the lock may not be able to enter its critical section because context switched out and another thread can keep check the lock!

How to implement lock/unlock

Poll close in 1:30

Disable interrupts?

- Which of the following can disabling interrupts guarantee for for the bounded-buffer example?
  A. At most one process/thread in its critical section
  B. A thread outside of its critical section cannot block another thread from entering its critical section
  C. A thread cannot be postponed indefinitely from entering its critical section
  D. The solution should work regardless the speed of executing threads and the number of processors
  E. None of the above

Poll close in 1:30

Disable interrupts?

- Which of the following can disabling interrupts guarantee for for the bounded-buffer example?
  A. At most one process/thread in its critical section
  B. A thread outside of its critical section cannot block another thread from entering its critical section
  C. A thread cannot be postponed indefinitely from entering its critical section
  D. The solution should work regardless the speed of executing threads and the number of processors
  E. None of the above

Disable interrupts?

- Which of the following can disabling interrupts guarantee for for the bounded-buffer example?

What if we have multiple processors?

A. At most one process/thread in its critical section
B. A thread outside of its critical section cannot block another thread from entering its critical section
C. A thread cannot be postponed indefinitely from entering its critical section
D. The solution should work regardless the speed of executing threads and the number of processors
E. None of the above

What if we have multiple processors?

Use locks

```c
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;
}
```

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

```c
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 context switch happens here?

— the lock must be updated atomically

```c
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;
}
```

Use locks

```
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(&l
        buffer[in] = item;
        in = (in + 1) % BUFF_
        Pthread_mutex_unlock(
    }
    printf("parent: end\n");
    return 0;
}
```

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

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

```
static inline uint xchg(volatile unsigned int *addr,
unsigned int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr),
"=a" (result) : "1" (newval) : "cc");
    return result;
}
```

exchange the content in %0 and %1

a prefix to xchgl that locks the whole cache line

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    // what code should go here?
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    // what code should go here?
}
```

Use locks

```
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(&l
        buffer[in] = item;
        in = (in + 1) % BUFF_
        Pthread_mutex_unlock(
    }
    printf("parent: end\n");
    return 0;
}
```

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

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

```
static inline uint xchg(volatile unsigned int *addr,
unsigned int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr),
"=a" (result) : "1" (newval) : "cc");
    return result;
}

void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1);
}

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

Semaphores

Semaphores

- A synchronization variable
- Has an integer value — current value dictates if thread/process can proceed
- Access granted if val > 0, blocked if val == 0
- Maintain a list of waiting processes

Semaphore Operations

- `sem_wait(S)`
  - if S > 0, thread/process proceeds and decrement S
  - if S == 0, thread goes into “waiting” state and placed in a special queue
- `sem_post(S)`
  - if no one waiting for entry (i.e. waiting queue is empty), increment S
  - otherwise, allow one thread in queue to proceed

Semaphore Op Implementations

```
sem_init(sem_t *s, int initvalue) {
    s->value = initvalue;
}
```

```
sem_wait(sem_t *s) {
    while (s->value <= 0)
        put_self_to_sleep(); // put self to sleep
    s->value--;
}
```

```
sem_post(sem_t *s) {
    s->value++;
    wake_one_waiting_thread(); // if there is one
}
```

Atomicity in Semaphore Ops

- Semaphore operations must operate atomically
  - Requires lower-level synchronization methods requires (test-and-set, etc.)
  - Most implementations still require on busy waiting in spinlocks
- What did we gain by using semaphores?
  - Easier for programmers
  - Busy waiting time is limited

Poll close in 1:30

Using semaphores

- What variables to use for this problem?

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

```
int buffer[BUFF_SIZE]; // shared global
sem_t filled, empty;
```

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

	W	X	Y	Z
A	empty	empty	filled	filled
B	empty	filled	filled	empty
C	filled	empty	empty	filled

Poll close in 1:30

Using semaphores

- What variables to use for this problem?

```
int buffer[BUFF_SIZE]; // shared global
sem_t filled, empty;
```

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

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

	W	X	Y	Z
A	empty	empty	filled	filled
B	empty	filled	filled	empty
C	filled	empty	empty	filled

Using semaphores

- What variables to use for this problem?

```
int buffer[BUFF_SIZE]; // shared global
sem_t filled, empty;
```

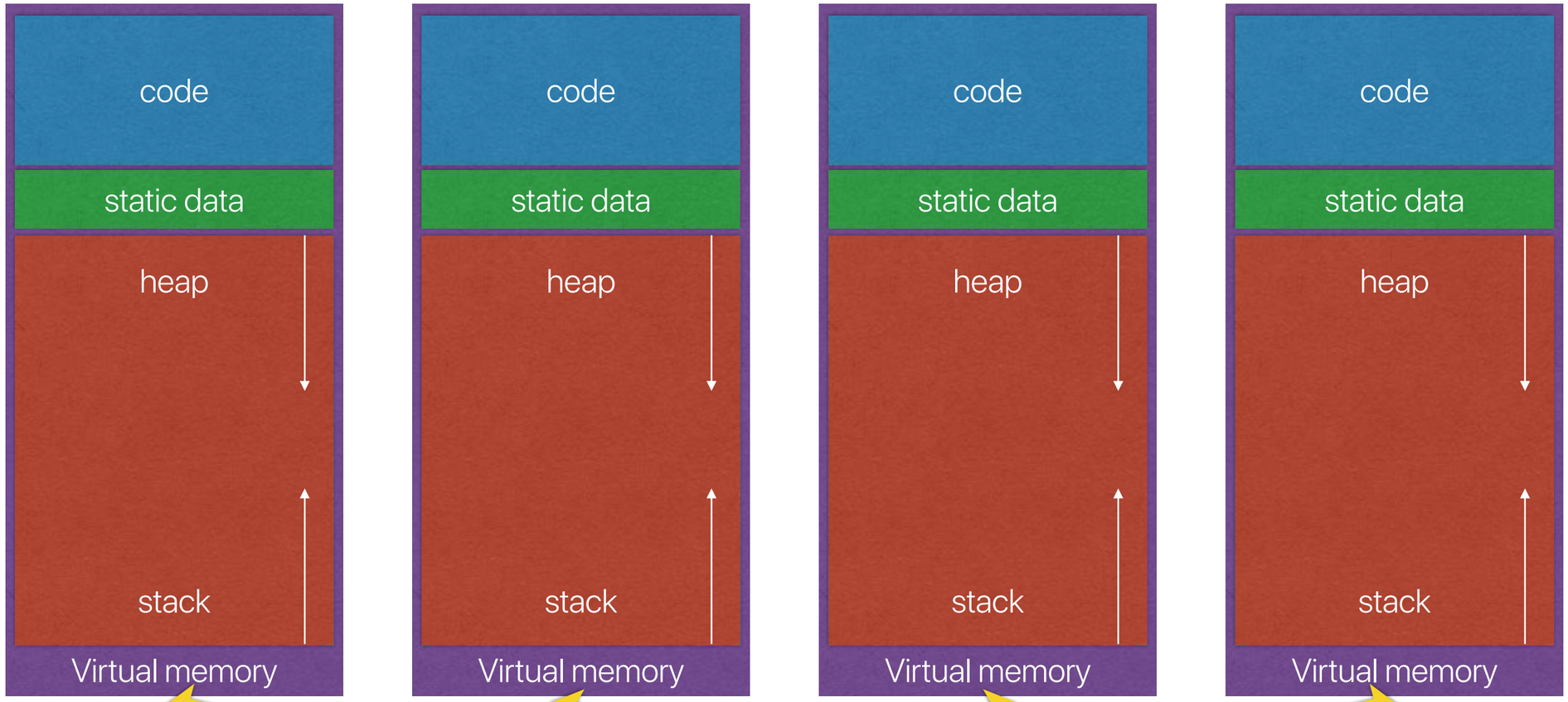
```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    Sem_init(&filled, 0);
    Sem_init(&empty, BUFF_SIZE);
    while(TRUE) {
        int item = …;
        Sem_wait(&W);
        buffer[in]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

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

	W	X	Y	Z
A	empty	empty	filled	filled
B	empty	filled	filled	empty
C	filled	empty	empty	filled

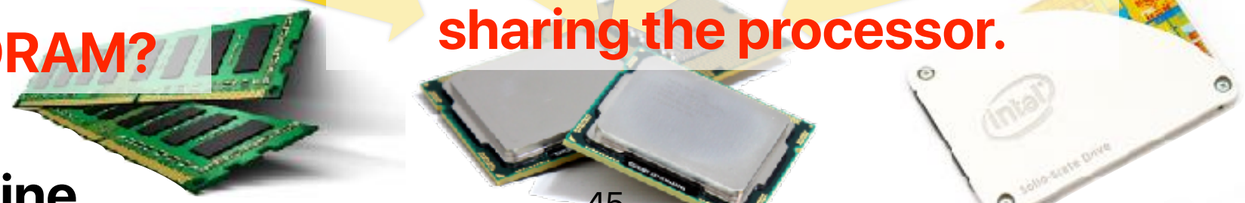
Let’s talk about virtual memory

Previously, we talked about virtualization



Previously, we've talked about sharing the processor.

How about sharing DRAM?



The Machine

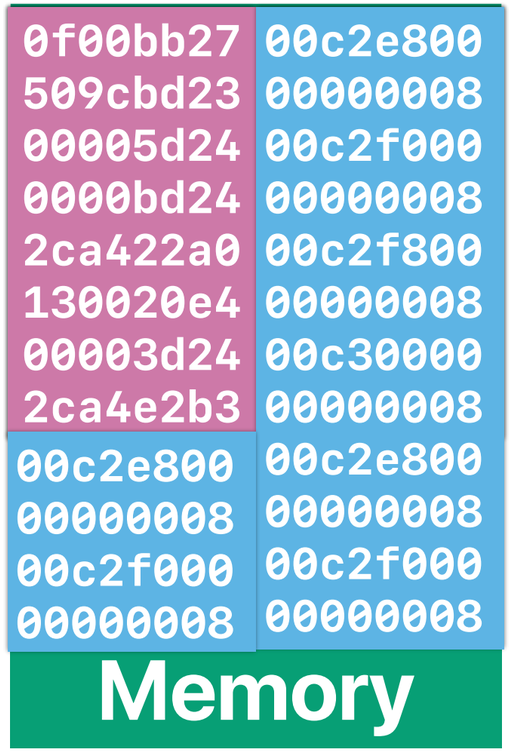
Why Virtual Memory?

If we expose memory directly to the processor (I)

00c2f800
00000008
00c30000
00000008

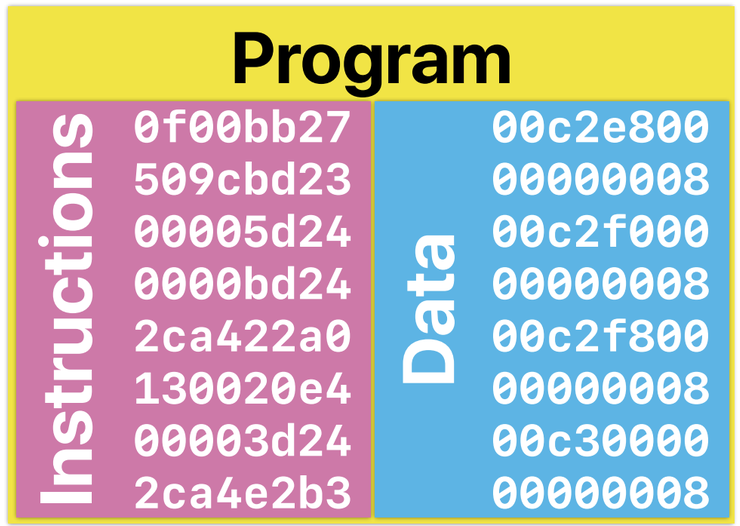
?

What if my program needs more memory?



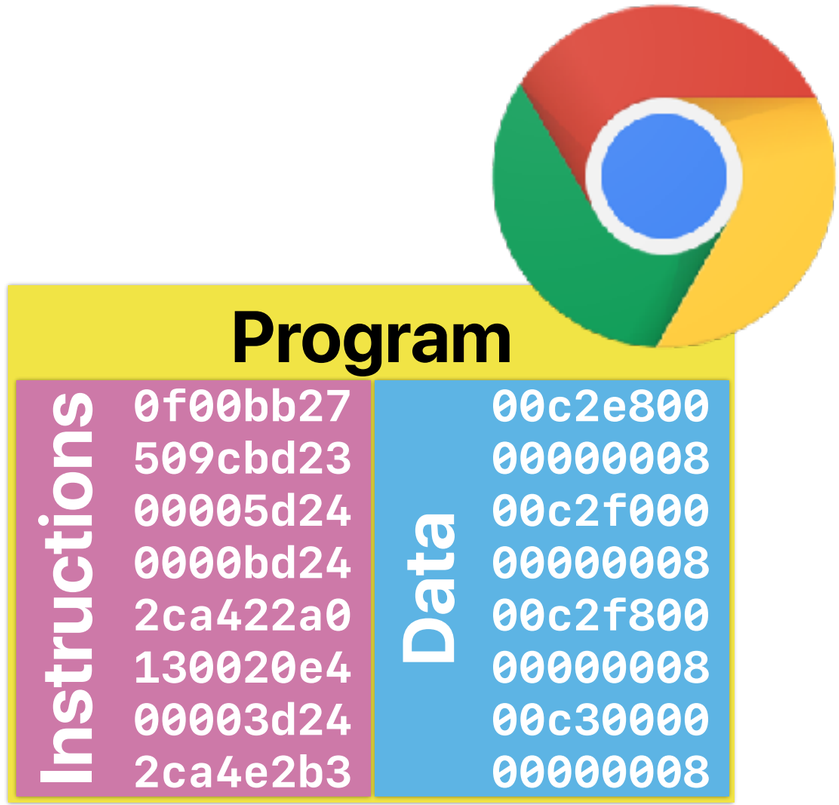
If we expose memory directly to the processor (II)

What if my program runs on a machine with a different memory size?



If we expose memory directly to the processor (III)

What if both programs need to use memory?



?

The Virtual Memory Abstraction

Virtual memory

instruction
0x0

data
0x80000000

data
0x80008000

instruction
0x0

data
0x80008000

Virtual Memory Space

Virtual Memory Space

Program

Instructions

0f00bb27
509cbd23
00005d24
0000bd24
2ca422a0
130020e4
00003d24
2ca4e2b3

Data

00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008

0f00bb27	00c2f800
509cbd23	00000008
00005d24	00c30000
0000bd24	00000008
0f00bb27	00c2f800
509cbd23	00000008
00005d24	00c30000
0000bd24	00000008

Memory

Program

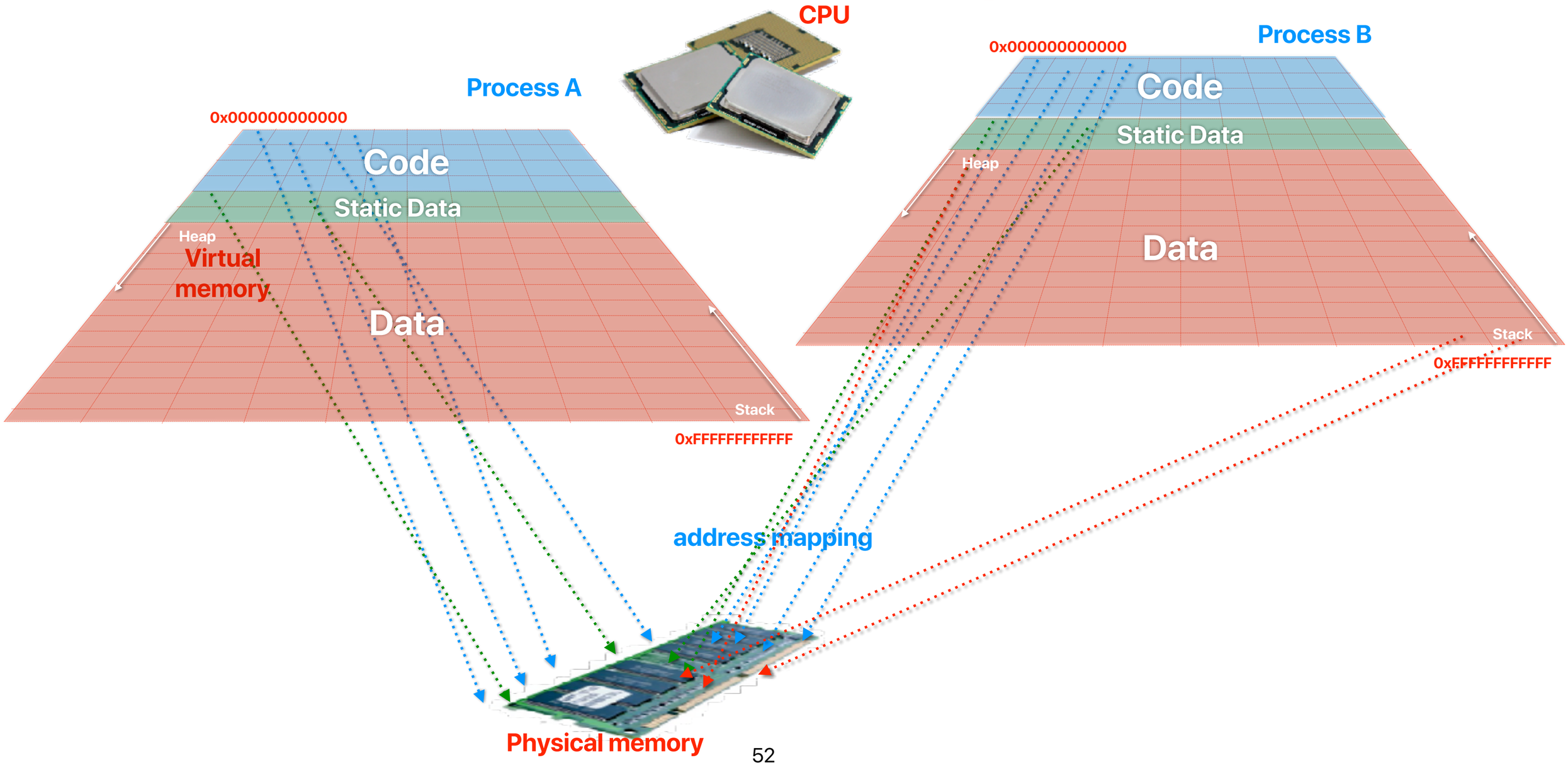
Instructions

0f00bb27
509cbd23
00005d24
0000bd24
2ca422a0
130020e4
00003d24
2ca4e2b3

Data

00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008

Virtual memory



Virtual memory

- An **abstraction** of memory space available for programs/software/programmer
- Programs execute using virtual memory address
- The operating system and hardware work together to handle the mapping between virtual memory addresses and real/physical memory addresses
- Virtual memory organizes memory locations into "**pages**"

Demo revisited: Virtualization

```
double a;

int main(int argc, char *argv[])
{
    int cpu, status, i;
    int *address_from_malloc;
    cpu_set_t my_set;        // Define your cpu_set bit mask.
    CPU_ZERO(&my_set);       // Initialize it all to 0, i.e. no CPUs selected.
    CPU_SET(4, &my_set);     // set the bit that represents core 7.
    sched_setaffinity(0, sizeof(cpu_set_t), &my_set); // Set affinity of this process to the defined mask, i.e. only 7.
    status = syscall(SYS_getcpu, &cpu, NULL, NULL);
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s process_nickname\n",argv[0]);
        exit(1);
    }

    srand((int)time(NULL)+(int)getpid());
    a = rand();

    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and address of a is %p\n",argv[1], cpu, a, &a);
    sleep(1);

    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and address of a is %p\n",argv[1], cpu, a, &a);
    sleep(3);

    return 0;
}
```

getcpu system call to retrieve the executing CPU ID

create a random number

print the value of a and address of a

print the value of a and address of a again after sleep

Demo revisited

```
Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
```

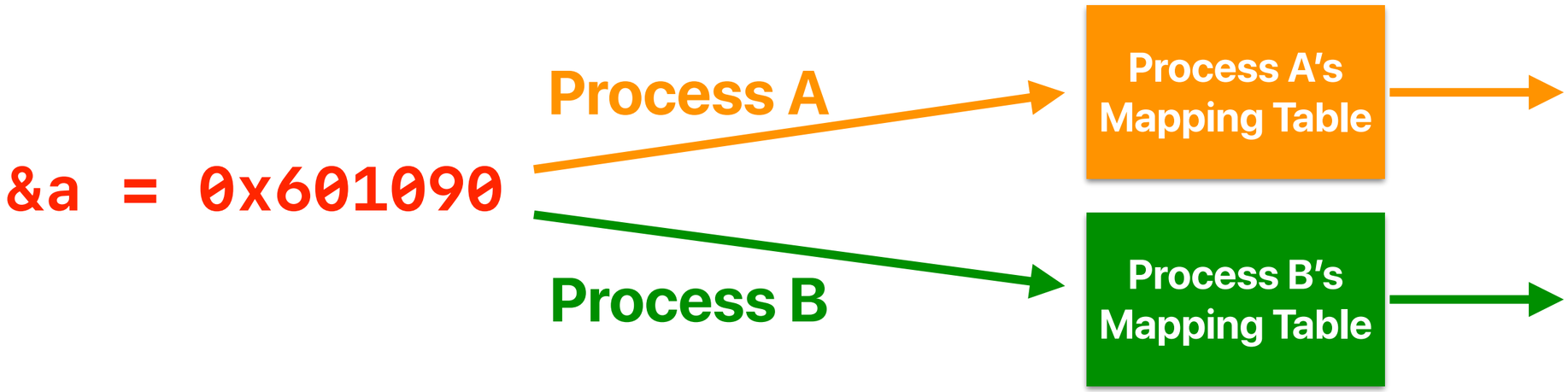
The same processor!

Different values

Different values are preserved

The same memory address!

Demo revisited

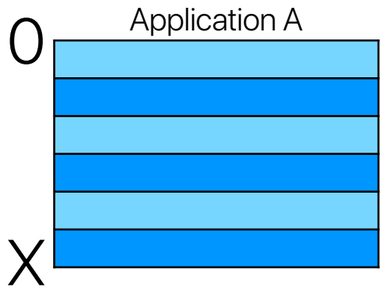


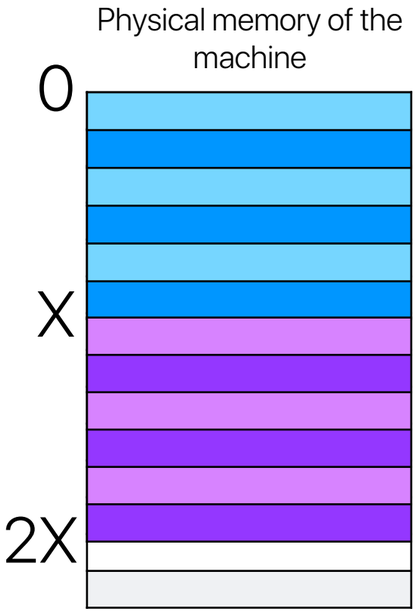

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr, "\nProcess %d is using CPU: %d. Value of a is %lf and address of a is %p\n",getpid(), cpu, a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d is using CPU: %d. Value of a is %lf and address of a is %p\n",getpid(), cpu, a, &a);
    return 0;
}
```

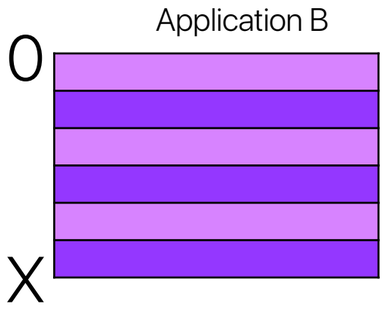
How to map from virtual to physical? Let’s start from segmentation

Segmentation

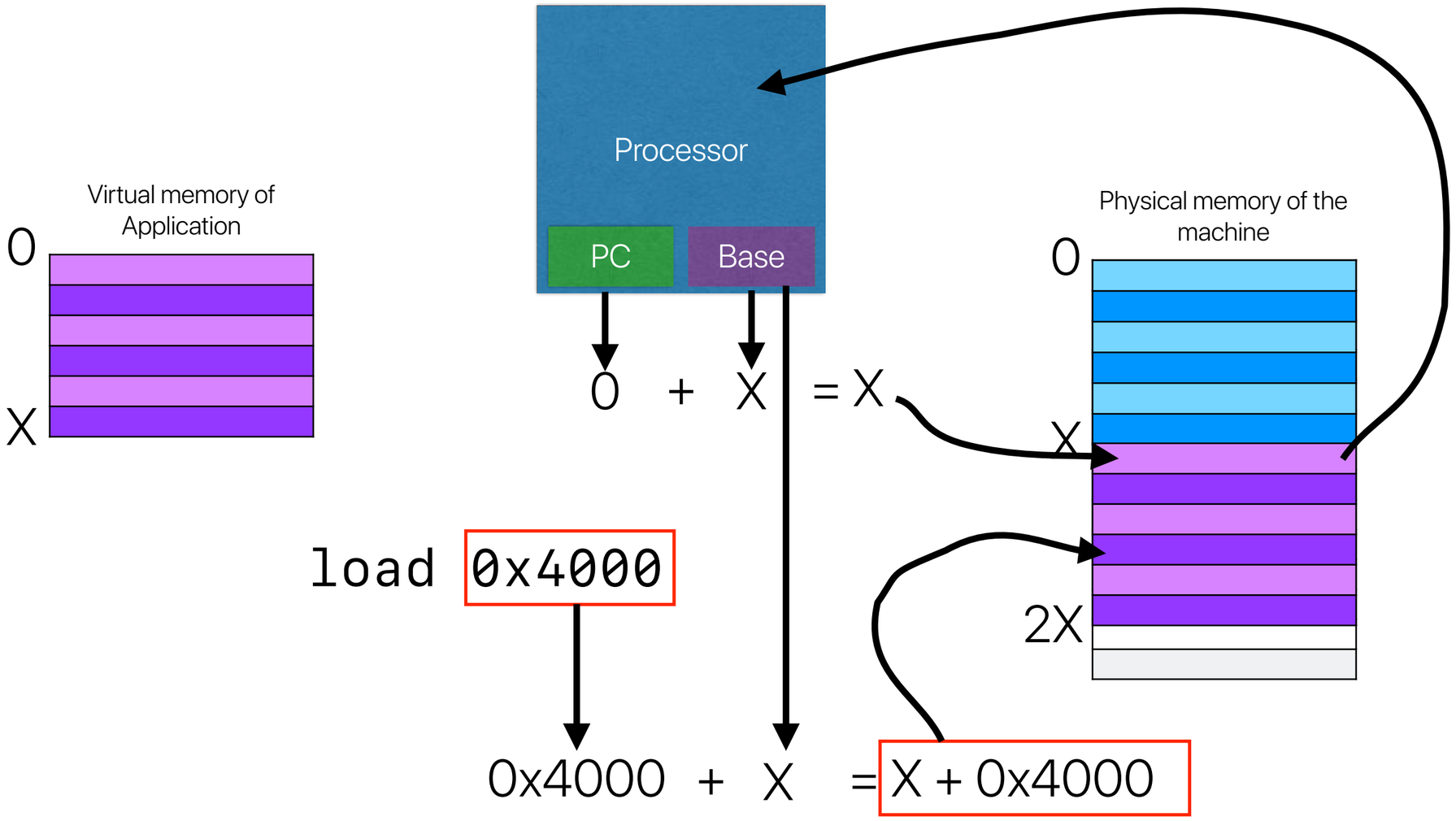
- The compiler generates code using virtual memory addresses
- The OS works together with hardware to partition physical memory space into segments for each running application
- The hardware dynamically translates virtual addresses into physical memory addresses



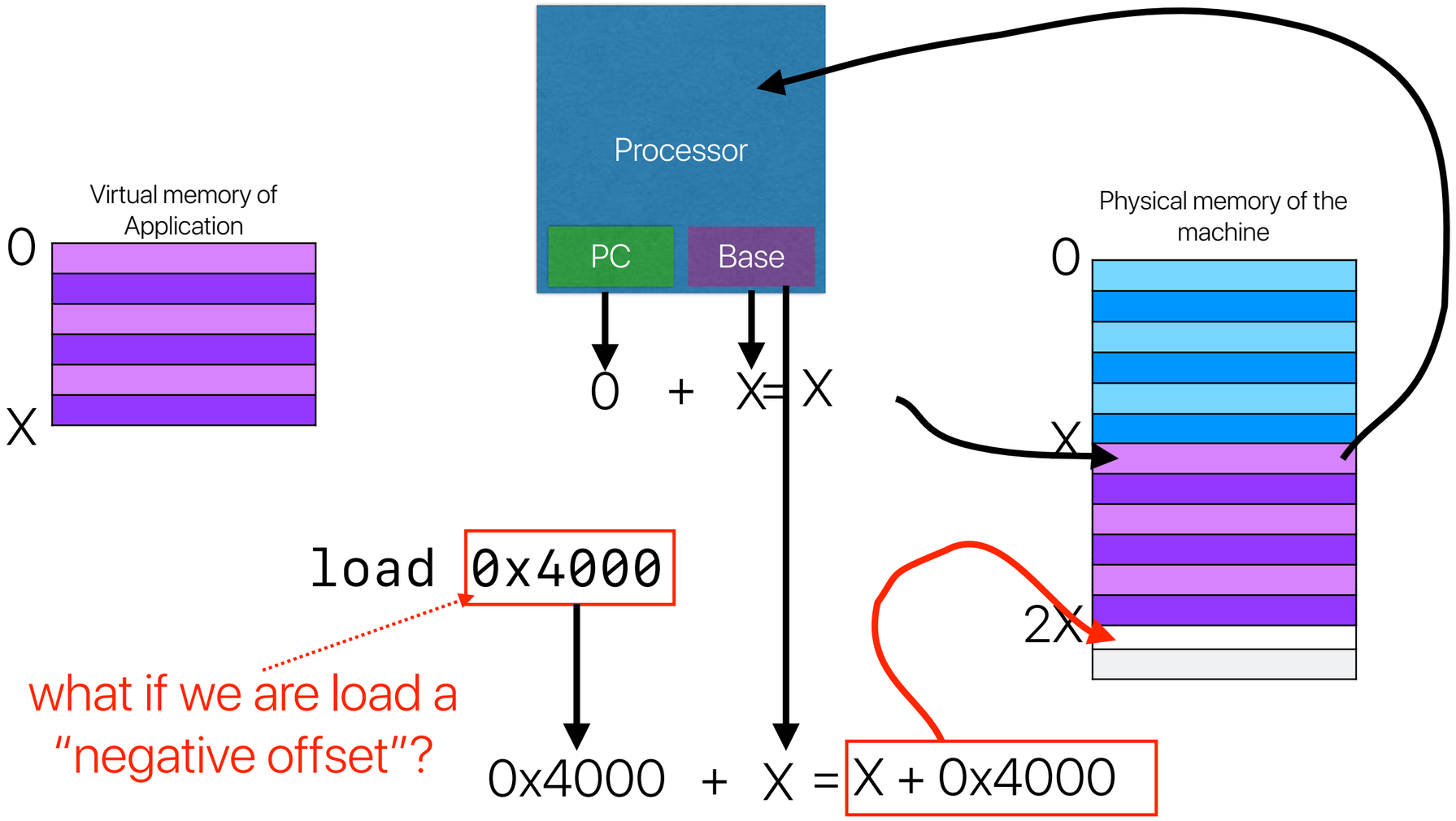




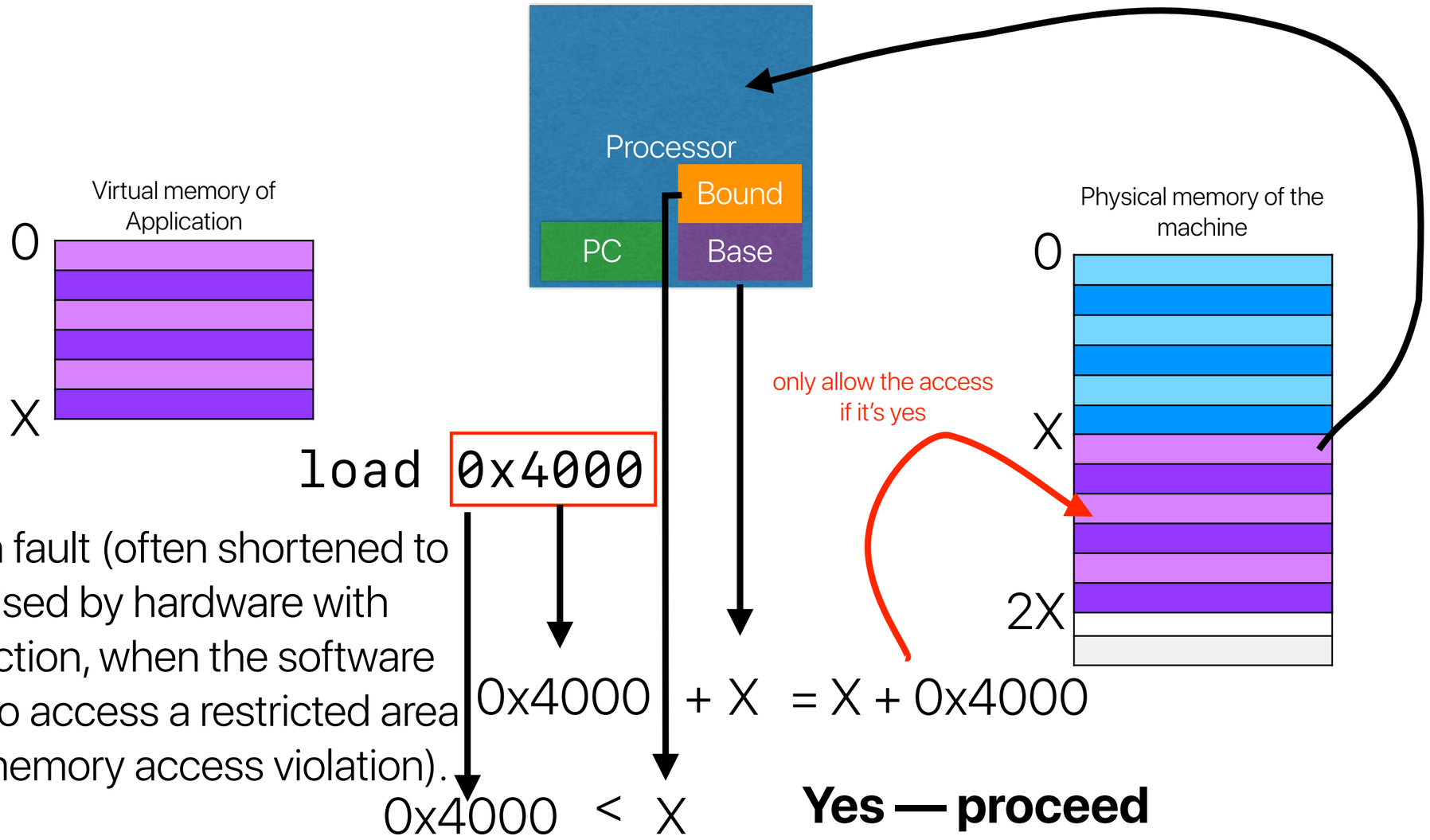
Address translation in segmentation



Protection again malicious processes



Protection again malicious processes



a segmentation fault (often shortened to segfault), raised by hardware with memory protection, when the software has attempted to access a restricted area of memory (a memory access violation).

Announcement

- Reading quizzes due next Tuesday
- New office hour
  - M 3p-4p and Th 9a-10a
  - Use the office hour Zoom link, not the lecture one
- Project released
  - Groups in 2
  - Start as soon as you can — due in about a month
  - **Pull the latest version — had some changes for later kernel versions**
    https://github.com/hungweitseng/CS202-ResourceContainer
  - **Install an Ubuntu Linux 16.04.07 VM as soon as you can!**
  - **Please do not use a real machine — you may not be able to reboot again**
- Midterm
  - Will release on 2/10/2021 0:00am and due on 2/15/2021 11:59:00pm
  - You will have to find a consecutive, non-stop 80-minute slot with this period
  - One time, cannot reinitiate — please make sure you have a stable system and network
  - **No late submission is allowed**

Computer
Science &
Engineering

202

つづく