CSE 451: Operating Systems

Lab Section: Week 5

Today

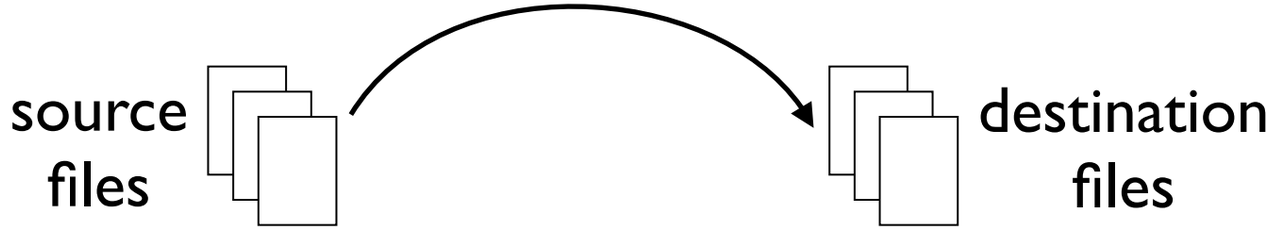
- Project 3
- Virtual Address Spaces
- Deadlock
  (this may be useful for tomorrow's quiz ☺)

Project 3

- Due Wednesday, Feb 16 at 11:59pm
- You can work in pairs
  - use discussion board to find a partner

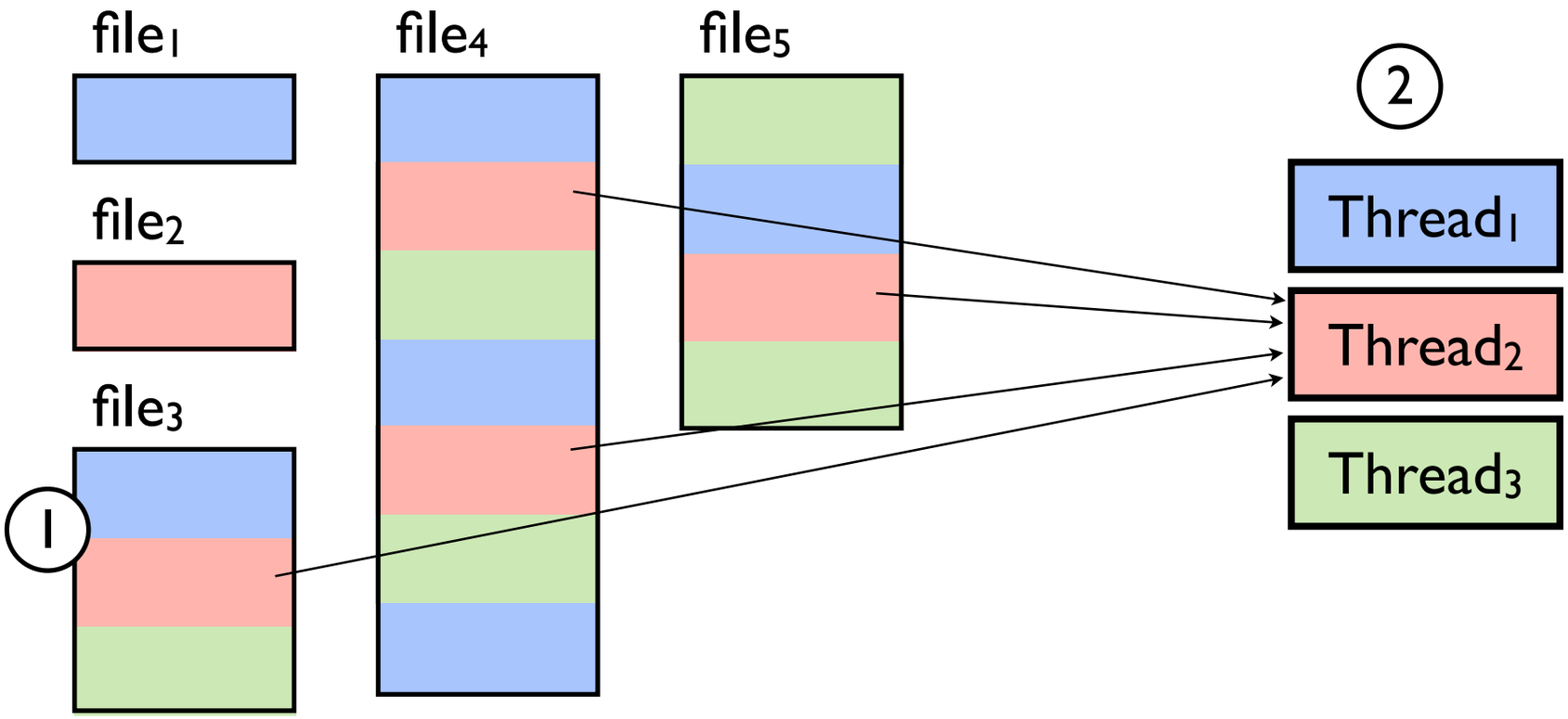
Project 3

- File copy program
  - implement entirely in user-space



- Three parts
  - implement MtFileCopy (multithreaded)
  - implement MtFileCopyAsync (single-threaded)
  - performance analysis

Project 3



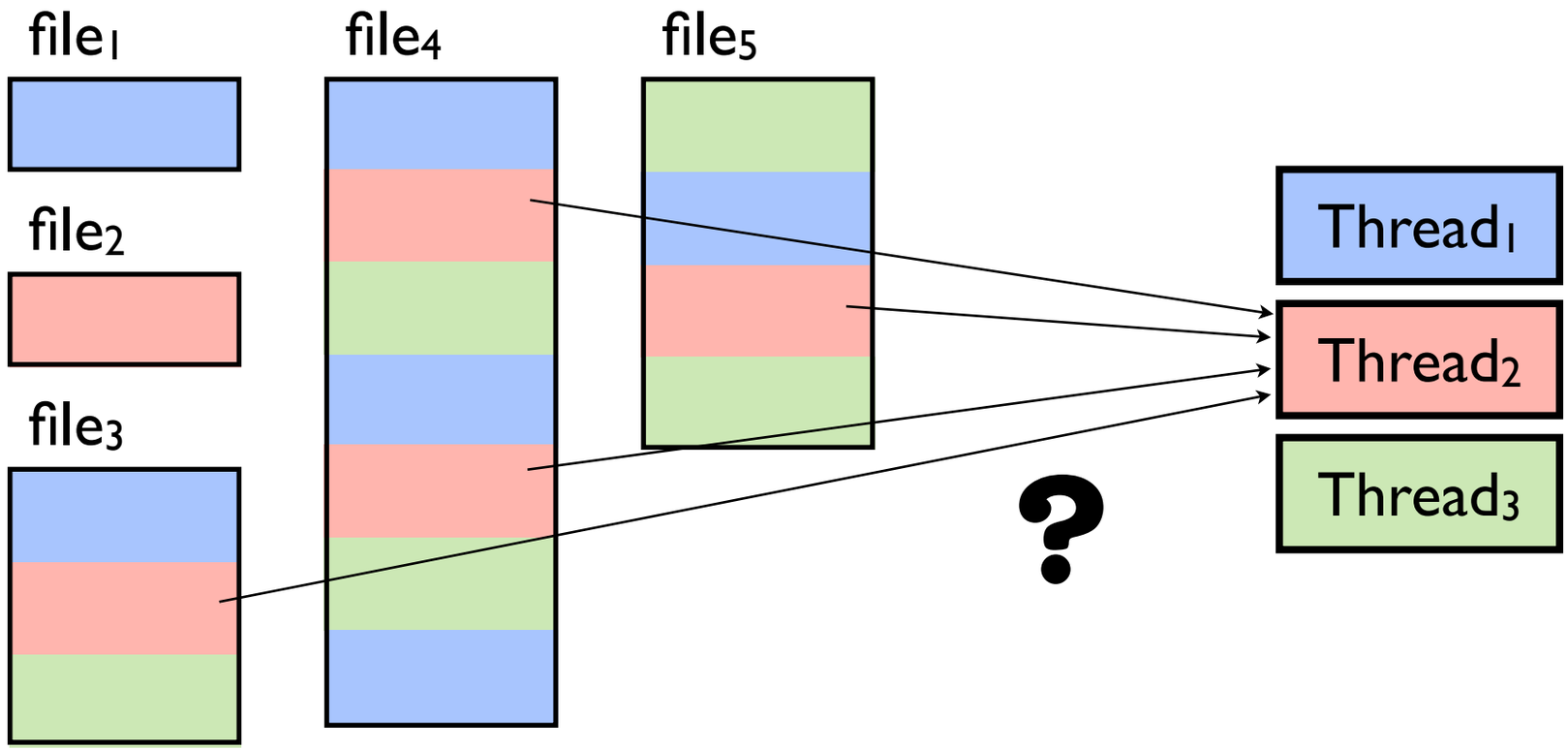

```
MtFileCopy( ThreadCount=3, BufferSize=4096, files .. )
```

① Break files into chunks of work (use chunkSize == BufferSize)

② Schedule chunks to threads (each thread copies one chunk at a time)

Project 3

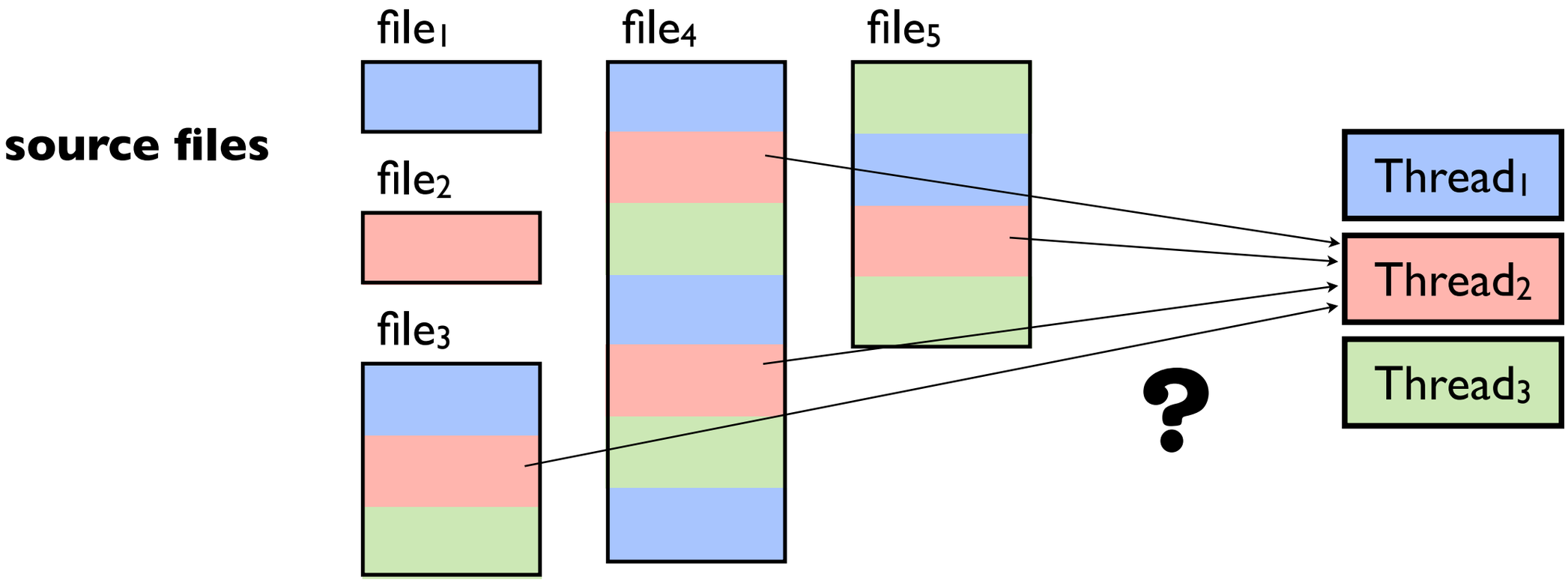
source files



- What goes into an efficient schedule?
  - load balancing (keep threads busy)
  - locality
    - ... assign threads to different files?
    - ... have threads gang up on the same file?

} I don't know which is better! That's why we run experiments

Project 3

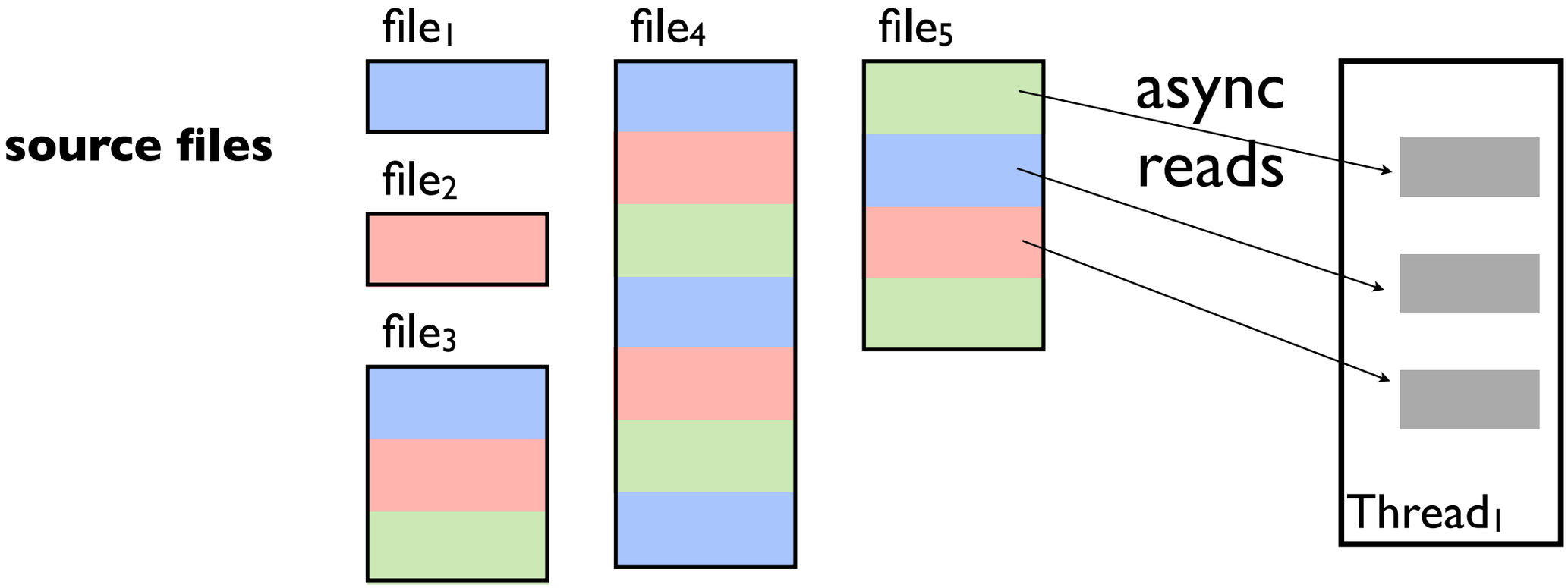


- Scheduling approaches
  - build a schedule up-front (doesn't respond well to performance glitches?)
  - put chunks in a FIFO queue
  - work stealing (a cool idea! ask google for more)

  . . .

Project 3

(async version)




```
MtFileCopyAsync( BufferCount=3, BufferSize=4096, files .. )
```

Same idea! Except ...

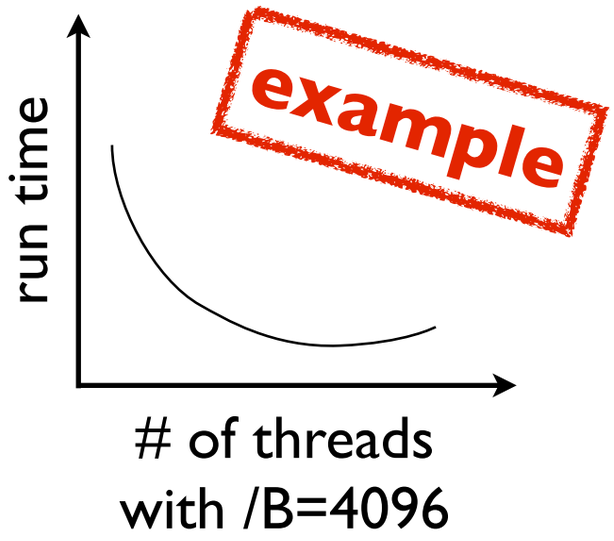
.... we have just **one** thread

.... that thread does 3 asynchronous chunk copies at once

What experiments could I run?

(these are just examples: you can do more!)

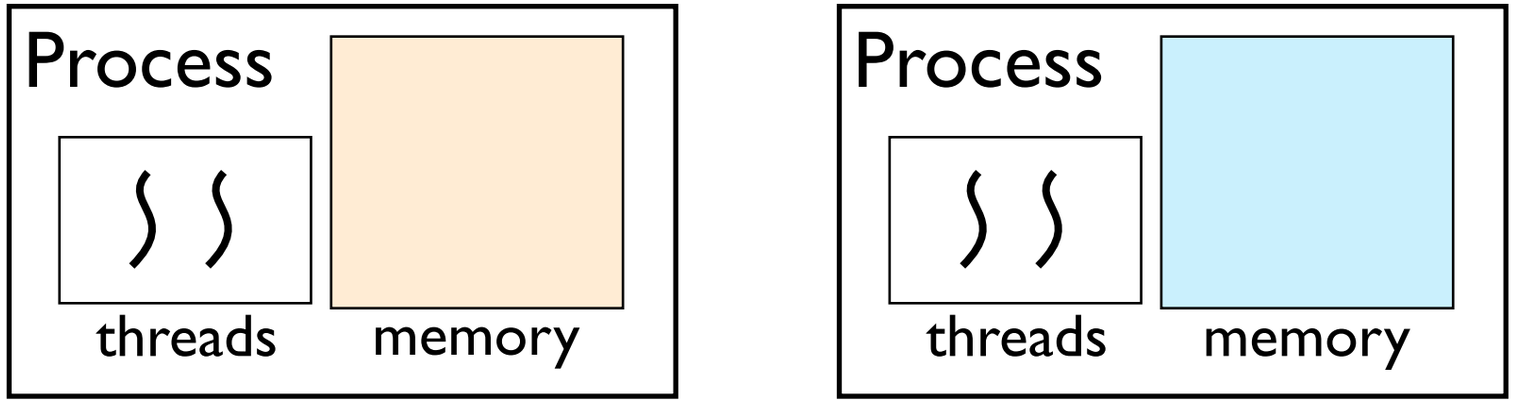
- Select some diverse inputs sets
  - ... big files many files using network drives
    small files few files using local hard drives
    using usb drives
- Time your program on each input set
  - use different values for /T and /B
  - use sync and async
- Analyze:
  - what is the best configuration?
  - what is the worst configuration?
  - make graphs



Today

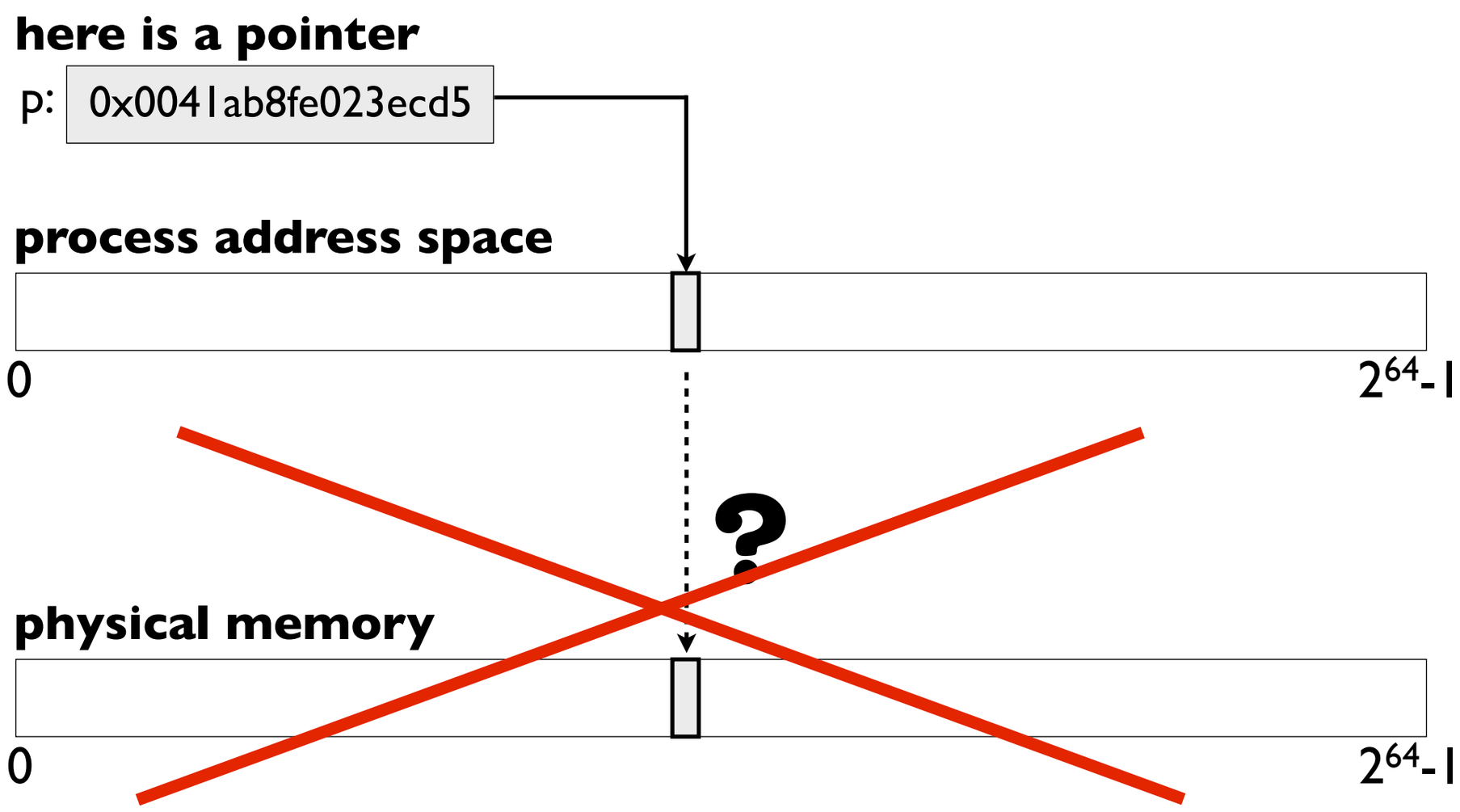
- ~~Project 3~~
- Virtual Address Spaces
- Deadlock
  (this may be useful for tomorrow's quiz ☺)

Virtual Address Spaces

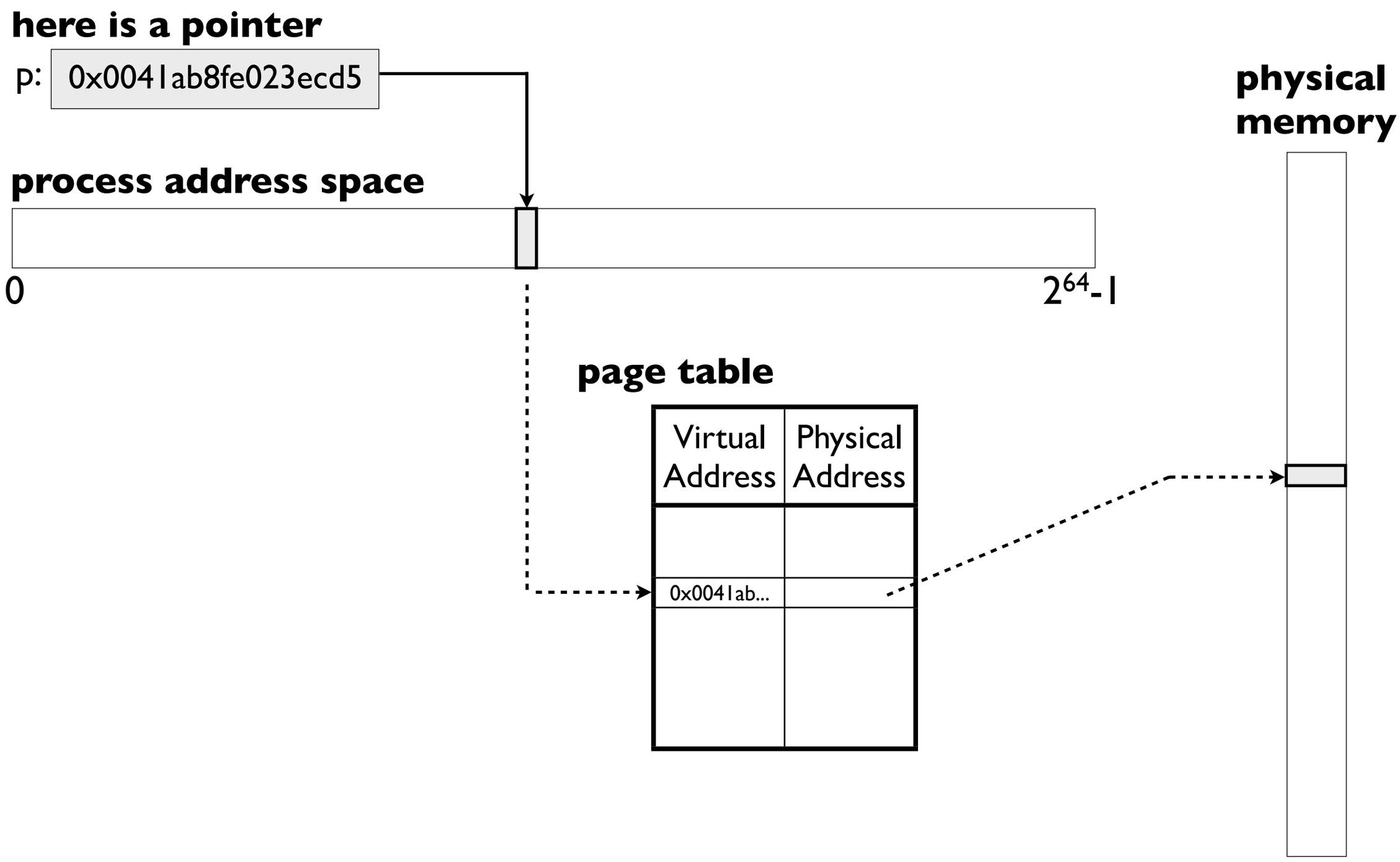


- Wait, if pointers are just numbers ...
  - how does each process get a private memory space?
  - how does the kernel get a private memory space?
  - how does the kernel access process memory?

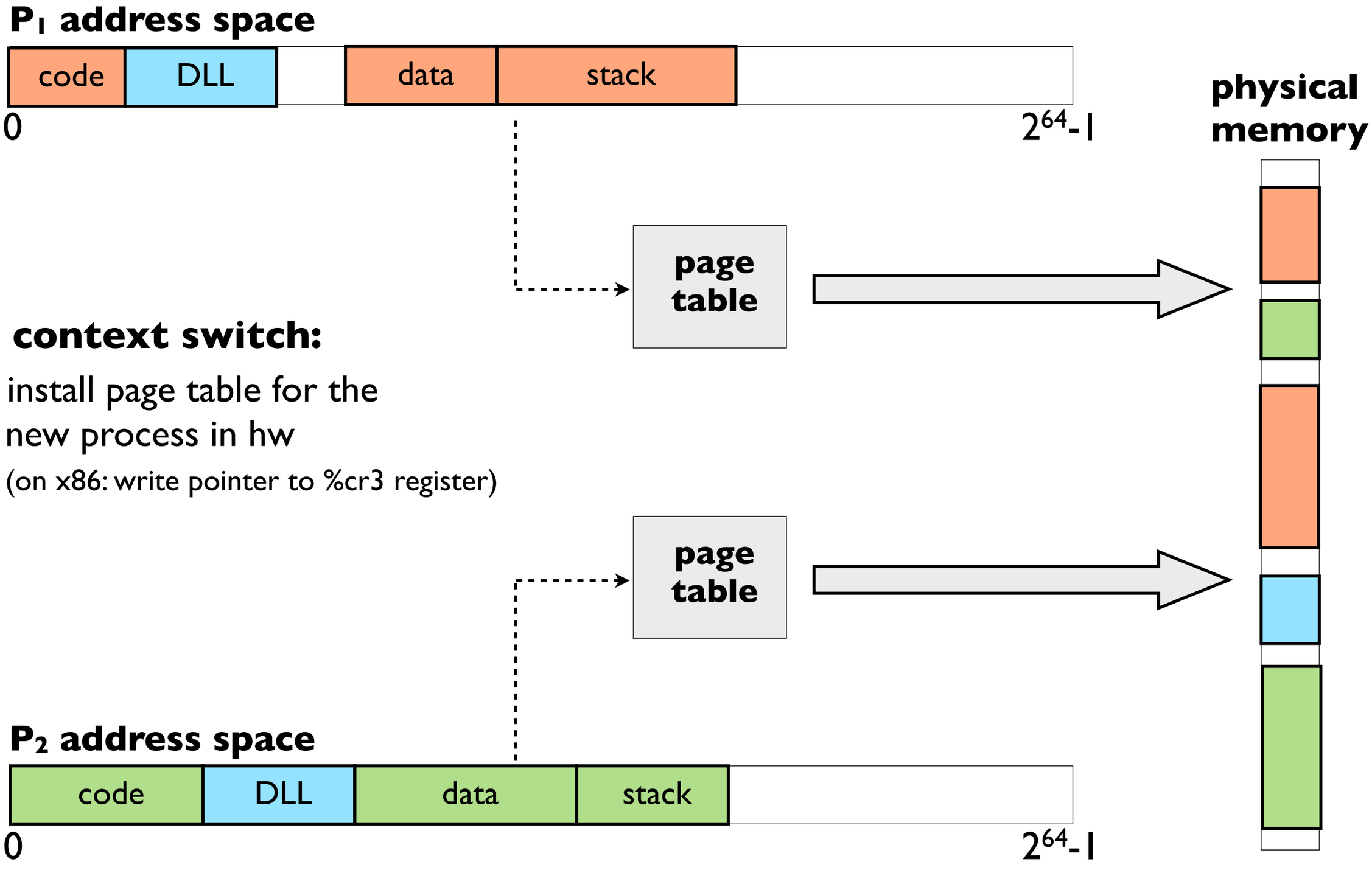
Virtual Address Spaces



Virtual Address Spaces



Virtual Address Spaces



On context switch:

- install page table for the new process in hw

  (on x86: write pointer to %cr3 register)

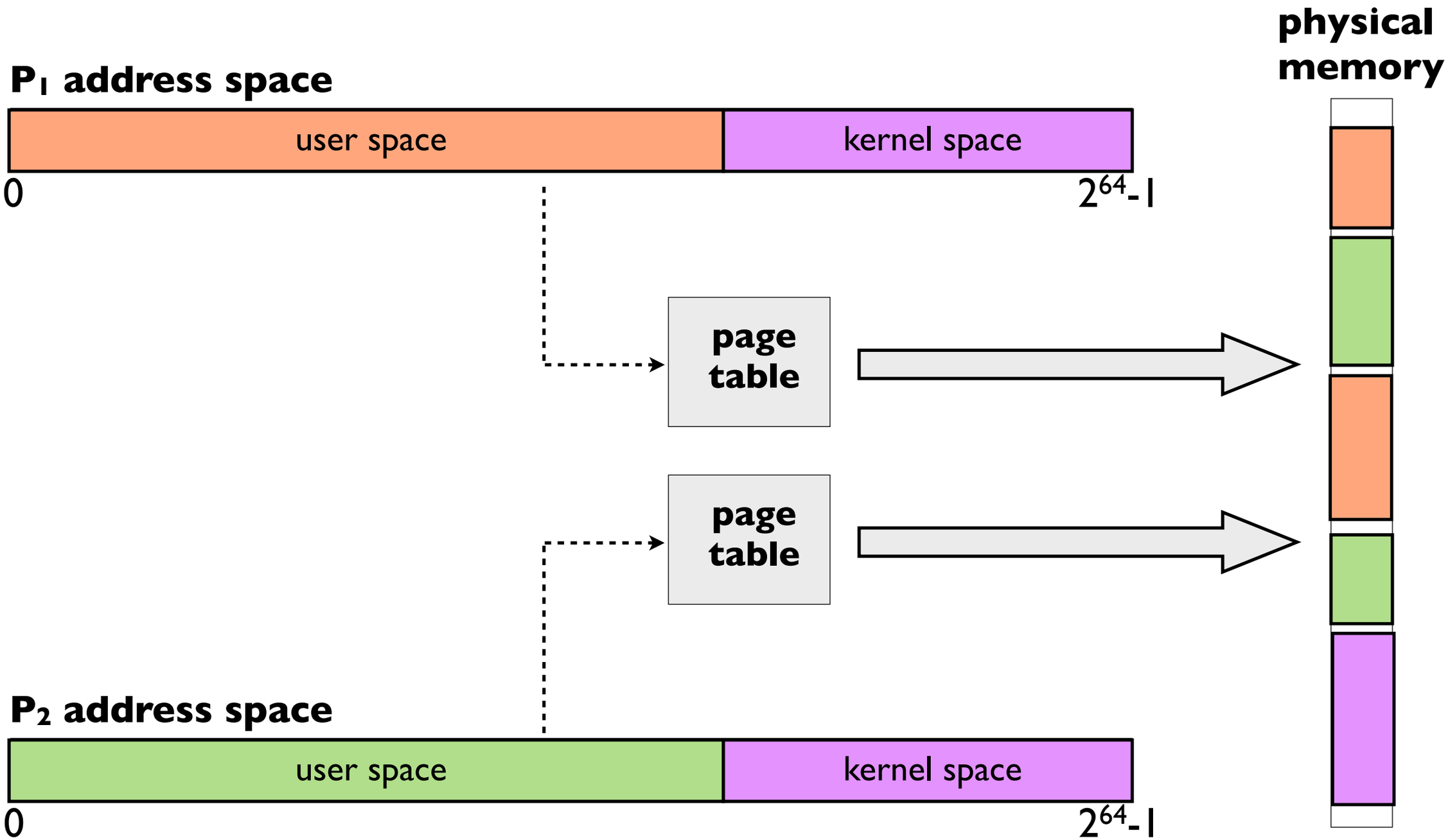
Virtual Address Spaces

- Great, that explains how processes are isolated
- What about the kernel?
  - how does the kernel get a private memory space?
  - how does the kernel access user memory?

```
NtReadFile(void* userbuf) {
    ...
    CSE451.readcalls++;
    ...
```

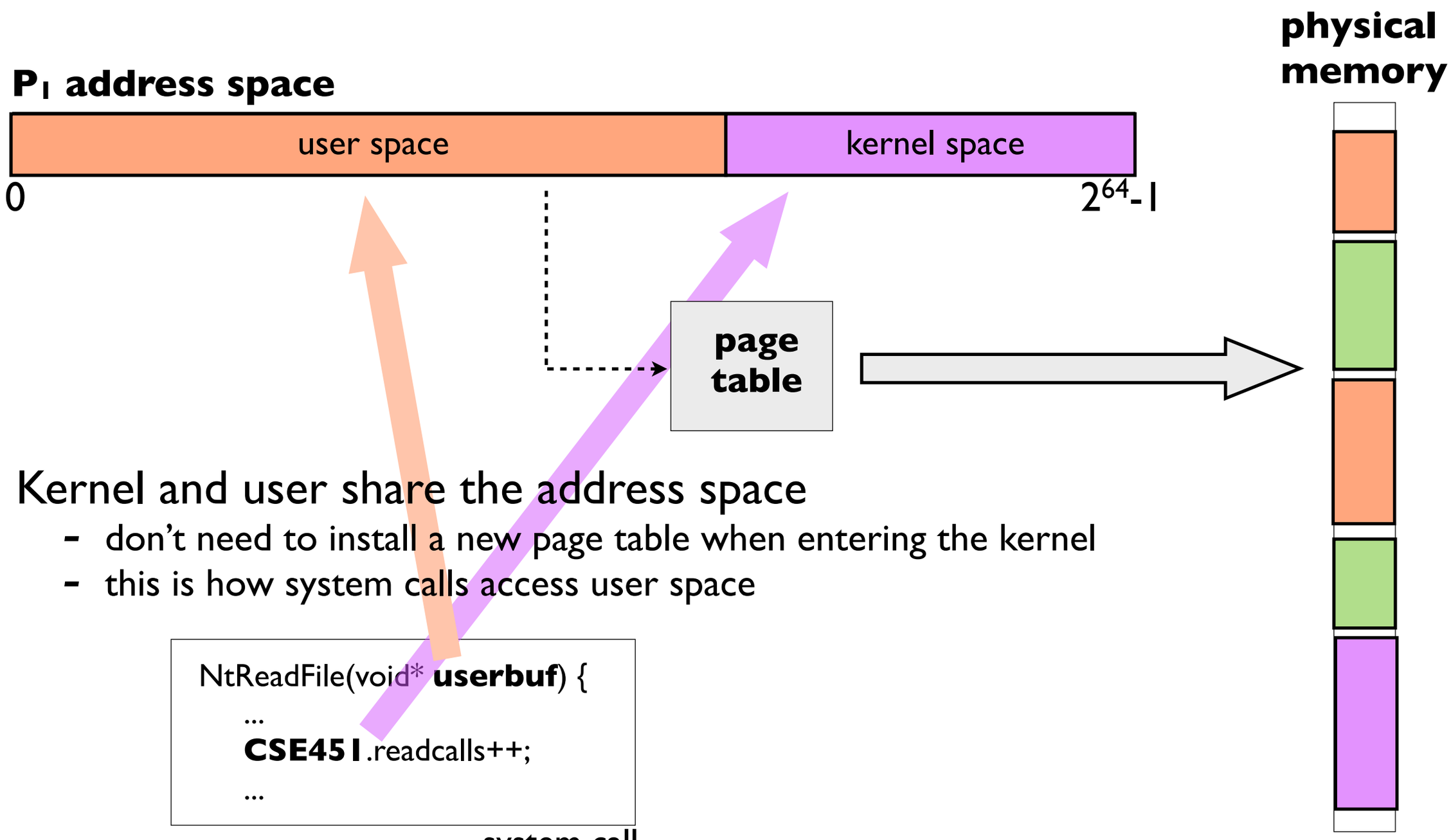
Virtual Address Spaces

(what about the kernel?)



Virtual Address Spaces

(what about the kernel?)



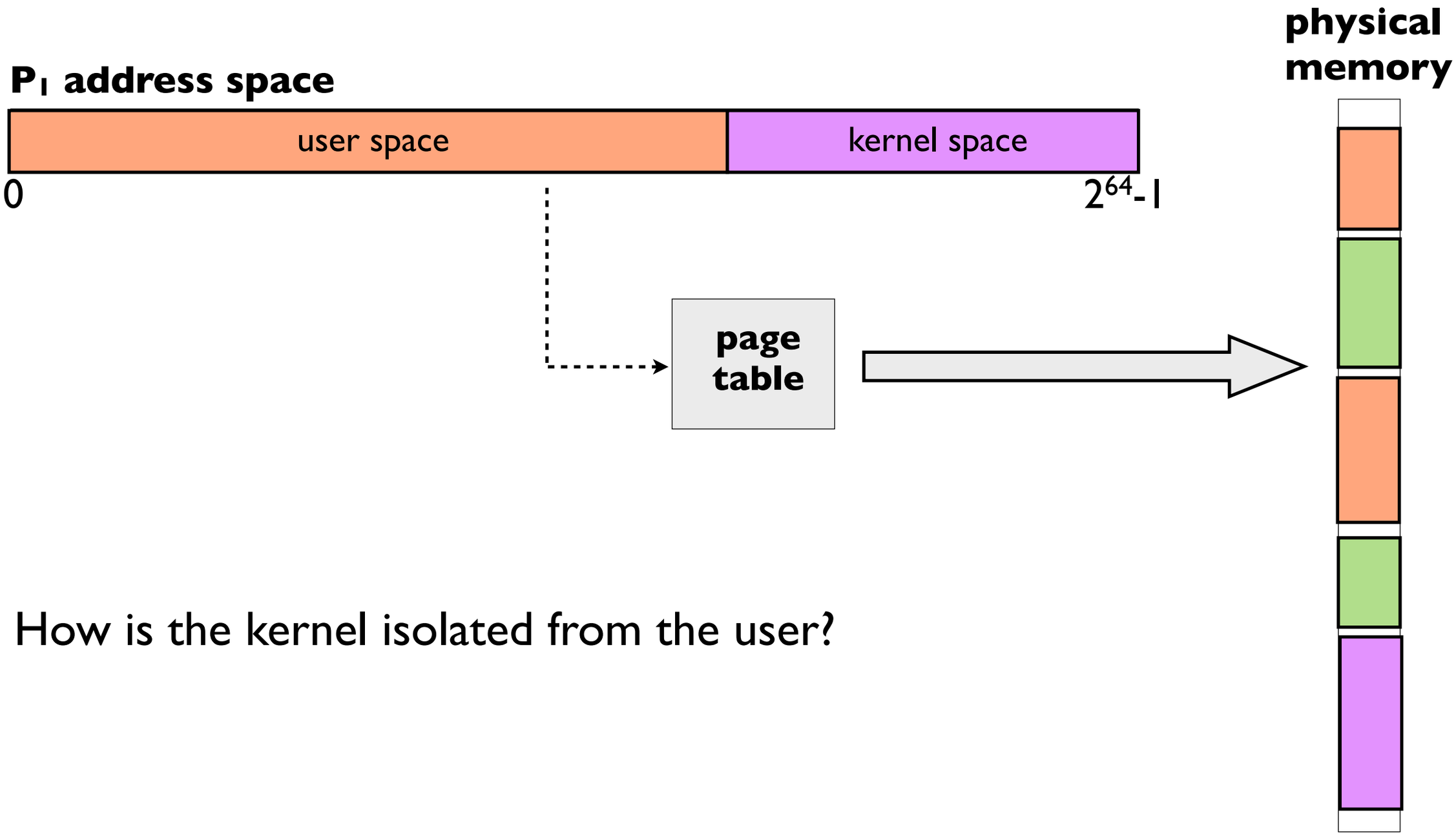
- Kernel and user share the address space
  - don't need to install a new page table when entering the kernel
  - this is how system calls access user space

```
NtReadFile(void* userbuf) {
    ...
    CSE451.readcalls++;
    ...
```

system call

Virtual Address Spaces

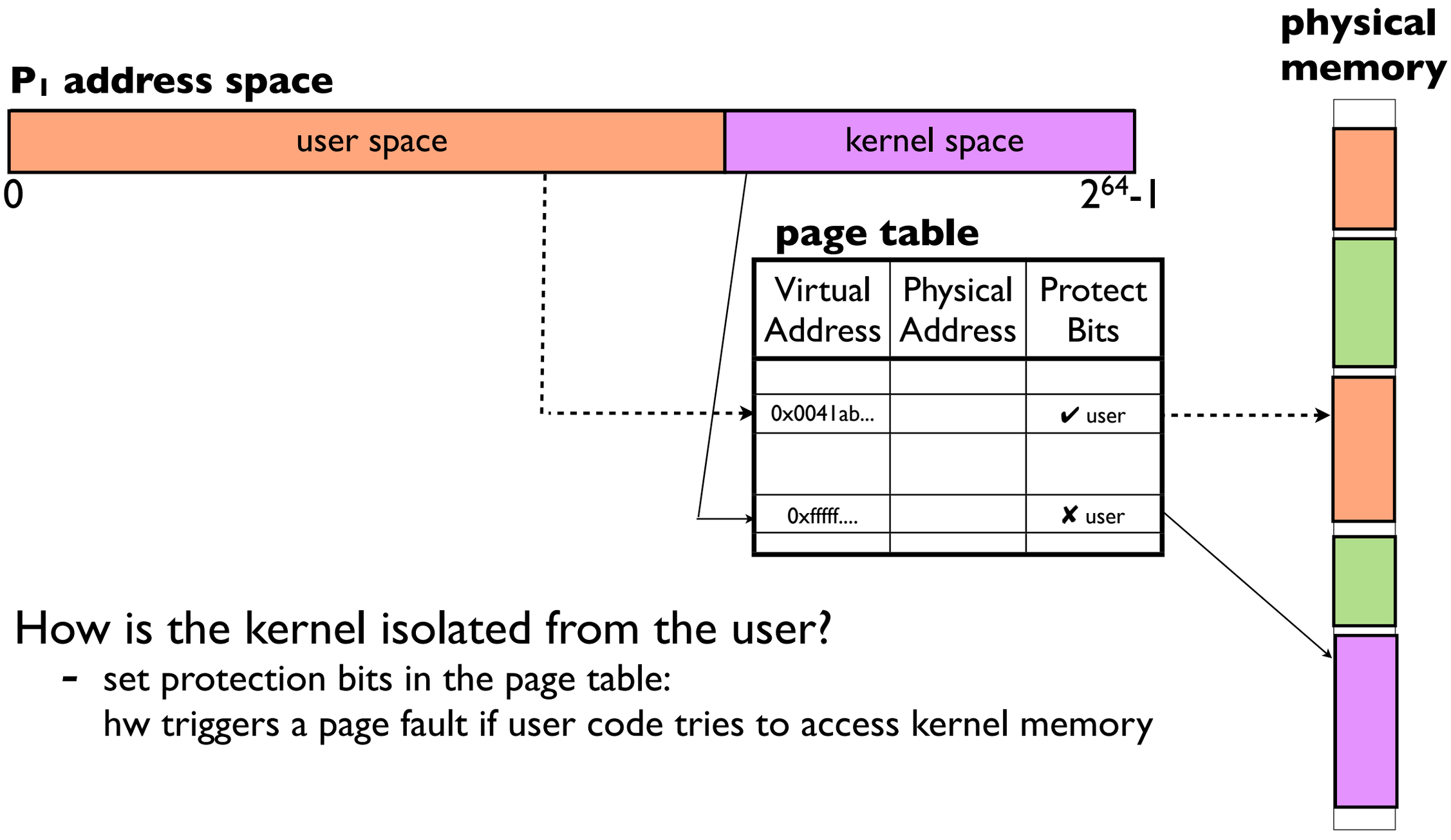
(what about the kernel?)



- How is the kernel isolated from the user?

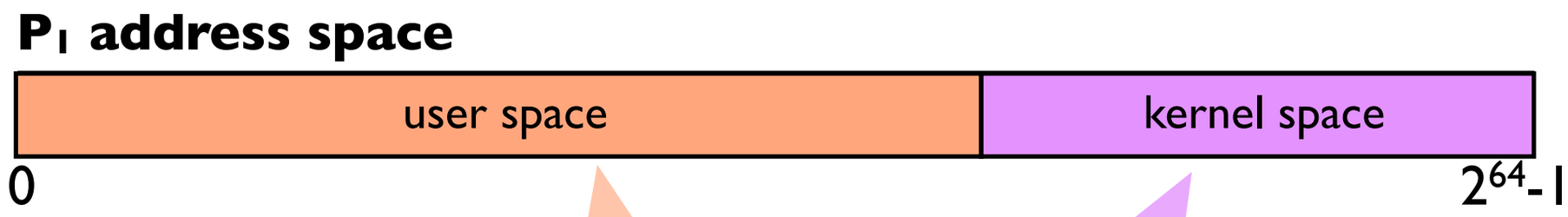
Virtual Address Spaces

(what about the kernel?)



- How is the kernel isolated from the user?
  - set protection bits in the page table:
    hw triggers a page fault if user code tries to access kernel memory

Virtual Address Spaces

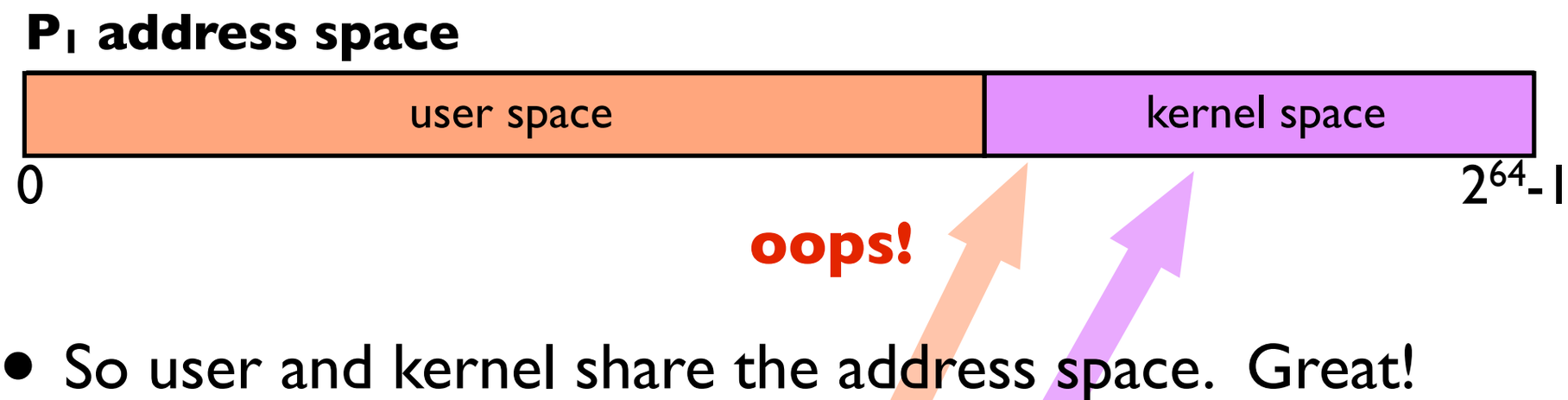


- So user and kernel share the address space. Great! What could possibly go wrong?

```
NtReadFile(void* userbuf,
           int   userlen)
{
    ...
    memcpy( userbuf,
            FileData,
            FileDataSize );
```

- What if **userbuf** is invalid?
  e.g. NULL or
  points at an unmapped page
- The kernel will segfault!

Virtual Address Spaces



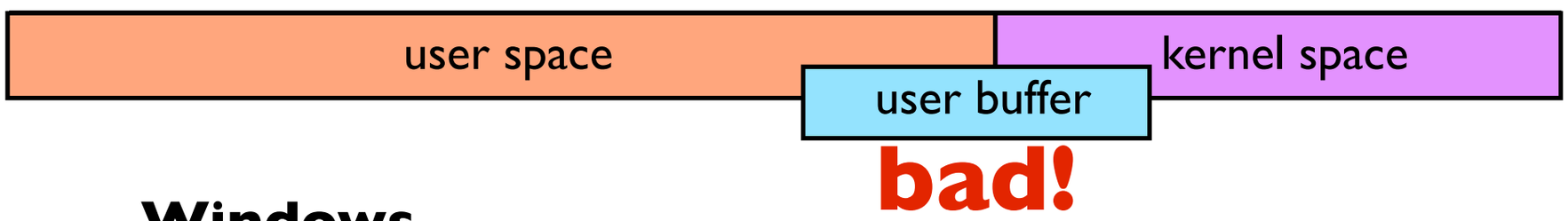
- So user and kernel share the address space. Great! What could possibly go wrong?

```
NtReadFile(void* userbuf,
           int   userlen)
{
    ...
    memcpy( userbuf,
            FileData,
            FileDataSize );
```

- What if **userbuf** points into kernel space?
  - e.g. malicious user code "guesses" a pointer value
- The kernel data structures will be corrupted!

Always always always validate user pointers in the kernel

- Check that user pointers point at **user** memory, not kernel memory



Windows

```
ProbeForWrite( userbuf, length );
```

Linux

```
access_ok( userbuf, length );
```

- Guard kernel code that accesses user pointers against segfaults

Windows

```
try {
   memcpy( userbuf,
           FileData,
           FileDataLen );
} except {
   x = GetExceptionCode();
   ... // oops, handle segfault
}
```

Linux

```
copy_to_user( userbuf,
              FileData,
              FileDataLen );
  // copy_to_user deals with
  // a segfault if it happens
```

http://msdn.microsoft.com/en-us/library/ms809962.aspx#drvrreliab_topic2

Always always always validate user pointers in the kernel

- An example from Project 2:

```
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    // copy event buffer to user space
    memcpy( info->buffer,  CseEventBuffer,  info->bufferSize );
    ....
}
```

Always always always validate user pointers in the kernel

- An example from Project 2.
  Added a fix. Is this enough? What could go wrong?

```
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    ProbeForWrite( info->buffer,  info->bufferSize );
    try {
        memcpy( info->buffer,  CseEventBuffer,  info->bufferSize );
    } except {
        ....
}
```

Always always always validate user pointers in the kernel

- An example from Project 2.
  Added a fix. Is this enough? What could go wrong?

What if another thread changes info->buffer after ProbeForWrite and before memcpy?

oops!

```
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    ProbeForWrite( info->buffer,  info->bufferSize );
    try {
        memcpy( info->buffer,  CseEventBuffer,  info->bufferSize );
    } except {
        ....
}
```

Buggy user code example

Thread$_1$

```
NtQuerySystemInformation(info);
```

Thread$_2$

```
info->buffer = 0xfff....;
```

(a kernel address)

Always always always validate user pointers in the kernel

- An example from Project 2.
  The full fix:

capture

```
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    tmpBuffer = info->buffer;   // capture pointer
    tmpSize   = info->bufferSize;
    ....
    ProbeForWrite( tmpBuffer,  tmpSize );
    try {
        memcpy( tmpBuffer,  CseEventBuffer,  tmpSize );
    } except {
        ....
}
```

Today

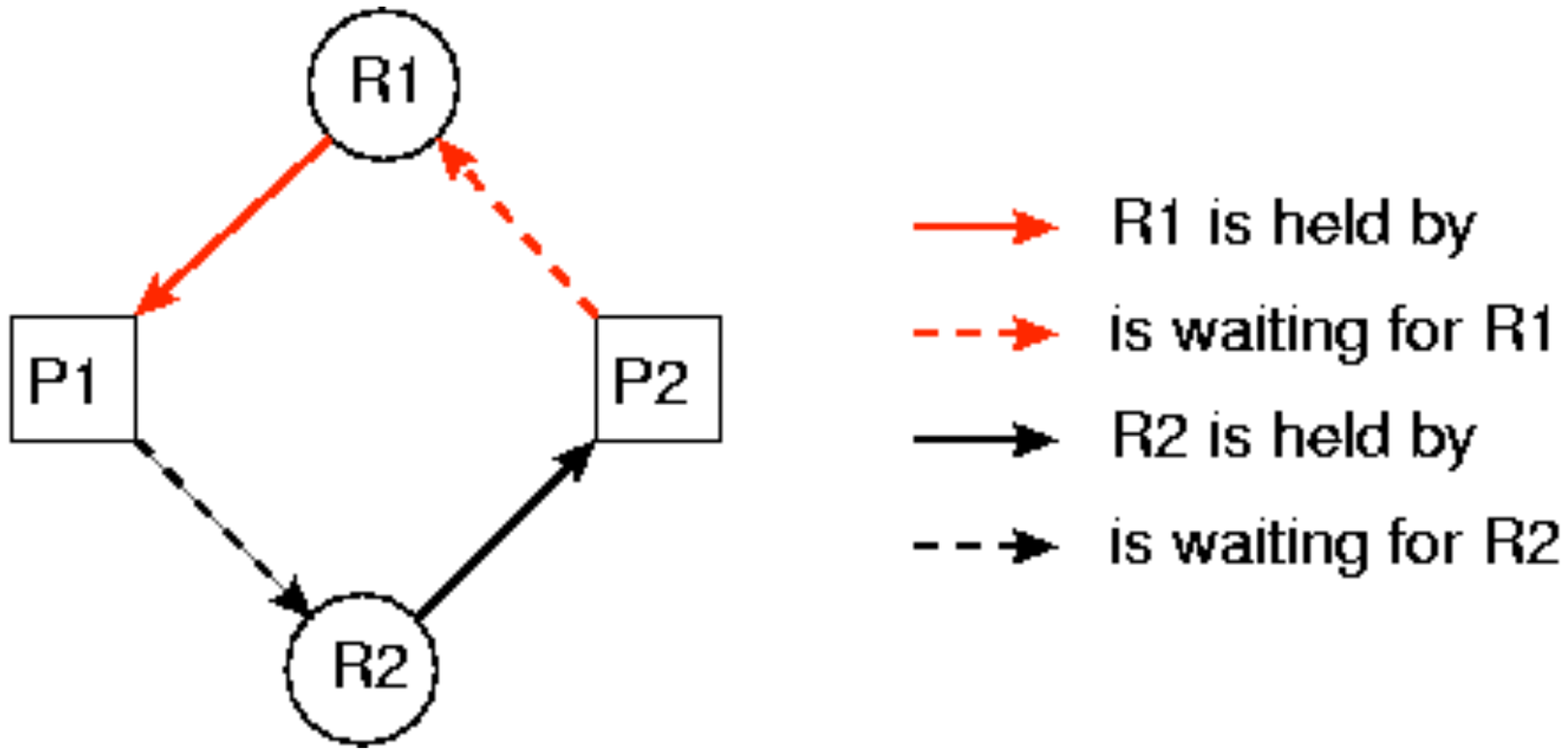
- ~~Project 3~~
- ~~Virtual Address Spaces~~
- Deadlock
  (this may be useful for tomorrow's quiz ☺)

What is deadlock?

Deadlock is an *irreducible* circular dependence.

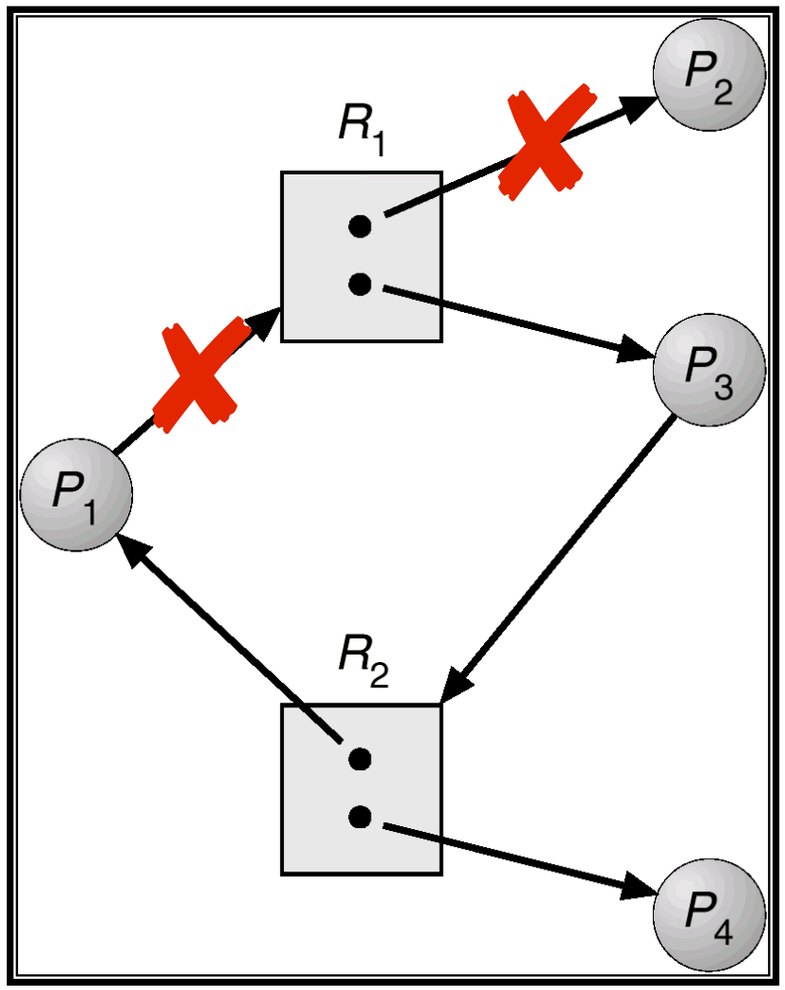
That's it.

Spot the deadlock!



stolen from lecture slides

Spot the deadlock!

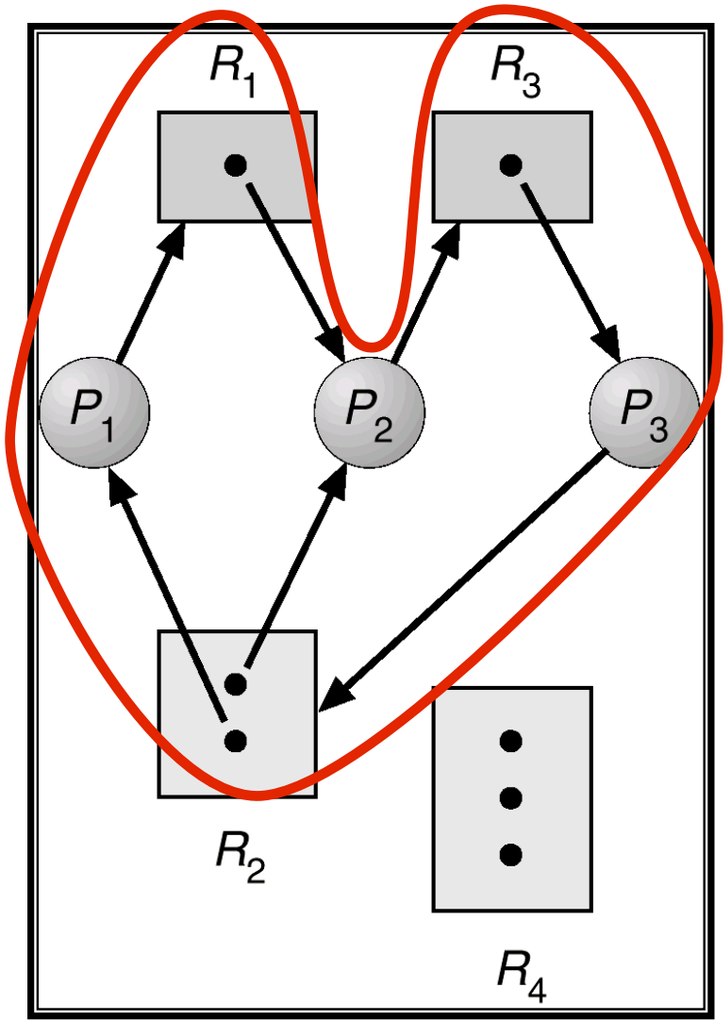


circle = process
square = resource
arrow = dependence

No deadlock!

P2 can release R1
... allowing P1 to acquire R1
... breaking the cycle

Spot the deadlock!



circle = process
square = resource
arrow = dependence

stolen from lecture slides

Spot the deadlock!

Thread$_1$

```
foo() {
    lock( &A );
    lock( &B );
    ....
```

Thread$_2$

```
bar() {
    lock( &B );
    lock( &A );
    ....
```

Spot the deadlock!

Thread$_1$

```
foo() {
    lock( &A );
    ....
    while (true) {
        x = ReadFromPipe();
        if (x == 42)
            break;
    }
    unlock( &A );
    ....
```

Thread$_2$

```
bar() {
    lock( &A );
    ....
    WriteToPipe(42);
    ....
    unlock( &A );
    ....
```

Spot the deadlock!

```
TransferMoney(account A,  account B,  int amount)
{
    lock( &A.lock );
    lock( &B.lock );
    //
    // transfer the money
    //
    unlock( &B.lock );
    unlock( &A.lock );
}
```

$Thread_1$	$Thread_2$
TransferMoney(A, B, x);	TransferMoney(B, A, x);