Today’s agenda

• Solving midterm from winter 2018
Question 1.a: Basic page tables

Problem description:

cr3 = 0x0

PD at address 0x0:

0 -> 0x1
1 -> 0x2
2 -> 0x1

PT at address 0x1000:

0 -> 0x3
1 -> 0x4

PT at address 0x2000:

0 -> 0x5
1 -> 0x4

Question: what is the mapping look like?

1. Basic page tables.
   Consider the following 32-bit x86 page table setup.
   
   %cr3 holds 0x00000000.

   The Page Directory Page at physical address 0x00000000:

   PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
   PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
   PDE 2: PPN=0x00001, PTE_P, PTE_U, PTE_W
   ... all other PDEs are zero

   The Page Table Page at physical address 0x00001000 (which is PPN 0x00001):

   PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
   PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
   ... all other PTEs are zero

   The Page Table Page at physical address 0x00002000:

   PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
   PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
   ... all other PTEs are zero
Question 1.a: Basic page tables

Problem description:

Problem description:

CR3 = 0x0

PD at address 0x0:

0 -> 0x1
1 -> 0x2
2 -> 0x1

PT at address 0x1000:

0 -> 0x3
1 -> 0x4

PT at address 0x2000:

0 -> 0x5
1 -> 0x4

Question: what is the mapping look like?

Remind virtual to physical address mapping:
Question 1.a: Basic page tables

Remind virtual to physical address mapping:

Problem description:

cr3 = 0x0

PD at address 0x0:

0 → 0x1
1 → 0x2
2 → 0x1

PT at address 0x1000:

0 → 0x3
1 → 0x4

PT at address 0x2000:

0 → 0x5
1 → 0x4

Question: what is the mapping look like?

Bits 31-22 can be either 0x0, 0x1, 0x2
Question 1.a: Basic page tables

Problem description:
cr3 = 0x0
PD at address 0x0:
0 -> 0x1
1 -> 0x2
2 -> 0x1

PT at address 0x1000:
0 -> 0x3
1 -> 0x4

PT at address 0x2000:
0 -> 0x5
1 -> 0x4

Question: what is the mapping look like?

Remind virtual to physical address mapping:

If bits 31-22 are 0x0:

Look at the page table (PT) at address 0x1000
Problem description:
cr3 = 0x0
PD at address 0x0:
0 -> 0x1
1 -> 0x2
2 -> 0x1
PT at address 0x1000:
0 -> 0x3
1 -> 0x4
PT at address 0x2000:
0 -> 0x5
1 -> 0x4

Question: what is the mapping look like?

Remind virtual to physical address mapping:
If bits 31-22 are 0x0:
Look at the page table (PT) at address 0x1000

Question 1.a: Basic page tables
Problem description:

cr3 = 0x0

PD at address 0x0:
0 -> 0x1
1 -> 0x2
2 -> 0x1

PT at address 0x1000:
0 -> 0x3
1 -> 0x4

PT at address 0x2000:
0 -> 0x5
1 -> 0x4

Question: what is the mapping look like?

Remind virtual to physical address mapping:

Page table (PT) at address 0x1000 has 2 entries 0x0 and 0x1 (all other zeros) =>

bits 21-12 can be either 0x0 or 0x1
Question 1.a: Basic page tables

Problem description:

\[ \text{cr3} = 0x0 \]

PD at address 0x0:

\[
\begin{align*}
0 & \rightarrow 0x1 \\
1 & \rightarrow 0x2 \\
2 & \rightarrow 0x1
\end{align*}
\]

PT at address 0x1000:

\[
\begin{align*}
0 & \rightarrow 0x3 \\
1 & \rightarrow 0x4
\end{align*}
\]

PT at address 0x2000:

\[
\begin{align*}
0 & \rightarrow 0x5 \\
1 & \rightarrow 0x4
\end{align*}
\]

Question: what is the mapping look like?

Remind virtual to physical address mapping:

Let's sum up the first PD entry range:

It maps addresses from 0x0 to 0x1FFF

\[ 0x0 = 0b \]
\[ 0x1FFF = 0b \]
Question 1.a: Basic page tables

Problem description:

- cr3 = 0x0

PD at address 0x0:

- 0 -> 0x1
- 1 -> 0x2
- 2 -> 0x1

PT at address 0x1000:

- 0 -> 0x3
- 1 -> 0x4

PT at address 0x2000:

- 0 -> 0x5
- 1 -> 0x4

Question: what is the mapping look like?

Remind virtual to physical address mapping:

Second PD entry range:

It maps addresses from 0x400000 to 0x401FFF

- 0x400000 = 0b
- 0x401FFF = 0b
Question 1.a: Basic page tables

Problem description:
cr3 = 0x0
PD at address 0x0:
0 -> 0x1
1 -> 0x2
2 -> 0x1

PT at address 0x1000:
0 -> 0x3
1 -> 0x4

PT at address 0x2000:
0 -> 0x5
1 -> 0x4

Question: what is the mapping look like?

Remind virtual to physical address mapping:

Third PD entry range:
It maps addresses from 0x800000 to 0x801FFF

0x800000 = 0b10000000000000000000000000000000
0x801FFF = 0b10000000000000000000111111111111
Question 1.a: Basic page tables

Problem description:

\( cr3 = 0x0 \)

PD at address 0x0:

\( 0 \rightarrow 0x1 \)
\( 1 \rightarrow 0x2 \)
\( 2 \rightarrow 0x1 \)

PT at address 0x1000:

\( 0 \rightarrow 0x3 \)
\( 1 \rightarrow 0x4 \)

PT at address 0x2000:

\( 0 \rightarrow 0x5 \)
\( 1 \rightarrow 0x4 \)

Question: which virtual addresses are mapped

Remind virtual to physical address mapping:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Table</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

Final answer:

\( 0x0 - 0x1FFF \)

\( 0x400000 - 0x401FFF \)

\( 0x800000 - 0x801FFF \)
Question 1.b: Basic page tables

Problem description:

cr3 = 0x0

PD at address 0x0:

0 -> 0x1
1 -> 0x2
2 -> 0x1

PT at address 0x1000:

0 -> 0x3
1 -> 0x4

PT at address 0x2000:

0 -> 0x5
1 -> 0x4

Page directory is at PHYSICAL address 0x0

You need to find a mapping from some virtual address into physical 0x0

Question: what is the virtual address of PD
Problem description:

cr3 = 0x0

PD at address 0x0:
0 -> 0x1
1 -> 0x2
2 -> 0x1

PT at address 0x1000:
0 -> 0x3
1 -> 0x4

PT at address 0x2000:
0 -> 0x5
1 -> 0x4

Question: what is the virtual address of PD

How to find it?

1. Look through page table mappings. You need to find an entry which map to 0x0

2. If you found, traverse to page directory and find index of the PDE corresponding for this PT

3. Create an address.

   1. Index in PD - first 10 bits
   2. Index in PT - middle 10 bits
   3. Offset - last 12 bits of physical address of PD (in our case 0x0)

If haven’t found - there is no mapping

Question 1.b: Basic page tables

Is equal 0x0?
Problem description:

int foo(int a) {
    ...
}  <- stopped here

int bar(int a, int b) {
    ...
}
foo(...); ...
}

int baz(int a, int b, int c) { ...
foo(...); ...
}

Question: What is in the stack?

Stack:

0x8010b5b8: ...
0x8010b5b4: 0x00010074
0x8010b5b0: 0x00000002
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e80
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b59c: 0x00000001
0x8010b598: 0x80102e32
0x8010b594: 0x8010b5a4  <- ebp
0x8010b590: 0x00000000  <- esp
0x8010b590: 0x00000000  <- esp

To solve remember how stack looks like in general when you just entered a function:

1. First local variable
2. ...
3. Last local variable
4. esp
5. ebp
6. Last function arg
7. ....
8. First function arg
9. Return address
10. Local variables  <- caller
11. Old ebp  <- caller
Question 2.a: Stack and calling conventions

Problem description:

```c
int foo(int a) {
    ...
} <- stopped here

int bar(int a, int b) {
    foo(...); ...
}

int baz(int a, int b, int c) {
    foo(...); ...
}
```

Question: What is in the stack?

Stack:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8010b5b8</td>
<td>...</td>
</tr>
<tr>
<td>0x8010b5b4</td>
<td>0x00010074</td>
</tr>
<tr>
<td>0x8010b5b0</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0x8010b5ac</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x8010b5a8</td>
<td>0x80102e80</td>
</tr>
<tr>
<td>0x8010b5a4</td>
<td>0x8010b5b8</td>
</tr>
<tr>
<td>0x8010b5a0</td>
<td>0x80112780</td>
</tr>
<tr>
<td>0x8010b59c</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x8010b598</td>
<td>0x80102e32</td>
</tr>
<tr>
<td>0x8010b594</td>
<td>0x8010b5a4</td>
</tr>
<tr>
<td>0x8010b590</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

To solve remember how stack looks like in general when you just entered a function:

1. **First local variable**
2. **...**
3. **Last local variable**
4. esp
5. ebp
6. Last function arg
7. ...
8. **First function arg**
9. Return address
10. **Local variables** <- caller
11. Old ebp <- caller
Problem description:

```c
int foo(int a) {
    ...
} <- stopped here

int bar(int a, int b) { ...
foo(...); ...
}

int baz(int a, int b, int c) { ...
foo(...); ...
}
```

Question: What is in the stack?

Stack:

```
0x8010b5b8: ...
0x8010b5b4: 0x00010074
0x8010b5b0: 0x00000002
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e80
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b59c: 0x00000001
0x8010b598: 0x80102e32
0x8010b594: 0x8010b5a4 <- ebp
0x8010b590: 0x00000000 <- esp
```

To solve remember how stack looks like in general when you just entered a function:

1. First local variable
2. ...
3. Last local variable
4. esp
5. ebp
6. Last function arg
7. ...
8. First function arg
9. Return address
10. Local variables <- caller
11. Old ebp <- caller

Don’t have
Don’t have
Don’t have
Already done
Problem description:

```c
int foo(int a) {
...
}                        <- stopped here

int bar(int a, int b) { ...
foo(...); ...
}

int baz(int a, int b, int c) { ...
foo(...); ...
}
```

Question: What is in the stack?

Stack:

```
0x8010b5b8: ...                   <- caller
0x8010b5b4: 0x00010074
0x8010b5b0: 0x00000002
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e80
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e32
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e32
0x8010b5a4: 0x8010b5b8
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e32
```

To solve remember how stack looks like in general when you just entered a function:

1. First local variable
2. ...
3. Last local variable
4. esp
5. ebp
6. Last function arg
7. ...
8. First function arg
9. Return address
10. Local variables
11. Old ebp
Problem description:

int foo(int a) {
    ...
}

int bar(int a, int b) {
    ...
}

int baz(int a, int b, int c) {
    ...
}

foo(...); ...

}  // end of baz

}  // end of bar

}  // end of foo

Question: What is in the stack?

Stack:

0x8010b5b8: ...
0x8010b5b4: 0x00010074
0x8010b5b0: 0x00000002
0x8010b5ac: 0x00000001
0x8010b5a8: 0x80102e80
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b59c: 0x00000001
0x8010b598: 0x80102e32
0x8010b594: 0x8010b5a4  <-- ebp
0x8010b590: 0x00000000  <-- esp

To solve remember how stack looks like in general when you just entered a function:

1. First local variable
2. ...
3. Last local variable
4. esp
5. ebp
6. Return address
7. Last function arg
8. ...
9. First function arg
10. Local variables  <-- caller
11. Old ebp  <-- caller

Question 2.a: Stack and calling conventions

Stack:

0x8010b5b8: ...
0x8010b5b4: 0x00010074
0x8010b5b0: 0x00000002
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e80
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b59c: 0x00000001
0x8010b598: 0x80102e32
0x8010b594: 0x8010b5a4  <-- ebp
0x8010b590: 0x00000000  <-- esp
**Question 2.a: Stack and calling conventions**

**Problem description:**

```c
int foo(int a) {
...
}
```

```c
int bar(int a, int b) {
...
}
```

```c
int baz(int a, int b, int c) {
...
}
```

**foo(...);** ...

```c
int foo(int a) {
...
} <- stopped here
```

```c
int bar(int a, int b) {
...
}
```

```c
int baz(int a, int b, int c) {
...
}
```

**foo(...);** ...

**Question: What is in the stack?**

**Stack:**

```
0x8010b5b8: ...
0x8010b5b4: 0x00010074
0x8010b5b0: 0x00000002
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e80
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b59c: 0x00000001
0x8010b598: 0x80102e32
0x8010b594: 0x8010b5a4  <-- ebp
0x8010b590: 0x00000000  <-- esp
```

To solve remember how stack looks like in general when you just entered a function:

1. **First local variable**
   - Don’t have
2. **…**
   - Don’t have
3. **Last local variable**
   - Don’t have
4. **esp**
   - Already done
5. **ebp**
   - Already done
6. **Return address**
7. **Last function arg**
8. **…**
9. **First function arg**
10. **Local variables**  <-- caller
11. **ebp**  <-- caller
### Question 2.a: Stack and calling conventions

**Problem description:**

```c
int foo(int a) {
  ...
}
```

```c
int bar(int a, int b) {
  foo(...);
}
```

```c
int baz(int a, int b, int c) {
  foo(...);
}
```

**Question:** What is in the stack?

#### Stack:

```
0x8010b5b8: ...
0x8010b5b4: 0x00010074
0x8010b5b0: 0x00000002
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e80
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b59c: 0x00000001
0x8010b598: 0x80102e32
0x8010b594: 0x8010b5a4
0x8010b590: 0x00000000
```

To solve remember how stack looks like in general when you just entered a function:

1. **First local variable**
2. ...
3. **Last local variable**
4. `esp`
5. `ebp`
6. **Return address**
7. **Last function arg**
8. ...
9. **First function arg**
10. **Local variables** <- caller
11. **Old ebp** <- caller

Don’t have

Don’t have

Don’t have

Already done

Already done

Don’t have

Already done

Don’t have
Problem description:

```c
int foo(int a) {
...
} <- stopped here

int bar(int a, int b) {
...
foo(...); ...
}

int baz(int a, int b, int c) {
...
foo(...); ...
}
```

Question: What is in the stack?

Stack:

```
0x8010b5b8: ...
0x8010b5b4: 0x00010074
0x8010b5b0: 0x00000002
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e80
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b59c: 0x00000001
0x8010b598: 0x80102e32
0x8010b594: 0x8010b5a4 <-- ebp
0x8010b590: 0x00000000 <-- esp
```

To solve remember how stack looks like in general when you just entered a function:

1. First-local-variable
2. ... Don’t have
3. Last-local-variable Don’t have
4. esp Already done
5. ebp Already done
6. Return-address
7. Last-function-arg
8. ... Don’t have
9. First-function-arg
10. Local variables <- caller
11. Old ebp <- caller
Problem description:

```c
int foo(int a) {
...
}

int bar(int a, int b) {
foo(...); ...
}

int baz(int a, int b, int c) {
foo(...); ...
}
```

Question: What is in the stack?

Stack:

```
0x8010b5b8: ...                <- stopped here
0x8010b5b4: 0x00010074
0x8010b5b0: 0x00000002
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e80
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b59c: 0x00000001
0x8010b598: 0x80102e32 0x8010b5a4
0x8010b594: 0x8010b590 0x00000000
```

To solve remember how stack looks like in general when you just entered a function:

1. First-local-variable
2. ...
3. Last-local-variable
4. esp
5. ebp
6. Return-address
7. Last-function-arg
8. ...
9. First-function-arg
10. Local-variables <- caller
11. Old ebp <- caller

Question 2.a: Stack and calling conventions

Stack:

```
0x8010b5b8: ...
0x8010b5b4: 0x00010074
0x8010b5b0: 0x00000002
0x8010b5ac: 0x00000001
0x8010b5a8 0x80102e80
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b59c: 0x00000001
0x8010b598: 0x80102e32 0x8010b5a4
0x8010b594: 0x8010b590 0x00000000
```

Old ebp

Local var or esp

Argument a of foo

Return address

<- ebp

<- esp

Don’t have

Don’t have

Don’t have

Already done

Already done

Return address

Argument a of foo

Local var or esp

Old ebp
Problem description:

```c
int foo(int a) {
    ...
}

int bar(int a, int b) {
    foo(...); ...
}

int baz(int a, int b, int c) {
    foo(...); ...
}
```

Question: What is in the stack?

Stack:

```
0x8010b5b8: ...  # 0
0x8010b5b4: 0x00010074  # Argument 3 of baz
0x8010b5b0: 0x00000002  # Argument 2
0x8010b5ac: 0x00000001  # Argument 1
0x8010b5a8 0x80102e80  # Return address
0x8010b5a4: 0x8010b5b8  # Old ebp
0x8010b5a0: 0x80112780  # Local var or esp
0x8010b59c: 0x00000001  # Argument a of foo
0x8010b598: 0x80102e32  # Return address
0x8010b594: 0x8010b5a4  # <-- ebp
0x8010b590: 0x00000000  # <-- esp
0x8010b5b4: 0x00000002  # Already done
0x8010b5b0: 0x00000000  # Already done
0x8010b5ac: 0x00000001  # Already done
0x8010b5a8 0x80102e80  # Already done
0x8010b5a4: 0x8010b5b8  # Already done
```

To solve remember how stack looks like in general when you just entered a function:

1. First local variable
2. ... (Don’t have)
3. Last local variable (Don’t have)
4. esp (Don’t have)
5. ebp (Already done)
6. Return address (Already done)
7. Last function arg (Already done)
8. ... (Already done)
9. First function arg
10. Local variables (Already done) <- caller
11. Old ebp (Already done) <- caller

Question 2.a: Stack and calling conventions
Question 2.b: Stack and calling conventions

Problem description:

```c
int foo(int a) {
    ...
    <- stopped here
}
int bar(int a, int b) {
    ...
    foo(...); ...
}
int baz(int a, int b, int c) {
    ...
    foo(...); ...
}
```

Stack:

```
0x8010b5b8: ...
0x8010b5b4: 0x00010074  Argument 3 of baz or local variable
0x8010b5b0: 0x00000002  Argument 2
0x8010b5ac: 0x00000001  Argument 1
0x8010b5a8 0x80102e80  Return address
0x8010b5a4: 0x8010b5b8  Old ebp
0x8010b5a0: 0x80112780  Local var or esp
0x8010b59c: 0x00000001  Argument a of foo
0x8010b598: 0x80102e32  Return address
0x8010b594: 0x8010b5a4  <-- ebp
0x8010b590: 0x00000000  <-- esp
```

Question: Can Alice make a conclusion if foo() is called from the context of bar() or baz()

Answer:

We can't decide which function called foo, if ebp in 0x8010b5a4 would point to 0x8010b5b4 then we could say that foo was called from a function that takes two arguments, i.e., bar but since we don't know what is there in 0x8010b5b4 we can't make this decision.
Question 3: Process organization

Problem description:

xv6 processes have the following memory layout created as part of the `exec()` function. First, the kernel allocates pages for the kernel text and data (not that these pages are both executable and writable). Then xv6 allocates two pages: stack and guard. The guard page is made is placed between the stack and the rest of the program to make sure that if the stack overflows the operating system can catch an exception caused by the access to the guard page and terminate the program early.

Question: is it possible to write a C program that escapes the guard page mechanism and accidentally overwrites the text section of the program
Question 3: Process organization

Problem description:

xv6 processes have the following memory layout created as part of the `exec()` function. First, the kernel allocates pages for the kernel text and data (not that these pages are both executable and writable). Then xv6 allocates two pages: stack and guard. The guard page is made is placed between the stack and the rest of the program to make sure that if the stack overflows the operating system can catch an exception caused by the access to the guard page and terminate the program early.

Question: is it possible to write a C program that escapes the guard page mechanism and accidentally overwrites the text section of the program

Answer:

Yes, it is possible to write a C program that escapes the guard page mechanism. If a C program has a local variable that is of size greater than 2 pages, we would skip the guard page and overwrite the text and data section.

Char xv6_hacked[PAGE_SIZE*2];

Int this_variable_is_allocated_in_text_section = 228;
Question 4.a: Physical and virtual memory allocation

Question: How xv6 keep track of available physical memory (using kalloc function)?

Original question: Xv6 uses 234MB of physical memory. But how does it keep track of available physical memory? Specifically, explain the following: the xv6 memory allocator (kalloc()) always returns a virtual address, but how does the allocator know which physical page to use for each virtual address it allocates?

How to solve questions like that:

1. Open xv6 source code: https://github.com/mit-pdos/xv6-public

2. Search for kalloc

3. Open a function and try to understand what’s going on
Question 4.a: Physical and virtual memory allocation

Question: How xv6 keep track of available physical memory (using kalloc function)?

1. Synchronization lock
2. Getting a linked list of available spaces
3. Pop first element from the list
4. Release the lock
Question 4.a: Physical and virtual memory allocation

Question: How xv6 keep track of available physical memory (using kalloc function)?

How you found out it is a linked list of free spaces?

- Look like linked list
- freerange calls kfree on every page available
- Init function calls freerange
- Add page into linked list
Question: Xv6 defines the V2P() macro that allows the kernel to convert between virtual and physical addresses:

#define V2P(a) (((uint) (a)) - KERNBASE)

Does V2P() macro work for virtual addresses that belong to the user part of the address space (i.e., a virtual address inside the user data or stack)?
Question: Xv6 defines the V2P() macro that allows the kernel to convert between virtual and physical addresses:

```
#define V2P(a) (((uint) (a)) - KERNBASE)
```

Does V2P() macro work for virtual addresses that belong to the user part of the address space (i.e., a virtual address inside the user data or stack)?

Answer: No, because the V2P mapping for kernel is simple - kernel is physically located at 0x0, but virtually at 2gb. It is not true for user programs. You need to go through page table mechanism.
Question: Xv6 defines the V2P() macro that allows the kernel to convert between virtual and physical addresses:

#define V2P(a) (((uint) (a)) - KERNBASE)

Does V2P() macro work for virtual addresses that belong to the user part of the address space (i.e., a virtual address inside the user data or stack)?

Answer: No, because the V2P mapping for kernel is simple - kernel is physically located at 0x0, but virtually at 2gb. It is not true for user programs. You need to go through page table mechanism
Problem description:

```c
#include "param.h"
#include "types.h"
#include "user.h"  #include "syscall.h"

int main() {
    char * message = "aaa\n";
    int pid = fork();
    if(pid != 0){
        char *echoargv[] = { "echo", "Hello\n", 0 };
        message = "bbb\n";
        exec("echo", echoargv);
    }
    write(1, message, 4);
    exit();
}
```

Question: What is the output

The fundamental question here is who would run first child or parent?
int main() {  
    char * message = "aaa\n";  
    int pid = fork();  
    if(pid != 0){  
        char *echoargv[] = { "echo", "Hello\n", 0 };  
        message = "bbb\n";  
        exec("echo", echoargv);  
    }  
    write(1, message, 4);  
    exit();  
}  

Question: What is the output

The fundamental question here is who would run first child or parent?

It is undefined

Answer:

There are two possible outputs:

1. aaa
2. Hello

Aaa
Problem description:

What would be if we remove mapping 0-4MB (Virtual) -> 0-4MB (Physical) from entrypgdir:

```c
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
    // Map VA's [0, 4MB) to PA's [0, 4MB)
    // [0] = (0) | PTE_P | PTE_W | PTE_PS,
    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
    [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

Question 6: Initial page tables

How to solve:

1. Open source code
2. Find entrypgdir
3. Try to analyze what's going on

What about those guys? Would they be executed?
Problem description:

What would be if we remove mapping 0-4MB (Virtual) -> 0-4MB (Physical) from entrypgdir:

```c
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
    // Map VA's [0, 4MB) to PA's [0, 4MB)
    // [0] = (0) | PTE_P | PTE_W | PTE_PS,

    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
    [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

Question 6: Initial page tables

Answer:

The code wouldn’t run, because as entry.S would load the page directory all other setup instructions would not be available anymore